

2013

SERVICE-BASED AUTOMATION OF SOFTWARE CONSTRUCTION ACTIVITIES

Zinn, Marcus

<http://hdl.handle.net/10026.1/2862>

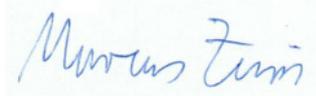
<http://dx.doi.org/10.24382/3396>

University of Plymouth

All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.

Copyright statement

This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's prior consent.

A handwritten signature in blue ink, appearing to read "Marcus Junii", is centered on a light blue rectangular background.

***SERVICE-BASED AUTOMATION OF SOFTWARE
CONSTRUCTION ACTIVITIES***

by

Marcus Zinn

A thesis submitted to the School of Computing
and Mathematics in partial fulfilment of the
requirements for the degree of

Doctor of Philosophy

In collaboration with
Darmstadt Node of NRG Network
at University of Applied Sciences Darmstadt

September 2013

Abstract

Service-based automation of software construction activities

Marcus Zinn

Master of Science

The reuse of software units, such as classes, components and services require professional knowledge to be performed. Today a multiplicity of different software unit technologies, supporting tools, and related activities used in reuse processes exist. Each of these relevant reuse elements may also include a high number of variations and may differ in the level and quality of necessary reuse knowledge. In such an environment of increasing variations and, therefore, an increasing need for knowledge, software engineers must obtain such knowledge to be able to perform software unit reuse activities. Today many different reuse activities exist for a software unit. Some typical knowledge intensive activities are: transformation, integration, and deployment. In addition to the problem of the amount of knowledge required for such activities, other difficulties also exist. The global industrial environment makes it challenging to identify sources of, and access to, knowledge. Typically, such sources (e.g., repositories) are made to search and retrieve information about software units and not about the required reuse activity knowledge for a special unit. Additionally, the knowledge has to be learned by inexperienced software engineers and, therefore, to be interpreted. This interpretation may lead to variations in the reuse result and can differ from the estimated result of the knowledge creator. This makes it difficult to exchange knowledge between software engineers or global teams. Additionally, the reuse results of reuse activities have to be repeatable and sustainable. In such a scenario, the knowledge about software reuse activities has to be exchanged without the above mentioned problems by an inexperienced software engineer. The literature shows a lack of techniques to store and subsequently distribute relevant reuse activity knowledge among software engineers. The central aim of this thesis is to enable inexperienced software engineers to use knowledge required to perform reuse activities without experiencing the aforementioned problems. The reuse activities: transformation, integration, and deployment, have been selected as the foundation for the research. Based on the construction level of handling a software unit, these activities are called Software Construction Activities (SCAc) throughout the research. To achieve the aim, specialised software construction activity models have been created and combined with an abstract software unit model. As a result, different SCAC knowledge is described and combined with different software unit artefacts needed by the SCACs. Additionally, the management (e.g., the execution of an SCAC) will be provided in a service-oriented environment. Because of the focus on reuse activities, an approach which avoids changing the knowledge level of software engineers and the abstraction view on software units and activities, the object of the investigation differs from other approaches which aim to solve the insufficient reuse activity knowledge problem. The research devised novel abstraction models to describe SCACs as knowledge models related to the relevant information of software units. The models and the focused environment have been created using standard technologies. As a result, these were realised easily in a real world environment. Software engineers were able to perform single SCACs without having previously acquired the necessary knowledge. The risk of failing reuse decreases because single activities can be performed. The analysis of the research results is based on a case study. An example of a reuse environment has been created and tested in a case study to prove the operational capability of the approach. The main result of the research is a proven concept enabling inexperienced software engineers to reuse software units by reusing SCACs. The research shows the reduction in time for reuse and a decrease of learning effort is significant.

Contents

Abstract	iii
Contents	iv
List of figures	viii
List of tables	xii
List of abbreviations and acronyms	xiv
Acknowledgements	xvii
Author's Declaration	xviii
1. Introduction	1
1.1. Missing activity knowledge in software unit reuse	2
1.2. Focus of research	5
1.3. Aims and objectives	9
1.4. Method overview and thesis outline	12
2. Problem of missing knowledge in software unit reuse	17
2.1. Secondary research methods	17
2.2. Software reuse academic background	20
2.2.1. Term definitions and software reuse aspects.....	20
2.2.2. Discussion of key definitions.....	37
2.2.3. Missing knowledge in software unit reuse	47
2.3. Conceivable research contribution	59
2.4. Summary	60
3. Missing software construction activity knowledge – problem analysis	63
3.1. Focused software construction activities	63
3.1.1. Console-based transformation of software units – transformation activities	64
3.1.2. Transformation activity example	65
3.1.3. Integration of software units in development environments – integration activities.....	69
3.1.4. Integration activity example.....	70
3.1.5. Deployment of software units into embedded devices – deployment activities.....	72
3.1.6. Deployment activity example.....	73
3.2. Problem analysis	75
3.2.1. Explanation of analysis contexts.....	76
3.2.2. Knowledge storing problem	86
3.2.3. Knowledge learning problem.....	90
3.2.4. Problem of searching and receiving of knowledge.....	93
3.2.5. Knowledge exchange problem.....	97
3.2.6. The problem of knowledge execution	100

3.2.6.	Problem significance.....	103
3.3.	Missing solution approaches	104
3.4.	Summary.....	106
4.	A general approach to realise knowledge-based automated reuse activities	108
4.1.	The basic idea	108
4.2.	Focused user profiles and scenarios.....	113
4.2.1.	Knowledge user (KU) – Reuse of software units	114
4.2.2.	Knowledge creator (KC) – Provision of software units and reuse activities knowledge	116
4.3.	Focused development project scenarios	118
4.3.1.	Separate user development projects	118
4.3.2.	Separate team development projects	120
4.4.	The fundamental concept.....	123
4.4.1.	Software construction as concept bases	123
4.4.2.	Relevant elements of the concept	130
4.4.3.	Use cases.....	134
4.5.	Concept of potential technical environments.....	136
4.5.1.	Communication concept.....	136
4.5.2.	Software unit model.....	146
4.5.3.	Reuse activity models.....	148
4.5.4.	Extensibility	149
4.6.	Summary.....	156
5.	Solution realisation.....	159
5.1.	Development approach.....	159
5.2.	Selected technical environment	159
5.2.1.	Distribution model and relevant architecture elements	161
5.2.2.	Interface definitions.....	161
5.2.3.	Used technologies and communication protocols	193
5.2.4.	Extensibility approaches.....	195
5.3.	Realised models.....	196
5.3.1.	Software Unit Model instance.....	196
5.3.2.	Transformation model instance.....	207
5.3.3.	Deployment Model instance	210
5.3.4.	Integration model instance	213
5.4.	Usage concepts	215
5.4.1.	Focused use cases.....	216
5.4.2.	Knowledge creator profile use cases	217

5.4.3.	Knowledge user profile use cases	238
5.5.	Summary.....	245
6.	Evaluation and research result analysis	247
6.1.	Focused case study and scientific viewpoint	247
6.1.1.	General overview about the case study.....	247
6.1.2.	Scientific case study theory.....	248
6.1.3.	Case study hypothesis	249
6.2.	Research theory and methods.....	249
6.2.1.	Relating focused problems and research question	250
6.2.2.	Selected research type	250
6.2.3.	Used research theory, methods and application area.....	251
6.2.4.	Theoretical viewpoint	252
6.2.5.	Methods	252
6.3.	Case study setup, procedure and measurement model	257
6.3.1.	Application domain	258
6.3.2.	General case study sequence.....	265
6.3.3.	Measurement and experiment results overview	266
6.4.	Result analysis.....	270
6.4.1.	Analysis methods definition	270
6.4.2.	Case study result analysis.....	275
6.5.	Case study hypothesis review.....	305
6.5.	Summary.....	306
7.	Conclusion	309
7.1.	Summary and achievements of the research.....	309
7.2.	Research contribution discussion.....	313
7.2.1.	Contribution to the problem of different technologies.....	314
7.2.2.	Contribution to the problem of different knowledge levels.....	315
7.2.3.	Contribution to the problem of knowledge distribution	317
7.2.4.	Contribution to definition of software reuse knowledge	319
7.2.5.	Final statement.....	320
7.3.	Objectives and limitations of research.....	320
7.4.	Future work.....	322
7.4.1.	Neutralisation of the limitations.....	322
7.4.2.	Extended research	323
7.5.	Technology review and epilogue.....	325
8.	Bibliography	xxi
9.	Appendix	xli

9.1.	Content of data medium.....	xli
9.2.	Methodology of literature review	xli
9.2.1.	Categories	xli
9.2.2.	Stages of a literature review	xliii
9.2.3.	Problem formulation.....	xliv
9.2.4.	Data collection	xlvi
9.2.5.	Data evaluation.....	xlviii
9.2.6.	Data analysis and interpretation	l
9.2.7.	Data presentation.....	liii
9.3.	Document evaluation protocol example.....	lv
9.4.	Additional research on software unit base technologies	lvii
9.4.1.	Object-oriented software construction.....	lvii
9.4.2.	Component-based software construction	lxii
9.4.3.	Service-based software construction	lxxii
9.5.	Schneider Electric internal reuse study and key notes	lxxxii
9.6.	Case study measurement value results.....	lxxxiii
9.7.	Additional identified problems and requirements for future studies	lxxxix
9.7.1.	Identified problems for software engineers.....	lxxxix
9.7.2.	Problem selection for reuse activities.....	xcvii
9.7.3.	Requirements definition based on focused Problems	cxiv
9.8.	Information demand model for software unit reuse	cxxxiv
9.8.1.	Description of information demand.....	cxxxiv
9.8.2.	Definition of information demand from the software unit reuse perspective.....	cxxxvi
9.8.3.	Use of information demand.....	cxxxvii
9.9.	Published papers.....	cxli

List of figures

Figure 1 - Relationship between aim and objectives.....	10
Figure 2 - Used research methods.....	14
Figure 3 - Landscape of reuse (Sommerville, 2011 p.429)	26
Figure 4 - Relation between relevant reuse types.....	33
Figure 5 - The knowledge pyramid (Ackoff, 1989, p. 5)	44
Figure 6 - Example of the DIKW hierarchy (Rowley, 2007, p. 186).....	45
Figure 7 - Software Reuse Information Demand Model based on Picot (2003, p. 106).....	53
Figure 8 - Users experience level (based on Ye, 2001, p.2)	54
Figure 9 - Dependency hierarchy of DPWS Java libraries (blue external libraries; red internal libraries)	66
Figure 10 - Example IKVM transformation execution	67
Figure 11 - Example IKVM transformation execution with dependency.....	67
Figure 12 - Example IKVM transformation execution of the DPWS4J.JAR file	68
Figure 13 - Dependency hierarchy of DPWS .NET libraries (ILSpy View)	68
Figure 14 - Dependency hierarchy of DPWS .NET libraries (blue external libraries; red internal libraries).....	69
Figure 15 - DPWS integration activity folder and project structure.....	72
Figure 16 - Single problem visualisation (a) and multiple problem visualization (b)	76
Figure 17 - Creation of a multiview SRID model out of single view SRID models.....	77
Figure 18 - Horizontal vs. vertical markets	79
Figure 19 - Distribution of reusable software units (Schoop, 2012).....	81
Figure 20 - Development distribution (a) STwS (b) MTwS; and MTwoS (c) based on Schoop (2012)	82
Figure 21 - Use of relevant software units in different business units	84
Figure 22 - Reused units in software development (based on Schneider; (cf. Appendix Section E))	85
Figure 23 - Future improvements identified by the study of Schneider Electric 2012 (Brick = software unit).....	85
Figure 24 - Example SRID model for knowledge storing	89
Figure 25 - Example process for knowledge interpretation	90
Figure 26 - Creation of the problem of repository localisation	96
Figure 27 - SRID model for the problem of search request formulation based on Picot (2003, p. 106).....	96
Figure 28 - SRID model for information demand for search and receipt of knowledge.....	97
Figure 29 - Example of focused reuse steps	109
Figure 30 - Basic Idea of this thesis.....	110
Figure 31 - Concepts used for problem solving	110
Figure 32 - Single KC and Single KU relation	118
Figure 33 - Single KC related to multiple KU	119
Figure 34 - Multiple KC related to single KU	119
Figure 35 - Single KC related to multiple related KU.....	120
Figure 36 - Single KC related to multiple-non separated teams.....	121
Figure 37 - Single KC related to multiple separated teams.....	121
Figure 38 - Parts of the Service-oriented Software Construction Process.....	123
Figure 39 - Content of a Software Construction Artefact.....	125
Figure 40 - Service interfaces of an SCA.....	127
Figure 41 - Reuse of software construction artefacts	127
Figure 42 - Automation concept	128
Figure 43 - UI as abstraction layer for the focused environment.....	129
Figure 44 - Data content of the SSCP environment.....	129
Figure 45 - Requesting knowledge inside the SSCP environment (Transformation activity example).....	130
Figure 46 - Use of the Software Unit model in the focused environment	131
Figure 47 - Communication concept of knowledge in the focused concept.....	132
Figure 48 - Use of active and passive knowledge	134
Figure 49 - Overview of the supported use cases.....	135
Figure 50 - Knowledge injection scenario	138

List of figures

Figure 51 - Knowledge extraction scenario.....	138
Figure 52 - Request for reuse activity execution.....	140
Figure 53 - Monolith scenario.....	142
Figure 54 - Completely distributed scenario.....	142
Figure 55 - Data-driven communication.....	143
Figure 56 - ID-driven communication.....	144
Figure 57 - Example of distribution of business logic for the concept-based environment.....	145
Figure 58 - Standardisation of the view on services, components, and classes.....	146
Figure 59 - Areas of the Software Unit Model.....	146
Figure 60 - Relevant views of the Software Unit Model.....	147
Figure 61 - Software Unit Model as fundamental information base for reuse activities.....	148
Figure 62 - Extension concept of reuse activity models.....	151
Figure 63 - Reducing view complexity on different repositories.....	152
Figure 64 - Concept environment as repository.....	154
Figure 65 - Typical development environments.....	155
Figure 66 - Multiple client system using the same service of the focused concept.....	156
Figure 67 - Prometheus architecture overview.....	161
Figure 68 - Overview of Prometheus server architecture.....	164
Figure 69 - Information flow of the Prometheus core.....	165
Figure 70 - Communication behaviour of a search request.....	166
Figure 71 - Distribution possibilities of the used Prometheus architecture.....	167
Figure 72 - Integration plugins for Visual Studio and Eclipse.....	169
Figure 73 - Relevant interfaces in the Prometheus architecture.....	172
Figure 74 - Relevant interfaces of the User Client (UC)-Plugin.....	173
Figure 75 - Advanced interface of the UC-Plugins.....	176
Figure 76 - Additional support Interface of the UC-Plugin.....	185
Figure 77 - C# notation of the integration plugin interface.....	188
Figure 78 - C# notation of the transformation client Interface.....	189
Figure 79 - C# Notation of the deployment web service.....	190
Figure 80 - C# Notation of the Integration client plugin Interface.....	191
Figure 81 - Used technologies.....	193
Figure 82 - Model restrictions.....	197
Figure 83 - Relevant elements of the area 1 - stakeholder view (U-R1).....	198
Figure 84 - Relevant elements of the area 2 – problem and solution view” (U-R2).....	200
Figure 85 - Relevant elements of the area 3 – technical view (U-R3).....	203
Figure 86 - Relevant elements of the area 4 – content view (U-R4).....	205
Figure 87 - Relevant relations between the different areas of the U-Model.....	207
Figure 88 - Data model extension for transformation activities maintenance.....	208
Figure 89 - Data model extension for deployment activities Part 1.....	210
Figure 90 - Data model extension for deployment activities Part 2.....	211
Figure 91 - Data model extension for Integration activities.....	213
Figure 92 - Focused stakeholder of the Prometheus environment.....	216
Figure 93 - Use Case digramm for the focused Prometheus environment.....	217
Figure 94 - Activity diagram for Use Case ‘UOM Creation’.....	219
Figure 95 - UOM creation UI.....	220
Figure 96 - File element creation UI.....	220
Figure 97 - Activity diagram of use case ‘UOM SEARCH’.....	221
Figure 98 - User Interface for UOM search.....	222
Figure 99 - Activity diagram of use case ‘UOM DISCOVERY’.....	222
Figure 100 - Detailed presentation of a software unit.....	223
Figure 101 - Activity diagram of use case ‘UOM Adaptation’.....	224
Figure 102 - Wizard to add new UOM file information.....	224
Figure 103 - Activity diagram of use case ‘ACTIVITY CREATION’.....	225

List of figures

Figure 104 - Main UI for integration activity creation	226
Figure 105 - Integration steps.....	228
Figure 106 - Example of an UI for transformation tool setup	229
Figure 107 - Main UI for transformation activity creation.....	229
Figure 108 - Main User Interface for transformation rule creation.....	230
Figure 109 - Relevant UI areas of the transformation output definition wizard.....	231
Figure 110 - Transformation steps.....	232
Figure 111 - Relevant steps for deployment activity creation.....	234
Figure 112 - Activity diagram of the use case 'ACTIVITY SEARCH'	235
Figure 113 - User Interface for activity search.....	235
Figure 114 - Activity diagram of the use case 'ACTIVITY DISCOVERY'.....	236
Figure 115 - Activity diagram of the use case 'ACTIVITY Adaptation'.....	237
Figure 116 - UI Wizard for UOM adaptation.....	238
Figure 117 - UI for downloading UOM information.....	239
Figure 118 - Activity diagram of the use case 'UOM INFORMATION RETRIVAL'	239
Figure 119 - Activity diagram of the use case 'ACTIVITY EXECUTION'	240
Figure 120 - UI for activity execution (a) in UOM overview (b) in activity detail	240
Figure 121 - Activity diagram of the use case 'INTEGRATION ACTIVITY EXECUTION'.....	241
Figure 122 - UI for integration activity execution	242
Figure 123 - Configuration UI for IDE service endpoints	242
Figure 124 - Activity diagram of the use case 'TRANSFORMATION ACTIVITY EXECUTION'	243
Figure 125 - UI for transformation activity execution.....	244
Figure 126 - Activity diagram of the use case 'DEVICE DEPLOYMENT ACTIVITY EXECUTION'	245
Figure 127 - Experimental environment and setup.....	259
Figure 128 - Basic experiment scenario	260
Figure 129 - Overview measurable content	261
Figure 130 - Estimated results of experienced and inexperienced user	271
Figure 131 - Average results for the DPWS Java transformation SCAC.....	278
Figure 132 - Average results for the DPWS Java integration SCAC	280
Figure 133 - Average results for the DPWS C stack transformation SCAC.....	282
Figure 134 - Average results for the DPWS C integration SCAC	284
Figure 135 - Average results for the Log4J transformation SCAC.....	285
Figure 136 - Average results for the Log4J integration SCAC	287
Figure 137 - Average results for the Log4Net transformation SCAC	289
Figure 138 -Average results for the Log4Net integration SCAC.....	290
Figure 139 - Average results for the EWSJ transformation SCAC	292
Figure 140 - Average results for the EWS J integration SCAC.....	294
Figure 141 - Average results for the EWS .NET integration SCAC.....	295
Figure 142 - Average results for the EWS .NET transformation SCAC	296
Figure 143 - Average results all measured transformation SCAC.....	299
Figure 144 - Average results all measured integration SCAC	300
Figure 145 - Final comparison of inexperienced software engineers.....	304
Figure 146 - Comparison between supported and unsupported software engineers.....	312
Figure 147 - Sketched process for document separation	xlvi
Figure 148 - Example of a cookbook form for data analysis and interpretation	liii
Figure 149 - Document evaluation protocol example.....	lv
Figure 150 - UML like representation of a class.....	lviii
Figure 151 - Technical architecture of the .NET Platform	lx
Figure 152 - NET runtime environment communication (Computerbase 2008, online)	lxi
Figure 153 - UML component diagramm example (Ambler, 2008)	lxvii
Figure 154 - Web Service Choreographie and Orchestrierung (Peltz, 2003, p. 47)	lxxiv
Figure 155 - Multi-tier architecture using services (based on Jiang and Willeam 2005)	lxxx
Figure 156 - General structure of activities used for explanation.....	xcix

List of figures

Figure 157 - Relation between a general transformation activity and focused knowledge based problems xcix
Figure 158 - Relation between a general integration activity and focused knowledge based problems civ
Figure 159 - Relation between a general deployment activity and focused knowledge based problems cx
Figure 160 - Information flow of a activity reuse..... cxv
Figure 161 – Oriiginal information demand model by Picot (2003, p. 106).....cxxxv
Figure 162 - SRID model related to original information modelcxxxvii
Figure 163 - Critical success factors (of Frakes and Fox, 1996) in the SRID model..... cxxxviii

List of tables

Table 1 - Extension mechanism for reusable software units (based on Jansen et al., 2008)	24
Table 2 - Typical software units in different software reuse landscape approaches	29
Table 3 -Types of reuse (based on Prieto-Diaz, 1993)	30
Table 4 - Example of base technology/concept in the software reuse landscape approaches.....	36
Table 5 - Briefly description of the supported use cases	135
Table 6 - Used interface groups	173
Table 7 - Parameters of the search operation	174
Table 8 - Parameters of the GetItems/GetItemsAsZip operation	174
Table 9 - Parameters of the PerformTransformation operation	175
Table 10 - Parameters of the PerformIntegration operation	175
Table 11 - Parameters of the PerformDeployment operation	176
Table 12 - Parameters of the CreateArtefact operation	177
Table 13 - Parameters of the CreateUOM operation.....	178
Table 14 - Parameters of the AddData operation.....	179
Table 15 - Parameters of the RemoveItem operation.....	179
Table 16 - Parameters of the LoadIntegrationActivity operation	180
Table 17 - Parameters of the RemoveIntegration operation	180
Table 18 - Parameters of the Create/UpdatesIntegration operation	180
Table 19 - Parameters of the GetAvailableTransformationApplication operation	181
Table 20 - Parameters of the GetTransformationApplication operation	181
Table 21 - Parameters of the RemoveTransformationApplication operation	182
Table 22 - Parameters of the Create/TransformationApplication operation.....	182
Table 23 - Parameters of the LoadTransformationActivity operation	183
Table 24 - Parameters of the RemoveTransformation operation	183
Table 25 - Parameters of the Create/UpdateTransformation operation	183
Table 26 - Parameters of the LoadDeploymentActivity operation.....	184
Table 27 - Parameters of the RemoveDeploymentActivity operation.....	184
Table 28 - Parameters of the Create/UpdateDeployment operation	185
Table 29 - Parameters of the GetServiceInformation operation.....	185
Table 30 - Parameters of the GetAvailableRepositoryInformation operation.....	186
Table 31 - Parameters of the DoIntegration operation	188
Table 32 - Parameters of the DoTransformation operation	189
Table 33 - Parameters of the DoDeployment operation	190
Table 34 - Parameters of the SetTransferType method	191
Table 35 - Parameters of the ReceiveZip method	192
Table 36 - Defintion of elements of the stakeholder view (U1).....	198
Table 37 - Defintion of elements of the problem solution view (U2)	199
Table 38 - Defintion of elemens of the technical view (U3)	202
Table 39 - Classification of unit types	204
Table 40 - Defintion of elements of the content view (U4)	205
Table 41 - Case study software units	264
Table 42 - Case study scenario summary.....	265
Table 43 - Overview variables of comparison methods used in case study scenarios	267
Table 44 - Case study scenario related to measurement methods	268
Table 45 - Summary of measured values of an SCA performed by a participant.....	274
Table 46 - Summary of measured values of a Prometheus preparation.....	274
Table 47 - Average values example of a software construction activity.....	275
Table 48 - Measured values of each participant (DPWS4J transformation SCAC KR - Knowledge Resource)	276
Table 49 - Average values of the DPWS Java transformation SCAC (KR Knowledge Resource)	277
Table 50 - Average values of the DPWS Java integration SCAC (KR Knowledge Resource).....	279

List of tables

Table 51 - Average values of the DPWS C transformation SCAC (KR Knowledge Resource)	281
Table 52 - Average values of the DPWS C integration SCAC (KR Knowledge Resource).....	283
Table 53 - Average values of the Log4J transformation SCAC (KR Knowledge Resource).....	284
Table 54 - Average values of the Log4J integration SCAC (KR Knowledge Resource).....	286
Table 55 - Average values of the Log4Net transformation SCAC (KR Knowledge Resource).....	288
Table 56 - Average values of the Log4Net integration SCAC (KR Knowledge Resource)	290
Table 57 - Average values of the EWSJ transformation SCAC (KR Knowledge Resource).....	291
Table 58 - Average values of the EWSJ integration SCAC (KR Knowledge Resource)	293
Table 59 - Average values of the EWS .NET integration SCAC (KR Knowledge Resource)	295
Table 60 - Average values of the EWS .NET transformation SCAC (KR Knowledge Resource).....	298
Table 61 - Setup time for focuses SCACs.....	303
Table 62 - Comparison of component-based procedural models (Stojanović, 2005).....	lxiv
Table 63 - Measured values for the DPWS Java Stack transformation.....	lxxxiii
Table 64 - Measured values for the DPWS Java Stack integration	lxxxiii
Table 65 - Measured values for the DPWS C Stack transformation.....	lxxxiv
Table 66 - Measured values for the DPWS C Stack integration	lxxxiv
Table 67 - Measured values for the Log4J transformation	lxxxv
Table 68 - Measured values for the Log4J integration.....	lxxxv
Table 69 - Measured values for the EWS .NET transformation.....	lxxxvi
Table 70 - Measured values for the EWS .NET integration	lxxxvi
Table 71 - Measured values for the EWS J transformation.....	lxxxvii
Table 72 - Measured values for the EWS J integration	lxxxvii
Table 73 - Measured values for the Log4.NET integration.....	lxxxviii
Table 74 - Measured values for the Log4.NET integration.....	lxxxviii
Table 75 - Single or combined visualisation.....	xcvii
Table 76 - Problems in the focused reuse activities	xcviii
Table 77 - Sub requirement list for the problem of key concepts.....	cxvii
Table 78 - Sub requirement list for the problem of different views	cxviii
Table 79 - Sub requirement list for the problem of multitude	cxix
Table 80 - Sub requirement list for the problem of different component models and worlds	cxx
Table 81 - Sub requirement list for the problem of availability	cxxi
Table 82 - Sub requirement list for the problem of context dependencies	cxxii
Table 83 - Sub requirement list for the problem of different perspectives	cxxii
Table 84 - Sub requirement list for the problem of completeness	cxxiii
Table 85 - Sub requirement list for the problem of knowledge storing	cxxiv
Table 86 - Sub requirement list for the problem of knowledge learning	cxxv
Table 87 - Sub requirement list for the problem of knowledge receiving	cxxvi
Table 88 - Sub requirement list for the problem of knowledge search	cxxvi
Table 89 - Sub requirement list for the problem of knowledge using	cxxvii
Table 90 - Sub requirement list for the problem of knowledge distribution	cxxviii
Table 91 - Sub requirement list for the problem of localisation (single).....	cxxix
Table 92 - Sub requirement list for the problem of localisation (single).....	cxxx
Table 93 - Sub requirement list for the problem of missing knowledge exchange	cxxx
Table 94 - Sub requirement list for the problem of reachable knowledge.....	cxxxii
Table 95 - Sub requirement list for the problem of excessive support requirements.....	cxxxii
Table 96 - Requirements relationship	cxxxiii

List of abbreviations and acronyms

A		EWS	Ecostructure Web Service
ACM	Association for Computing Machinery	EXE	Executable
AIS	Actual Information State	F	
API	Application Programming Interface	FTP	File transfer protocol
B		G	
BCF	Business Component Factory	GUI	Graphical User Interface
BIN	Binary file	GUID	Globally Unique Identifier
BPM	Business Process Management	H	
BPML	Business Process Modelling Language	HTML	Hypertext Markup Language
C		I	
CAD	Computer-Aided Design	IBM	International Business Machines Corporation
CCM	CORBA Component Model	ICP	Integration Client Plugins
CLI	Common Language Infrastructure	ID	Information Demand
CLS	Common Language Specification	ID	Identifier
CMI	Collaboration Management Infrastructure	IDE	Integrated Development Environment
CMS	Content Management System	IDL	Interface Definition Language
COM	Component Object Model	IEEE	Institute of Electrical and Electronics Engineers
Corba	Common Object Request Broker	IP	Internet Protocol / Information Provision
COTS	Commercial of the shelf	IQ	Information Query
CSC	Computer Science Corporations	IT	Information Technology
D		J	
DCOM	Distributed Component Object Model	JAR	Java Archive
DIKW	Data–information–knowledge–wisdom	JDBC	Java Database Connectivity
DLL	Dynamic Linked Library	JMS	Java Message Service
DPWS	Device Profile for Web Service	JNDI	Java Naming and Directory Interface
DSL	Domain specific language	JTA	Java Transaction API
DVD	Digital Video Disk	JVM	Java Virtual Machine
E		K	
EJB	Enterprise Java Beans	KC	Knowledge Creator
EOF	Entity Object Framework	KD	Knowledge Discovery
ERD	Entity-relationship diagram	KM	Knowledge Management

List of tables

KMS	Knowledge Management System	SCA	Software Construction Artefact
KU	Knowledge User	SCAc	Software Construction Activity
KV	Knowledge Vaporization	SCS	Software Construction Service
M		SDK	Software Development Kit
MDS	Model-driven software development	SID	Subjective Information Demand
MEF	Microsoft Extensible Framework	SME	Small and Medium sized Enterprises
MTwoS	Multiple Teams without Support	SOA	Service-oriented Architecture
MTwS	Multiple Teams with Support	SOAP	<i>Simple Object Access Protocol (not used anymore)</i>
O		SPI	Software Process Improvement
ODBC	Open Database Connectivity	SPL	Software Product Line
OID	Objective Information Demand	SQL	Structured Query Language
OMT	Object Modelling Technique	SRE	Software Reuse Environment
OO	Object orientation	SRID	Software Reuse Information Model
OOA	object-oriented analysis	SSCP	Service-based Software Construction
OOD	object-oriented design	STARS	Software Technology for Adaptable , Reliable Systems
OOP	object-oriented programming	STwoS	Single Team without Support
OPF	OPEN Process Framework	STwS	Single Team with Support
OS	Operating System	SVG	Scalable Vector Graphic
OSGi	<i>Open Services Gateway initiative (not used anymore)</i>	U	
OWL	Web Ontology Language	UC	User Client
R		UCP	User Client Plugin
RAS	Reuse Activity System	UDDI	Universal Description, Discovery and Integration
RCP	Repository Client Plugins	UI	User Interface
RDF	Resource Description Framework	UID	Unique Identifier
REST	Representational State Transfer	UML	Unified Modelling Language
RMI	Remote Method Invocation	UOM	Unit of Modelling
RNIF	RosettaNet Implementation Framework	URL	Unified Resource Identifier
RUP	Rational Unified Process	V	
T		VS	Visual Studio
TrCP	Transformation Client Plugins	W	
S		WCF	Windows Communication Foundation
SC	Software Construction	WS	Web Service
		WS-CDL	Web Service choreography Description Language

List of tables

WSCI *Web Service Choreography
Interface*

WSDL Web Service description
language

X

XAML Extensible Application Markup
Language

XML Extensible Markup Language

Acknowledgements

This thesis was produced during my work as a system architect, manager of software engineers and software architects in industry business, and the work as a scientific researcher at the University of Plymouth in cooperation with the University of Applied Science Darmstadt (HDA). It is the result of continuous technical discussion with business partners, colleagues, students, and the personal experience of industry projects, research projects, and the given teaching at the university.

This Ph. D. thesis could not have been produced without support from several people. Therefore, I would like to thank all parties involved. First I want to thank my supervisors.

Dr. Günther Turetschek and Dr. Klaus Peter Fischer-Hellmann, (First supervisors)
Dr. Alois Schütte (Second supervisor)
Dr. Andy D. Phippen (Third supervisor)

At this point it is relevant for me to say something about Mr. Turetschek: Mr. Turetschek died in December 2009, and unfortunately cannot see the results. I have to thank him for the scientific support, personal motivation, confidence, opportunities given and the clearance for this dissertation. Mr. Turetschek believed in the scientific and industrial relevance of this work, and it is, therefore, an honour for me to fulfil this study.

Thank you Mr. Turetschek for last 13 years. I will miss you.

It is also relevant for me to mention Mr. Klaus-Peter Fischer-Hellmann who takes over the position of the first supervisor. He gave me the same support that Mr. Günther Turetschek gave. Hopefully we can work together in the future.

Thank you very much Mr. Fischer-Hellmann.

I also want to thank the Schneider Electric Automation GmbH especially Dr. Ronald Schoop and Ralf Neubert. Schneider Electric was the business partner for the experimental study of this work.

I will also think back very positively about the years of studying for the Ph.D. at the University of Plymouth with my friend Benjamin Heckmann. This dissertation should not be the end of our teamwork.

A Ph.D. thesis cannot be written without the support of family and friends. Especially, Michael and Jenna. So finally I would like to thank my family Kerstin, Belana and Daniel for their motivation, their support and endurance.

Thank you to all my friends for supporting me and my aim.

Author's Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

This study was supported by Schneider Electric GmbH Automation Germany.

The postgraduate research project has been conducted and supervised under the agreement between the University of Plymouth and the University of Applied Sciences Darmstadt concerning the establishment of the Darmstadt Node of the NRG Network at the University of Applied Sciences Darmstadt (HDA).

Relevant publications, seminars and presentations were regularly attended at which work was often presented, and several papers were prepared for publication.

Presentation on internal conferences

Zinn, M.: *Service-based software construction process*, Proceedings of the 3th Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2007), 2007, Plymouth, UK, ISBN: 978-1-8410-2173-7, pp. 169-184.

Zinn, M., Turetschek, G. and Phippen, A.D.: *Definition of Software Construction Artefacts for Software Construction*, Proceedings of the 4th Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2008), 2008, Wrexham, UK, ISBN: 978-1-84102-196-6, pp. 79-90.

Zinn, M., Fischer-Hellmann, K.P. and Phippen, A.D.: *Development of a CASE-tool for the Service-Based Software Construction*, Proceedings of the 5th Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2009), 2009, Darmstadt, Germany, ISBN: 978-1-84102-236-9, pp. 134-144.

Publications and conference presentation

Zinn, M., Fischer-Hellmann, K.P., Phippen, A.D. and Schütte, A.: *Finding Reusable Units of Modelling - an Ontology Approach*, Proceedings of the 8th International Network Conference (INC 2010), 2010, Heidelberg, Germany, ISBN: 978-1-84102-259-8, pp. 377-386.

Zinn, M., Bepperling, A., Schoop, R., Phippen, A.D. and Fischer-Hellmann, K.P.: *Device services as reusable units of modelling in a service-oriented environment - An analysis case study*, Proceedings of the 2010 IEEE International Symposium on Industrial Electronic (ISIE 2010), 2010, Bari, Italy, ISBN 978-1-4244-6391-6, pp. 1728-1735.

Zinn, M., Fischer-Hellmann, K.P., Schütte A. and Phippen, A.D.: *Information Demand Model for Software Unit Reuse* Proceedings of the 20th International Conference on Software Engineering (SEDE 2011), 2011, Las Vegas, USA, ISBN: 978-1-880843-82-6, pp. 32-39.

Author's Declaration

Zinn M., Fischer K.P., Schütte A., Phippen A.D.: *Reusable Software Units Integration Knowledge in a Distributed Development Environment*, Proceedings of the 2nd International Workshop on Software Knowledge (SKY 2011), 2011, Paris, France, ISBN: 978-989-8425-82-9, pp. 24-35.

Zinn M., Fischer K.P., Schoop R.: *Reuseable software unit knowledge for device deployment*, Proceedings of the 3th conference of conception of complex automation systems (EKA 2012), 2012, Magdeburg, Germany, ISBN: 978-3-940961-72-3, pp. 99-110.

Zinn M., Fischer K.P., Schoop R.: *Case-based reasoning approach for re-use activities*, Proceedings of the 3th International Workshop on Software Knowledge (SKY 2012), 2012, Barcelona, Spain, ISBN: 978-989-8565-32-7, pp. 31-42.

Zinn M., Fischer K.P., Schoop R.: *Automated Reuse of Software Reuse Activities in an industrial environment – Case Study Results*, Proceedings of the 6th International Conference on Software Engineering Advances (ICSEA 2012), 2012, Lisbon, Portugal, ISBN: 978-1-61208-230-1, pp. 331-340.

Heckmann B., Zinn M., Phippen A. D., Moore D. J., Wentzel C.: *Economic Efficiency Control on Data Centre Resources in Heterogeneous Cost Scenarios*, Proceedings of 7th International Conference for Internet Technology and Secured Transactions (ICITST 2012), 2012, London UK, ISBN: 978-1-908320-08-7, pp. 675-679.

Other presentations

Zinn M.: *Service based Software Construction*, CeBIT, 3-8 March, Hannover, Germany, 2009.

Zinn M.: *Service based Software Construction*, Schneider Electric Global Architecture Meeting, 9 January, Seligenstadt, Germany, 2010.

Zinn M.: 13 presentations of research results in the Ph.D. seminar of the Darmstadt Node of NRG Network, Darmstadt, Germany, 2006-2013.

Word count of the main body of the thesis is: 76207

(Tables and captions not included)

Date & Signature: _____
Marcus Zinn

This is for my family.
My deepest gratitude to my wife
Kerstin,
my children
Belana, Daniel, and Lennard.
I love you!

1. Introduction

“...So spake Zeus in anger, whose wisdom is everlasting; and from that time he was always mindful of the trick, and would not give the power of unwearying fire to the Melia race of mortal men who live on the earth. But the noble son of lapetus outwitted him and stole the far-seen gleam of unwearying fire in a hollow fennel stalk. And Zeus who thunders in high was stun in spirit, and his dear heart was angered when he saw amongst men the far-seen ray of fire” (Hesoid and Evelyn-White, 1914, line 545).

Among other things, this excerpt from the Greek hero mythology ‘Theogony’ describes a human dilemma. Against the will of Zeus, *son of lapetus* (so called Prometheus) gave mankind the knowledge of certain skills, such as cooking or producing tools. To punish mankind, Zeus took the skill to make fire away from them. This meant that, although Prometheus had given them the necessary knowledge, mankind was no longer able to carry out relevant tasks like cooking or producing certain tools.

It seems that this punishment was only possible because mankind was unable to independently create an relevant detail necessary for these activities. The different activities like cooking food or producing tools depended on fire and the activities to create it. However, the people did not know how to generate fire by themselves. They were only able to keep an already existing fire burning. Prometheus helped the people again so this part of Theogony had a happy ending.

The dilemma came about because humans in this story are only able to conduct processes when they have the required knowledge for all the necessary sub-activities to manufacture parts, or they are able to use parts that have been made before. If this knowledge is absent, some activities of a process cannot be performed. As a result, the dependent processes cannot be performed at all, only incompletely or not in the calculated time. This dilemma, occupying the minds of poets and authors 2300 years ago, still exists today and is particularly apparent in manufacturing of products. Car manufacturers, for example, do not usually create all parts of a car by themselves and are only able to assemble the parts into a finished product. Zheng (2007)

shows the dilemma which arises if one of the individual parts is not available, the car cannot be completed.

Regarding the area of software this dilemma seems to be solved. Install tools and package management systems on different operation systems are able to perform the activities of installation and configuration of software for a user. The user has not to know these activities and can continue. But the problem of missing knowledge exists in the area of software development. Sandhu et al. (2010) and Böckle, Pohl and van der Linden (2005) describe a special field of software engineering attempting to develop software products (such as software units, software applications, and software product lines) by utilising those previously developed. Sommerville (2011) and Sandhu et al. (2010) call this software reuse, which should help to save resources, (e.g., time and money). In software reuse, existing software products are integrated into another software product because they solve a particular sub-problem and, therefore, create added value. Sandhu et al. (2010) and Garlan, Allen and Ockerbloom (2009) figured out that usually, trying to solve problems that can be solved by reuse existing functionality by themselves uses more resources than reusing an existing software unit.

However, Jha and O'Brien (2011) discuss the problem of missing reuse knowledge impacting the whole software development process. Therefore, reuse of existing software products needs knowledge which has to be available for the human user (e.g., software engineer). Regarding the Theogony, not knowing the knowledge of a reuse activity seems to be a dilemma.

1.1. Missing activity knowledge in software unit reuse

The mentioned impact of missing knowledge on software reuse is well-known, especially at the end of the last century, reuse based software development projects were analysed. For example, Mohagheghi et al. (2004), Edward, Ali, and Sherif (1999), Fichman and Kemerer (2001), and Schmidt (1999) show in their multi-project analysis that knowledge is one of the critical success factors of software reuse.

Introduction

Regarding the analysis results of Ravichandran and Rai (2003) and Ajila (2006), missing knowledge may also lead to an increase of required resources (i.e., time or costs) or to the complete failure of a development project. Jansen et al. (2008) also analysed different software development projects more recently and conclude the same result. Based on such results, different researchers focus on the exchange and application of knowledge (cf. Qu, Ji and Nsakanda, 2012; Choi, Lie and Yoo, 2010). Qu, Ji and Nsakanda (2012) stated that software development teams are knowledge intensive groups and the exchange of knowledge is relevant for the project success.

Inside this thesis, knowledge describes the information a software engineer has to know to perform a reuse activity. For example, if a software engineer wants to reuse a software unit (e.g., a Java component), the engineer has to know how to insert this component into the Eclipse development environment for further use. Another reuse activity example is the transformation of a software unit into another technology. The transformation of a Java component into a .NET component requires specific tools (e.g., IKVM by Frijters, 2011) and a list of settings with specific values. Such settings are related to the used transformation tool and the software unit to transform. This is knowledge that a software engineer has to know in order to perform the transformation activity. Here, missing knowledge can lead to reuse activity failure.

This thesis focuses on two types of knowledge. The first one is the above mentioned knowledge needed in order to perform a reuse activity. The second one is the knowledge required to exchange the other knowledge type between software engineers. This is the knowledge to store, search, and receipt and perform reuse activity knowledge by using a technical environment.

Also the scenarios which create a lack of knowledge for software engineers are well-known. Shiva and Shala (2007) discuss a typical scenario of young professionals (e.g., students) without any experience. Another example is the unsuitable knowledge of senior software engineers for a new project or task. Based on Boh (2008) this is a typical scenario based on

Introduction

missing knowledge for experienced software engineers. Ven et al. (2006) use the term knowledge vaporisation. It describes the scenario that existing knowledge gets lost. Typically, this happens if a person leaves a project and previous decisions cannot be implemented anymore. As a result, none of the remaining people has access to knowledge of this person.

Another fact is the multitude of different technologies that require different knowledge for software unit reuse. If the engineer is not aware of the necessary knowledge for reuse, the whole reuse process might be at risk. This is a dilemma comparable to the situation implied in the Greek myth. The knowledge problem is based on two relevant aspects:

1. The knowledge for reuse activities is based on the technology of the software unit it relates to (McCarey, Ó Cinnéide and Kushmerick, 2008). There are a high number of different software units and related technologies which require knowledge (Isoda, 1992). As a result, the amount of existing knowledge is huge.
2. In recent years the problem has been aggravated by the dramatic growth of new technologies, necessary tools, and the opportunities to use new software units. If this trend continues, an even greater amount of knowledge will be required to employ the technology for reusing software units successfully in future. Regarding the analysis of Ajila and Zheng (2004) knowledge is always increasing and, therefore, has to be maintained.

Garcia et al. (2006), Tsai et al. (2010), Ye and Fischer (2005), Bjørnson and Dingsøyr (2008), and Boh (2008) show in their discussion that approaches may exist in the area of knowledge management (i.e., search recognition and storing of knowledge). But this does not include the handling of reuse activity knowledge. Especially in the area of application and automation of reuse activity knowledge such approaches are rare (see also Bjørnson and Dingsøyr, 2008).

Next to the point that a suggestion system for reuse activities does not exist, there is another problem with knowledge received from a knowledge management system. As seen in the

discussion of Bjørnson and Dingsøyr (2008), Ajila and Zheng (2004), and Ajila (2006) about knowledge interpretations, this may lead to the wrong interpretation of knowledge. This means the reuse result might be not what the user or system that enters or creates the knowledge expects. Therefore, this may not result in a successful reuse. Furthermore, long learning processes may fail or may be useless because this reuse is never repeated by the same person, or the learning process was not sustainable to perform a reuse activity.

1.2. Focus of research

Qu, Ji and Nsakanda (2012) analysed different software development projects and stated that software engineers and teams are knowledge intensive. Success of a project depends on the sharing and execution of this knowledge. Choi, Lie and Yoo (2010) identify information technology (IT) as a relevant factor for the exchange and use of knowledge. McCarey, Ó Cinnéide and Kushmerick (2008) conclude that a lack of techniques to store and subsequently distribute reuse activity relevant software unit knowledge among software engineers also exists. As a result, software engineers are not supported by the techniques focusing on the problems described in Section 1.1.

In general, this research creates an added value on the aforementioned problem of unsuccessful reuse caused by missing or misinterpreted reuse activity knowledge in the field of software unit reuse. A technique will be created to limit the lack McCarey, Ó Cinnéide and Kushmerick (2008) have identified, but with a focus on a specific type of reuse activity. As a result, of this added value, software engineers require less knowledge to perform this type of reuse activity.

Thereby, a technique has to handle the following knowledge based problem areas which are the main focus of this research:

1. Problem of knowledge intensive technology

The first problem is the multitudes of different technologies of software units, environments, and tools necessary for a reuse activity. McCarey, Ó Cinnéide and Kushmerick (2008) discuss the relation between reuse activity knowledge and related

software unit technology. Software engineers have to handle the amount of different technologies (Isoda, 1992) which increases significantly (Ajila and Zheng, 2004). As a result, the numbers of possible reuse activities and the required knowledge for these increases, too.

2. Problem of knowledge level of software engineers

Ye and Fischer (2005) state that software engineers may have inadequate knowledge levels. This means they have no, or less, experience with reuse activities of a specific software unit (e.g., the transformation of a Java component into a .NET component). As a result, a software engineer has to gain knowledge to increase their knowledge level (cf. Qu, Ji and Nsakanda, 2012).

Additionally, the creation of variants by learning and the interpretation of activity knowledge may be a problem (based on the experience of individuals; cf. Johansson, Hall and Coquard, 1999). As discussed before, such variants can lead to non-adequate reuse results. This is combined with the problem that reuse activities (including the learning of knowledge and the setup of these activities) are time consuming tasks for each software engineer.

3. Problem of knowledge intensive distribution environment.

Relating to the secondary knowledge, the multitude of different existing repositories containing primary knowledge and information is the focused problem to search, receive and perform reuse activity knowledge. A software engineer has to know how each of these repositories can be used. This is similar to the multitudes of different technologies. Another related problem is the location. This problem is typical for global development teams placed on different locations all over the world. Often, team members have no idea about the repositories of other teams, how to locate them, and how to access them (cf. Qu, Ji and Nsakanda, 2012, Vlaar et al., 2008).

Introduction

Global software development projects are used as application areas of these problems in this thesis. Typically, the knowledge exchange between software development teams is difficult. Thereby, software engineers have the mentioned problems. (cf. Qu, Ji and Nsakanda, 2012)

The research aims to find a solution that enables inexperienced software engineers to perform reuse activities and, thereby, handle the aforementioned problems. Choi, Lie and Yoo (2010) conclude that a technical infrastructure support people to exchange and execute of knowledge. Qu, Ji and Nsakanda (2012) relate this statement to software engineers. Based on this statement, the identified problems are addressed by the solution of this thesis through following points:

1. The problem of different technologies of software units, environments, and tools used in reuse activities are handled by an abstraction model in relation to a reuse environment. The creation of models based on abstractions is a widely used methodological approach. A good example is the topic of model-driven software development where the core idea is the use of abstraction models for creating software or new models (cf. Selic, 2003). Abstraction is also an relevant method in software reuse (Krueger, 1992).

In this thesis, a common model for software units is created. To do so, this research focuses on classes, components, and services as examples of reusable software units. As a result, the common Software Unit Model includes these three types of units. Additionally, reuse activity models are created and related to the common software unit model. This relation represents the software unit information used by reuse activities. The reuse activity models represent (as example) integration, transformation, and deployment activities (i.e., the focused reuse activities in this research) of software units. To be more precise, the integration of software units into development environments, the console tool based transformation of software units, and the deployment of software units into embedded devices are used as examples of activities

in this thesis. Based on the construction behaviour of these three reuse activities, they are called software construction activities (SCAc) in this thesis. The reuse activity models also describe how a unit should be integrated, transformed, and deployed. This includes the description of necessary environments, tools, and the required configuration. The reuse environment is hosting instances of these models. As a result, all different software units, environments, tools, and SCAc are described by abstract models. This generates a single view on the multitudes of technologies and makes it easier to handle.

2. To avoid a learning process which may end in a misinterpretation or insufficient knowledge of a reuse activity, an automation environment is created. An experienced user fills this system with reuse activity knowledge. This information is stored in the reuse activity models. An inexperienced user can use this system to execute the focused reuse activity based on the stored information. This means an inexperienced user selects an SCAc and executes it in the user's technical environment. As a result, the inexperienced user is not constrained to learn and interpret the activity knowledge. Additionally, the reuse activity results do not vary. The automation system performs the stored SCAc in the same way the experienced user expects and, therefore, produces invariant results. The related problem of time intensive reuse activities is addressed by creating an environment which is able to host different setups and configurations required by different SCAcs. An experienced user creates this SCAc setup and the inexperienced user is then able to perform the SCAc with less preparation and learning time for the SCAc. The expected effect is a reduction of time in performing an SCAc.
3. The problem of the knowledge intensive distribution environment (e.g., the use of different repositories) is also based on abstraction. The focused environment handles repositories and the reuse automation environment for an experienced and inexperienced user. To simplify the handling, a service interface is used. By using this

interface an experienced user can store SCAC related information. An inexperienced user is then able to search and retrieve SCAC information, as well as perform SCACs in their environment. Next to the abstraction this service should address the location problem by a simplification of provided infrastructure.

This also includes the location problems of repositories. Whether a repository is placed next to the user or in a different location should be not relevant. Even though the structure of the focused environment is changing (i.e., repositories will be added, replaced or removed) the user should not be aware of these changes. Using such an SCAC service limits the knowledge to find and access different repository systems.

This thesis focuses on the problem of software engineers if knowledge that is necessary to perform reuse activities (i.e., transformation, integration, and deployment) of reusable software units (i.e., classes, components, and services) is missing. The research question is formulated as follows:

How does one provide successful reuse of different software units considering the possibilities of reusing and performing related software construction activities even if software engineers do not have the required knowledge?

By focusing on the reuse of activities, this research can be classified as a reuse of procedure using the reuse classification of reuse types by Prieto-Diaz (1993).

This research will identify one possible technique to enable inexperienced users to perform the focused SCACs and analyse the effects on software engineers in a case study.

1.3. Aims and objectives

The principal aim of this thesis is defined as follows: To define a concept to enable software engineers to reuse software construction activities of reusable software units even if these engineers do not have enough knowledge to perform these activities on their own.

Basically, the concept is based on the idea to create different models that are able to store SCAC related information. Additionally, an environment will be created that is able to perform the

Introduction

SCAc stored in the SCAc models. This environment is service-oriented which means that the functionality of the environment is provided by one service to users and the environment itself uses internal services.

By achieving this aim, one possible approach is identified to limit the lack of techniques to store and subsequently distribute reuse activity relevant software unit knowledge among software engineers. As a result, this contributes to McCarey, Ó Cinnéide and Kushmerick (2008) who identify this lack but with the focus on three software construction activities (i.e., integration, transformation, and deployment). This aim will be complemented by a series of further objectives. These objectives and their relationships are summarised in Figure 1 and will be defined as follows:

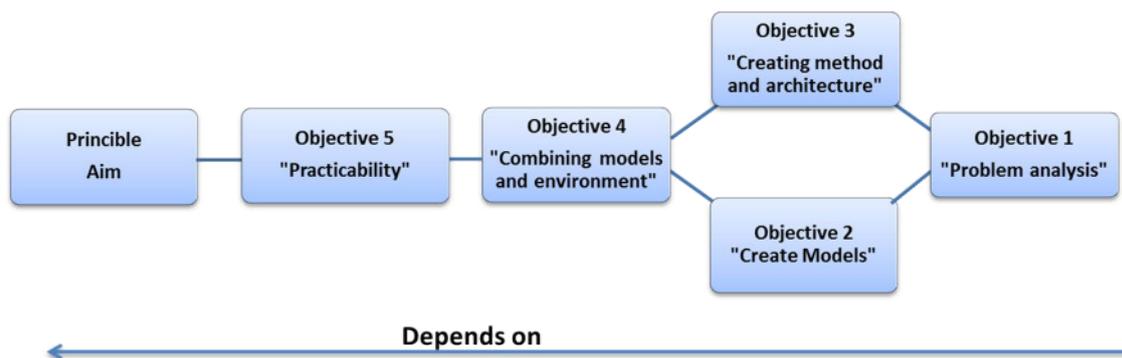


Figure 1 - Relationship between aim and objectives

1. Objective - Problem analysis

This objective includes analysing the problems of missing knowledge in software unit reuse for SCAc, the potential causes generating this problem, and finally the impact created by this type of problem. This approach is necessary to achieve the principal aim of this work. The briefly described problems of missing SCAc knowledge in this chapter will be analysed.

The result of this objective is the analysis of the problems which can be used for the discussion of the success or failure of the approach.

Introduction

To achieve this objective, the software reuse information demand model is used to represent the missing knowledge problems of software engineers and, therefore, the elements of missing knowledge (see ‘Objective 1’ in Figure 1).

2. Objective - Model creation

The second objective of this thesis is the definition and realisation of two model types which allows a unified view on software units and related software construction activities. The first includes a way of describing software units using a unified viewpoint. From the scientific point of view this new model is not an relevant research result. It is used to underline two aspects. The first one is the fact that an abstract model is sufficient to support the creation of a solution to solve the focused problems. The second aspect focuses on the demonstration of the focused solution. The research will show that little information of this model is necessary for the research.

The second model type describes the three focused software construction activities (see ‘Objective 2’ in Figure 1). This provides the basis for solving the problem that arises through missing SCAc knowledge.

3. Objective - Creating a service-oriented environment concept

The principle aim supports software engineers in performing software construction activities. This thesis tries to achieve this by the specification of a technical service-oriented environment. This environment is the result of this objective. It includes the definition of how to use the infrastructure and how the infrastructure itself makes use of the models defined in the second objective (see ‘Objective 3’ in Figure 1).

Note: Even though this objective includes the creation of a service-oriented environment, it is not the aim of the research to create yet another reuse environment or repository.

4. Objective - Combining models and environment

In order to achieve this objective it is relevant to combine the results of Objectives 2 and 3. Based on the problem identified in the first objective, the models for a unified view of software

units, and their related software construction activities (Objective 2) will be combined with the service-oriented environment (Objective 3). This environment has to manage different repository systems, user requests for managing SCAC, and software units, as well as the execution of SCAC in the inexperienced users' environment. The result of this objective is the solution (i.e., SCA model and service-based environment for performing SCAC) which this thesis aims to discover. This result is relevant for the principal aim of the thesis, because from that point onwards, the solution is defined and can be investigated, validated, and discussed (see 'Objective 4' in Figure 1).

5. Objective – Practicability

The fifth objective of the thesis is verifying the principal aim. By using the result of Objective 4 and including the models defined in Objective 2, together with the environment created in Objective 3, the evidence resulting from this work's principal aim is investigated and replicated in a real environment. This realised environment includes existing software units, software construction activities, software engineers, and an existing industrial environment. This environment is used in a case study to measure different values. In the last step of this objective the values are compared and their impact on the focused problems is discussed (see 'Objective 5' in Figure 1).

1.4. Method overview and thesis outline

The following section describes the outline of the thesis, the chapters' content and the research methods briefly.

This introduction has laid the foundations for the thesis by introducing the research problem and research question.

In Chapter 2, relevant literature is analysed and discussed to demonstrate the scientific gap (i.e., missing support of SCAC knowledge execution for software engineers) that this research focuses on as well as the relevance of the research. The discussion in Chapter 2 also includes a

Introduction

overview about the research methods used for the literature (secondary) research and the definition of the focused problem.

Chapter 3 discusses the focused problems of missing SCAC knowledge and shows examples of SCACs. The problem discussion uses the perspective of software engineers and examples of the previously described SCACs and the industry context as the application area. Additionally, the software reuse information demand model is used here to demonstrate the missing SCAC knowledge problem. This model is one result created during the Ph.D. research. At the end of Chapter 3 a short discussion about existing solution types is included.

Chapter 4 presents this thesis' advocated solution concept. This includes the description of the solution's methodology, the concept of a unified view of software units, and the concept for models describing software construction activities. From the scientific research perspective this chapter includes the creation of abstraction models for the focused SCAC types and the abstraction model for the different types of software units (e.g., classes, components, and services). This simplifies the view on knowledge and creates a solution based on this simplification. Additionally, this chapter describes the service-oriented environment used to focus on the problems based on distribution environment knowledge.

Chapter 5 extends the theoretical discussion of Chapter 4 by describing one possible realisation of the solution. Furthermore, the concept of a unified view of software units will be achieved by creating model instances. The same procedure will be performed with the concept for models describing software construction activities. The realisation is a service-oriented environment which is used in the case study in Chapter 6.

From the methodical point of view this chapter fulfils the creation of a software reuse environment which is able to handle user requests and the created models for SCACs. Such method is often used to demonstrate the reliability of an approach. (cf. Garcia et al., 2006; Santana de Almeida et al., 2004).

Introduction

Based on the examples and the realisation demonstrated in Chapter 5, Chapter 6 describes the verification of the proposed solution's concept by means of a case study. This implies the preparation of a case study including a demonstration of the chosen inputs (i.e., SCAC and software units) and participants. It is also relevant to show the different ways of measuring relevant research results. Such a method is often used in approaches for software unit reuse and has the advantage of demonstrating the real world properties of approaches. (cf. Santana de Almeida et al., 2004; Edward, Ali, and Sherif, 1999)

The results of the study are collected and discussed objectively. Chapter 7 summarises the results of the previous chapters, thereby, presenting the research in a compact and well-structured form. After this, the results will be compared with the results of the previous chapters especially those of Chapter 2 and Chapter 3. This results in the conclusion which demonstrates the usability of the proposed solution. The result of this chapter is a discussion and evaluation of the results of the primary research. The possible future work will also be discussed for this purpose.

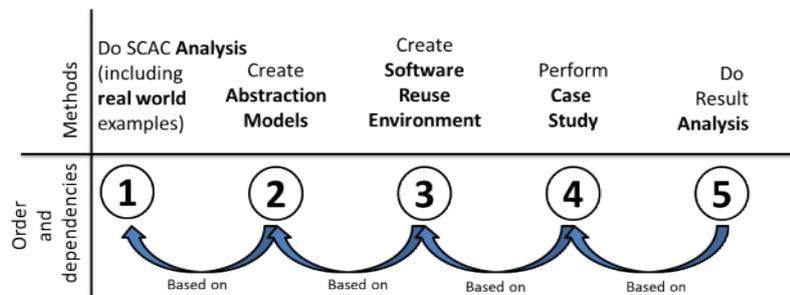


Figure 2 - Used research methods

Figure 2 shows the chosen research methods as a process. The process starts with the analysis of different SCACs (1). The result includes necessary knowledge for SCACs, detailed problem properties, as well as, a state of the art analysis of existing solutions. This knowledge is used to create abstraction models (2). These models are used in a new reuse environment (3). This environment is filled with data (stored in the models) and used with different participants in a

Introduction

proof of concept case study (4), the result of input analysis (1) and the proof of concept (4) is used for the results analysis (5).

The next chapter includes the secondary research and, therefore, discusses the basics, relevant definitions, and problems of software unit and activity reuse from the perspective of the used literature.

2. Problem of missing knowledge in software unit reuse

This chapter discusses the literature focusing on the topic of software unit reuse, the problem of deficient reuse knowledge, as well as the lack of handling software construction activities knowledge in the area of software reuse. Thereby, the chapter starts with a short overview about the secondary research analysis characteristics and methods. It is followed by a general perspective on software unit reuse knowledge. This includes the definition of relevant terms and an overview about software unit reuse, as well as reuse areas. After this overview section, the general problem of missing knowledge in software unit reuse will be discussed. This is followed by a conclusion that the used literature includes a lack of support of software construction activities knowledge and, therefore, a need for an adequate solution.

2.1. Secondary research methods

This section presents an overview of the secondary research methods to describe the procedure model the author used to identify literature and relevant statements for the literature review. For further information, a detailed discussion about these methods and their realisation in this thesis can be found in Appendix Section B.

In general, the research methods follow the discussion of a literature review published by Randolph (2009) in the *Journal of Practical Assessment, Research & Evaluation*. The discussion shows two types of information in literature: characteristics and used analysis methods. The used characteristics of the literature review are focus, goal, perspective, coverage, organisation, and audience. Regarding the focus, the literature review in this thesis is used in order to:

1. Explain reuse in general.
2. Identify and discuss relevant keywords in the field of reuse.
3. Show different research and problem areas of reuse (including SCAC related topics).

Problem of missing knowledge in software unit reuse

4. Underline the problem of missing knowledge in reuse (including SCAC related topics).
5. Show the historical view on reuse and the problem focused upon (including SCAC related topics).
6. Critically discuss problems of existing solution approaches (focusing SCAC related topics).
7. Discuss the contribution of this thesis to the research area of reuse.

Note: The discussion of the existing solution approaches can be found in Section 3.3.

Using these objectives, the overall goal of the literature review is to: (1) generalise the findings and outcomes of ‘missing knowledge in reuse research’, (2) identify central issues, and (3) create a line of argument for an innovative solution of a service-oriented provision of software construction activities.

The literature review is structured by using the aforementioned objectives. Literature used for an objective discussion is first discussed in a neutral position. The different literature will be related also from this neutral position. In some of the cases, the discussions have to be related to the research of this thesis or require a critical analysis.

The coverage characteristic shown by Randolph (2009) is defined as follows: They range from a review of all existing literature to a purposive collection of literature. This literature review focuses on a purposive selection of literature, therefore, only journal papers, conference papers, and specifications of standards (e.g., processes or technologies) were used. These documents were searched using scientific digital libraries (e.g., IEEE, Association for Computing Machinery (ACM), Springer, CiteSeerX, and Thinkmind). Also, documents were selected by analysing references cited previously in studies, journals or conference papers. The literature used is deemed adequate to explain the focused problem and the solution.

Problem of missing knowledge in software unit reuse

For the organisation characteristic, this literature review mainly uses a conceptual format and is structured using the above mentioned objectives. As a result, the review follows the order of these objectives. Inside each objective discussion the conceptual format is also used, but the structure differs. Regarding the audience characteristic, the complete thesis including the literature review is used to demonstrate the capability of the author to perform a Ph.D. study. Therefore, the primary audience is the review committee of the University of Plymouth.

Following the characteristics, the analysis methods are relevant. Using the characteristics as the main information following analysis, methods were used in the literature review, as follows (see Appendix Section A for more detailed description):

Problem formulation: for each focused aim, an objective is formulated. Additionally, the inclusion and exclusion criteria for literature are defined.

Data collection: In the next step, a basic questionnaire is formulated. At the same time, a process to separate irrelevant literature is defined based on the inclusion and exclusion criteria of the problem formulation step.

Data evaluation: For each objective it will be defined which kind of information is seen as interesting for the research.

Data analysis and interpretation: Using the data evaluation criteria for interesting information, each selected literature has to be read and checked to see if specific statements can be used as evidence or a contrary statement. Therefore, this section defines what is evidentiary or contrary for each objective. For each of the literatures, all relevant information (e.g., statement and evidence value) is listed in a personal cookbook. An example of this listing can be found in Appendix Section C.

Data presentation: This last step defines the structure of the literature discussion. The resulting structure is presented in Section 2.2. An exception is the discussion about other solution approaches. This is given in Section 3.3. A detailed description of the used literature review methods can be found in the Appendix Section A.

2.2. Software reuse academic background

In this section the necessary background information for software reuse will be discussed. This includes term definitions, as well as a discussion about the reuse landscape and related research. The section concludes with a discussion regarding relevant statements showing that a lack of techniques exists to support software engineers in exchanging software construction activity knowledge.

2.2.1. Term definitions and software reuse aspects

In the following, the terms used in this thesis to explain the research are discussed and defined.

2.2.1.1. Software reuse term definition

Reuse based software engineering, also called ‘Software Reuse’ (see McClure, 2001), is a development approach that focuses on the reuse of previously developed software parts. Without regard to different characteristics in the realisation of software reuse, this term is more or less defined using the same meaning. Amongst others, the following definitions for software reuse exist:

“Reuse-based software engineering is an approach to development that tries to maximize the reuse of existing software. The software units that are reused may be of radically different sizes.” (Sommerville, 2011, p. 426)

“*Software reuse is the process of building or assembling software applications and systems from previously developed software parts designed for reuse. Software reuse is practiced to save time and money, and to improve quality.*” (McClure, 2001, p. 3)

“*Software reuse is the process of creating software systems from existing software rather than building software systems from scratch.*” (Krueger, 1992, p. 131)

From the perspective of this thesis, all definitions are correct and applicable. Especially, in relation to this work, the definition of McClure (2001) including a process view, is applicable. From the perspective of this definition, the term ‘Software Reuse’ is defined as a process.

Another relevant point is mentioned by Krueger (1992, p. 131): “*The reuse is more the use of existing units rather than newly writing software units*”. Krueger’s statement also implies the possibility that in one project, the reuse of existing units as well as the creation of a new unit may occur.

In the scope of this thesis, the term reuse based software engineering is defined from the software engineer's point of view. To understand the approach discussed in this thesis, it is useful to identify ‘Reuse based software engineering’ as a **software engineering discipline** which is realised using a **development process**, including **different activities**, to **create software systems** by using **software development methods** and **reusing existing software units**.

Note: This definition is based on the aforementioned statements and is applicable for this thesis. For other approaches, this perspective may not be suitable. From this point of view ‘reuse based software engineering’ is called ‘software reuse’ in the thesis.

2.2.1.2. Expected characteristics of software reuse

The following is generally expected from software reuse: higher reliability, lower risk of wrong development, more effective use of specialists, standards compliance, and accelerated development (cf. Sommerville, 2011).

The reduction of costs and the improvement of product quality are seen as the relevant aims of software reuse. This is discussed, for example, by Morisio, Ezran and Tully (2002), Ha, Sun and Xie (2012), and Ajila (2006).

Ha, Sun and Xie (2012), Sandhu et al. (2010), Poulin (1997), and White et al. (2009) demonstrate different ways to calculate costs of software reuse, but mention that different reuse approaches are difficult to compare. The measurement of quality seems to be a major problem in the area of research (see Leite et al., 2005).

The requirements of software reuse are based on the expectation that a previously developed software unit is tested and, therefore, of high quality and simple to reuse. A software engineer

has only to select the correct software unit, integrate it into a development project, and use it because the high quality is known. The surveys of Slyngstad et al. (2006), Isoda (1992), and Morad and Kuflik (2005) in different countries show that this basic idea is correct. Here, companies are good examples demonstrating reuse as an approach which is able reach the aim of higher quality and lower costs. The same studies as well as the studies of Frakes and Kang (2005) and Rothenberger et al. (2003) also demonstrate that the effort for reuse has also to be considered. Effort is, for example, the training of people for reuse or the search and validation of a suitable software unit in an in-house repository. Morisio, Ezran and Tully (2002), Card and Comer (1994), and Frakes and Kang (2005) stated that software reuse includes a preparation from the technical as well as from the organisational point of view. As a result, software reuse is not as simple as expected.

Another expected characteristic on reuse is the handling of reusable software units. The studies of Frakes and Kang (2005), Morisio, Ezran and Tully (2002), Tomer et al. (2004), and Alferez and Pelechano (2011) are based on different reuse research areas (reuse knowledge, costs, and software product lines) and identify together source-code (e.g., object-oriented classes), components, services, and applications (or application parts) as reusable software units.

The last interesting expected characteristic for this research is the expectation of reuse plans and organisational support of reuse Jansen et al. (2008), Jha and O'Brien (2011), Jakobson, Griss, and Jonsson (1997), Frakes and Kang (2005), Morisio, Ezran and Tully (2002), and Ajila (2006), for example, discuss the need for organisational support. Therefore, software reuse has to be supported by the management, meaning the management has to allocate a budget for reuse costs, human resources, processes, and time into their business plans.

2.2.1.3. Units and landscape of software reuse

To create an overview of relevant areas where software reuse is used, it is helpful to identify reusable software unit types first. In software reuse, units of different sizes are reused in a new

context or environment. Three typical types of reusable software units are defined by Sommerville (2011, p. 416):

1. Reuse of software systems: a system as a whole or parts of a software system can be used either by integrating with other systems and customising it for different customers or by developing application, having a shared architecture, but with the added features that a current customer needs.
2. Reuse of components: application components that adjust the size of the subsystems to one simple class or object can be reused. It is, for example, possible for a pattern matching system, which was developed as part of a text processing system, to be reused in a database management system.
3. Reuse of objects and functional software components that implement functions such as a mathematical function or object classes that can be reused. Indeed, reuse of components was common for 40 years (Paulisch, 2008). Today a complete market exists for libraries, different types of function classes, and application development platforms. They can be easily used by another application code that is linked. This approach is effective, especially in the areas of mathematical algorithms and graphics, particularly where domain specific experienced users are needed to develop objects and functions.

Different scientific studies handle the first and second type of Sommerville's classification as typical reusable software unit's categories. In their analysis, Jansen et al. (2008) use services and components as reusable software units and focuses on the topic of the reuse of components. Wang and Fung (2004) also see service and components as relevant units of modelling in software reuse from an architectural perspective. This is different from the viewpoint of Tomer et al. (2004) and Böckle, Pohl and van der Linden (2005). Their research focused on software

Problem of missing knowledge in software unit reuse

parts or whole software products in a software product line (SPL) environment. Bayer et al. (1999) and Dikel et al. (1997) are examples of research that focused on the third type during the 1990s.

From the perspective of this thesis the second type is the focused category. This research focuses on classes, components, and services as reusable software units.

Additionally, to the discussion of the reusable software units it is relevant to see how these units are reused. Next to the typical use of objects and classes Jansen et al. (2008) discussed in detail the two software unit types (components and services) of reuse inside development environments (see Table 1).

Note: A detailed discussion about different base technologies of classes, components and services can be found in the Appendix Section D. This also includes an overview of typical usage behaviours.

Unit of inclusion	Call type	Interaction method
Component	Direct	Pipe and filter
		Component library reuse
	Indirect	Glue code
		Shared data object
		Component bus
Service	Direct	Service framework
	Indirect	Enterprise service bus

Table 1 - Extension mechanism for reusable software units (based on Jansen et al., 2008)

Table 1 shows that service frameworks and enterprise service busses are used to integrate services into a development project or application environment. The use of a service framework is defined as direct method call. This means the software engineer reuses the service by a direct link between the existing code and the service using the service framework. Windows

Problem of missing knowledge in software unit reuse

Communication Foundation (WCF), for example, is the service framework of the .NET architecture. An indirect method is the configuration of an enterprise bus to act as a mediator between a service and the project or application environment. Here, the software engineer does not use the service directly. The task is to configure an enterprise bus for forwarding request or responses of the service.

Table 1 shows direct and indirect methods for integration of components into a development project or application environment. Pipes and filters, as well as method calls of component libraries are typical direct methods of a software engineer to reuse a component. Four indirect methods exist for components.

- Glue Code means to write some extra source-code to integrate the component into the project.
- Shared data objects are instantiated components used like a service and capsule the existing implementation.
- Component bus is equal to the service bus approach.
- Plugin-architecture describes an infrastructure as part of the existing development project. This infrastructure is able to load/integrate an existing component in the development project or application environment. These methods are similar to the component methods of Jansen et al. (2008) and used in the area of SPL.

Also, it is relevant to know if software units are reused for critical or non-critical operations. Jansen et al. (2008) conclude that software units (in this case services and components) are used for non-critical and critical operations. Leite et al. (2005) also support this statement and, therefore, mention that the quality of the software unit is relevant. If a software unit contains errors or has an unexpected behaviour this may lead to errors in the system using this unit.

Next to the discussion of the types of units and their usage in development, is the relevant discussion about the potential content of a reusable unit. The examples used before discuss software units as reusable units. Ajila (2006), for example, uses the term component or reuse

Problem of missing knowledge in software unit reuse

component to summarise all reusable software units. Other studies as Lopez and Niu (2011) and Morisio, Ezran and Tully (2002) use the term artefact or asset and define it as a reusable unit which includes, e.g., the following information: knowledge, documentation, design, code, and cost information for assets. This demonstrates that additional information only uses binaries or source-codes are the focus of software reuse. Rothenberger et al. (2003) discuss these terms as containers of multiples values. Note: For a more detailed discussion about the different terms see Section 2.2.2.

The different software units of the three categories mentioned by Sommerville (2011) can be related to the different approaches in software unit reuse. Sommerville (2011) shows a software reuse landscape including these approaches.

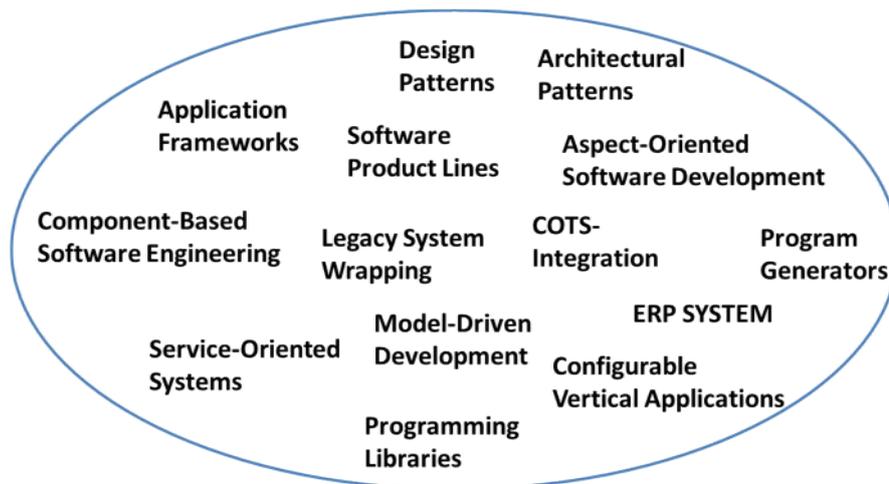


Figure 3 - Landscape of reuse (Sommerville, 2011 p.429)

Figure 3 shows different reuse approaches from the area of software engineering. These are described briefly as follows (based on the listing of Sommerville, 2011):

1. Design Patterns: Across applications, abstractions are commonly presented as design patterns denoting either abstract or concrete objects or interactions. Design patterns are concepts which can be reused as a concept or implemented in adaptable source-codes or components (see Gamma, 1995 for example).

2. **Component-based Software Engineering:** A software systems will be created by a mix of components meeting the standards of the component model. Typically, the software units used here are components (Szyperski, 2002a).
3. **Application Frameworks:** These consist of concrete and abstract class sets which can be expanded or adapted to build application systems. In application frameworks usually application parts can be used. An example for reuse of application parts is the Microsoft Office application. Often, application parts of Microsoft Word or Microsoft PowerPoint are reused between these two applications in the same version of Office as well as between different versions of Office. Therefore, a specific framework is used that is provided to software engineers using the Microsoft Component Object Model (COM) technology (Microsoft, 2012a).
4. **Legacy System Wrapping:** A system that can be included in a wrapper and is defined by the associated use of an interface over which access takes place. This type of reuse often uses components, application parts or services. Especially if a wrapper or legacy system creates a connection to another technology and is used in the system providing the wrapper. An example is the use of Java byte code in a .NET application using the IKVM wrapping system (Frijters, 2011).
5. **Service-Oriented Systems:** These comprise systems developed by linking shared services that can be supplied by external sources. Typically, services are the used software units for reuse in this area (Singh and Huhns, 2005; Wang and Fung, 2004).
6. **Software Product Lines:** A type of application which is generalised by using a common architecture so that it can be adapted to different customers. Usually, whole applications are used in such approaches (see Fayad and Johnson, 2000).

7. Commercial of the shelf (COTS) integration: These systems are developed through the integration of existing application systems. Typically, the software units used here are components (Sommerville, 2011) that are provided by COTS selling companies (e.g., Componentsource, 2012). An example for COTS is the WebCAD web service provided by Componentsource (2012) which includes numerical procedures to either construct a function of one or two variables from a set of points (i.e., interpolate), or solve an equation of one variable.
8. Configurable Vertical Applications: A generic system designed to make it adaptable to the specific needs of the customer. Compared to the application product lines, the software units here are reused by configuration and not by adaptation of the software units. The typical software units are also whole application systems.
9. Programming Libraries: Consists of class and function libraries that implement functionality for reuse. Typically, the software units used here as components are used as libraries today. An example is the topic of Dynamic Linked Libraries (DLL). Often, these libraries include a set of classes that may be not related to each other or to the same business domain. This is the difference to components used in component-based development (see Szyperski, 2002a)
10. Program Generators: A generator system storing knowledge about a particular type of application purchase system or system fragment created for this application. Czarnecki and Eisenecker (2000) discuss such programme generator technology called Generative Programming.
11. Aspect-Oriented Software Development: In this process, shared components are ‘woven’ into the composing program in different places. In this type of reuse

Problem of missing knowledge in software unit reuse

development, smaller software units (e.g., source-code, classes, components or services) are usually used (see Rashid and Akşit, 2006).

12. Model-Driven Development (or Model-Driven Software Development; MDSD):

Development discipline using domain models and implementation independent models to represent software or software units. By transforming models, software units or systems can be created (see MDSD; Petrasch and Meimberg, 2006).

13. ERP System: System for organisational use. These include business functionality and rules.

14. Architectural Patterns: Software architecture used to create software. An architectural example is the use of a plugin infrastructure (e.g., Microsoft Extensible Framework; cf. Microsoft, 2012b)

Table 2 summarises the relation between reuse approaches shown in the landscape of Sommerville (2011, p. 430) and the typical software unit types used in the approaches.

<i>Landscape approach</i>	<i>Typical used software units</i>
Design Patterns	<i>Components, Source-Code</i>
Component-Based Development Systems	<i>Components</i>
Application Frameworks	<i>Application parts</i>
Legacy System Wrapping	<i>Components, Services, Application parts</i>
Service-Oriented Systems	<i>Services, Interface descriptions</i>
Software Product Lines	<i>Applications, Application parts</i>
COTS integration	<i>Components</i>
Configurable Vertical Applications	<i>Applications, Application parts</i>
Programming Libraries	<i>Components</i>
Program Generators	<i>Models (Source-Code)</i>
Aspect-Oriented Software Development	<i>Components, Source-Code, Classes, Services</i>
Model-Driven Development	<i>(typed) Models</i>
Architectural Patterns	<i>Components, Source-Code, diff. Models</i>
ERP System	<i>Components, Source-Code, Configuration files</i>

Table 2 - Typical software units in different software reuse landscape approaches

2.2.1.4. Perspectives on software reuse

In used literature, different perspectives on software reuse can be identified. For example, Prieto-Diaz (1993) identified six facets of reuse: by substance, by scope, by mode, by technique, by intention and by product. Based on these facets the author describes different types of reuse.

Facets		By substance	By scope	By mode	By technique	By intention	By product
Reuse types	Idea reuse		Vertical reuse	Planned reuse	Compositional reuse	Black-box reuse	Reuse products
	Artefact reuse		Horizontal reuse	Ad-hoc reuse	Generative reuse	White-box reuse	
	Procedure reuse						

Table 3 -Types of reuse (based on Prieto-Diaz, 1993)

The facet ‘*by substance*’ describes reuse from the perspective of the reusable content. Ideas, artefacts, and procedures can be reused. The facet ‘*by scope*’ focuses on vertical and horizontal reuse. Therefore, a reusable unit can be used in similar (e.g., vertical areas in industrial automation) or in different business contexts (e.g., horizontal areas of automation, power, and building). In the facet ‘*by mode*’ the focus of reuse is set to an organisational planned (so called planned or systematic) reuse or not planned (so called ad-hoc) reuse. The facet ‘*by technique*’ is not relevant from the literature review point of view, and is similar to the COTS integration and program generators described by Sommerville (2011). While the facet ‘*by substance*’ describes the content to reuse the facet, ‘*by intention*’ describes how content is used. Two aspects are given: in the first, (black-box) content is reused as it is which means without changes. In the second, (white-box) the reuse content will be adapted to fit the requirements for reuse. The last facet ‘*by product*’ focuses the use of the whole application or application parts. Also other classifications exist. Rada (1995), for example, identifies the type of focus (methodology centric or user centric) as an relevant facet in software reuse. Both types are named; development-with-reuse and reuse-within-development.

The research described in this thesis can be classified as reuse '*by substance*'. This is explained as follows: The facet '*by substance*' describes reuse from the perspective of the general content. As described in the previous sections, this research focuses on classes, components, and services as software units and their SCACs. The research shown by this thesis focuses on handling missing SCAC knowledge. Therefore, the primary research of this thesis is a procedure reuse. As a result, the research can be classified in the facet '*by substance*'. But the other aspects are also relevant for the research.

In the following, the relevant terms will be explained in more detail, related to each other, and related to the context of the focused research. The used literature distinguishes planned (systematic), ad-hoc (opportunistic), white-box, black-box, development with reuse, reuse-within-development, vertical reuse, and horizontal reuse.

Systematic (Planned) and ad-hoc (opportunistic) reuse: The literature shows two typical classifications for reuse: systematic and ad-hoc reuse. Morisio, Ezran and Tully (2002), Rada (1995), Ye and Fischer (2005), and Ha, Sun and Xie (2012) are examples which defines systematic reuse as a process of previously (long term) planned reuse. This includes organisational and technical project planning. On the other side they show that the ad-hoc reuse is also planned (short term) reuse. Usually, ad-hoc reuse is the use of an existing software unit in different application/development projects sporadically. Therefore, ad-hoc reuse is also called '*opportunistic reuse*'.

For Morisio, Ezran and Tully (2002) and Ye and Fischer (2005) systematic reuse has to be established inside an organisation. As a result, management decides and plans how to reuse. To do systematic as well as ad-hoc reuse different infrastructure is necessary (see Tomer et al., 2004). Tomer et al. (2004) describes systematic and controlled reuse scenarios. In a controlled reuse, a repository is prepared with reusable software units ready for reuse in one product line. In a systematic reuse, all reusable software units are prepared to be reused in multiple product lines. This view differs from the other views of systematic and ad-hoc reuse. Tomer et al.

(2004) showed that not only the level of planning is relevant, but also the level of used infrastructure.

Morisio, Ezran and Tully (2002) use the term systematic practise to describe the term of systematic reuse but with a focus on organisation-wide reuse behaviour.

White-box and black-box reuse: Ha, Sun and Xie (2012) shows by using the example of software development for embedded devices the white-box reuse or black-box reuse is relevant. Szyperski (2002a, pp. 40-42) describes white-box and black-box abstraction related to the reuse of components which is summarized as follows:

- Black-box: A component is included into the system to be developed, as a complete unit. This component cannot be altered. Furthermore, no statement can be made about its internal construction and functionality. The use of the component happens exclusively on the basis of defined interfaces and specifications of the component.
- White-box: The component is reused as an open or editable unit that can be adapted to the new requirements. For that purpose, its internal construction is visible and thus analysable. Hence, the component is considered as a software fragment. The use of the component does not happen exclusively on the basis of defined interfaces, but also by analysing the actual implementation of this component.

Additionally, glass-box and grey-box reuse is discussed by Szyperski (2002a, pp. 40-42). These are not relevant for this research.

Tomer et al. (2004) describe a white-box reuse approach as an adaptable reuse approach for software development. Software units as well as processes are adaptable. On the other side Ampatzoglou et al. (2011) describes black-box reuse as a process of reusing software unit or design patterns and its processes as '*it is*'. This explicitly excludes the adaptability of a software unit which is equal to the black-box view on components.

Development-with-reuse and reuse-within-development: McCarey, Ó Cinnéide and Kushmerick (2008) use two different scenarios for reuse: Development-with-reuse and reuse-

Problem of missing knowledge in software unit reuse

within-development. Development-with-reuse is described by Rada (1995). The main properties of this methodology centred approach are:

- Reuse of software units is an optional activity in the development process
- Reuse of each software unit has to be planned
- The user has to handle given reuse models

The reuse-within-development scenario is described by McCarey, Ó Cinnéide and Kushmerick (2008) as a user centred approach. Mili, Mili and Mili (1995) and Ye and Fischer (2005) describe case tool supported scenarios which fit the reuse-within-development scenario. This ‘user centred’ approach has the following properties:

- Focus on behaviours and actions of software engineers
- Combine different reuse based development activities (McCarey, Ó Cinnéide and Kushmerick (2008)) focuses on Component-Based Development activities)
- The used tools (e.g., development environments) have to be adapted for supporting the users and fulfil the user’s requirements.

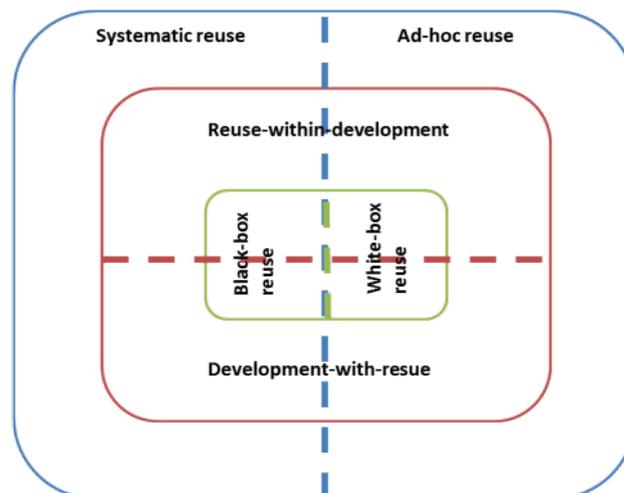


Figure 4 - Relation between relevant reuse types

While systematic and ad-hoc reuse describes the different levels of long and short term planning, white-box and black-box defines the level of adaptability of a software unit. Development-with-reuse and reuse-within-development describes two type of reuse for

software engineers. In this thesis the relation of the three different perspectives on reuse are related as shown in Figure 4.

The relation between the different perspectives is defined for this thesis as follows: Black-box or white-box is in the scope of development-with-reuse and reuse-within-development. These two perspectives can be used in systematic and ad-hoc reuse. This thesis targets knowledge based reuse problems generated in both scenarios: white-box and black-box reuse. The solution approach created in Chapter 5 focuses on reuse of SCAs of software units without any manual adaptations by a software engineer. As a result, of hiding the structure and the execution of reuse activities, this approach is related to the perspective of black-box reuse. Therefore, it can be used on development-with-reuse and reuse-within-development, and the related systematic and ad-hoc (or opportunistic) reuse.

2.2.1.5. Impact of reuse

The literature shows positive impacts of software reuse on software development projects and software products. Jansen et al. (2008) mentioned based on McConnell (1996a) that the reuse effort is between 1% and 19% of effort of reinvention in start-up companies. White et al. (2009) create a formula based on existing cost metrics to calculate the costs for the development and the reuse of a domain specific language (DSL). They conclude that reuse does reduce costs. Ajila (2006) discusses several studies for the topic of costs and productivity that shows the same conclusion.

Some examples in literature show a reduction of cost (e.g., 43% and 15%) and the increase (e.g., 10% and 57%) of quality (see discussion of Tomer et al., 2004) by focusing on software unit reuse.

The different studies focus on different types of companies and different countries (and different cultures). Therefore, it is difficult to compare these statements. But for this thesis the fact is relevant that reuse supports software development. The research of this thesis tries to simplify software reuse in a special topic to generate similar positive impacts.

As mentioned, the cost of reuse is relevant information for organisations. Usually, it is a problem to compare different reuse projects because they're not comparable one by one. This is particularly a problem between white-box and black-box projects. In the SPL environment, Tomer et al. (2004) create a cost function to compare these two different reuse approaches and mention reuse development costs, product construction costs, core asset costs, and infrastructure costs. Thereby, they mentioned an relevant perspective for this thesis - costs can be related to activities necessary in a reuse process. As a result, a specific activity can have a cash value. In a SPL environment these are, for example, transformation costs (e.g., adaptation, creation, white-box reuse, and new development) and transition costs (e.g., integration of reuse results into the SPL environment).

An relevant statement here is the use of activities in cost calculations for software reuse (see Tomer et al., 2004). This indicates that reuse activities are a relevant (cost) factor in software reuse. Therefore, an approach for optimisation of the reuse activities has impacts on the costs of software reuse. The research in this thesis focuses the simplification of the execution of reuse activities. The result of this research may show that this has positive impacts on costs in an indirect way. If the focused approach is able to show that time for reuse can be reduced, it can be stated that the labour cost for this reuse is also reduced. This presumption is based on the relation between time and costs (cf. Ajila, 2006).

2.2.1.6. Reuse technologies and environments

This section describes reuse technologies and a special environment type identified in the literature. In general, for each element of the software reuse landscape, supporting technology approaches can be identified. Table 4 shows example technologies and concepts that are related to the typical software unit of the different software reuse landscape approaches. As a result, Table 4 links landscape approaches and base technologies and concepts based on the used software unit type.

<i>Landscape approach</i>	<i>Typical used software units</i>	<i>Base technology/concept</i>
----------------------------------	---	---------------------------------------

Problem of missing knowledge in software unit reuse

	<i>examples</i>	
Design Patterns	<i>Components, Source-Code</i>	<i>CB, OO</i>
Component-based development systems	<i>Components</i>	<i>CB</i>
Application frameworks	<i>Application parts</i>	<i>SPL, CB</i>
Wrapper and legacy systems	<i>Components, Services, Application parts</i>	<i>SPL, CB, SB</i>
Service-Oriented Systems	<i>Services, Interface descriptions</i>	<i>SO</i>
Application (software) product lines	<i>Applications, Application parts</i>	<i>SPL</i>
COTS integration	<i>Components</i>	<i>CB</i>
Configurable vertical applications	<i>Applications, Application parts</i>	<i>SPL</i>
Libraries	<i>Components</i>	<i>CB</i>
Program generators	<i>Models (Source-Code)</i>	<i>UML, MDSD, OO</i>
Aspect-oriented software development	<i>Components, Source-Code, Classes, Services</i>	<i>SO, CB, OO</i>

Table 4 - Example of base technology/concept in the software reuse landscape approaches (CB = component based technology ; OO = object oriented technology; SPL = Software Product Line; SO = service oriented technology; UML = Unified Modelling Language; MDSD = Model-driven software development)

A base technology concept for components is the component-based technology used in component-based development. For example, Morisio, Ezran and Tully (2002), Szyperski (2002a), and Naur and Randell (1968) identify reuse as the composition or generation of software units. The reuse landscape component-based technology can be used in the area of design patterns, application frameworks, wrappers, COTS integration, libraries, and aspect-oriented software development.

Service-based technologies use services and interfaces as typical software units. This is stated for example by Papazoglou et al. (2007), Wang and Fung (2004), and Singh and Huhns (2005). Regarding the reuse landscape, service-based technologies can be used in the area of wrapper and aspect-oriented software development.

In the area of advanced software units, software product lines (SPL) are used as the technology concept. In this area, core asset development (see Tomer et al., 2004) and product development are seen as different types of software unit creation. The reuse landscape SPL can be used in the

area of applications frameworks, wrappers, application product lines, and configurable vertical application.

A source-code is used in object-oriented form, for example as class structure. Morisio, Ezran and Tully (2002) for example, found examples of object-orientated technologies as a typical technology that supports reuse. Regarding the reuse landscape, object-oriented technologies can be used in the area of design patterns, programme generators, and aspect-oriented software development.

Another landscape approach is the area of programme generators. Here, source-codes, as well as modelling technologies, for example Unified Modelling Language (UML), may be used to generate reusable software units. Generating models as reusable software units, for example; model-driven software development can be used.

Next to the technologies in Table 4, another approach in the reuse of software units is the Software Process Improvement (SPI). Pino, García and Piattini (2007) review different case studies for software development improvements and argue that case tools that improve reuse processes are very relevant.

Garcia et al. (2006) describes an abstract case tool necessary for software reuse called software reuse environments. Software reuse environments (SRE) support software engineers by addressing, for example, reuse activities. Garcia et al. (2006) concludes that current integrated development environments (IDEs) are SRE systems and do not completely fulfil the requirement for SREs to include all possible reuse functions as, for example, for integration purposes.

2.2.2. Discussion of key definitions

Next to the term software reuse, the area of software reuse research includes other typical terms. Often, these terms can be interpreted differently. Following, a short overview is given discussing and concluding relevant term definitions for this thesis.

2.2.2.1. Software construction

The research focuses on three special reuse activities (i.e., integration, transformation, and deployment) which are not related to the domain specific knowledge of the software unit. To separate these activities from other activities which are related to software unit domains, the term software construction is used. In contrast to software engineering, the definition of software construction differs. Basically the construction of software relates to the development and integration of bigger components (McConnell, 1996b). These components are linked with each other in a previously specified manner and result in a finished product or another component. Therefore, software construction is a part of software engineering. This form of construction is the result of a long chain of developments and has a large number of aspects. The term software construction is often applied to the meaning of component-based software development (Szyperski, 2002b) or component-based software engineering (Sommerville, 2011). On account of the ambiguity arising from this, using the term ‘software construction’ is not unproblematic.

According to the software engineering glossary (McConnell, 2006) construction in the area of software engineering is defined as follows:

“...The activity in software development consisting of detailed design, coding, unit testing, and debugging. Also called programming or development.” (McConnell, 2006, p. 128)

Taking this definition, software construction includes software unit domain related reuse activities. Accordingly, software construction is only one synonym for software programming or development. This is similar to the definition of software construction two years before by SWEBOK (2004). In the procedure model SWEBOK (2004), software construction is one of ten knowledge areas which are necessary in a software development procedure model. Software construction is described as follows: *“Software construction refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging.”* (SWEBOK, 2004, Chapter 4 HTML Version) Here,

software construction is a description for typical object-oriented software development and contrary to the other mentioned definitions.

Hunt and Thomas (2004) adopt a pragmatic view. Here, software construction relates, in this case, to the joining of bigger units. Furthermore, the authors show that construction is also about architecture and design. Also, the software construction becomes more separated from the software unit domain.

Another view focuses on processes. Software construction describes a process in which clusters within a software development process are built and joined (Baudoin and Hollowell, 1996; Meyer, 1997). Thereby, a cluster is defined as follows:

“The module structure of the object-oriented method is the class. For organisational purposes, you will usually need to group classes into collections, called clusters [...] A cluster is a group of related classes or, recursively, of related clusters.” (Meyer, 1997, p. 923)

This description corresponds to component-based software development. Due to the many-sided definitions, the base definition for software construction used within this thesis is the general definition of component-based software engineering (cf. Sommerville, 2011):

“Component-based software engineering is an approach based on reuse for the definition, implementation and composition of loosely coupled, independent components to systems.” (Sommerville, 2011, 775).

Combining this statement with the view of Hunt and Thomas (2004) this view on software construction can be defined as follows: Instead of focusing on components, as in component-based software engineering, software construction focusses on different independent software units. Because software construction is mostly seen as software development in the use of bigger previously developed software units (e.g., objects, components, and services) which serve as a base for the creation of software, it is similar to component-based software development. Additionally, software construction is the reuse of a software unit without, or with less, relation to its domain.

Note: This definition is applicable for the research of this thesis. As shown above, other researchers have other definitions for software construction. For other investigations this might not be a useful viewpoint.

2.2.2.2. Software artefact

Morisio, Ezran and Tully (2002) and Lopez and Niu (2011) describe software artefacts as part of a software unit. Typical content might be: code, design, requirements, test cases, and so on. For Tomer et al. (2004) artefacts are components, test cases, or documentation.

But also other interpretations exist. Petrasch and Meimberg (2006, p. 12, translated.) define artefacts as follows: “*Artefacts are work results (final results or intermediate data) which are produced during a project. Artefacts are used to hold or to transmit project specific information*”.

The view of the different authors is based on granularity of reusable software units. In that they can be classified from single source-code, to libraries, to complete software products (see Tomer et al., 2004; Morisio, Ezran and Tully, 2002).

In the scope of this thesis a software artefact is defined as a container including different information. This view is close to the perspective of Rothenberger et al. (2003) who discusses this term as a container of multiples values. The granularity of a software artefact is not relevant for this thesis. An interesting point is given by the definition of Petrasch and Meimberg (2006). In primary research of this thesis knowledge (and all reusable parts of a software unit) about software construction activities will be stored in software construction artefacts. This knowledge is the result of the previous work of a software engineer and it can be identified as a ‘work result’. This is similar to the discussion of Petrasch and Meimberg (2006) in which artefacts store models which are intermediate results in an MDS based process. As a result, the research shown in this thesis defines knowledge description as an optional part of an artefact. This is contrary to the opinion of Morisio, Ezran and Tully (2002) who’s defines knowledge (i.e., experience) as explicitly excluded in the definition of software artefacts.

2.2.2.3. Reuse activity and software construction activity

Isoda (2001) uses the term software reuse activity to describe all activities related to software reuse. Basically in this research all activities (e.g., identification, search, test, validation, integration, etc.) that are used in a reuse process are named reuse activities.

Software construction activities are seen as specialised variants of reuse activities. Based on the differences between software construction and software development (cf. Section 2.2.2.1) or engineering, SCACs include activities focusing on the reuse of bigger software units without relating to software unit domain reuse. In the scope of this research, only reuse activities are named as SCAC that focus on activities, as for example, integration, transformation or deployment of a software unit.

Note: Examples of activities of software construction (i.e., integration, transformation, and deployment) can be found in Section 3.1.

2.2.2.4. Software asset

Basically assets describe software artefacts from the commercial view of business management (Tomer et al., 2004). Therefore, assets are seen as reusable software units including a cash value for business calculations. Seedorf (2010) calls this a business object.

Additionally, software assets are described differently in literature based on their usage. Tomer et al. (2004) for example defines an asset as a software unit which can be used in cost calculation. Due to the SPL perspective of their research, they separate two different types of assets: Private Asset, which indicates an asset not available for reuse because it exists only in a software engineer's private environment; and Repository Asset, which indicates a software unit available for reuse in a company repository. In the research the term asset is used to identify a software artefact which includes cost values.

2.2.2.5. Knowledge and knowledge perspectives

In this research, knowledge of software construction activities is focused upon. Different existing definitions are discussed this section. The term knowledge is widely used but with many different definitions, as for example:

- *“Knowledge is the combination of data and information, to which is added experienced user opinion, skills, and experience, to result in a valuable asset which can be used to aid decision making.”* (Chaffey and Wood, 2005, p. 223, quoting the European Framework for Knowledge Management)
- *“Knowledge is data and/or information that has been organized and processed to convey understanding, experience, accumulated learning, and experienced userise as they apply to a current problem or activity.”* (Turban, Rainer and Potter, 2001, p. 38)
- *“Knowledge builds on information that is extracted from data [...] While data is a property of things, knowledge is a property of people that predisposes them to act in a particular way.”* (Boddy, Boonstra and Kennedy, 2004, p. 9)
- *“Knowledge is the capability of a man (or an intelligent machine) to use information for problem-solving.”* (Bobillo, Delgado and Gómez-Romero, 2008, p. 1903)
- *“Knowledge consists of that mix of contextual information, values, experience, and rules [...] Knowledge involves the synthesis of multiple sources of information over time. The amount of human contribution increases along the continuum from data to information to knowledge.”* (Bellinger, Castro and Mills, 2004, pp. 13-14)

The different definitions of knowledge highlight the relationship with the terms ‘data’ and ‘information’. The data–information–knowledge–wisdom hierarchy (DIKW hierarchy) defines the terms data, information, knowledge, and wisdom and is defined as follows (Rowley, 2007):

- *“Data are defined as symbols that represent properties of objects, events and their environment. They are the products of observation. But are of no use until they are in a*

useable (i.e., relevant) form. The difference between data and information is functional, not structural.

- *Information is contained in descriptions, answers to questions that begin with such words as who, what, when and how many. Information systems generate, store, retrieve and process data. Information is inferred from data.*
- *Knowledge is know-how, and is what makes possible the transformation of information into instructions. Knowledge can be obtained either by transmission from another who has it, by instruction, or by extracting it from experience.*
- *[...]Wisdom is the ability to increase effectiveness. Wisdom adds value, which requires the mental function that we call judgement. The ethical and aesthetic values that this implies are inherent to the actor and are unique and personal.” (Rowley, 2007, p. 166)*

Regarding the definition of Rowley (2007), knowledge is the combination of information in order to fulfil a specific purpose. In the focused research of this thesis, this knowledge can both be part of a person or part of a system (Bobillo, Delgado, and Gómez-Romero, 2008). To be more specific, this research distinguishes between two types of knowledge that is possessed by humans or stored in a system (based on Horeis and Sick, 2007): data-driven knowledge and human-driven knowledge.

“Data-driven knowledge is application-specific knowledge which is extracted from data by conventional Knowledge Discovery (KD) systems. ...” (Horeis and Sick, 2007)

“Human-driven knowledge is application-specific knowledge, too, but this kind of knowledge originates from human experienced users. They have a certain experienced userise concerning an application area.“ (Horeis and Sick, 2007, p. 422)

Therefore these two terms can be described by assigning them to existing definitions:

“If we want to describe data-driven and human-driven knowledge by means of some existing terms, [...] we can state that data-driven knowledge is often provided in an implicit way (it must be extracted from data). It typically has a quantitative nature and it is less abstract (with

Problem of missing knowledge in software unit reuse

respect to the application) than human-driven knowledge. Human-driven knowledge is (at least in an initial phase of knowledge acquisition) explicitly provided by human experienced users.”

(Horeis and Sick, 2007, p. 422)

In this thesis, the term ‘knowledge’ represents both data-driven knowledge and human-driven knowledge. Both terms are used. In this thesis, this is the knowledge to store, search, if it necessary to differ the meaning.

After discussing the definition of the term knowledge it is necessary to explain the context of knowledge, information, and data.



Figure 5 - The knowledge pyramid (Ackoff, 1989, p. 5)

Based on the initial definition of the knowledge pyramid (Ackoff, 1989; see Figure 5), the data–information–knowledge–wisdom hierarchy (DIKW Hierarchy; see Figure 6) was developed. Basically, both models explain that wisdom is based on knowledge which is based on information. Data is the basis for information. Both models are generally accepted for the definition of knowledge (Jennex, 2009), but treat it differently.

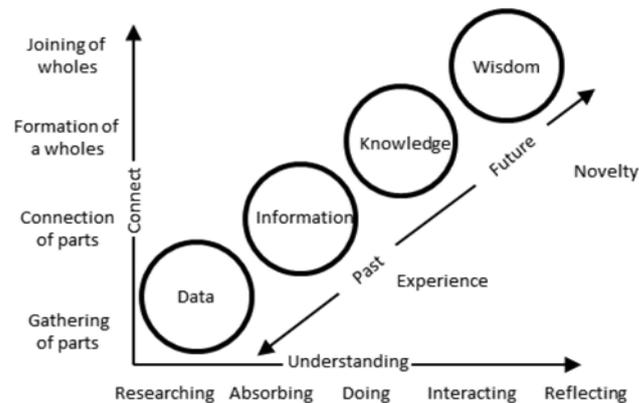


Figure 6 - Example of the DIKW hierarchy (Rowley, 2007, p. 186)

However, both models see knowledge in a different manner. While the basic knowledge pyramid indicates a classification of different information types, newer approaches focus on the transition between the different classification types. Figure 6 shows five different transitions between the mentioned terms:

1. Researching: Research is necessary to collect basis data, for example, values from different sensors (Rowley, 2007).
2. Absorbing: When different data (values) are linked in systematic relationships, information is created. This can be seen as understanding data, (Rowley, 2007; Bellinger, Castro and Mills, 2004), for example, the relationship between rain and an increasing water level.
3. Doing: By creating a pattern which indicates what will happen when specific information is available, knowledge is created. Therefore, knowledge can be seen as a rule to handle information, (Rowley, 2007; Bellinger, Castro and Mills, 2004), for example, the water level increases if it rains more than 1 litre per square metre.
4. Interacting: Using knowledge is called interacting. The result of a knowledge process is relevant for the next transition (Reflecting; Rowley, 2007). Based on the used example, a reaction to an increasing water level is to install protectors on the river banks.

5. Reflecting: This interaction results in wisdom whether the applied knowledge is correct or not. Within this thesis, the base definition of the knowledge pyramid (Figure 5) is used. In relation to the DIKW hierarchy, the interactions ‘absorbing’, ‘doing’, and ‘interacting’ are relevant for this work.

Regarding the research of this thesis, knowledge is the use (so called ‘Interacting’, Rowley, 2007) of information. Information is the relation of existing data (so called ‘Doing’, Rowley, 2007).

In this thesis, the term knowledge defines data-driven and human-driven knowledge related to the reuse of SCACs. Regarding the DIKW hierarchy, this is the interaction with SCAC related information. The exchange of knowledge is a relevant research area. Theories such as, for example, the Transactive Memory System focusses on the exchange of knowledge inside and outside of a team. Choi, Lie and Yoo (2010) show in a field study that the search, exchange and use of knowledge is relevant for teams to perform projects. The authors conclude the use of IT as infrastructure is relevant for the exchange of knowledge for software development teams. Qu, Ji and Nsakanda (2012) relate the topic of knowledge exchange to software development teams and agree to the importance of an IT infrastructure. An relevant concept which can be seen as an IT based infrastructure is a knowledge management system (KMS). Such systems provide functionality to manage different knowledge. In this thesis, knowledge management (KM) is understood to mean “*a method that simplifies the process of sharing, distributing, creating, capturing and understanding of company’s knowledge.*” (Davenport, 2000, p. 4)

In this research the discussion of different viewpoints on knowledge management is not necessary. What is relevant is the analysis of existing solutions or approaches to handle knowledge of software construction activities.

2.2.3. Missing knowledge in software unit reuse

This section discusses the problem area of missing knowledge for software unit reuse and describes the focused problem of the research regarding missing knowledge for software construction activities. As shown in Chapter 1, typical scenarios including the fact of missing knowledge are:

- young professionals (e.g., students) without any experience (Shiva and Shala, 2007)
- senior software engineers in new projects (Boh, 2008)
- team members who leave the team ('knowledge vaporisation', Ven et al., 2006)

In the following, relevant research statements about missing knowledge for software units will be discussed.

2.2.3.1. Historical perspective on the problem of missing knowledge for software unit reuse

The problem of missing knowledge in software unit reuse is not a problem created today or in the last three years. Reuse itself can be seen as a solution for missing knowledge. Naur and Randell (1968) suggest creating reusable components based on the examples in the area of hardware to make it easier to develop software. The idea was not to build software functionality every time it is needed from scratch. The result from knowledge perspective is that it is not necessary for a software engineer to know how to build the functionality.

In the 1980's this issue was identified as a problem of 'experience' of a software engineer and management (Deming, 2000). Such examples can be found in the 90's also (e.g., Johansson, Hall and Coquard 1999; Isoda 1992).

The discussion about software units and their content differs in the past. Originally a reusable component consisted only of source-code or binary code (see Naur and Randell, 1968). Later on with the use of video and audio systems it becomes clear that also other values e.g., audio, video, or text can be reused. But the focus was still on elements directly used by an application. Childs and Sametinger (2012) and Blok and Cybulski (1998) are examples who discuss the

reuse of documentation, specification and models used in the 1990s. The same studies connect the reusable content with information (may be seen as knowledge from the perspective of the DIKW hierarchy) needed by a software engineer. Another example is the area of reusing tests. Bagnasco et al. (2001) discusses the reusing of unit test. In the 1990s reuse knowledge was combined with existing reuse technologies as object and component orientation (see Szyperski, 2002b). As a result, software engineers have to know such technologies to handle a software unit or their content. The same requirement was created by the use of SPL. SPL also changes the view on the content of a reusable software unit. From now on a software unit does not include only a few functionalities and has only a small size. In the beginning of the 21st century a change in thought occurs. Ye (2001) and Morisio, Ezran and Tully (2002) for examples uses the term software artefact that is able to contain different content. This means different software units (i.e., components, classes, service, etc.) as well as different additional values (e.g., binary data, documentation, models, tests, etc.). This perspective does not change till today. With the increase of content types and variations the amount of necessary knowledge grows for software engineers (Ajila, 2006).

The changes on the view on software units and related knowledge can be identified in the past. But the view on reuse activities is not so easy to identify. In the used literature, two views on reuse activities can be identified. Prieto-Diaz (1993) includes in their classification of reuse types the reuse of procedures. Here, an activity will be reused. On the other side, Isoda (1991) concluded after a 4 year reuse project, that the main activities are: registration and the reuse of reusable modules, construction of a reusable module library, compilation of reusability guidelines and the development of software reuse support tools. In the 1990s different reuse processes such as, for example, Software Technology for Adaptable, Reliable Systems (STARS) and ROSE PM were developed (cf. De Almeida, 2005). A clear relation between the reuse activities and missing knowledge was not identified in the past.

From the view of this thesis the historical consideration of the different missing knowledge problem classes concludes in the following statements:

- The problems of missing knowledge is still known, but related to software units and its technical structure
- Based on technology changes / evolution the complexity of the knowledge increases
- The idea of reuse activity is known
- A clear definition of knowledge for software reuse activities was not identified

2.2.3.2. Relevant research statements of literature

Literature shows different problems in the area of software reuse regarding missing knowledge.

Tracz (1994) highlights different software reuse myths (i.e., software reuse is a technical problem, special tools are needed for software reuse, reuse results in huge increases of productivity, software reuse is equal to hardware reuse, reused software is equal to reusable software, and software reuse) will just happen. The conclusion of Tracz (1994) is that reuse is not a problem anymore. It is only a problem to the organisation level.

Analysing other studies shows a different picture: Frakes and Isoda (1994) argues that support for reuse is difficult to create because of the variants of different technologies and user domains.

Morisio, Ezran and Tully (2002) conclude in their survey that often the use of a repository and reuse supporting technologies (e.g., object-oriented technologies) are adequate for performing reuse. But they also mention the need for reuse processes and the human factor. Repositories are relevant in the area of software reuse. Ajila (2005) and Cummings and Teng (2003) use the term 'intellectual capital' for information and knowledge stored in repositories. Ajila (2006) mentions this capital as an relevant success factor in software reuse.

Frakes and Fox (1996) indicate that 24% of software development projects using reuse fail because not all software engineers try to reuse. The study analysed 29 American organisations in 1991 and 1992. Based on this statement McCarey, Ó Cinnéide and Kushmerick (2008)

Problem of missing knowledge in software unit reuse

concludes in another study that the human factor is the relevant factor in reuse of software units. Ye and Fischer (2005) describe a negative scenario which is based on this human factor and related to knowledge and concludes:

- Software engineers may not be able to perform reuse because of lack of knowledge (e.g., for accessing and handling repositories) or cannot anticipate.
- This limits investment in reuse-based development projects
- No investment in reuse projects limits reuse activities for software engineers.

While Ye and Fischer (2005) shows that a lack of knowledge exists, McCarey, Ó Cinnéide and Kushmerick (2008) identifies the following three problems in software reuse:

- Inability of support tools to automatically identify reuse opportunities
- The separation of reuse from mainstream development
- The lack of techniques to store and subsequently distribute task relevant component knowledge among software engineers

The last statement of McCarey, Ó Cinnéide and Kushmerick (2008) is relevant for this discussion and for the research of this thesis. It contains three relevant statements:

1. Task has/needs knowledge to be performed
2. Task relevant knowledge has to be stored
3. A lack of techniques for storing such knowledge

The first statement is also noted by Ajila and Zheng (2004) who claim that knowledge is the relevant factor in software development. Because of the changes in technologies and the knowledge of software engineers, this knowledge increases and has to be maintained. The second statement is also supported by Ajila and Zheng (2004) and Qu, Ji and Nsakanda (2012). The size of a company is also related to reuse problems regarding missing knowledge. Ha, Sun and Xie (2012) mentioned based on Mishra and Mishra (2009) that more small and medium

Problem of missing knowledge in software unit reuse

sized enterprises (SME) exist then large companies. An example is shown by Mishra and Mishra (2009): in 2009, 77% (Germany) 69% (Brazil) of the software development companies in Germany and Brazil were SME. Fayad, Laitinen, and Ward (2000) showed that in USA 99,2% of all software development companies were smaller than 250 people. The different studies used different metrics and characteristics to identify SME. Also they are performed in different countries and cultures. Among these differences, SMEs typically have the following problematic attributes (based on Mishra and Mishra, 2009):

- Insufficient development environments
- Low budget
- Customer dependencies
- Development teams usually consist of only one or a few team members.

Thörn (2010) concludes the missing of reuse in SMEs. This statement is complementary to the conclusion of Jansen et al. (2008) and Ajila (2006). Ajila (2006) especially showed in their multi company/project analysis the difference between large and medium (small) sized companies. Large companies are able to store knowledge for reuse but based on organisational problems the expected reuse is limited. Medium sized companies on the other side are efficient when using knowledge but usually the processes to store or search knowledge are not available for cost reasons. Additionally, the exchange of knowledge between software development teams is seen as difficult (cf. Qu, Ji and Nsakanda, 2012).

The limit of team member size is also discussed by Johansson, Hall and Coquard (1999) and it appears that not only small software engineer teams have problems performing reuse, but also multiple teams in global companies and different companies working together have the problem of exchanging reuse knowledge based on the team member size and the global distribution of teams (see also O'Sullivan, 2003; Qu, Ji and Nsakanda, 2012).

The size of teams is not the only impact factor. The knowledge of each individual software engineer is different and is handled differently by each person. Ye (2001) categorises the

Problem of missing knowledge in software unit reuse

activity of reuse focusing the search of reusable software units into three classes: reuse-by-memory, reuse-by-recall, and reuse-by-anticipation.

The reuse-by-memory scenario describes a software engineer identifying a software unit that can be reused and fits with the given requirement. In a reuse-by-recall scenario, a software engineer knows of one or more existing software units, but is not sure where to find or how to access it. The last scenario, reuse-by-anticipation, is given if a software engineer has no idea about useful software units which may be useful.

Based on this classification McCarey, Ó Cinnéide and Kushmerick (2008) argues that the first two classes (reuse-by-memory and reuse-by-recall) are state of the art in software development. Therefore, no problem exists. Software engineers may find the correct software unit. But the third class (reuse-by-anticipation) includes several problems based on missing knowledge:

- *“A software engineer may incorrectly anticipate a component that does not exist.*
- *It is difficult for a software engineer to clearly express their reuse intentions.*
- *A software engineer cannot easily evaluate retrieved components due to the engineer’s knowledge limitations.”* (McCarey, Ó Cinnéide and Kushmerick, 2008, p. 54)

Picot (2003) shows a demand model for management knowledge which can be used to show the classification of Ye (2001) and the three problems shown by McCarey, Ó Cinnéide and Kushmerick (2008). This model was adapted by Zinn et al. (2011a) for software unit information demand to support the research focused by this thesis.

Figure 7 shows four relevant areas:

- OID – includes all (theoretical) software units which can solve a specific problem
- SID – includes all software units a software engineer believes can solve a specific problem
- IP – includes all software units which are provided/accessible at the moment.
- IQ – This area indicates a search request of a single person. It is based on the area SID; as a result, the IQ area overlaps the SID every time.

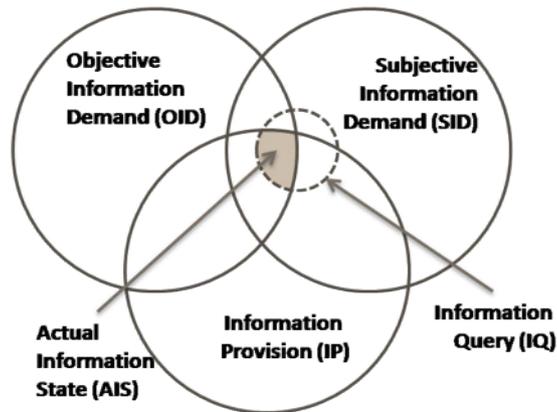


Figure 7 - Software Reuse Information Demand Model based on Picot (2003, p. 106) adapted by Zinn et al. (2011a)

The reuse-by-memory scenario is presented in the Actual Information State (AIS) area that is overlapping area of OID, SID, ID, and IQ. From the perspective of the information demand model the software units in this overlapping area are valid, known by the user, accessible, and describable (in a search request) by a user.

The reuse-by-recall can be represented in this model. Because the user is not sure which unit is the correct one and how to access it the two overlapping areas between (SID and OID / SID and IP) can contain the unit a person mentioned. Because this person cannot describe the searched unit the IQ will be very small. Regarding to Picot (2003) this will limit the useful results.

The reuse-by-anticipation may be presented in an information demand model as the area of SID. In this case the user has no idea about adequate software units. This reduces the area of SID and limits the overlapping areas to OID and SID. Because of the mentioned restriction of creating an IQ the valid results in this approach are very small. Two of the discussed problems of McCarey, Ó Cinnéide and Kushmerick (2008) can be also shown in this model. The first problem is that the SID may be containing correct and incorrect solutions. As a result, a person may think they know the correct solution when it could be wrong. The second problem is indicated by Picot (2003) as the difficulty of a person in creating the correct search request based on missing knowledge. In the Software Reuse Information Model (SRID) model a non-

Problem of missing knowledge in software unit reuse

correct search request may be reduce the area of IQ or change the location of the area to the field of SID. In both cases the overlapping with the AIS area will be reduced.

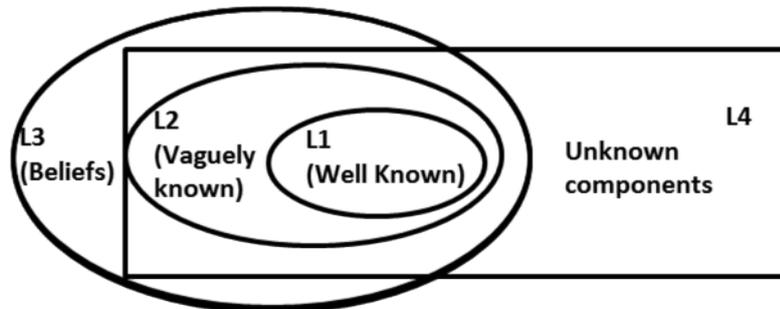


Figure 8 - Users experience level (based on Ye, 2001, p.2)

Ye and Fischer (2005) discuss a model (based on Ye, 2001) which demonstrates the view of reuse repositories on the reuse knowledge of software engineers (cf. Figure 8). This model shows that the knowledge of software engineers partly consists of information the engineers knows well (L1), knows vaguely (L2), knows of (L3), and doesn't know (L4). This is similar to the above discussed information demand model and the perspective of McCarey, Ó Cinnéide and Kushmerick (2008). An relevant difference to other knowledge perspectives is the relation of knowledge and task-relevant information. For Isoda (1992), Budhija and Ahuja (2011), and the author of this thesis reuse includes a set of specific activities, each of which need different information. But the necessary information is not only defined by the activities. Visser (1990) showed in a case study that different software engineers use different ways to do the same activity, often including different information. The same effect is explained by Sen (1997). This leads to the discussion of knowledge or information interpretation. Qu, Ji and Nsakanda (2012) and Choi, Lee and Yoo (2010) argue that the interpretation of knowledge leads to different results. Such results may be different to the expected results in the knowledge creator. As shown by Qu, Ji and Nsakanda (2012) this is a typical problem of knowledge transfer between different teams. Using distributed teams is a typical scenario in software development and software reuse (see O'Sullivan, 2003; Johansson, Hall and Coquard, 1999). Interpretation may be one reason why people repeating a task use different ways and/or information or knowledge.

Problem of missing knowledge in software unit reuse

The same information provided to another person may not be supporting this person. On the other hand it shows that a solution which supports persons in reuse has to be flexible to handle different information from different perspectives. (cf. discussion of Johansson, Hall and Coquard, 1999; Ye and Fischer, 2005)

LaToza, Venolia and DeLine (2006) in a case study, show that a lot of software engineers assist others by performing development tasks. This implies that lot of software engineers need support for software reuse. For McCarey, Ó Cinnéide and Kushmerick (2008) this lack is based on knowledge and will be increased in component-based development because components may be larger and more difficult to integrate. This statement is also seen by Ajila (2006) which indicates that the growing number of information a software unit includes has to be maintained or will result in project problem (e.g., time issues). Therefore, the limited view on components of McCarey, Ó Cinnéide and Kushmerick (2008) can be enlarged to other software units (e.g., service and classes) as well.

To conclude this section is to show relevant research statements of literature that relate problems of missing knowledge to software development activities. Regarding this relation, McCarey, Ó Cinnéide and Kushmerick (2008), partly supported by the statements of Frakes and Kang (2005), Ajila (2006), and Cummings and Teng (2003) come to the following conclusions:

- Software development requires an amount of experienced user knowledge.
- Often, software engineers do not have the required knowledge for specific development tasks.
- Knowledge about technology is not transferred between engineers or teams.

Thereby, the fact that knowledge is missing was identified by the literature. Also, the impact is well known (cf. Section 2.2.1.5):

- Reduced quality (see Morisio, Ezran and Tully, 2002; McCarey, Ó Cinnéide and Kushmerick, 2008)

Problem of missing knowledge in software unit reuse

- Longer development times (see Ajila, 2006; McCarey, Ó Cinnéide and Kushmerick, 2008)
- Failure of development (see Morisio, Ezran and Tully, 2002; Ajila, 2006)
- Rise in costs (see Ajila and Zheng, 2004; McCarey, Ó Cinnéide and Kushmerick, 2008)

An interesting point is that missing reuse knowledge may lead to (from the view of the literature) a contrary result than the estimated results in improvements in quality, shorter development times, success of development, and reduction of costs (see Section 2.2.1.2).

2.2.3.3. A key challenge for software unit reuse

In the previous section, different relevant research statements from the literature are discussed.

These statements will now be consolidated to identify the key challenge for this research.

One fact the literature shows is that knowledge is relevant for software unit reuse. For example, Ajila (2006) and Cummings and Teng (2003), identify knowledge as a critical success factor in software unit reuse. Isoda (1992), Budhija and Ahuja (2011), and McCarey, Ó Cinnéide and Kushmerick (2008) state that the task of reuse is based on knowledge.

McCarey, Ó Cinnéide and Kushmerick (2008) conclude that a lack of techniques to store and subsequently distribute task relevant component knowledge among software engineers exists.

Using the other statements identified in the literature by the previous section, this lack is based on following problems:

Problem 1 – Knowledge required based on variants of different technologies: Frakes and Isoda (1992) state that reuse is difficult because of different technologies and user domains. The knowledge of reuse activities, which is called task relevant component knowledge by McCarey, Ó Cinnéide and Kushmerick (2008), is based on the technology of the related software unit. Next to the multitude of existing technologies, Ajila and Zheng (2004) state that the rapid changes in technologies and required knowledge have to be maintained. As a result, the first challenge to limit the lack, McCarey, Ó Cinnéide and Kushmerick (2008) focuses on, is

Problem of missing knowledge in software unit reuse

to handle the problem of different technologies and the related software construction activity knowledge of different software units.

Problem 2 – Different knowledge level of software engineers: Another challenge is to find a way to distribute reuse activity knowledge at the level of software engineers' knowledge. Ye (2001) discusses three types of software engineers' knowledge related reuse types. The analysis using the models of Zinn et al. (2011a) and Ye and Fischer (2005) show that software engineers can have different knowledge levels. As a result, the conclusion of Ye and Fischer (2005) is that software engineers may be not able to perform reuse because of a lack of knowledge. This statement is also made by McCarey, Ó Cinnéide and Kushmerick (2008).

On the one side, this knowledge is required to perform a reuse activity. In this case the previously discussed problem of knowledge interpretation occurs. Qu, Ji and Nsakanda (2012) and Choi, Lee and Yoo (2010) identifies this, particularly in an environment where multiple teams exchange knowledge. This may lead to variations in the reuse activity result or to the failure of the reuse activity. Another point of interpretation is that software engineers use different ways of working to perform the same activity even if the underlying information is equal (Visser, 1990; Sen 1997).

On the other side, the software engineers have to know how to use a knowledge resource or know somebody who is experienced (Qu, Ji and Nsakanda, 2012).

As discussed in the previous section, it is difficult for a software engineer to clearly express their reuse intentions (McCarey, Ó Cinnéide and Kushmerick, 2008; Picot, 2003). In the case of a reuse activity a user has to describe what software unit is required and what kind of reuse activity has to be done.

Problem 3 – Distribution of knowledge: McCarey, Ó Cinnéide and Kushmerick (2008) states that the distribution of knowledge about technology between engineers or teams is not adequate. Next to the problem of interpretation and the use of knowledge, an experienced software engineer has to distribute knowledge in a way that other engineers are able to

understand (see Taweel et al., 2009; Boden and Avram, 2009). This implicates an infrastructure which provides the functionality to upload activity information and knowledge. Additionally, it has to provide the possibility to find and access this infrastructure for searching, receiving, uploaded, and execution of knowledge (cf. Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010). Frakes and Kang (2005), Ajila (2006), and Slyngstad et al. (2006) discuss the need of repositories; usually software engineers have repositories, but these are different in type and distribution. This can range from personal project files to a team or department repository. As a result, a software engineer has to know where to find a repository, how to access it, and how to use it. The last point relates to the previously mentioned problem of mind-set and capability of formulating a request. As a result, an inexperienced user has to know how to find and access this knowledge source (cf. Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010). As shown by Ajila (2006), large companies are able to store knowledge for reuse, but based on organisational problems the expected reuse is limited. To limit the lack described by McCarey, Ó Cinnéide and Kushmerick (2008) one challenge is to create such an infrastructure. Following the discussion of Qu, Ji and Nsakanda (2012) and Choi, Lee and Yoo (2010) such an infrastructure has to be information technology (IT) based. Visser (1990) and Morad and Kuflik (2005) state the use of special teams or single experienced users for a single software unit - as support for other software engineers or development teams in bigger companies. Ha, Sun and Xie (2012) and Thörn (2010) also mentioned that this is not usually possible in SMEs.

Problem 4 – Definition of software reuse knowledge: The last challenge discussed in this section is the definition of software reuse activity knowledge. In the focused problem statement, task relevant component knowledge should be exchanged between software engineers (see McCarey, Ó Cinnéide and Kushmerick, 2008). But a definition of this knowledge is not given. Based on the amount of possible knowledge, for example, based on the technology variations, this is challenging. On the other side, activities are recognised by the literature as typical activities of reuse. Bosch and Bosch-Sijtsema (2010) and Shiva and Shala

(2007) for example indicate the need for integrating a reusable software unit into the development environment. Also, for Vliet (2008) and Mens and Vangorp (2006) it is necessary to adapt an existing unit before reuse. Especially in the area of embedded devices (see Carlson et al., 2010; O'Connor et al., 2009), the deployment is an relevant part and often depends on previous created software units.

Note: Also other problems may exist which can be seen as challenges. The selected problems are identified by using scientific literature. The research determines if a new approach is able to limit these four problems. Whether the research is able to do this or not, it creates a small added value for this research area.

2.3. Conceivable research contribution

The discussion in Section 2.2.3.3 supports the statement of McCarey, Ó Cinnéide and Kushmerick (2008) whereby the lack of techniques to store and subsequently distribute task relevant component knowledge among software engineers exists. The discussion shows that a technique has to handle four different problems to successfully handle knowledge based problems in the exchange of reuse activity knowledge.

The research of this thesis aims to contribute to the field about the lack of techniques to store and subsequently distribute software construction activity relevant software unit knowledge. This includes the execution of software construction activity activity knowledge as part of the distribution. To identify an approach to handle the four problems would create a contribution to this research area and support software engineers in today's problem of software unit reuse.

The reuse or use of previously stored SCAC is an procedure reuse type (cf. Petro-Diaz, 1993). Regarding the knowledge area, the research deals with the knowledge necessary to perform the SCAC and the knowledge of the distribution environment. The literature discussion about the type of reuse (black-box, withe-box, reuse-within-development, etc.) is not relevant for the

problem discussion. This part of the literature review is used to create a solution approach in Chapter 4. The historical view in the literature does not show these problems.

2.4. Summary

This chapter starts with an overview about the research methods for the literature review. The main part of this chapter is the literature review itself. First of all, this review defines the view of this thesis on software reuse as a software development including different software reuse activities. After this, a definition of typical estimated characteristics of software reuse is discussed (e.g., a reduction in costs or an increase of product quality). Software reuse is separated by the literature into 11 areas (e.g., design patterns, component-based software development, software product lines, and so on). These different areas use different software unit types which require different software units to handle them. Also, different impacts of inadequate reuse are discussed (e.g., rising costs and limitations in quality). A relevant result of this chapter is found by the definition of reuse activities (i.e., all tasks that are necessary for reuse) and software construction activities which is a subset of reuse activities (e.g., integration or transformation). After the discussion of relevant definitions, an overview of different research areas is given including the focused research area of this thesis that handles (missing) knowledge for software reuse. In the next part of the literature review different examples of problems of missing knowledge and their relation to human-driven knowledge is discussed. This discussion also includes the focused problem of missing knowledge for the execution of software construction activities of this thesis. The literature discussion identifies a lack of techniques to store and subsequently distribute software reuse activity relevant software unit knowledge among software engineers. Following problem areas were identified that make the creation of such a technique challenging:

- Insufficient knowledge level of software engineers
- A high variant of existing technologies and related (activity) knowledge
- The distribution in global company environments

Problem of missing knowledge in software unit reuse

- Missing definition of reuse knowledge for reuse activities

The chapter concludes with a discussion concerning the targeted contribution which focuses on the four identified problem areas and the aim to enable the inexperienced user to perform software unit reuse.

3. Missing software construction activity knowledge – problem analysis

The literature review in Chapter 2 shows that a lack of techniques to store and subsequently distribute software reuse activity knowledge exists. Also, problems are explained which show that the creation of adequate techniques is challenging. This research investigates an approach for a technique to store and subsequently distribute software reuse activity knowledge. Therefore, this chapter shows additional analysis results focusing on the identified problems of the creation of an adequate technique. The first step is the definition and explanation of focused software construction activities: integration, transformation, and deployment. The second one is the analysis of the different problems based on missing knowledge. Thereby, a low knowledge level of an inexperienced user is discussed for each problem. The final analysis shows an overview of existing approaches and concludes with the need of an approach to focus on all mentioned problems to limit the lack of techniques of handling software reuse activity knowledge (on the example of software construction activities) and, therefore, enable inexperienced users to perform software unit reuse.

3.1. Focused software construction activities

This section explains the focused software construction activities that are the research objects of the research. On the one hand this is necessary to identify necessary reuse activity knowledge. On the other hand the aim is to give the reader an overview of the focused software construction activities for the description of the focused problems of these activities (relation between analysed problems and reuse activities) in Section 3.2. The following reuse activities are in the scope of this thesis:

- a) Integration of software units in integrated development environments
- b) Console-based transformation of software units into other technologies
- c) Deployment of software units into embedded devices.

Note: The topic of integration and deployment activities was published by the author (see Zinn, Fischer-Hellmann, and Schoop, 2012a; Zinn et al., 2011b). Other reuse activities also exist. These activities were chosen because they have been identified in literature (cf. Bosch and Bosch-Sijtsema, 2010; Shiva and Shala, 2007; Vliet, 2008; Mens and Vangorp, 2006; Carlson et al., 2010; O'Connor et al., 2009). In the following section, these activities and their special scope will be introduced. These are used as examples to support further discussion of the identified problem areas (i.e., insufficient knowledge level of software engineers, high variance of existing technologies and related activity knowledge, and the knowledge distribution in global company environments).

3.1.1. Console-based transformation of software units – transformation activities

Different activities within development projects with software reuse require different kinds of transformation expertise. For example the modification of a software unit into another form may be reached by using transformations (Mens and Vangorp, 2006). The results of such transformation processes are manifold. They range from simple content-adaptation to an adaptation that changes the basic technology of the software units. An example of a content adaptation is changing the content (domain) of a method within a class or a component (Seriai, Bastide and Oussalah, 2006). An example of a technology change is the conversion of a Java-based component to a .Net-based component (Frijters, 2011). Another frequent form of transformation is the extension of a software unit with new information and interfaces. This, for example, allows simple transformation of Java-based components to a web interface (Lee et al., 2005). Other examples are: compiler or interpreter transformation using source codes and model-driven software development transformation using different models.

The transformation of a software unit can be defined by using the definition of model transformation from the area of model-driven software development:

“A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.” (Kleppe, 2003, p. 24)

Note: The focused transformation activity is based on console applications. As a result, the software engineer is using an application to perform the transformations.

3.1.2. Transformation activity example

There is no typical or standardised transformation activity step identified in literature, but the following example will illustrate a real transformation process of a software unit. To transform a Java-based byte code into .NET-based byte code the tool IKVM (Frijters, 2011) may be used. In this example the Device Profile for Web Service (DPWS) is used. The DPWS software unit enables embedded devices to handle different web service protocols (called WS*) like Web Service discovery and security (see Jammes, Mensch and Smit, 2007). The DPWS software unit based on Java should be transformed into a .NET compatible software unit. The IKVM tool offers two different transformation scenarios. The first one is to run the original Java byte code in a Java virtual machine inside the .NET application. Therefore, only some interfaces are required by the software engineer. The second scenario is to make a real transformation. The last scenario is used to describe a transformation activity.

Aim: The aim of this transformation activity is to transform the DPWS Java-based software unit into .NET binary libraries.

Precondition: The software engineer has searched and identified the software unit, as well as identified the IKVM transformation tool. The used operating system (OS) is a Windows OS.

Preparation: The software engineer has to download the IKVM tool and transfer the ZIP package to a folder. The engineer has to set Java path information in the OS settings because

the IKVM is based on the Java runtime. As an additional precondition is that the Java runtime has to be installed. The tool is installed and can be executed by typing starting the IKVM executable that is located in the IKVM binary folder in the windows console window.

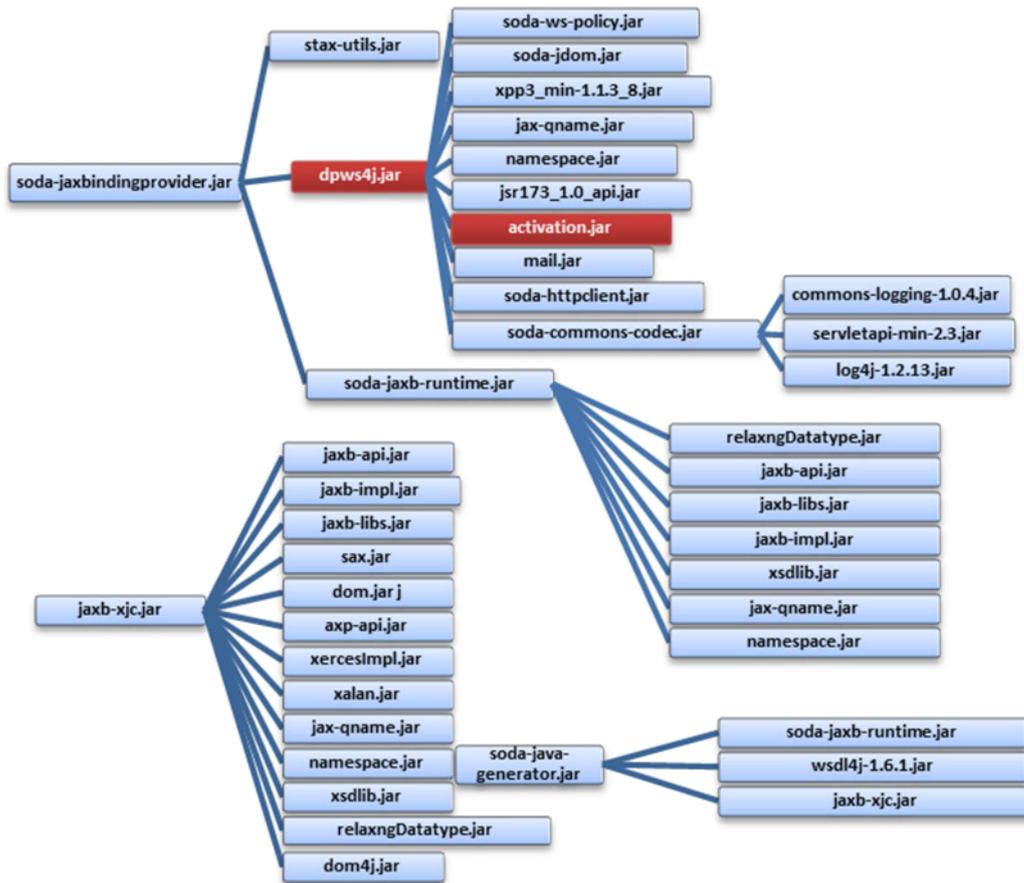


Figure 9 - Dependency hierarchy of DPWS Java libraries (blue external libraries; red internal libraries)

Additionally, the engineer has to download the DPWS Java stack which includes 23 Java binary files. These files have to be extracted to a folder.

As additional task in the preparation phase is that the engineer has to identify references between the DPWS Java libraries or other external libraries. These are necessary to perform the transformation. Figure 9 shows the dependency hierarchy between DPWS libraries and external libraries. All libraries here have direct or indirect dependencies and have, therefore, been transformed. The file DPWS4J.jar contains the relevant DPWS functionality.

Execution: The software engineer has to identify all libraries without any other dependencies.

These can be now transformed by using IKVM as shown in the example in Figure 10.

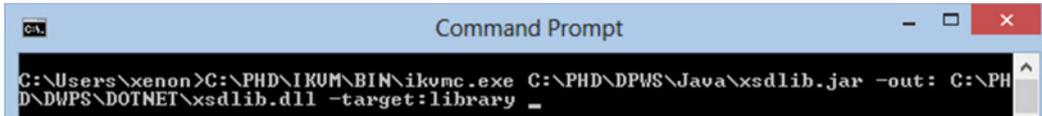


Figure 10 - Example IKVM transformation execution

The engineer has to be experienced enough to set the parameters so that the IKVM tool is able:

- to create a DLL file ('-target:library'),
- to load an input file ('C:\PHD\DPWS\Java\xsdlib.jar'), and;
- to identify the location and name of the output file ('C:\PHD\DPWS\DOTNET\xsdlib.dll').

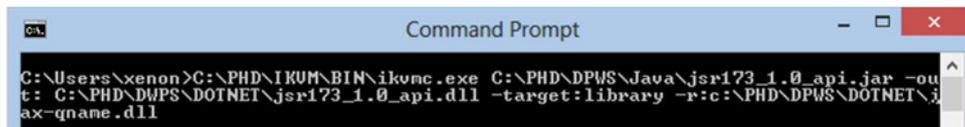


Figure 11 - Example IKVM transformation execution with dependency

This procedure has to be repeated for each Java library at the end of the dependency hierarchy.

For each of the libraries with dependencies the execution of IKVM appears as follows; the call has a new parameter `-r` or `-reference` and a path to the referenced dependency. Figure 11 shows an example of an IKVM transformation using a dependency ('`-r:c:\PHD\DPWS\DOTNET\jax-gname.dll`').

Additional to the parameters from the first call, the engineer has to specify the location of the previously transformed dependencies. To transform the top of the DPWS dependency hierarchy, multiple dependency parameters have to be used. Figure 12 shows the transformation call including the dependency parameters which are shown as references in Figure 9. To complete the transformation, 23 single transformation activities have to be executed.

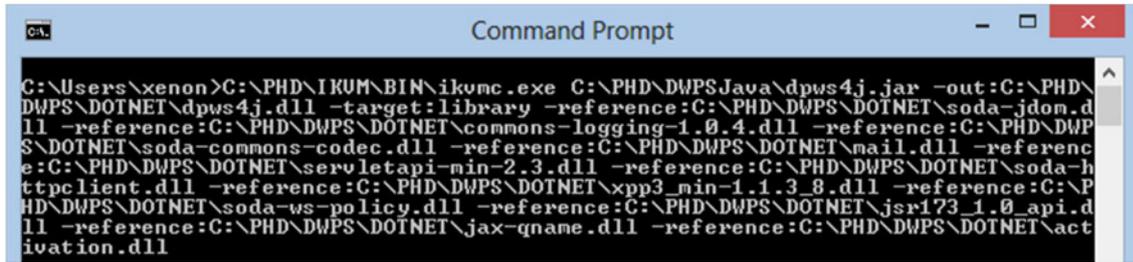


Figure 12 - Example IKVM transformation execution of the DPWS4J.JAR file

Output: The output of this transformation example is a set of .NET libraries including the functionality of the Java DPWS stack. The engineer has to know that the transformation result has special dependencies. On the one hand the dependencies between the original DPWS Java libraries exist. This is shown in the resource view in Figure 13. Also, the dependencies to external libraries exist. This is shown in the reference view in Figure 13.

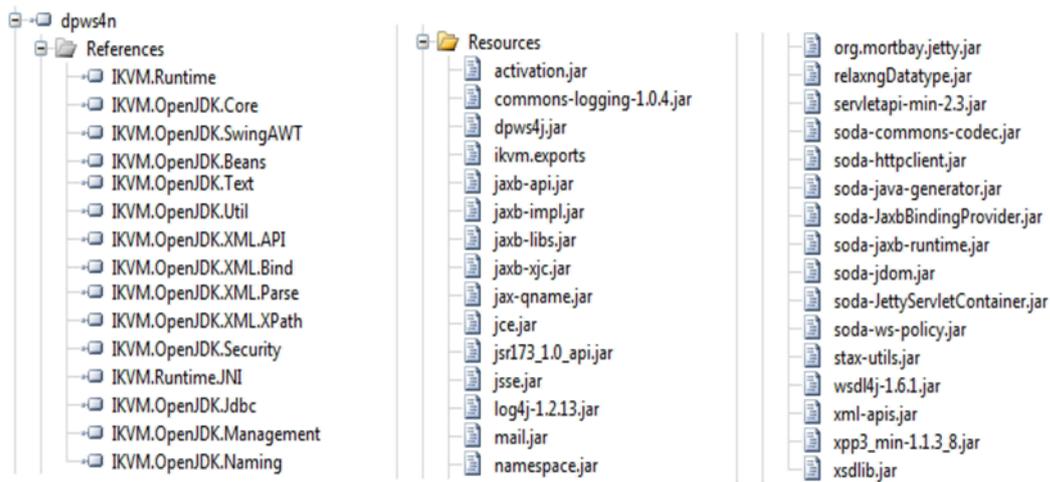


Figure 13 - Dependency hierarchy of DPWS .NET libraries (ILSpy View)

To handle these libraries, the transformation output refers to special IKVM libraries. As a result, these special libraries have to be shipped and deployed together with the transformation result. Figure 14 shows the dependency hierarchy of the software unit created by the transformation result, references to external libraries (blue), and internal (red) which are results of the transformation process.

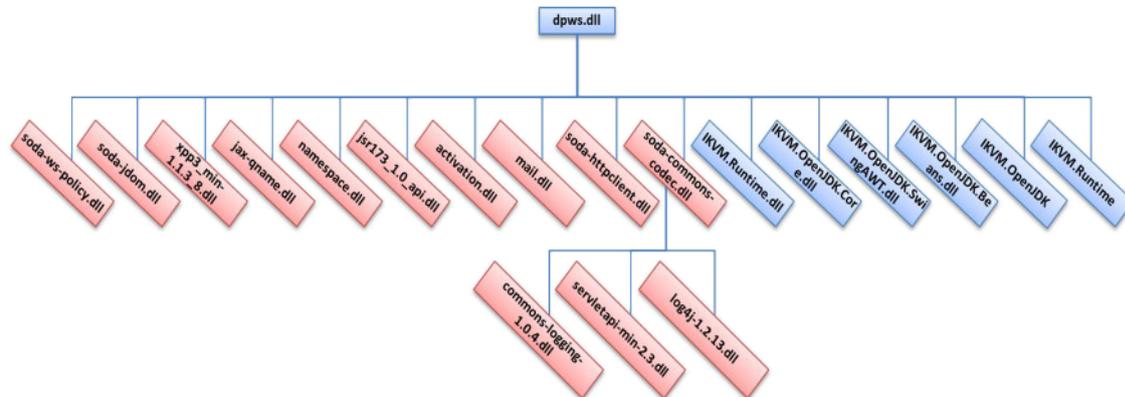


Figure 14 - Dependency hierarchy of DPWS .NET libraries (blue external libraries; red internal libraries)

3.1.3. Integration of software units in development environments – integration activities

In the scope of software unit reuse it is necessary to integrate these units into development environments or, more precisely, into a development project. Software reuse environments (SRE) support software engineers by addressing these activities. The SRE discussion in Section 2.2.1.6) shows that IDEs are such SRE systems.

Note: In this research the integration of software units into an IDE is equal to the integration of a software unit into a development project. Because the focused problems are distinguishable from the different knowledge requirements of the different IDEs the first term will be used in this work. Additionally, it is relevant to know that the focused integration activity does not include the activity of interface adaptation or mapping. These are examples of transformation activities.

In the scope of the integration activity four interesting perspectives exist: the first perspective is concerned with the integration of software units. Software engineers have to be aware of the technical properties of a software unit. Software units may consist of different parts (e.g., files or environment variables). Each of these parts may need to be integrated differently into an IDE. (see discussion of Zinn et al., 2011b).

The second perspective is the required file structure of software units. The different parts of a software unit may require a specific location in the project. Each part of a software unit may have dependencies themselves. Also, these dependencies may have a required location which can be specific or relative to the dependent part of the software unit.

The third perspective is the IDE in use. Today different IDEs exist, and some of these are specialised. The development of software for embedded devices for example, often requires a special IDE for the targeted device or special libraries in a normal software development project. Other IDEs, for example, Eclipse and Visual Studio can be extended and used for different application languages or technologies. For each IDE the provided functionality for software unit integration is different. As a result, the knowledge to use this functionality differs too.

The last relevant perspective is concerned with the different scenarios of integration. The scopes of the previous perspectives can be used in distributed and non-distributed scenarios. Typically, for an SME scenario, the decision maker, the person who decides to reuse a specific software unit, is the same as the integrator, implementing the reuse. However, there are scenarios which include the decision maker and the integrator not being the same person. In the scope of this thesis this is called a distributed scenario because the individuals can be located in different locations and differ in their domain of experienced users (see Section 2.2.2.3). Typically, software architects are this kind of decision makers in software development (see discussion of Kruchten, Capilla and Dueñas, 2009).

3.1.4. Integration activity example

There is no typical or standardised integration activity process identified in the used literature, but the following example will illustrate a real integration process of a software unit.

Aim: The aim of this integration activity is to integrate the DPWS .NET library of the transformation example in Section 3.1.1 into the Visual Studio IDE.

Precondition: The software engineer has searched and identified the software unit. The used operating system is a Windows OS and a Visual Studio 2010 with a loaded development project.

Preparation and Execution: Figure 14 shows all files necessary for a runnable integrated software unit. Therefore, all files marked blue in the figure (includes all created .NET files and special files of IKVM for .NET) have to be inserted as references in the Visual Studio project. Due to special behaviours of IKVM the engineer has to copy all additional IKVM files into the same folder as the .NET DPWS files. It is also possible to insert a library path in the project configuration instead of copying all additional IKVM files. Additionally, the Java Virtual Machine (JVM).dll file has to be copied into the DPWS .NET folder without referencing. The engineer has to select the 32bit or 64bit version, depending on the development project configuration. To get the file automatically copied to another build or debug folder of the project, the engineer can choose between two (automatic) options:

- Create a copy order in the pre-condition or post-condition settings of the project
- Add the file to the project tree and set its reference property to ‘Copy’ or ‘Copy if newer’

In both cases, Visual Studio will copy the file if the debug or release folder is changed.

Output: The result of this integration activity is a DPWS .NET library integrated into a Visual Studio project including all dependencies. Figure 15 show the resulting folder structure of the integration activity and the project structure in a Visual Studio environment.

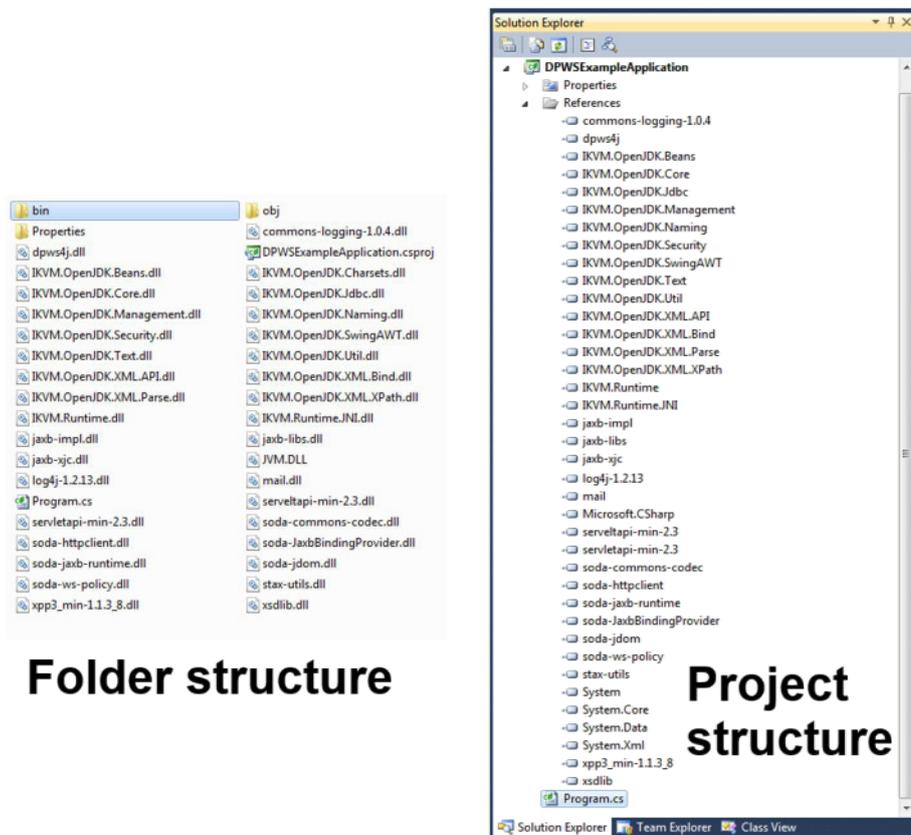


Figure 15 - DPWS integration activity folder and project structure

3.1.5. *Deployment of software units into embedded devices – deployment activities*

Deployment of embedded devices is seen as the physical set up of devices in a specific environment (e.g., wireless sensors; Bohn, Bobek and Golatowski, 2006; and medical devices; Burg et al., 2009). From a software development perspective, deployment may be seen as the installation of software on a system (Burg et al., 2009). From here on, the term 'deployment' will refer to the latter definition. The conception of embedded devices has changed in the past. Originally, such devices were perceived as follows (based on Gill, 2005):

- Specialised on a specific task by limited functionality.
- Built for an unchanging environment.
- Limited by resources.

Nowadays, they are perceived as embedded systems, which are characterised as follows (based on Gill, 2005):

- (Self-)adaptive, open and more efficient
- Capable of dynamically handling multiple tasks
- 'Plug and Play'-able for integration

The reason for this change of perception can be seen “...as a consequence of the integration of IT” (Gill, 2005, p. 7) into the field of embedded systems. Gill (2005) perceive this change to be a result of advancements in hardware and software of embedded systems. Over time, hardware has become more capable of handling increasingly complex software instructions, more advanced software technologies, and platforms (see Gilart-Iglesias et al., 2006; Gill, 2005). This increased flexibility enables the implementation of special software features, namely: Fault Tolerance (Pinello, Carloni and Sangiovanni-Vincentelli, 2008), Security (Gogniat et al., 2008), and Dynamic Infrastructure (Karnouskos and Tariq, 2009). The high number of available embedded devices poses a problem for software reuse. This problem is especially apparent in the area of automation where many different types of device exist. Usually, available devices are distinguished by hardware technology, software technology, form factor, and safety features. This results in the fragmentation of both software platforms and libraries for embedded devices. Therefore, the task of reusing such software units for embedded devices is becoming increasingly more complicated and requires special knowledge for regarding deployment. This is similar to the discussed problem of an increasing number of technologies for software units.

3.1.6. Deployment activity example

There is no typical or standardised deployment activity step, but the following example will illustrate a real deployment process of a web service to a device.

Aim: The aim of this deployment activity is to deploy a web service written in Java into an embedded device.

Precondition: The software engineer has searched and identified the software unit. The used operating system is a Windows OS, and an Eclipse IDE is used to build the deployment packages. For this example a GX300 Gateway with OSGi (earlier called Open Services Gateway initiative) device platform is used.

The deployment of a web service to an embedded device needs the following information:

1. Interface description file – Extensible Markup Language (XML) (Text)
2. Resource description file – Binary file (BIN) (Binary)
3. Manifest File (Project) - XML (Text)
4. Classes for the web service - Java Class (Java source-code)

Some of this information (1-3) is OSGi specific. By developing the web service, context dependencies to the OSGi platform has to be created inside the Java classes. The OSGi website provides different features like sample Java projects and Eclipse plugins for easier handling. Even by using such a support feature, the software engineer has to do following steps:

The first step is to create the interface description files that are a special text file. This file describes the interfaces of the new web service. The syntax and format are specified by OSGi. This is also true for the manifest file. This file describes all project information and file locations that are necessary to build a deployment package. The last file to create is the resource file, which is only a pre-formatted list of all files that are a part of the deployment package at the end. The last file is the service description itself. To create this file, the engineer has to develop a Java class with special dependencies (i.e., to Java libraries from OSGi) that have to be included into the project as references or path settings in the Eclipse environment.

Execution: The deployment package is created by compiling the project. To upload the file into a device, a console tool from OSGi has to be used. This tool needs the information from the new Java Archive (JAR) file and the IP address of the device. After deployment, the device has to restart. Depending on the device type, the restart has to be done automatically or manually by the user.

Output: The result of this deployment activity is a SOAP based web service running on a GX300 device and an OSGi platform. The running web service can be now reused by other software systems.

The shown procedure model differs depending on other platforms or device types (see Zinn, Fischer-Hellmann and Schoop, 2012a).

3.2. Problem analysis

The main problem this research focuses on is that an inexperienced software engineer is not able to perform SCAs. McCarey, Ó Cinnéide and Kushmerick (2008) conclude that a lack of techniques to store and subsequently distribute task relevant component knowledge among software engineers is responsible for this dilemma. The discussion of this statement in Section 2.2.3.3 identifies four major problem areas:

1. Amount of different knowledge exists based on different technology.
2. The inadequate knowledge level of inexperienced software engineers.
3. The amount knowledge required by distribution environments.
4. Missing reuse activity knowledge specification.

These problems have to be challenged by an approach to handle this lack. In the following these, problems are related to each other by using the view of typical problems in the area of knowledge management: knowledge storing (Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010), knowledge learning (Bjørnson and Dingsøyr, 2008; Ajila and Zheng, 2004), search and receiving of knowledge (Garcia et al., 2006), knowledge exchange (Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010) and knowledge execution (Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010). To discuss each problem, the software reuse information demand (SRID) model and the industrial context with the example studies of Schoop (2012) and an internal study of Schneider Electric (performed 2012; cf. Appendix Section E) is used.

3.2.1. Explanation of analysis contexts

In the following, the software reuse information demand model and the industrial working environment of software engineers are explained. Both topics are used for the problem discussion in this chapter.

3.2.1.1. Software reuse information demand model

For the problem analysis the SRID model introduced in Section 2.2.2.3 is used. In the following, two visualisation types of knowledge problems for inexperienced user are explained.

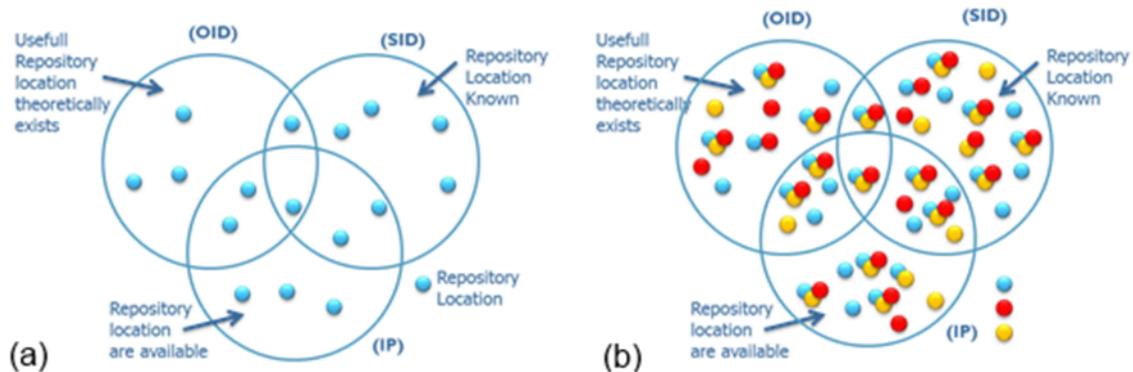


Figure 16 - Single problem visualisation (a) and multiple problem visualization (b)

Figure 16 shows both visualisations (a) called single problem visualisation and (b) called multiple problem visualisation. These types of visualisation are explained using the knowledge problem of accessible knowledge. Different locations may use different repositories. A user or a team has to know how to localise and access these repositories in a localisation scenario. From the perspective of the SRID model, this can be analysed separately or together. Figure 16 shows the two separate SRID models examples. In the first one (a) SID contains all repositories a user might think is useable. OID contains all repositories that may exist in the environment of the user. IP represents all existing repositories in the environment of the user. The SRID model (b) analysis is the same scenario with different perspectives. In (a) an element contains different problems (i.e., localisation, access, and use of a repository). On the other side (b) shows each knowledge problem separately.

A problem to visualise may contain different sub problems (e.g., knowledge about service, object, and component technology). Each of these sub problems can be visualised using single problem visualisation type (Figure 16a). These single problems can be now combined. In a software development project where a software engineer has to use all three technologies together a solution has to consist of triple all three technologies types. The amount of information is higher because OID may contain a lot of different variations of triple, double or single items based on the three single problem visualisations (see Figure 16b and Figure 17).

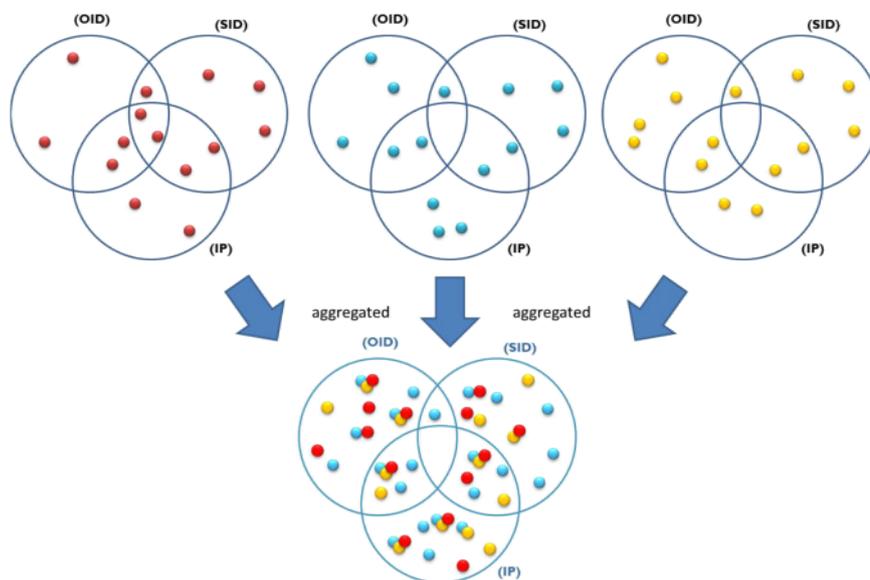


Figure 17 - Creation of a multiview SRID model out of single view SRID models

3.2.1.2. Reuse in industrial environment

In this thesis, the term industrial environment or context describes software reuse in an industrial environment. In such environments, software engineering is applied to solve problems, for example, in the areas of automation, building, power, software, and conveyance. Software development of desktop applications is one of the typical parts of software engineering in an industrial context. But this also includes software development for smaller devices or specialised environments (e.g., software for mobiles or out of space projects).

An relevant view for the research on software construction activities is the environment software engineers have to work in. Such an environment may create problems or difficulties. In the literature review different studies of development projects were discussed. The discussed problems arise for single persons and for whole working teams (cf. Qu, Ji and Nsakanda, 2012; Desouza, Awazu and Baloh 2006).

The environments for software engineers are different. As shown in Section 2.2.2.3 the size of companies differ, ranging from SME to large global companies. In such environments software engineers also try to perform reuse. Section 2.2.2.3 also shows the problems companies have with software reuse.

Next to the efforts of companies different European research projects also try to create software units which are reusable for different platforms (e.g., Bohn, Bobek and Golatowski, 2006; Jammes, Mensch and Smit, 2007). In this way, standardised software units for different companies and different industry areas are created.

Reusable software units should not only reuse in one project or one vertical market. It is also interesting to reuse in horizontal markets (see discussion of Szyperski, 2002b; Wang and Fung, 2004). Figure 18 shows examples for vertical markets as well as horizontal markets for software development projects. Often, software units are market or domain-specific (Frakes and Isoada, 1994). As a result, a software unit has not only to be reused in a vertical market. In horizontal markets the requirements may be different depending on the domain knowledge. In Figure 18 different industrial domains are shown (i.e., Power, Automation, and Building). Inside each domain different vertical markets exists. The building domain for example handles office, university, and hospital buildings. A software unit may used in each of these vertical markets. Additionally, the same software unit may be used on other horizontal markets, as for example, in factories of the automation domain. Global companies are involved in more than one vertical market. This may lead to distributed development team (cf. Qu, Ji and Nsakanda, 2012). In this thesis the terms horizontal projects and vertical projects are used to identify

software development projects which are used in one domain (vertical) or different domains (horizontal).

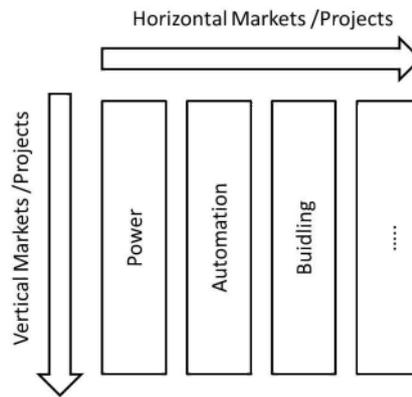


Figure 18 - Horizontal vs. vertical markets

Next to the development of usual desktop application, also other development specialisations exist. One specialisation is the development of software for embedded devices reusing reusable software units. Such devices are typically limited in their physical size and in the amount of available hardware resources and will be used for specific tasks. These devices are applied in all previously named industrial areas. In the automation area, for example, they control small parts of factory lines (cf. Jammes, Mensch and Smit, 2007). In the building industry, embedded devices are used as sensors for single rooms or whole buildings. Embedded devices are also used in the area of power to measure energy consumption or to predict bigger control devices. In conveyance they are mostly used as sensors. A gold mine, for example, may use more than 10,000 of such devices to measure sensor values (e.g., air pressure). Typical software engineering tasks are the development of software platforms (Firmware) for such devices and software units based on these platforms, including special functionality (Karnouskos and Tariq, 2009). Also, controller applications (so called Enterprise Systems) have to be created by software engineers.

Next to the specialised domains another relevant property for software engineers exists. In the industrial context often teams or people work together but do not share the same location. Large

companies, for example, outsource software development to other teams located in foreign countries to save money (cf. Desouza, Awazu and Baloh, 2006). Also, customers or development partners may be located in different countries. In this global industrial context software engineers have to work together and share experiences (see discussion of distributed software development of Boden and Avram, 2009; Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010; Taweel et al., 2009).

Chapter 2 shows different problems from the perspective of the literature. Two are relevant for this research: reuse problems based on the size of a company (see Chapter 2, Section 2.2.2.3) and the work in distributed teams. In this section the missing knowledge problems based on the distribution aspect are focused upon (cf. Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010).

The reuse of software units is an relevant part of software development in industrial areas (Henry and Faller, 1995). Its aim is to reduce cost and time in development projects (Morisio, Ezran and Tully, 2002). However, reuse in industrial projects does not guarantee a project's success; a fact that has been demonstrated by several project studies (see for example Morisio, Ezran and Tully, 2002). Next to the discussed organisational reuse problems, based Morisio, Ezran and Tully (2002) typical problems in industrial software development are:

- Misconceptions: Often, reuse is seen as technology; therefore, the use object-oriented technology is equal to reuse.
- No non-reuse specific processes are modified: Often, existing development processes are adapted to handle reuse.
- No reuse specific processes are installed: Reuse needs special processes which have to be prepared for systematic reuse.
- No training/awareness actions: Software engineers are not prepared or trained to perform reuse.
- Reusable assets produced but then not used: Often, reusable software artefacts are created but not reused or provided for reuse.

Missing software construction activity knowledge – problem analysis

- No production of assets: Often, non-reusable software units which are able to reduce costs are produced.

Due to the software units having to be developed in order to be reusable (Garlan, Allan, and Ockerbloom, 2009), the last problem: ‘no production of assets’, has special focus on the literature from others. Garlan, Allen and Ockerbloom (2009) and Morisio, Ezran and Tully (2002) argue that that companies often do not produce software units which are reusable. This requires more resources if these units are to be reused. Usually, the effort (e.g., costs and development time) to reuse should decrease after the creation of a software unit and should remain the same value continuously for each reuse. This requires software units prepared for reuse (Morisio, Ezran and Tully, 2002).

The advantages and disadvantages of reuse now are focused upon by creating artefacts in industrial contexts which are explained by using a practical example. An internal study performed by Schneider Electric (cf. Schoop, 2012) Appendix Section E shows an interesting picture. A set of 50 software units was created by different development groups and widely reused in different development projects in other development teams (including vertical and horizontal markets). The average reuse number was between 9 and 10. The reuse distribution of different software units is shown in Figure 19.

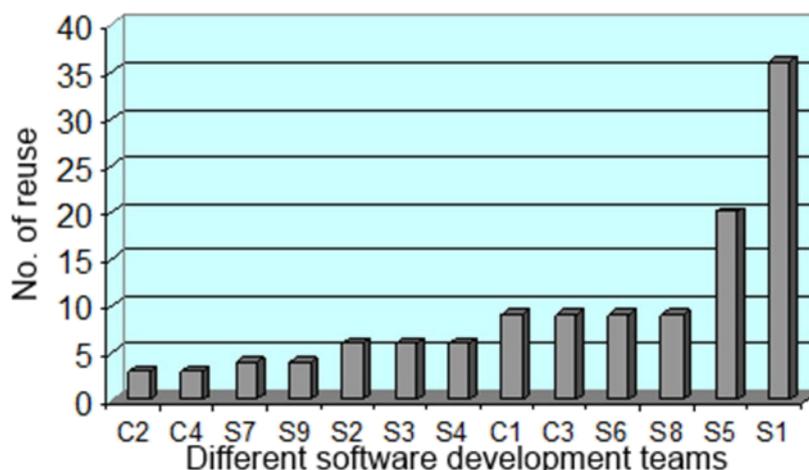


Figure 19 - Distribution of reusable software units (Schoop, 2012)

It starts with a minimum of 3 reuses (the point where development costs are typically recovered compared to a non-reuse scenario) and spans up to 36 reuses. All software units have been created for the purpose of reuse. This study of Schneider Electric identifies three typically reuse scenarios:

- a. **Multiple Teams with Support (MTwS):** Multiple teams reuse the same software units in different product lines, supported by the creation team.
- b. **Single Team with Support (STwS):** A single team reuses a software unit in different product versions that is not the same as the creation team, but supported by it.
- c. **Multiple and Single Team(s) without Support (MTwoS; STwoS):** Multiple teams reuse the same software unit in different product lines without the support of the creation team.

The corresponding development effort distribution is shown in Figure 20 ((a) STwS, (b) MTwS, (c) MTwoS) respectively.

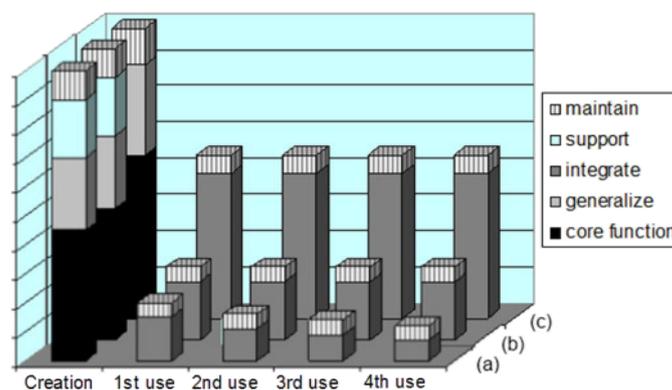


Figure 20 - Development distribution (a) STwS (b) MTwS; and MTwoS (c) based on Schoop (2012)

The creation of software units is equal for the STwS and MTwS scenarios regarding the maintenance, preparation of support and the development of the software unit (cf. Figure 20a and b). The highest effort is dedicated to designing the core function, to be followed by the effort to generalise the functionality as a basis for reuse (e.g., extended customisable function,

extended documentations, application notes, test case specification, and so on). Furthermore, a dedicated effort is needed to support the reuse later on and a maintenance effort for bug fixing, and later small evolutions are needed. In the STwS scenario the single team reusing the software unit reduced the integration effort in the first 4 reuses significantly. In the MTwS scenario each team reuses the software unit only once. A learning curve as seen in the STwS scenario is not created. Therefore, the effort for each team in an MTwS is similar. This is also true for the MTwOS scenario. The effort for each team is similar, but the general effort for the integration of the software unit is higher than in the other scenarios. The difference is the missing support of the creation team.

An internal survey at Schneider Electric from 2012 (see notes in Appendix Section E) aims to identify the effort involved and need for software unit reuse. It identifies different behaviours of software unit reuse for software engineers. Thereby, 86 people (engineers and technical managers working in development projects with reuse units) of different business units were asked 18 questions.

Figure 21 shows the reuse of different software units by different business units of Schneider Electric. These units are reused in different software products of these business units or in different product versions. While the reuse of general software units (e.g., eula information; cf. Figure 21a) is done by most of the business units, the picture changes to then focus on the device level (e.g., PC drivers, object Software Development Kit (SDKs), Schneider Electric specific programming SDKs; Figure 21c). Here, based on the difference on working with devices (e.g., different technology or platforms) not all business units reusing existing software units. What is interesting is the reuse of one software unit including a graphical user interface for device handling. This unit is reused several times (30x) by three business units. Figure 21b shows the use of platform specific software units (e.g., PC drivers, object SDKs, Schneider Electric specific programming SDKs). The number of reuses differs. While the signature and the PC driver units are reused several times, the specific SDK software units are not reused.

Missing software construction activity knowledge – problem analysis

These results show that software units are reused by different software development teams of different horizontal (in different business units) and vertical (inside a specific software unit) projects.

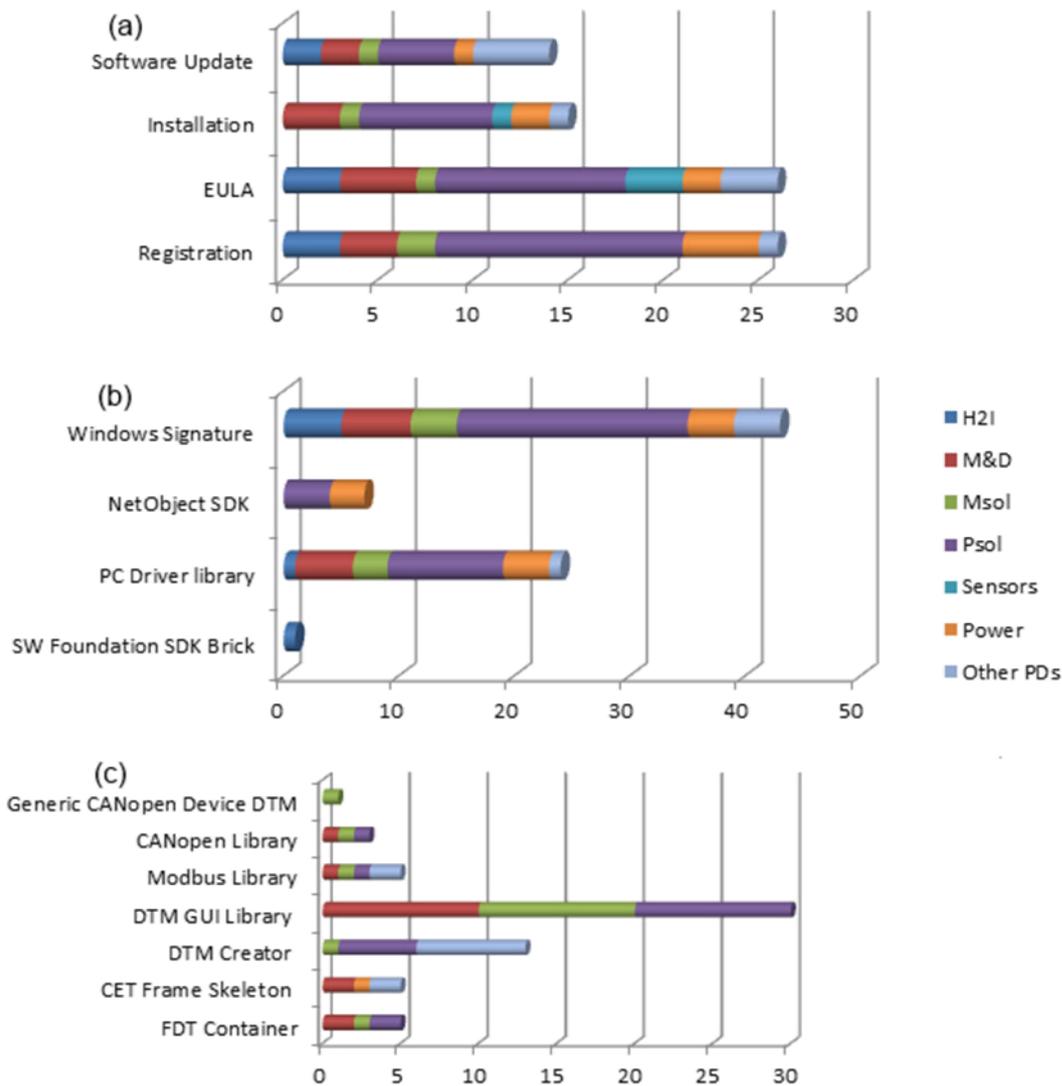


Figure 21 - Use of relevant software units in different business units (Schneider Electric; cf. Appendix Section E)

Additionally, the study shows what kinds of units are reused by the different business units. The survey concludes that source code, software modules (set of classes), and binary code and guidelines are the most recent reused units. Figure 22 shows the study results.

Missing software construction activity knowledge – problem analysis

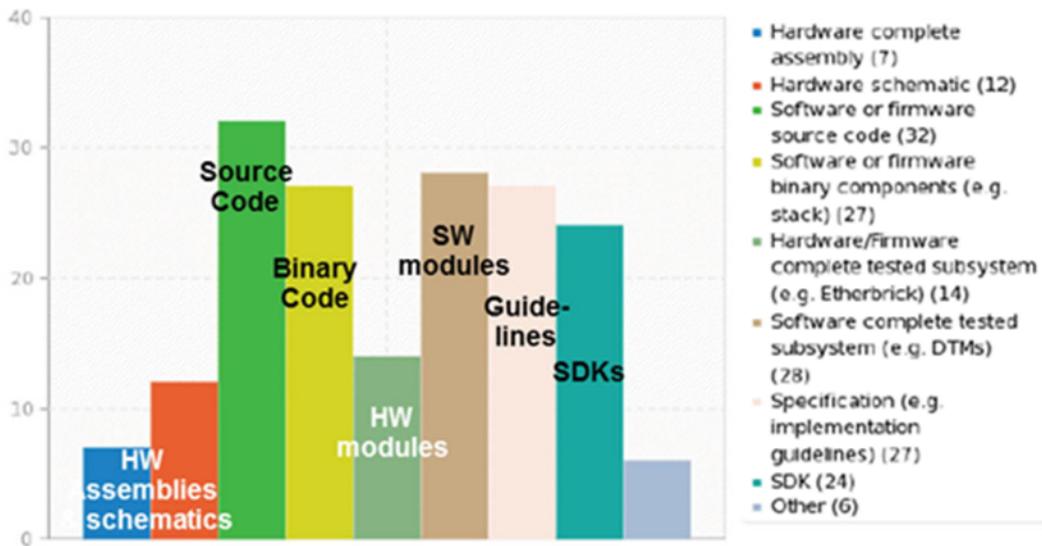


Figure 22 - Reused units in software development (based on Schneider; (cf. Appendix Section E))

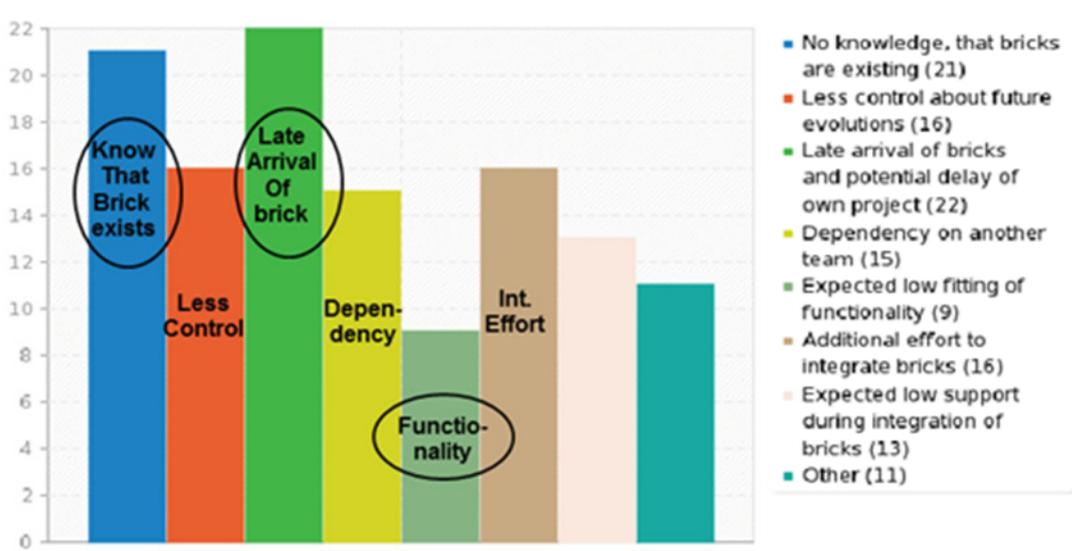


Figure 23 - Future improvements identified by the study of Schneider Electric 2012 (Brick = software unit)

The final interesting result in the survey is the topics of future improvements for software unit reuse at Schneider Electric. The study identifies this as shown in Figure 23. Most of the interviewed persons identify the late arrival and unknown storage as a relevant point. The development teams had no idea where the software units could be found and when they are released. The next point is the missing knowledge that software units still exist. Both points are

described by Qu, Ji and Nsakanda (2012). The next point is the internal effort to integrate a software unit. This includes the low support during the integration. Figure 20 shows the difference between supported and unsupported teams. An interesting point in this study is that the reuse of the domain of a software unit is not seen as critical as the aforementioned points.

The next section discusses the challenges identified in Chapter 2 based on the topic of missing SCAc knowledge. Therefore, the topic of industrial environments shown in this section is used to underline the problems.

3.2.2. Knowledge storing problem

For software engineers the problem of knowledge storing occurs if no external storing process description exists (see Boden and Avram, 2009). The storing of data and information is not a problem technical in software engineering. Today, knowledge management systems are able to store knowledge and to put information into a context and constitute knowledge (Bjørnson and Dingsøyr, 2008). This relationship and the associated information can be stored in systems. Therefore, different technologies exist. A typical technology is so-called semantic model (Seedorf, 2010). Such a model is not only able to describe the information itself, but also the meaning (semantics). This allows the linking of different semantic models representing the knowledge of different individuals or groups. An example of semantic mark-up language models are: Speech Framework (W3C, 2000), OWL2 (W3C, 2009), and Resource Description Framework (RDF) (W3C, 2004). Specialised knowledge is required to store the knowledge in such systems and their (semantic) models. Content management tools (e.g., GForgeGroup, 2012) support development processes by storing of information. However, a commonly used tool or process to force the problem of storing of SCAc related knowledge and information is not identified in the used literature.

If knowledge is not stored it can be get lost. This is called knowledge vaporisation (see Ven et al., 2006). This is an effect which occurs when people leave a company, for example. Often, these people are experienced users of technologies, relevant concepts, processes or domains.

However, their knowledge is part of their private experienced users and is mostly not documented. As a result, this knowledge is inaccessible to a company if such a person leaves (Bosch, 2004). The result of Knowledge Vaporization (KV) is an increase in cost because of adaptation of processes or people to replace the missing knowledge (see Seedorf, 2010).

To store information or knowledge, a user may access a repository (e.g., knowledge management system) and use it. Therefore, the user has to know where to find a repository, how to access it and how to use it. From the SCAC point of view, such a repository system has to store different information. The problem is that SCAC related information may consist of different technology forms (e.g., service description or binary files). This is based on the fact that SCACs are related to different software units which are based on different technologies and component worlds. Additionally, within each different software unit, technology forms (e.g., object orientation, component orientation, and service orientation), there can exist multiple sub technology forms or concepts (cf. Appendix Section D).

This is also valid for component models. Special context dependency is where a software unit belongs to a component model. Beside the exact form and the properties of the components which correspond to the model, a component model also specifies how components can communicate with each other (interaction standard) and connect to each other (composition standard). Moreover, a component model can be constructed by implementations from different manufacturers using different technologies. Similar to the multitude of object-oriented languages, a number of component models specify different approaches which may be incompatible with each other. (Szyperski, 2002b; Gruhn and Thiel, 2000)

The problem of component models is valid for component-based software construction and not found in this form in other construction forms. Though in the object-oriented software construction, a strong (economic) relation to the special paradigms is found (e.g., .NET and JavaEE). This is also found in component-based software development. Typically, component

models are related to a component world (e.g., .NET and Java). In service-based software construction there is currently no such dependency. Information from professional and/or market-political viewpoints, however, can be relevant for the software engineer (Szyperski, 2002b) and, therefore, part of the SCAC description. Services do show a kind of world perspective looking on different protocols. Commonly used protocols are, for example, SOAP and Representational State Transfer (REST) (Singh and Huhns, 2005), and both are incompatible. Next to the stored software unit information, also additional information, such as description files or additionally binary files, of an SCAC has to be stored.

Regarding the industrial environment described in Section 3.2.1.2 the problem of knowledge storing occurs especially for supporting teams. Knowledge has to be stored in a way that other teams can use it (cf. Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010).

Next to the software unit information the SCAC examples in Section 3.1 show additional information. The shown integration SCAC requires the information which software unit is needed and how each unit is inserted. Additionally, an integration SCAC includes information about settings for the IDE, the development project or an integrated software unit. The transformation and deployment SCAC shown also identifies the need of additional tools. The parameter and the relation to software unit parts has to be described. The result of transformation Software Construction Artefact (SCA) is a new software unit. which has to be described also.

As a result, of this view on the problem of knowledge storing following sub problems based on knowledge are identified

- Problem of identification of a repository
- Problem of access of a repository
- Problem of use of a repository

Regarding the use of the repository it is identified that the content (software unit and SCAC knowledge) has to be stored. Therefore, the special characteristics of the related technologies (software unit and SCAC) has to be known and how it insert in the system.

Using the identified knowledge based sub problems, an inexperienced user has a similar Software Reuse Information Demand model with multiple (i.e., three combined) solution elements as shown in Figure 16. If one of these elements is not known by the user the storing of knowledge might be incomplete or take a long time or will not happen. Figure 24 shows an possible SRID model for an inexperienced user. Here, the solution for problems of identification of a repository (red), access to a repository (yellow) and the use of a repository (blue) are shown.

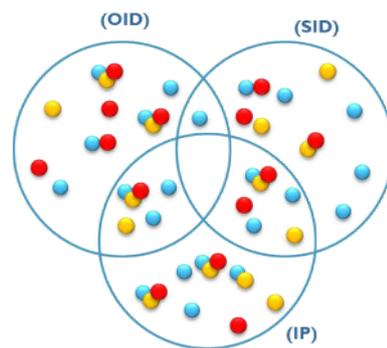


Figure 24 - Example SRID model for knowledge storing

Regarding the problem area of the distribution environment knowledge, the problem of knowledge storing can be related as follows. This problem occurs for experienced software engineers (focused on a software unit and SCAC) who wants to store and, therefore, distribute an SCAC. Figure 24 shows the problem of the inexperienced knowledge level to store this knowledge. Additionally, the knowledge to store relates to the problem about distribution knowledge and the problem of missing technology knowledge. All three problem areas are related, but the primary research focuses on the inexperienced user who wants to perform SCACs. Because storing SCAC knowledge is required for reuse, the problem of knowledge storing is discussed in this research.

3.2.3. Knowledge learning problem

Some research groups (e.g., Human Brain Project, 2011) deal with the acquisition of knowledge. Their aim is to find out how the human body absorbs knowledge and generates the corresponding semantic link. The underlying idea in this area of research is to emulate the human brain to gain positive results to improve systems of knowledge (e.g., for the medical sector; see Human Brain Project, 2011).

Knowledge management tools provide knowledge to a user. However, the content of such provisions differs. Typically, a user gets a text which contains the searched knowledge or describes it (see Horeis and Sick, 2007). It is also possible that knowledge is provided using graphic or animated processes (see Bjørnson and Dingsøyr, 2008). Users have different ways to learn knowledge.

The need to learn new knowledge is an relevant activity for software engineers (cf. Qu, Ji and Nsakanda, 2012). This is based on two facts in the area of software development: the changing tasks and fast growing nature of technologies and information (see Ajila, 2006). Software engineers have different ways to learn such new knowledge. Typically, professional or self-training sessions, magazines, or podcast support are common examples. However, the problem is the given time, interpretation, and the learning possibilities of a person (see discussion in Section 2.2.2.3).

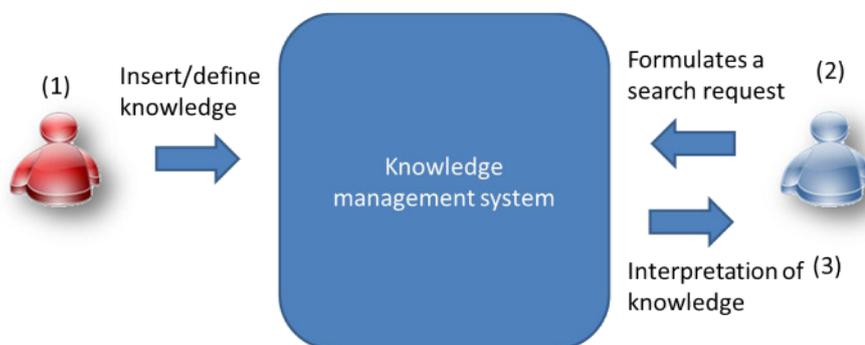


Figure 25 - Example process for knowledge interpretation

Figure 25 shows the following scenario: (1) The experienced user combines the knowledge they are aware of with the input user interface. (2) Another user formulates a search request. The system uses an algorithm to compare stored knowledge with information of the search request. (3) This user interprets the provided knowledge.

The problem of the process shown in Figure 25 is that the experienced user entering the knowledge cannot be sure the system interprets their perspective correctly. Also, the experienced user cannot be sure that the inexperienced user understands the knowledge in the same way (see Bjørnson and Dingsøy, 2008; Ajila and Zheng, 2004) or is able to formulate a search request correctly (Picot, 2003).

From the SCAC point of view, the knowledge to learn depends on the SCAC and the technology of the related software units. In general, a user has to learn how to prepare and execute an SCAC. Also, it can be necessary for a user to know how to handle the SCAC result. Especially in the creation or execution of an SCAC the problem of different technologies and component models may exist. A user has to learn different SCACs for different software unit technologies. Using the SCAC examples in Section 3.1 this is also valid for additional tools used by different SCAC. A transformation SCAC for example uses transformation tools. These tools have to be installed and configured for the SCAC. This is also valid for deployment activities. In the example of an integration SCAC a user has to learn about different IDE tools and its technologies.

The industrial environment of software engineers shows the problem of missing time and support. Reuse needs time and has to be planned (see Ajila, 2006; Frakes and Isoda, 1992). The discussion about industrial environment shows that if a team is not supported by experienced software engineers the investment in resources (e.g., time) increases. (cf. Section 3.2.1.2).

The reuse of 'unknown' software units may speed up with the performing or learning support of experienced users. If such experienced users are not available, the software engineer or a development team is under constraint to investigate the possibilities and limitations of a

software unit by themselves (see study of Schneider Electric Section 3.2.1.2). To share knowledge is seen as a relevant success factor (cf. Qu, Ji and Nsakanda, 2012).

Additionally, the learning curve is not shared. If one team discovers a way to simplify a reuse step, this knowledge may be not shared with other teams. In this team, profitability may be reached earlier. Therefore, shared knowledge between teams that could enable all teams to avoid the repetition of mistakes and generally accelerate the learning processes of all teams is missing; each team has an individual ‘learning curve’ (see study of Schneider Electric Section 3.2.1.2; Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010).

These team-related problems become more pronounced as the number of developed reusable software units increases. The reuse teams are different for each product line, therefore; sharing the learning process with other (horizontal or vertical) teams is difficult. Often, the learning curve of one reusable software unit which was reused by the same team can be re-applied to later units with additional work (see STwS example in the study of Schneider Electric Section 3.2.1.2).

During the research no special solution or approach was identified supporting software engineers by learning SCAC related knowledge. To summarise the discussion about the problem of learning following knowledge based problems are identified:

- Problem of different technologies and component worlds
- Problem multiple variations of SCAC related actions
- Problem multiple existing tools used in SCACs
- Problem of knowledge interpretation
- Problem of variant results

Using the SRID model to demonstrate the information demand an picture is created which differs to the example of Figure 16. While the SID and the OID can contain multiple interpretations of the SCAC knowledge or information the IP area only includes one set of related solution elements. In the real world this set is represented by the knowledge that is used

for learning and the interpretation of this knowledge by the experienced user who creates it. If the inexperienced user does not have the same interpretation the use of the learned knowledge might create a not valid SCAC from the perspective of experienced user. A user might use multiple knowledge resource to learn an SCAC. This can change the number of elements in OID and especially in the SID, but not the number of elements in the IP area.

Regarding the three problem areas focused by the research. The problem of knowledge learning can be related to the research as follows. The knowledge required by the technologies (i.e., software unit and SCAC technology) and the distribution environment has to be learned by the inexperienced user.

3.2.4. Problem of searching and receiving of knowledge

Usually, a user can search for knowledge by using a specialised search engine of such systems. Simple knowledge management systems provide knowledge as textual information. Another feature of these systems is the conclusion from existing knowledge to new data-driven knowledge. This can be reached, for example, by using a case-based reasoning approach (Allen, 1994). Usually, semantic models are used in such cases. For users, it is difficult to use these tools because of the diverse knowledge requirements (Seedorf, 2010; Picot, 2003).

For knowledge to be received, first it has to be searched for. The problem is based on the fact that a search request (description) of the searched object has to be created by a user (see SRID model discussion in Section 3.2.1.1). Due to differing search algorithms and search technologies, a user might not be familiar with the use of search technology (Garcia et al., 2006). Also, describing the information appropriately in a search query may represent a problem (see Picot, 2003; Garcia et al., 2006).

From the SCAC perspective, a user might describe the SCAC input, result, or its behaviour. As discussed before, SCACs differs in their used technology or component model. A user might have problems to use this information for a search query. The SCAC examples in Section 3.1 show three different types of SCAC with different behaviours. Also, the results (integrated

software unit, a new software unit, and a deployed software unit) differ. The inputs are similar but differ in their use. In a transformation SCAC, for example, the input information is used to setup a transformation tool. Such actions of an SCAC should to be searchable.

In addition to the experience of SCAC related information, a software engineer has to know how to reach or access such sources of information (see discussion of knowledge storing in Section 3.2.2) Each repository system is in place to advance different approaches since, for example, it may be necessary to authenticate in some repository systems (see Ajila, 2006). A user requires knowledge about an authentication system (e.g., user name and password).

Some systems offer standardised approaches such as web portals, while others use advanced specialised applications, in addition, different types or use.

Usually, software engineers are familiar with their own special in-house or free open source repository where they are able to search for information. The number of internal corporate repositories increases with the size of the company. A software engineer is not aware of all existing repositories in their environment (i.e., in a global company). This is particularly true for private repository of other software engineers.

For software engineers, the problem arises in the functionality of finding information using the request results. Search engines such as Google allow to search in many different systems for information. Search results of general search engines such as these provide a variety of results that do not match the desired result also.

From the SCAC point of view the problem of receiving SCAC knowledge includes another problem. The requested knowledge has to be complete. As logical result the input, configuration, needed tools and the output has to be described completely to repeat the SCAC. An example for this is the knowledge base CodeProject (see Maunder, 2012). Here, software engineers provide different software unit and description how to use the provided functionality of these units (domain context). But the SCAC related information is missing in most cases. As a result, an inexperienced user does not know, for example, how to integrate this unit.

The industrial environment of software engineers shows the problem of localisation for software engineers (see Bosch and Bosch-Sijtsema, 2010). One team member can be located on a different site than others of the same team. To exchange data is not only a problem of different time zone or culture but also a question of communication (see Taweel et al., 2009). The problem of localisation also occurs for a multiple teams of software engineers (see Taweel et al., 2009; Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010). Different teams may be placed in different locations. Teams, as well as, single software engineers, have to communicate with each other. To search and receive knowledge (e.g., formulation of a search request and the download of software units) about a repository in different locations is required. Additionally, software engineers use different types of repositories. These types of repositories reach from handmade notes or files on the personal hard disc to a team, department, companywide or community repository system (see Ajila, 2006; Ha, Sun, and Xie, 2012; Qu, Ji and Nsakanda, 2012). As discussed before, to know how to connect to these repositories is a problem. A software engineer (or a team) has to know where these repositories may be found and how to use them (formulate search requests).

The discussion about different views on the problem of searching and receiving knowledge can be summarised as follows. Search and receipt requires knowledge about the searching and receiving infrastructure. Additionally, a software engineer has to formulate a search request which requires knowledge about software units and SCACs (e.g., domain or technical knowledge). A search result has to be learned and interpreted (cf. Problem of learning, Section 3.2.3). As a precondition, knowledge has to be stored beforehand (cf. Section 3.1.4).

From the SRID model view, this problem has to be demonstrated by using multiple SRID instances. The information demand of each single knowledge problem (i.e., find, access, and use of a repository) of repository location can be demonstrated by using the SRID model (see Figure 26).

Additionally, the formulation of a search request can be presented by aggregating the problems of technology software units and SCACs (see Figure 27). Picot (2003) identifies that inexperienced user are not able to formulate a correct search request this limits the useful information. This limitation is shown in Figure 27 by a black circle representing a search request of an user.

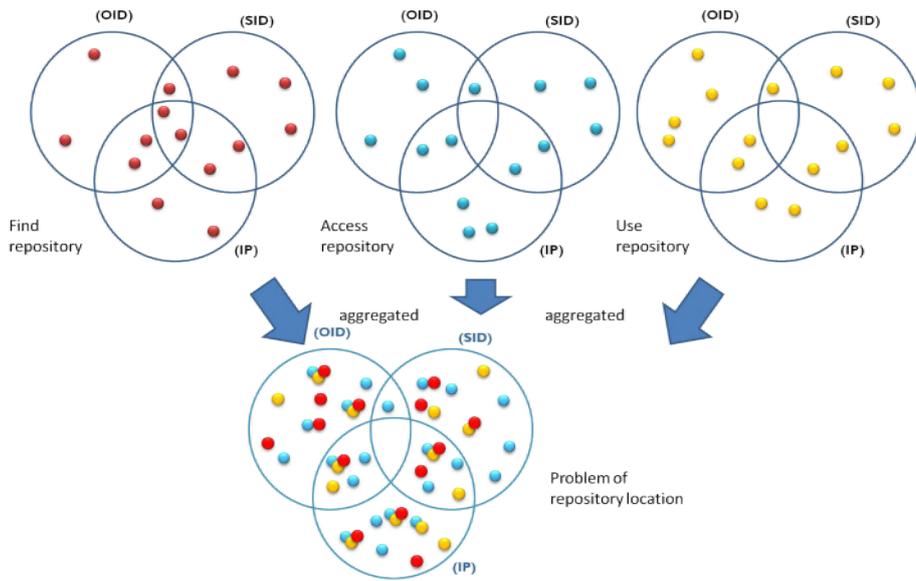


Figure 26 - Creation of the problem of repository localisation

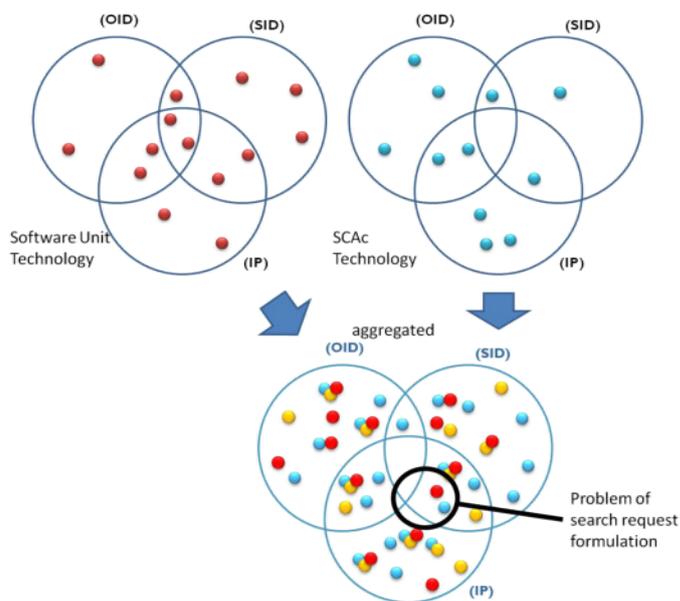


Figure 27 - SRID model for the problem of search request formulation based on Picot (2003, p. 106)

In the next step the final SRID model can be summarised by aggregating the SRID model for the problem of localisation, formulation a search request, and interpretation. Figure 28 shows the aggregation problem of searching and receiving. This problem is based on knowledge of repository location, search request formulation, and knowledge interpretation.

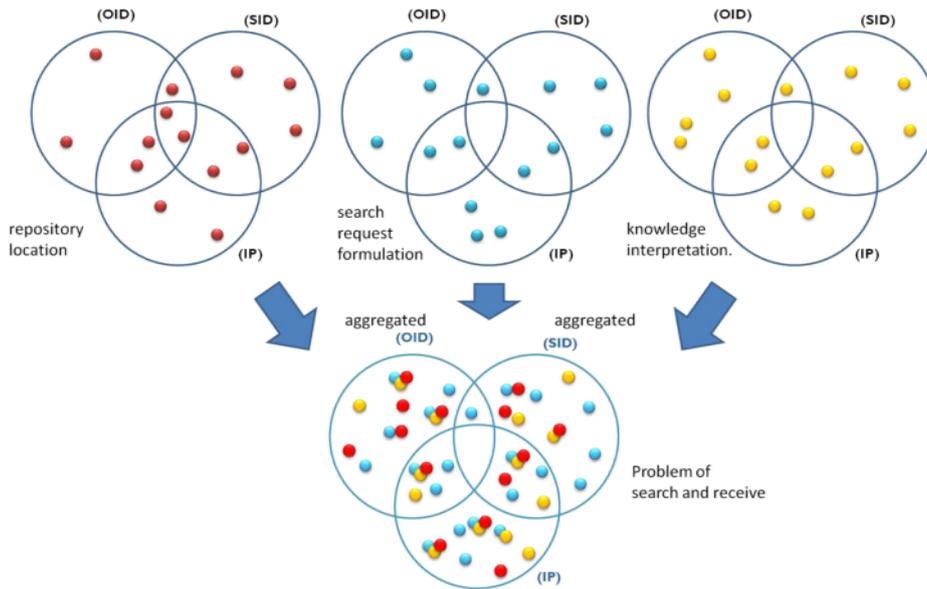


Figure 28 - SRID model for information demand for search and receipt of knowledge

The SRID model in Figure 28 demonstrates that the problem of searching and receiving knowledge is related to the problem area of the inexperienced knowledge level. Also, the remaining two problem areas are related to this topic. Especially the distribution environment which requires knowledge of an inexperienced user for searching and receiving SCAC related knowledge. The received knowledge about software units and SCACs has to be interpreted.

3.2.5. Knowledge exchange problem

In different projects software engineers have to share their knowledge. Typically, this can be done by arranging meetings supported by different presentation media (i.e., audio, video, or pictures) or by using knowledge management tools. Next to the discussed problems of searching and using of knowledge the problem arises to distribute knowledge in a way that it can be understood correctly by others. In contrast to the knowledge user, the software engineer

who is the knowledge creator has to look for the distribution possibilities (see Taweel et al., 2009; Boden and Avram, 2009).

Knowledge management systems can store information as knowledge and usually use concepts like location technology-independent, as for example, web pages or services (see Huang et al., 2005). As a result, the user no longer has problems accessing such systems because on technology dependency (e.g., missing runtime for Java-based tools). A web page can be used independently of special runtimes. Using a web technology makes it possible to build knowledge systems that can be accessed and used from different locations (i.e., by using a network connection). However, the problem is more focused on entering the knowledge into a system and enabling others to find and use it (see discussion in Section 3.2.2). Considering this, three points are pertinent: insert knowledge in a system, request knowledge from a system, and receive knowledge (see Figure 25).

The problem of knowledge exchange bases on the problem of storing, learning, searching, and receiving (cf. Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010). Software engineers have to store knowledge and other has to find and receipt it. As a result, the problem of knowledge exchange includes the same problems for software engineers. Additionally, in the multiple team scenarios, software units are created by a singular team, but multiple reuse teams then use them. In the study of Schneider Electric shown in Section 3.2.1.2, one team would create a software unit and related SCACs and teams from different industry areas would reuse the software unit. Qu, Ji and Nsakanda (2012) and Choi, Lee and Yoo (2010) also identify this problem, but focus on knowledge in general instead of SCAC or software unit knowledge.

On the side of the reuse teams, two different effects of the multiple team scenarios are identifiable. When multiple teams reuse software units, the effort required in all these teams is nearly the same. Also, the learning curve may be nearly identical. If a single dedicated ‘reuse’ team exists, the effort decreases with each reuse due to the learning process. This decrease is not linear and stops at a certain minimum (see STwS scenario discussed in Section 3.2.1.2).

Even if this minimum seems to be a positive effect, the constraints of organisations (i.e., missing management of reuse) prohibit this effect in some cases (see discussion about organisational problems of software reuse Section 2.2.2.3). This may lead to the following scenarios: Single Team without Support (STwoS) and Multiple Teams without Support (MTwoS; cf. Figure 20c). Here, no knowledge exchange to the supporting team exists. The study of Schneider Electric shows that such a scenario a significantly higher effort for reusing software units is required, since there is no support. This also leads to a decrease in profitability of the reuse approach compared to developing from scratch. While scenarios with support reach profitability with 3 or 4 reuses, the last scenario only starts to be profitable after five reuses (cf. study of Schneider Electric in Section 3.2.1.2). Additionally, the learning curve is not shared between teams (see problem discussion in Section 2.2.2.3).

These team-related problems become more pronounced as the number of developed reusable software units increases. (see problem discussion in Section 2.2.3.3).

Another typical problem is that teams or team members may be situated in different locations and have to cooperate over a distance (Qu, Ji and Nsakanda, 2012). Distributed software development scenarios cause problems in software architecture, engineering processes, and R&D organisation (Bosch and Bosch-Sijtsema, 2010). Also, the sharing of reusable software units between teams has a great impact on costs:

“A problem observed [...] is that when decoupling between shared software assets is insufficiently achieved is excessive coordination cost between teams. One might expect that alignment is needed at the road mapping level and to a certain extent at the planning level. When teams need to closely cooperate during iteration planning and have a need to exchange intermediate developer releases between teams during iterations in order to guarantee interoperability, the coordination cost of shared asset teams is starting to significantly affect efficiency.” (Bosch and Bosch-Sijtsema, 2010, p. 70)

The problem of missing knowledge exchange is identified by the analysis of other real development projects (see Boden and Avram, 2009). Software engineers may not be able to work with other solutions than the solutions they already know. In a worst case scenario, people are not able to fulfil their work or cooperate with teams using different versions of the same knowledge (based on interpretation issues; see Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010).

To summarise the discussion of the problem of knowledge exchange, the following statements can be made. Knowledge exchange consists of knowledge storing by an experienced software engineer, searching and receiving by an inexperienced software engineer, and the interpretation of knowledge by these engineers. Therefore, the problem of knowledge exchange includes the same sub problems (i.e., location, different technologies, learning, interpretation of knowledge, and the creation of search requests) as the problem areas it consists of. As a result, this problem is related to the three focused problem areas. Because of this problem is related to the other mentioned problems the SRID models explained in Sections 3.2.2, 3.2.3, and 3.2.4 explain the sub these as sub problems.

3.2.6. The problem of knowledge execution

To avoid misunderstanding to the term application, which means software program in the scope of the thesis, the term knowledge execution is used instead of knowledge application.

The difficulty is due to the fact that the definitions of knowledge and information are different (see knowledge term definition in Section 2.2.2.5).

However, the difficulty is due to the fact that knowledge can be applied in different ways; based on experiences of a person. (cf. Qu, Ji and Nsakanda, 2012). There is no rule or process which defines how knowledge can be applied. Usually, knowledge has to be described in a form whereby other users can use it (cf. discussion of Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010).

Humans use knowledge to relate information ('interacting'; see definition of knowledge in Section 2.2.2.5). This is a result of one or more learning processes. If knowledge is not adequate, humans may be able to work around this issue (Human Brain Project, 2011). To create such behaviour at system level is challenging. The problem is to create a system which is able to reuse the given knowledge to reach the same aim or intention as the user who creates this knowledge (cf. Qu, Ji and Nsakanda, 2012).

To execute SCAC related knowledge (i.e., information about software units, tools, and their usage), it is necessary to handle different problems. The first one is the availability which is of crucial importance for software engineers. To guarantee the operability, (e.g., the frictionless execution and operation of software), all units of software have to be available. If a unit is not available, an application has to be able to react to it. With the handling of objects, in most cases local resources which can be verified, are meant. Though with the handling of components, local resources also exist in most cases, during the construction time, however, only the interfaces are handled. The existence of the resource is not always mandatory. This behaviour is even more pronounced with service-based construction (Breivold and Larsson, 2007) and leads to the problem of the availability at runtime (Kumar et al., 2007). The knowledge how to use information about software unit in an IDE (integration SCAC) or configure special tool (transformation and deployment SCAC) is necessary for an SCAC (cf. SCAC examples Section 3.1). This kind of information have to be available for a user or a system for knowledge execution.

Additionally, the SCAC knowledge and information has to be complete in order to perform the SCAC. Otherwise (i.e., knowledge is missing) problems can be created in a project (cf. Qu, Ji and Nsakanda, 2012). In the area of components, for example, the software engineer uses interfaces to execute functions of a service or component. But engineers have to know the structure of a component (including external dependencies; Sommerville, 2011). On the service side the internal structure of a service is not so relevant (Breivold and Larsson, 2007). Since the

implementation of services is encased, only a low dependency exists (cf. Breivold and Larsson, 2007). Often, services provide all information (e.g., in an interface description). This description can describe dependencies which a client can try to find or to create an adequate alternative. In the area of classes (i.e., source code), the software engineer can influence the context dependency to a certain degree. This is done by adding, removing or changing the source-code. So, an engineer can manage the use of external dependencies or reduce such dependencies by independently writing missing functionality. Since these dependencies are necessary at different levels of the development, the software engineer has to know them intimately. In the area of components context dependencies (e.g., relation to other components or the runtime environment) are usual. The integration and transformation SCAc examples shown in Section 3.1 show this problem. For the transformation of each Java library the references to previously transformed software units has to be used. In the integration example the mail library has a lot of dependencies to other transformed libraries as well as to a special IKVM library. The internal study of Schneider Electric also shows that the topic of dependencies has to be improved in the future (see Section 3.2.1.2)

Following list summarises the knowledge based sub problem identified in this discussion:

- Problem of completeness (including dependencies)
- Problem of availability
- Problem of interpretation and realisation of existing knowledge

Using the SRID model for the problems based on these three sub problems and, therefore, similar to the SRID model shown in Figure 26.

To summarise the proof knowledge execution, it can be stated that this is an relevant problem identified by the literature (cf. discussion of Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010). The execution of knowledge depends on knowledge about the technology of a software unit and the technical environment for execution. Therefore, it can be concluded that this problem also depends on the other discussed problems of knowledge.

3.2.7. Problem significance

To highlight the problem significance, the following two questions will be discussed:

- How often is reuse used in industry?
- What happens if problems occur?

The questions are answered using a different discussion to Chapter 2. For the first question ‘How often is reuse used in industry?’ no investigation was identified defining a percentage number. But different viewpoints indicate a high usage of software unit reuse in industry projects. Following the discussion of Ajila (2006), the reuse of a software unit is a standard process used in software development projects. This statement is supported by other reuse discussions of Morisio, Ezran and Tully (2002) and Ha, Sun and Xie (2012). Thereby, reuse can appear, for example (see McCarey, Ó Cinnéide and Kushmerick, 2008), as a main focus (development-with-reuse) or inside a development process (reuse-within-development: see Chapter 2 Section 2.2.1.4).

Another indicator is the development environments used today. Often IDEs (SREs) with high reuse support are used (see discussion of Garcia *et al.*, 2006). Typical examples are Visual Studio and Eclipse and these support developers to perform software unit reuse, sometimes this occurs automatically, for example the automated creation of a web service client out of a web service description file.

The IDEs mentioned are related to another indicator. Object-orientation, component-orientation and service-orientation are relevant models for software development. As shown in Chapter 2 Section 2.2.1.6 these are reuse technologies (or concepts). As a logical result, these concepts are often used. Another related indicator is the reuse landscape (see Figure 3 and Table 2). The reuse topics represent relevant fields in software engineering and most of them use the mentioned reuse models (or concepts; see Table 2)

However, software development can be done without any software unit reuse. As a personal opinion this particularly occurs in the area of new technology research, and where software

developers have no knowledge of reuse in their specific field. Sometimes reuse may be avoided based on the risk of failure. In such situations, software development has to be done without reuse support.

3.3. Missing solution approaches

In Chapter 2 four problems are identified from the literature. The discussion in Chapter 2 concludes that these problems are challenges for the exchange and execution of SCAC related knowledge. This chapter discusses the problem areas in more detail. In the literature solution approaches focusing on the three knowledge problems related to reuse activities could not be identified.

However, approaches with a specialisation focusing on one problem area, such as technology, and one single SCAC can be found. In the following, some examples are discussed briefly.

McCarey, Ó Cinnéide and Kushmerick (2008) focus on supporting software engineers by using agents. In this approach, agents perform, amongst other activities:

learning from a human user and

- sharing knowledge between software engineers.

In this approach an agent is used to study and analyse the activities of a software engineer in an Eclipse environment focusing on Java source code creation. All relevant information about the reuse of a software unit (i.e., source code) is stored in an information retrieval model. Together with a repository this system is able to store knowledge about the use of a software unit. The system can analyse this knowledge and provide this to other users. The solution described by McCarey, Ó Cinnéide and Kushmerick (2008) is useful from the perspective of the author of this thesis and can be seen as a ‘learning by doing’ approach. The advantage of this type of approach is that the knowledge level of a software engineer is not changed. Regarding the focused problem areas some disadvantages of such a type of approach can be identified. The first one is the focus on one technology and one type of software unit. The example of McCarey, Ó Cinnéide and Kushmerick (2008) focuses on a Java source code in an Eclipse IDE, and the

research does not focus and does not show the possibilities of focusing on other software unit technologies. Ye and Fischer (2005) follow a similar approach to McCarey, Ó Cinnéide and Kushmerick (2008). Both approaches focus on the integration of a single software unit type (i.e., source code) with a special technology type (i.e., Java in the case of McCarey, Ó Cinnéide and Kushmerick, 2008). The problem area of different software unit SCAC knowledge is not covered by this approach.

Regarding the transformation and deployment, a similar picture is identified. In transformation, for example, different specialised solutions can be found. For Java and .NET, for example, compiler and interpreter exists which transform source-codes into component byte codes. Tools like SVCUtil (cf. Microsoft, 2012d) transform web service information into source codes or components. Even if such tools provide the possibilities to support two different software unit types, they focus on a special platform (in this case .NET). The same can be found with Java side using WsGen (Sun, 2013). Transformation tools like IKVM (Frijters, 2011) are able to transform one technology into another. In this case Java components are transformed into .NET components. The problem area of different software units or SCAC technologies is not covered by such tools. Even if these transformation tools avoid the manual performance handling of the transformation, these tools require ‘handmade’ knowledge (e.g., configuration parameters). The required knowledge depends on the different tools. For the problem area of distribution knowledge no approach was identified.

Next to the idea of using similar firmware (cf. discussion of device deployment of Zinn, Fischer-Hellmann and Schoop, 2012a) one example was identified that focuses on the problem of different technologies. Some vendors provide tools to supports a user in creating a configuration for different devices (e.g. Altera, 2012).

Device and firmware vendors provide their own application for device deployment (cf. Zinn, Fischer-Hellmann and Schoop, 2012a). Even if these different tools avoid the manual handling of device deployment knowledge, these tools still require knowledge which has to be handled

by users. For the problem area of distribution environment knowledge and the problem of a user's knowledge level, no approach was identified.

A solution or discussion about the definition of reuse activity knowledge focusing on one or more of the three focused SCACs was not found.

The research presented in this thesis focuses on the support of inexperienced software engineers to perform a SCAC. Thereby, the research focuses on an approach handling the three focused SCACs. While a similar approach (i.e., focusing on the three problem areas) was not found during the research phase, such an approach may be a contribution to the topic of SCACs.

3.4. Summary

This chapter analyses the problems of missing software construction activity knowledge. Therefore, it starts with an overview of the three focused software construction activity types chosen for this thesis: integration, transformation, and deployment software construction activities. For each activity type, three examples are given in the beginning of the chapter. After this discussion the industrial environment as typical environment of software engineers is explained. Additionally, the Software Reuse Information Demand Model is presented. This model is a research result of the Ph.D. study and is used to visualise the knowledge level of a person.

In the second part of this chapter five knowledge problems are discussed: knowledge storing, knowledge learning, search and receiving of knowledge, knowledge exchange, and knowledge execution. These knowledge problems are discussed as problems for three of the focused problem areas. Additionally, the SRID model and the industrial environment are used to explain negative effects of the knowledge problems to the area of software unit reuse and the use of SCACs. The result of the discussion is that the three problem areas are challenges to create an approach for the use of SCACs which performs single software unit reuse activities.

The analysis shows that the focused problems are based on different knowledge problems. The last section of this chapter discusses examples of existing approaches and concludes that these

Missing software construction activity knowledge – problem analysis

are not adequate to solve all problem areas at the same time. The next chapter defines a basic idea and a concrete concept of a solution focusing the knowledge level of the discussed problems of this chapter.

4. A general approach to realise knowledge-based automated reuse activities

The previous chapter discusses the challenges for creating an adequate technique to handle software construction activity knowledge. This chapter provides a solution to enable inexperienced software engineers to perform software construction activities and it also explains the concept which focuses on these challenges. Therefore, this chapter begins with an explanation of the basic idea and the solution approaches focusing on the different problems identified in the literature review. The concrete concept is explained in the second section of the chapter.

4.1. The basic idea

The basic idea is to enable inexperienced software engineers to perform software reuse activities on special software units that require experienced user knowledge. This support should be done automatically or partially automatically without a long learning process for software engineers. As concluded in the study of Qu, Ji and Nsakanda (2012) the use of IT is relevant for the exchange and execution of knowledge. Therefore, the basic idea is based on a technical infrastructure. The development of software can be seen as a set of sub-steps, for example, the reuse of a software unit, integration of a unit, or deployment. The basic idea focuses on handling such steps, and sketches a solution. This does not mean that a complete software unit development process or a complete reuse process of a software unit is focused by this idea. It means only the support of single and specialised sub-steps (i.e., software construction activities) is focused upon. Figure 29 shows an example process and the focused sub-step support by the basic idea of the focused approach. The focused approach is useable in the implementation phase of a development process. Here, different activities (e.g., integration, transformation or deployment) are performed. The figure uses an example login software unit. The download, transformation and integration of software units are focused steps in this research. This figure is only an example. One characteristic of the basic idea is that the focused

A general approach to realise knowledge-based automated reuse activities

approach is not a development process and does not depend on such a process. From the viewpoint of the author, software construction activities are typical reuse activities which occur with and without specific reuse development processes (cf. Perspectives on reuse in Section 2.2.1.4).

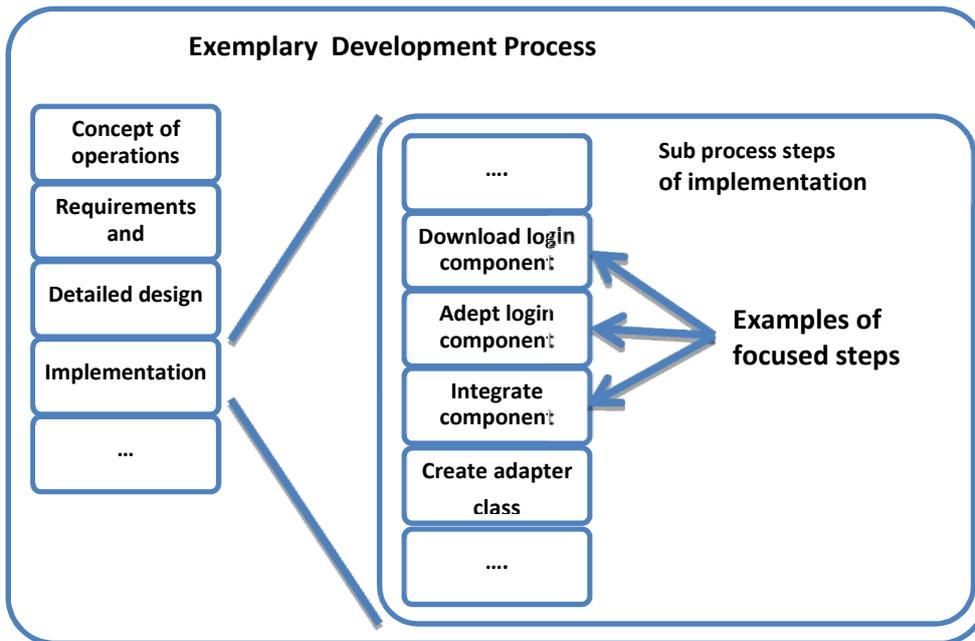


Figure 29 - Example of focused reuse steps

From a more technical perspective, the idea focuses a solution to combine a knowledge database with service provision. Regarding the discussion about inexperienced software engineers it became clear that, in most cases, knowledge about software unit reuse activities had to be learned from scratch (cf. Section 3.2.3). Sometimes the knowledge learned may never be reused by individuals. So the idea is to create a service which provides the capability of:

- searching for reusable software unit activities,
- automating single reuse steps,
- storing reuse activity information permanently,
- being accessible independently of location or technology platforms, and
- extending the knowledge database with new reuse activity information.

A general approach to realise knowledge-based automated reuse activities

Figure 30 shows this idea.

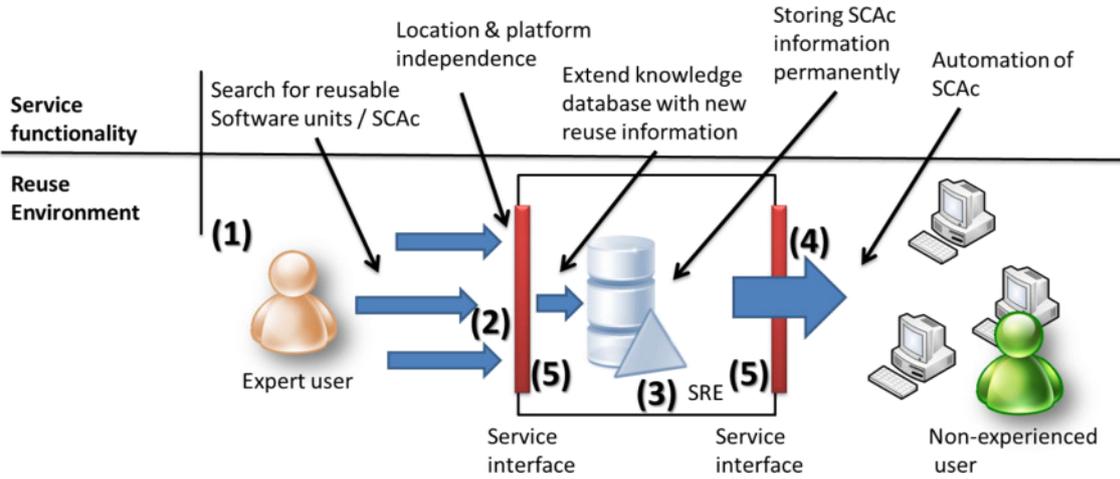


Figure 30 - Basic Idea of this thesis

(1) It is relevant for the idea to have an experienced user (2) storing knowledge about software reuse activities of a specific software unit (3) in an environment. (4) This stored knowledge may be used by a person who is not an experienced user in this particular software unit and/or the stored reuse activity. (5) This is done by using a service.

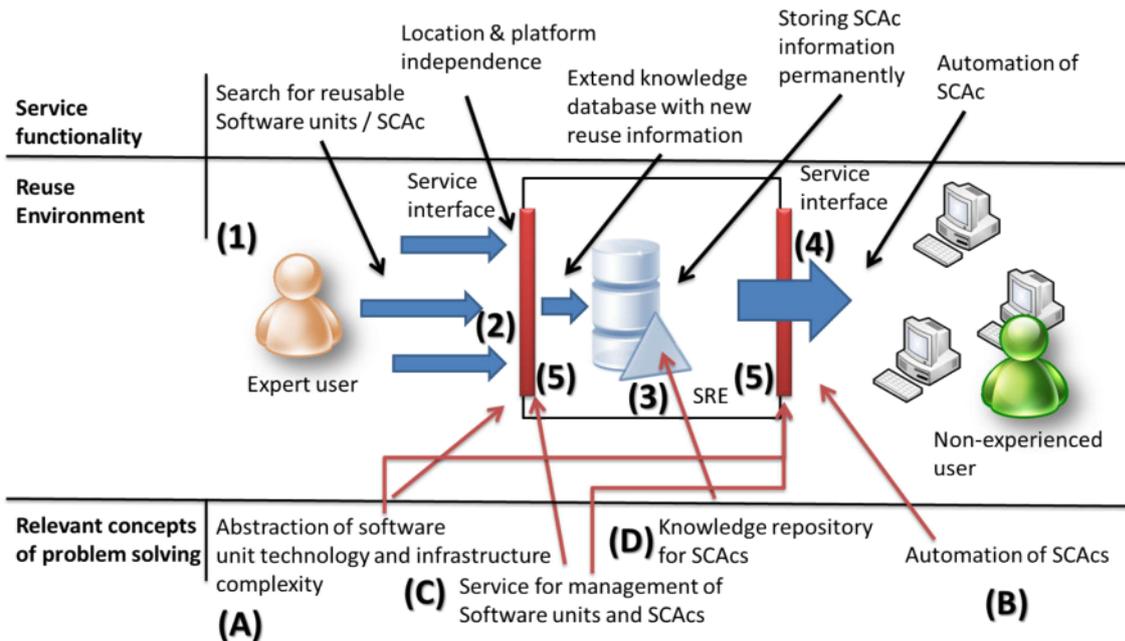


Figure 31 - Concepts used for problem solving

A general approach to realise knowledge-based automated reuse activities

The focus here is on the different challenges discussed in Section 2.2.3.3 by the combination of four concepts:

- (A) Abstraction of software unit complexity
- (B) Automation of reuse activities / Abstraction of distribution environment
- (C) Service for management of software units and reuse activities
- (D) Knowledge repository for reuse activities

Figure 31 shows where the different concepts can be found in the idea.

The approach uses abstraction to handle software unit complexity (i.e., changing technologies) to support a user in searching, handling, and reusing a software unit. A user can search for a software unit and an SCAC. The idea is based on the concept of abstraction which is widely used in software development (for example, see the use of abstract data types in Ludewig and Lichter, 2010 or abstract specification in Sommerville, 2011).

The idea to use automation for SCAC is based on the example of industrial manufacturing. For example, in the automobile industry, cars are built using tools and computer-aided design (CAD) models (cf. Zheng, 2007). Intellectual properties (knowledge parameters) are part of such models. As a result, different variations of cars can be built using a single model. Manufacturing machines can be programmed using the model and most of the construction process steps can be performed automatically. Likewise, it may be possible to automate special reuse activities (i.e., software construction activities) to support software engineers.

Within this idea, a service is used for handling (storing, removing, editing, and executing) of software construction activities. Therefore, the concept of service-oriented architectures (Singh and Huhns, 2005) is used. This allows a distributed environment. This enables people to access or use these functionalities independently of their location. In terms of automated software reuse activities; this feature (i.e., service-oriented architecture) reduces complexity. This means that the user does not have to know how to create a setup or how to perform a reuse activity. Also a user does not have to think about locations and environments.

The concept of using a repository is based on the fact that repositories are already widely used in the area of software engineering to store data and provide management functionality (see the discussion of Morisio, Ezran and Tully, 2002). The basic idea focuses on two different types of repositories. The first stores information about software units similar to a Content Management System (CMS). Here, software unit data and information can be stored using an abstraction model. This model simplifies the view on software units. The second repository type stores information and knowledge about software reuse activities which will be relayed to the software unit information. The second repository behaves more as a Knowledge Management System.

The four focused concepts are not new in the scope of software engineering or industrial behaviour. The literature review in Chapter 2 did not identify approaches using these concepts together to focus on the automation of software unit reuse activities for inexperienced software engineers. Therefore, the idea and the related concept (see Section 4.4 and the realised solution in Chapter 5) have to be proofed to be a valid technique to the identified lack.

The expected mode of operation is now summarised and related to the problems discussed in Section 2.2.3.3. The problem of the knowledge required based on variants of different technologies is handled for the inexperienced software engineer by storing the information about software units into a common model which does not focus on the variants (i.e., abstraction). Additionally, by use of the described service, the inexperienced software engineer does not need to handle the differences in the technologies. The service is also relevant for the problem of different knowledge levels of software engineers. The service will deliver the same information, use the infrastructure to perform the SCAC in the same way, and deliver the same results. This is independent of the knowledge level of a user. It is expected that the behaviour of the service also has a positive effect on the problem of the distribution of knowledge. The service holds the infrastructure (e.g., repositories and SCAC execution tools). As described, the user does not need to know this infrastructure and how the information is distributed. The

problem of a missing definition of software reuse knowledge should be handled by the model for software construction activity. Here, information is described which is used as knowledge by the service.

4.2. Focused user profiles and scenarios

The previous section illustrates that the focused approach is intended to support software engineers with insufficient knowledge in performing the specific reuse activities in software development projects. To define the corresponding user profiles, it is first necessary to classify the corresponding application scenarios.

The approach focuses on traditional reusable software units (classes, components, and services) in the field of object-oriented, component-based, and service-based development (see Appendix Section D.1). This means using these software units as artefacts in a black-box reuse project. These units represent a solution, either individually or in context from a domain specific view. A solution from the perspective of the focused approach is, therefore, the provision of the software unit information which may consist of classes, components and services, as well as any additional information, such as configuration and dependencies. This includes SCAC information. From a technical point of view, this solves at least the problem for the user. In addition to the provision of software units, the extra information collected can be deposited. Reuse activities are represented in this thesis by focusing on transformation, integration, and the deployment construction activities of software units. In detail of this, this thesis means the adaptation, integration into IDEs as well as the deployment of software units into embedded devices.

The following application scenarios are focused:

- 1) Software engineers want to use these software units to fulfil a requirement that they are developing or fixing a problem that is encountered in the development of a software

unit. Therefore, the engineer has to reuse the activities (i.e., reuse of procedures cf. Section 2.2.1.4).

- 2) Software engineers want to provide software units and activity information/knowledge for general reuse.

Generally, these two application scenarios identify two different user profiles: (1) ‘knowledge user’, and (2) ‘knowledge creator’. These profiles will be discussed in the following sections.

4.2.1. Knowledge user (KU) – Reuse of software units

The following two hypothetical examples illustrate the focused application scenario for the use of software unit activity knowledge:

Example 1 "Service Discovery": In automation industry, the use of web services has become standard (Jammes, Mensch and Smit, 2007). Devices with limited resources (so-called embedded devices) are equipped with Web Service interfaces. In this example, a software engineer is given the task of creating web services on such a device ‘discoverable’, (i.e., a device which is connected to a network can be found automatically by an application or other device and used). In the field of automation, the Device Profile for Web Service standard is used for this purpose (Jammes, Mensch and Smit, 2007). This profile defines a protocol extension for SOAP-based web services that enables web services to provide discovery functions in their web operations (which may differ from device to device) to be used dynamically. The SOA4D Group (see Jammes, Mensch and Smit, 2007) provides components for devices based on conventional technologies (e.g., C++ and Java). Software engineers can download them and use them in their software development projects to fulfil requirements. However, an extensive setup of the used development environment is needed. One example is the DPWS units which have dependencies that need to be configured (cf. DPWS example in Section 3.1.4).

Example 2 "Corporate Identity": A software engineer has to integrate the company's usual AboutBox into their latest program based on C#. However, this results in the following problem

A general approach to realise knowledge-based automated reuse activities

for the software engineer. This AboutBox was developed using the Java technology that is incompatible with its .NET based program. The engineer now has several options for solving the problem (e.g., develop their own AboutBox or build an adapter).

Another possibility is using an existing component (e.g., Java-.Net Bridge) which solves the problem for software engineering, or performing the conversion of AboutBox by using a transformation application (e.g., IKVM) into a C# AboutBox. The latter provides a problem for the software engineer; because IKVM applications have to be parameterised extensively. As in the first example it is necessary to prepare a complex configuration.

In both examples, an existing software unit (i.e., DPWS unit and the AboutBox) may be used to solve the problems. The units themselves are smaller parts of software and focus on small functionality, but require the use of additional knowledge (e.g., for configuration). The first example is the configuration of the dependencies in the development environment. In the second example, complex transformation parameters are needed for additional applications (cf. SCAc examples in Section 3.1).

Also, specialised knowledge is necessary in both cases. For people who have this knowledge, there is no direct need for an environment such as the focused approach that supports the use of this knowledge. An exception may be the time saved by using an automated reuse activity. For people who do not know this, the focused approach might be an added value from the perspective of the author. An interesting aspect in the use of the focused approach is the fact that users of this platform are not under obligation to acquire the knowledge.

Thus, the focused approach is aimed at a user profile that has the following properties:

1. The people do not have the necessary knowledge to reuse a certain software unit.
2. The people, however, are under constraint to use such an approach generally.
3. The people are not interested in acquiring the knowledge for independent use at a later time.

Such a user profile is related in this thesis to following groups:

1. ‘Young Professional’: Young professionals with software development backgrounds, who do not yet have enough knowledge (Garcia et al., 2006).
2. ‘End User’: Software engineers who do not have a software development background yet develop personal or professional software. Typically, these people develop web pages, macros or small applications (Ko et al., 2009).
3. ‘Senior software engineer’: Experienced software engineers who have no knowledge of the specific software unit and its activities.

The research of this thesis focuses on young professionals and senior software engineers. The user profile is called the Knowledge User (KU) profile.

4.2.2. Knowledge creator (KC) – Provision of software units and reuse activities knowledge

The following hypothetical examples are intended to clarify the focused application scenario in the provision of software units and reuse activity knowledge:

Example 3 "Corporate Web Services": A software engineer has to create a standardised Web Service that allows information to be queried regarding alarm information (i.e., sensor alarms) and devices (i.e., embedded devices). This Web Service is to be used to exchange data uniformly between horizontal and vertical software applications of a company. After completing the task (the development of such an interoperability Web Service), the software engineer has to select an appropriate repository to provide the development result to the other software engineers.

Example 4 "Corporate Web Service Integration": A task is given to a software engineer to integrate the service developed in Example 3 into an existing software application. The software engineer has to be aware of the integration of this Web Service and its additional artefacts in the used development environment. The service requires extensive knowledge of copying, referencing, and configuring activities. Due to these specific requirements, the software engineer chooses to automate these activities to perform this integration process more

easily a second time. In addition, the software engineer also wants to provide this automated integration to other users.

Example 5 "Corporate Web Service Transformation": By using the other previously transformed software unit (Corporate Web Service), the engineer of Example 4 recognises that it is possible to handle the problem of transformation. The unit solves the problem of unified communications, but in a different technology type. For this reason a transformation tool is used (e.g., SvcUtil) to change the original software unit into the required technology type. This automates the transformation step for easier reuse at a later date. In addition, the software engineer wants to make this automated transformation available to other users.

The software engineer of Example 3 has various problems, all based on knowledge. Firstly, it has to be ensured that all other teams have access to a repository and are able to use it. Knowing 'where' to find information and 'how' somebody gets the information to decide is part of the process of knowledge acquisition.

If the software engineer of Example 3 only uses this software unit for themselves, this will not raise any problems. When spreading to other software engineers, however, the engineer is confronted with various issues. First, the knowledge (automated integration) has to be described in a format that other people are able to use. Second, it has to be found as seen in Example 3, using a repository to distribute this knowledge. Here, the question of 'where' and 'how' arises accordingly (see Example 3), revealing the same knowledge problem as seen in Example 4 and Example 3.

These examples point to relevant application scenarios for the focused approach. A user will need to provide knowledge for others to use. In addition, there is a need to automate recurring activities that are of use. This knowledge can also be maintained by other users as well as the producer of a software unit. Each of these activities is based on knowledge.

In these examples, the focused approach is directed to a user profile that has the following properties:

A general approach to realise knowledge-based automated reuse activities

- Users have the knowledge of a software unit and want to make this available to other users.
- - Users have knowledge about reuse activities (e.g., integration and transformation) and want to make these available to other users.

Such a user profile is expected in this research by the following groups: ‘senior software engineer’: Experienced software engineers of certain software units, applications integration, and transformation scenarios that have the relevant knowledge.

In the scope of this research, the focus is on such senior software engineers. The user profile is called the Knowledge Creator (KC) profile.

4.3. Focused development project scenarios

The previous section describes both of the user profiles used in the basic concept. In the following section, the focused development project scenarios are described using these user profiles. This mainly based on the statement of Qu, Ji and Nsakanda (2012) that teams and knowledge are distributed. The result is the description of the application area of software engineers that is focused by the research of this thesis.

4.3.1. Separate user development projects

The first perspective on development project scenarios is separated into development projects or sub development projects both of which are only handled by one software engineer.

4.3.1.1. Single KC – single KU



Figure 32 - Single KC and Single KU relation

A general approach to realise knowledge-based automated reuse activities

Figure 32 shows a typical scenario in smaller development projects. An inexperienced user (KU profile) uses the knowledge of reuse activities for a single software unit in this development project.

4.3.1.2. *Single KC – multiple separated KU*

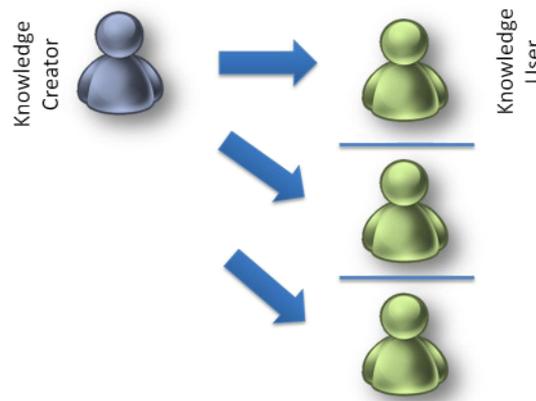


Figure 33 - Single KC related to multiple KU

Multiple inexperienced users handling the same knowledge of reuse activities for a single software unit are shown in Figure 33. However, not all users are involved in the same development project. This scenario is typical for software engineers using repository communities (e.g., CodeProject; cf. Maunder, 2012).

4.3.1.3. *Multiple KC – single KU*

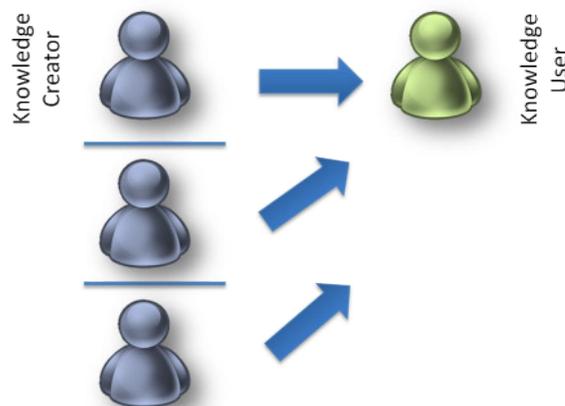


Figure 34 - Multiple KC related to single KU

A general approach to realise knowledge-based automated reuse activities

The third interesting scenario is the use of multiple software unit reuse activities by a single user. This is also a special variation of 1 KC – 1 KU and can be handled by looking on each relationship individually. This is shown in Figure 34.

4.3.1.4. Single KC – multiple related KU

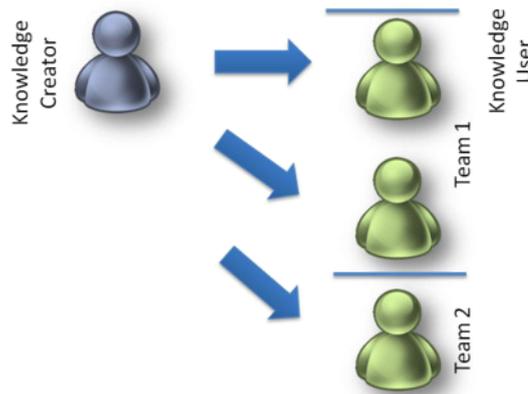


Figure 35 - Single KC related to multiple related KU

An interesting scenario is a development project where different team members are working together. Such scenarios are created multiple consultants or if a number of smaller companies work together for one customer. Such a scenario may include that different team members are not located on the same site yet have to reuse the same activities, for example to create their own working environment (cf. Qu, Ji and Nsakanda, 2012). This is shown in Figure 35.

4.3.2. Separate team development projects

In the previous section, only single software engineers with a KU profile are described. Often, in global companies, such development projects are done by development teams (cf. Qu, Ji and Nsakanda, 2012).

4.3.2.1. Single KC – Multiple non-separated teams

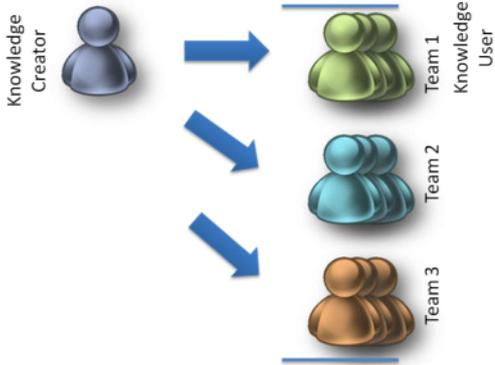


Figure 36 - Single KC related to multiple-non separated teams

Figure 36 shows a typical scenario in global development projects (cf. Qu, Ji and Nsakanda, 2012).; multiple KU teams working together on a development project. Often, such teams are not located on the same site and are mostly divided by culture or time zone differences which have negative effects on communications (see Taweel et al., 2009). It may be that reuse activity knowledge has to be used by all teams. The complexity in this scenario can be increased if teams or single team members use some of the scenarios in Section 4.3.1.

4.3.2.2. Single KC – Multiple separated teams

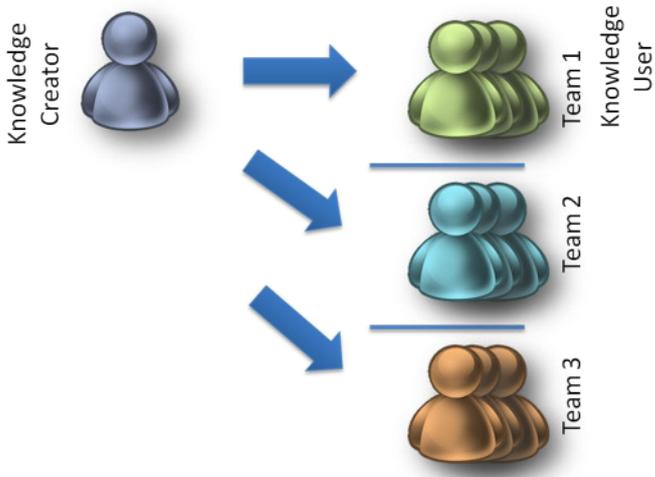


Figure 37 - Single KC related to multiple separated teams

Multiple, separated teams using the same knowledge of reuse activities for a single software unit are shown in Figure 37. However, not all teams are related in the same development project. This scenario describes horizontal development projects. The complexity in this scenario can be increased if teams or single team members use some of the scenarios in Section 4.3.1. This scenario is similar to the internal case study of Schneider Electric (cf. Section 3.2.1.2).

The research of this thesis focuses on all shown scenarios. Because of limitations of time and number of participants, this research performs a case study showing the scenario of a single KC and multiple separated KU (cf. Figure 33). The analysis result of this scenario is also valid for the other scenarios.

4.3.2.3. Decision maker

When changing the perspective to the decision maker of used software units, two interesting scenarios can be identified: distributed and non-distributed decision scenarios. In non-distributed decision scenarios, the decision maker - the person who decides to reuse a specific software unit - is the same as the person performing the reuse activity. However, there are scenarios in which the decision maker and the performer are not the same person. Within the scope of this thesis, this is called a distributed scenario because the individuals may be located in different places and may differ in their domain of experienced users (cf. Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010). Software architects are typically these kinds of decision makers in software development. Section 3.1.3 described these focused scenarios.

A distributed scenario can be explained as follows: Based on the scenario of Schneider Electric (see Section 3.2.1.1), two teams may be situated in different locations (e.g., France and India) while working together in a software development project. The French team defines the architecture and pre-selects existing software units for reuse that are developed by the same team. The Indian team is responsible for the real implementation and integration. The decision

A general approach to realise knowledge-based automated reuse activities

about software units is made by the French team. If the Indian team does not have all the relevant knowledge it cannot start the development process or may do so only partially.

4.3.2.4. Focused development scenarios

All the scenarios presented in Section 4.3.1 and Section 4.3.2 are relevant within the scope of the focused approach, especially the scenarios which involve handling multiple KU profiles in different locations.

The decision maker, who decides which kind of software unit should be reused, is not as relevant for the focused approach in this thesis. However, it is relevant to know that this decision is one way to create the KC and KU profile relationships in the different development scenarios.

4.4. The fundamental concept

4.4.1. Software construction as concept bases

A relevant part of the fundamental concept is adopting the perspective of software construction (see Section 2.2.2.1). Therefore, the concept focused approach has to be explained first. Two elements are relevant within this perspective: Service-based Software Construction Process (SSCP) and Software Construction Artefact (SCA).

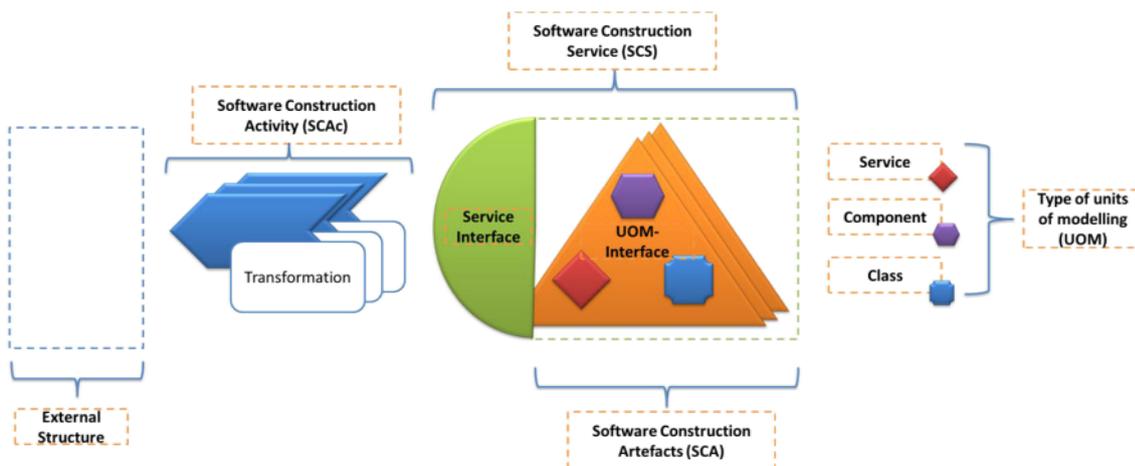


Figure 38 - Parts of the Service-oriented Software Construction Process

The SSCP describes the method of the focused approach. This has been briefly described by the basic idea (see Section 4.1); it handles Software Construction Artefacts for the user, these artefacts contain different information about software units and related reuse activities. Both perspective elements are described in more detail as follows.

Figure 38 shows the basic design of an SSCP. This contains five elements. In the following, an overview of each element is shown.

4.4.1.1. Representation of the SSCP-components

The SSCP uses four different parts: external structure, software construction artefact, software construction service and an optional process description.

External structure: The external structure represents a target platform (platform specific implementation) in which software construction artefacts are inserted at a later time. Often, reuse activities (e.g., the focused SCAs) have a relationship to a specific external structure. This depends, in particular, on the respective design phase of a reuse activity or a software unit. The external structure is part of the user's environment and not specified by the focused concept. Therefore, it is not relevant for the SSCP how the software engineer or designer creates the required external structure.

In the scope of the research, an integration activity uses an IDE, a deployment activity uses an embedded device or deployment platform, and a transformation activity uses, for example, a file and folder structure as an external structure.

Software Construction Artefact: *“A software construction artefact (SCA) is a typified unit which is the basis for the construction of software.”* (Zinn, 2008, p. 80)

Therefore, an SCA represents a container for software units within the SSCP. The basic idea of an SSCP is to enable the execution of SCAs, and not to change the software unit content. An SCA includes the necessary information. Therefore, an SCA consists of (cf. Figure 39):

- the different units of modelling (UOM, software units, i.e., objects, components, and services),

A general approach to realise knowledge-based automated reuse activities

- readable data for software engineers and designers (human readable data),
- data for reuse activities (i.e., transformation or integration; mostly non-human readable data),
- software construction activity information related to the unit of modelling,
- an SCA type which describes the contents of the units of modelling, and
- a service interface.

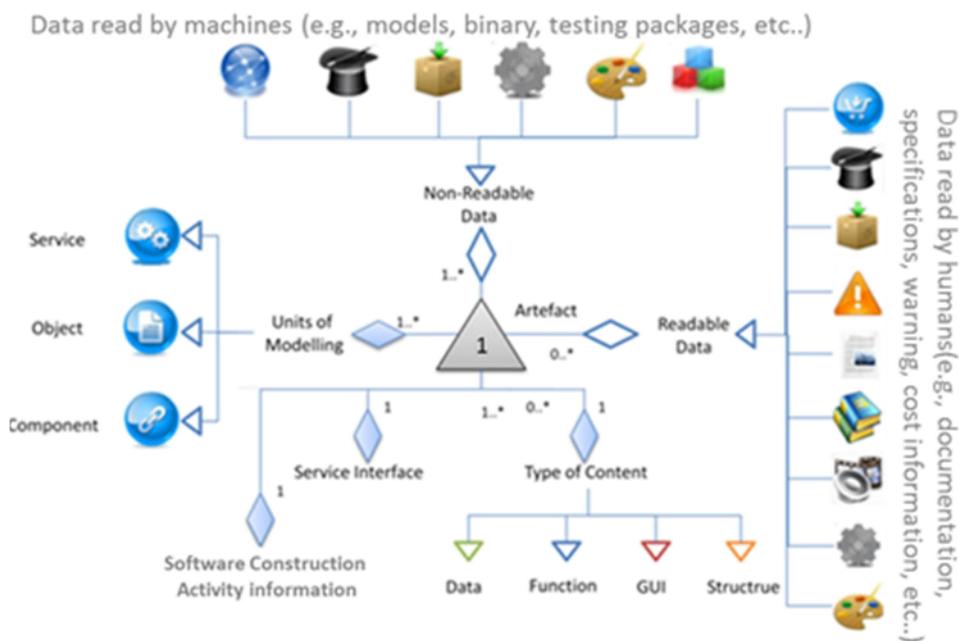


Figure 39 - Content of a Software Construction Artefact

An SCA contains different implementation solutions (e.g., non-readable data in Figure 39) from which a software engineer is able to choose. Again, the solutions can be realised in the technologies of the different software technology approaches or their combinations as objects, components, and services. Each variant may also have multiple reuse activities (cf. Figure 39).

SCA types: The classification of SCAs corresponds to meta-information and serves to differentiate the contents. It describes the professional content which is carried by the units of modelling and, therefore, is relevant for the software engineers to identify a software unit or an SCA. Note: The SCA type is explained in this section and shown in the realisation in Chapter 5.

These types are not relevant for this research but are used for further research. The types are published throughout the Ph.D. research (see Zinn, Turetschek and Phippen, 2008).

Within this thesis, four of the internal content-dependent types are distinguished: data, functions, structure and (graphic) interface elements. These constitute the basic types of software unit (Zinn, Turetschek, and Phippen, 2008). This differentiation serves to display the software-technical contents of the artefact for the software engineer. The view and expectation is limited to these four types:

- **Data:** Data represents all the information that is worked with. A software unit that belongs to a Data SCA typically provides data or information. In comparison to Function SCAs, Data SCAs have no or very low costs at the data gathering stage.
- **Functions:** With the function type, functions, methods or operations are described. They exist locally and/or externally. The useable content of a software unit that is related to a Function-SCA are functions, methods or operations.
- **Structure:** The structure type is a carrier of information in the form of interfaces, class structures, patterns, and architecture defaults.
- **(Graphic) user interfaces:** This type of software unit includes (graphic) user interfaces. Therefore, lightweight (e.g., Extensible Application Markup Language - XAML or Scalable Vector Graphic-SVG and library-based, as for example, Windows Forms) technologies are suitable. The useable content of a software unit that is part of a User Interface (UI)-SCA are user interfaces.

The shown definition of the SCA types is a summary of the definition given by Zinn (2007).

Service Interfaces: For the realisation of the SCA, different interfaces are necessary. Two interfaces are distinguished: the software construction service interface and the interfaces for handling individual units of modelling inside the SCA (see Figure 40).

A general approach to realise knowledge-based automated reuse activities

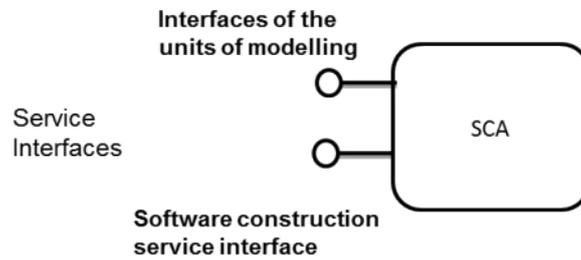


Figure 40 - Service interfaces of an SCA

Software construction service interface: Each artefact is offered by a software construction service (SCS). This service provides standardised access methods to the information (the artefact and the included software units).

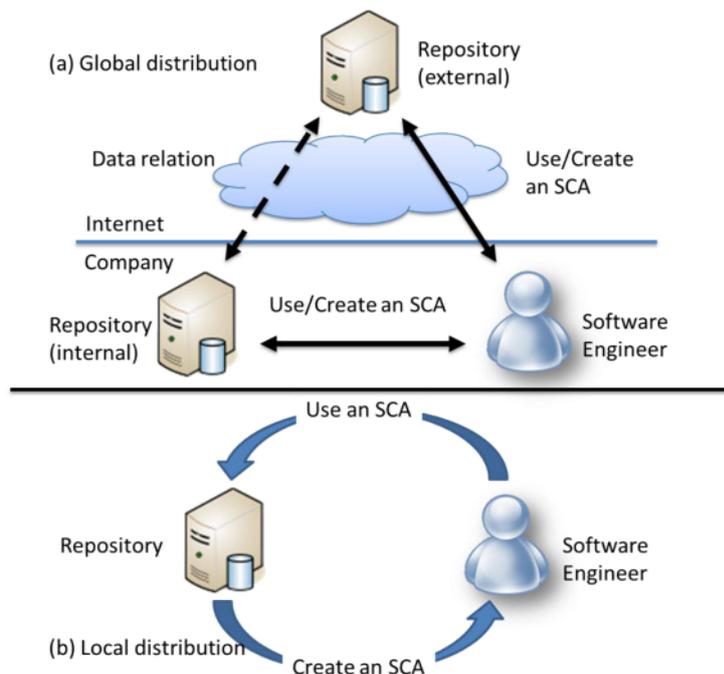


Figure 41 - Reuse of software construction artefacts

Interfaces of the units of modelling: The interface for the single units of modelling provides technical, domain, and additional information which are necessary for the development. Figure 41 shows examples of local (b) and global (a) distribution, as well as showing the reuse of

A general approach to realise knowledge-based automated reuse activities

software units by repositories providing these services. This service also includes the management of SCAs.

Units of modelling: This defines single software including components, objects, and services including their required descriptions, as well as, the related reuse activities.

The focused approach of this research is to create a service-oriented environment for the storage and execution of knowledge of software unit construction activities. This thesis does not refer to all possible aspects of these activities, focusing rather on the aspects of:

1. search of the focused SCAc information (and software units),
2. tool-based transformation,
3. integration within development environments, and
4. embedded device-based deployment of software units.

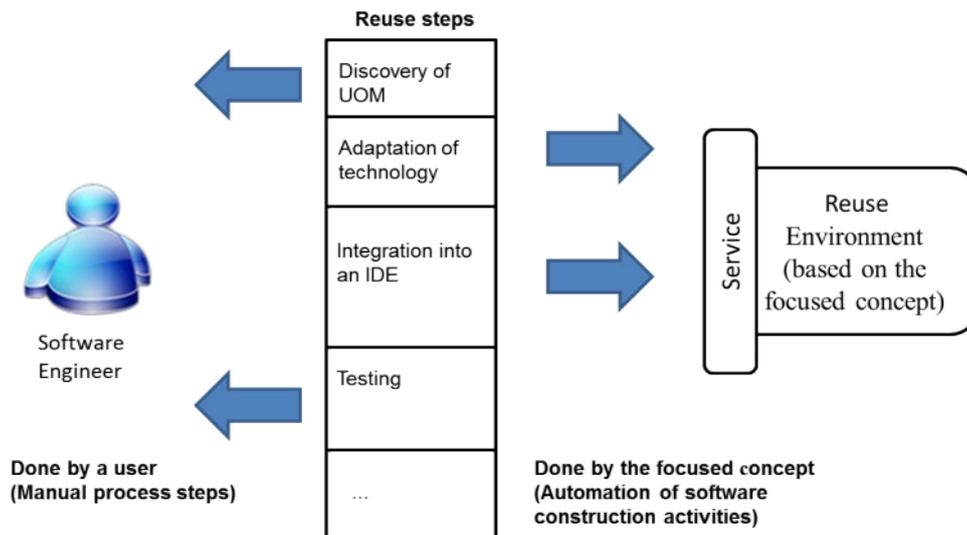


Figure 42 - Automation concept

The goal of the approach is to support software engineers to reuse specific software units. The inexperienced software engineers are focused (KU profile), which means they do not have the necessary knowledge to perform a specific software reuse activity. This goal is reached by (partial) automation of these specific reuse activities. Figure 42 shows an example. Here, a user

A general approach to realise knowledge-based automated reuse activities

has to discover the software units manually. The focused reuse activities are performed automatically for the user by the SSCP.

The input of knowledge information can be performed by use of a graphical user interfaces (GUI) or other knowledge detection approaches (i.e., McCarey, Ó Cinnéide and Kushmerick, 2008). An UI is a defined interface that communicates with the environment of the focused approach (see Figure 43).

Note: For the explanation of the concept a UI is used. But the definition of a UI is not part of the focused concept. Also, how the described service is used or integrated in an existing environment is not defined. The thesis uses one possible way of usage and service integration.

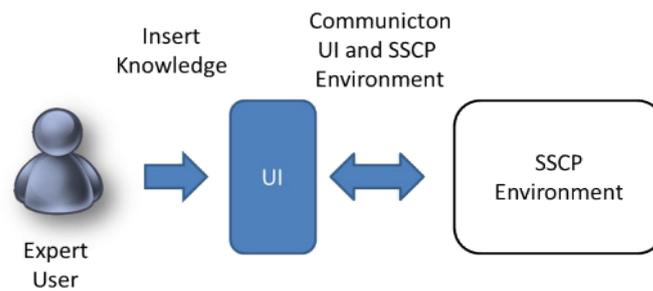


Figure 43 - UI as abstraction layer for the focused environment

The knowledge given by an experienced user is stored within a semi-semantic model. On a basic level, this knowledge consists of software units, additional information (such as documents, video, audio, and so on), knowledge about adaptation of the units, as well as knowledge about their integration into software development environments. Figure 44 demonstrates this property.

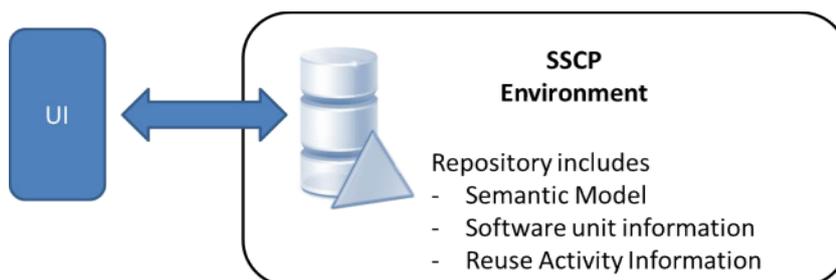


Figure 44 - Data content of the SSCP environment

A general approach to realise knowledge-based automated reuse activities

A service-oriented architecture is used to enable the integration of service-based extensions (e.g., plugin) into the focused approach. Thereby, the plugins are able to implement and perform the stored activity knowledge to the focused reuse activities (transformation, integration, and deployment) as well as all other management functions (i.e., search or storage of software unit information). Knowledge is entered into the system by experienced users and can be reused by inexperienced users. Figure 43 and Figure 45 show these relationships.

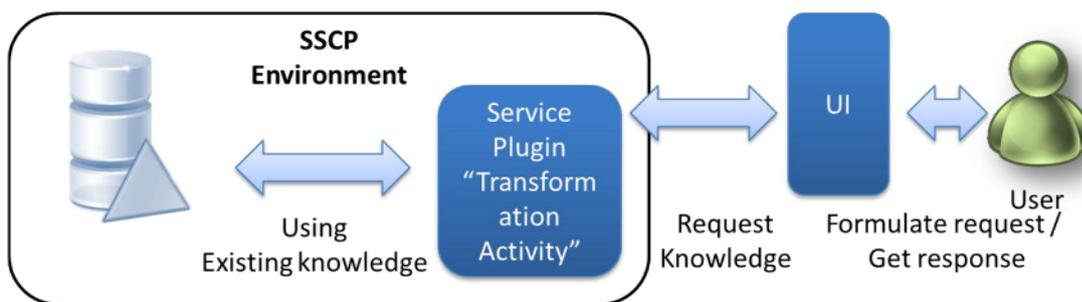


Figure 45 - Requesting knowledge inside the SSCP environment (Transformation activity example)

The focused approach behaves as a content management system (CMS) on the one hand and as an knowledge automation system on the other. In the following sections, the above briefly described aspects are explained as a concept.

4.4.2. Relevant elements of the concept

Based on this basic idea, a concept was developed. This concept is comprised of three parts:

- (1) The knowledge database called the ‘Software Unit Model’ has the task of storing knowledge about software units. This knowledge describes software units and their behaviours for reuse activities like transformation and integration into a common description. This description includes technical and business information.
- (2) The service called ‘Software Construction Service’ provides operations which support the user in typical software reuse tasks like searching, adapting, integration, and deployment as well as the execution of such reuse activities. For search functionality, the service uses the information stored in the Software Unit Model.

A general approach to realise knowledge-based automated reuse activities

(3) To perform SCAs the different activities are implemented by special adapters (e.g., plugins). These adapters use the common information/knowledge together with specialised information stored in the adapters to support the user. The Software Unit Model includes all information about the software units and activities. This is necessary information for these adapters.

Application example: The user requests a software unit to be transformed from Java-based into .NET-based technology. The Software Construction Service will then search (1) for an adapter (e.g., plugin) which is able to execute (2) the transformation and respond to the request with the transformed software unit. Figure 46 shows the relationship between Software Construction Service and the Software Unit Model.

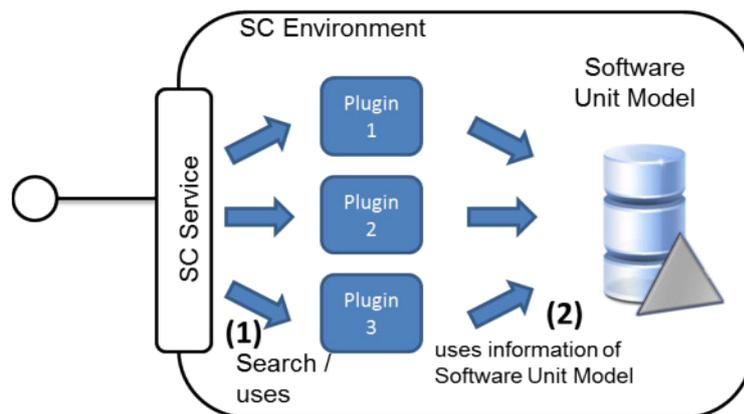


Figure 46 - Use of the Software Unit model in the focused environment

The example above shows only a small part of the whole concept. The main concept is an environment which has to be filled with reuse activity information. This information can be extended continually and reused by other users. Figure 47 shows the focused life cycle concept of reusable knowledge.

A general approach to realise knowledge-based automated reuse activities

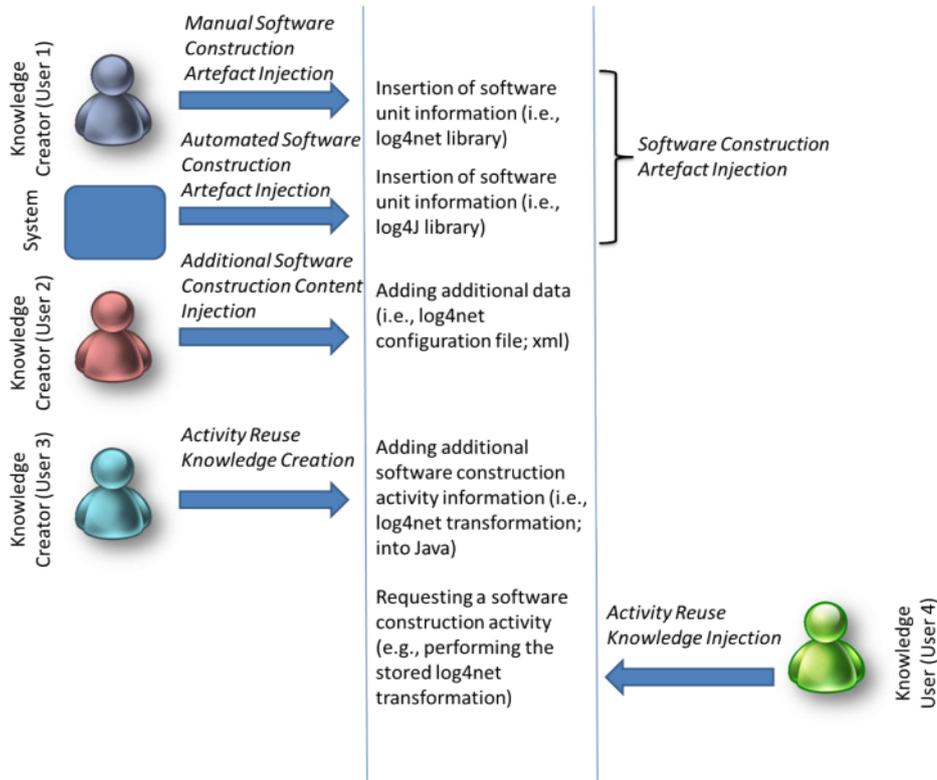


Figure 47 - Communication concept of knowledge in the focused concept

The concept procedure can be explained as follows: A reusable software unit may be inserted into the focused environment. This process is called ‘Software Construction Artefact Injection’, which means to store a software unit (i.e., a class, component or service) in an existing repository system which is connected to the environment. Usually, storing is done by an experienced user. In this case the process is called ‘Manual Software Construction Artefact Injection’. This process is carried out by users of the KC profile, (see User 1 in Figure 47), but can also be performed by an adapter that reads information from a repository and adds it to the environment automatically. This process is called ‘Automated Software Construction Artefact Injection’. The result of the Software Construction Artefact Injection is a software unit which is described by the Software Unit Model. KC profile users can add additional information, like specification or documentation (see User 2 in Figure 47). This process is called ‘Additional Software Construction Content Injection’ and is a functionality typically known in content

A general approach to realise knowledge-based automated reuse activities

management systems. This process is carried out by users of the KC profile. Users can add information about reuse activities related to the stored software units. This means that users are able to add information about the transformation of, or integration into, the software unit, for example. This process is called ‘Activity Reuse Knowledge Creation’, a process carried out by the KC profile users. The focused environment is now able to execute these rules which transform, integrate or deploy the specific software unit (mostly) automatically. For example, User 3 in Figure 47 injects knowledge about a transformation of the original .NET library into a Java library. The user also injects another rule which describes how to integrate the transformed software asset into an Eclipse development environment project. Such knowledge can now be used by other users by requesting the transformation or integration. The focused environment will execute the request. As a result, there is no need for the requested user to know all the information which is necessary for the two processes. In the example of Figure 47, User 4 uses the stored knowledge about transformation and integration. Using active reuse knowledge is called ‘Activity Reuse Knowledge Injection’ within the scope of the general concept. Section 4.4.1 demonstrates the fact that reuse knowledge is stored in the Software Unit Model and in the adapters (plugins) of the Software Construction Service. The difference between the model and the adapters is the abstraction level of the stored knowledge. Knowledge stored in the Software Unit Model is described abstractly by the model and is used for different purposes (for example, in transformation or integration). On the other hand, knowledge stored in the adapters (plugins) is specialised; not abstracted knowledge. Adapters control other applications or systems for reuse purposes (for example, in adaptation or integration). The input for these applications or systems is the knowledge stored in the Software Unit Model. The creation and integration of an adapter in the focused environment is called ‘Passive Reuse Knowledge Creation’. During the Active Reuse Knowledge Creation, a user stores the data for executing a rule on a specific adapter. The user then combines the input for the adapter with data that has

A general approach to realise knowledge-based automated reuse activities

been saved in the Asset Injection or Additional Content Injection. Figure 48 shows which knowledge repository is used by the different phases in the focused concept.

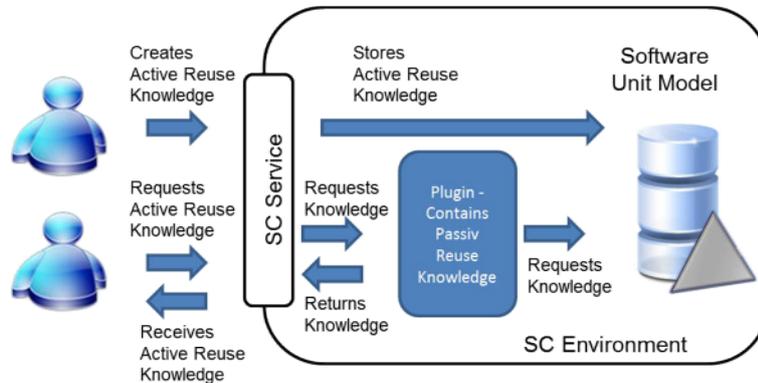


Figure 48 - Use of active and passive knowledge

Note: The research focus is via a concept to enable exchange of knowledge between inexperienced and experienced software engineers. The division of knowledge in active and passive parts is viewed as an interesting possibility for further research. Passive knowledge, as for example, the knowledge about the use of a special IDE is part of the plugins. This is created by the plugin developers and not by the experienced user. In this thesis the difference is know but not considered. Both knowledge types are necessary for the focused approach.

4.4.3. Use cases

The fundamental concept supports 12 different use cases. These use cases are required by the discussed profiles KC and KU; therefore, KC and KU are stakeholders. The use cases are now described briefly to support the discussion about the concept (see Table 5). A more detailed description is shown in Section 5.4.1. Figure 49 summarises the supported use cases of the focused concept.

A general approach to realise knowledge-based automated reuse activities

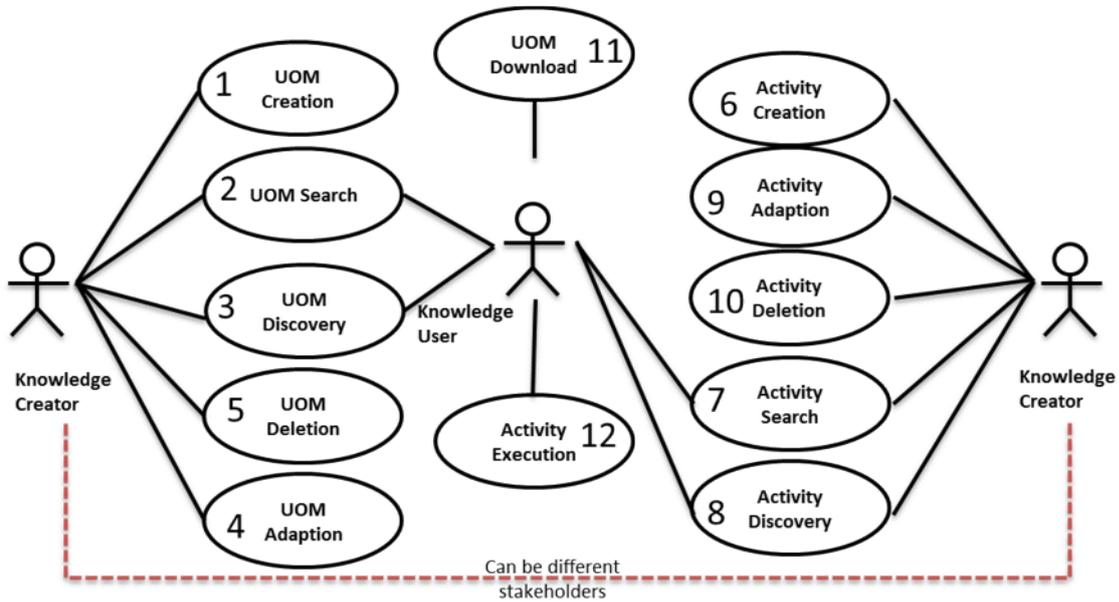


Figure 49 - Overview of the supported use cases

No.	Title	Description	Used by
1	UOM Creation	This use case describes the creation of a software unit.	KC
2	UOM Search	This use case describes the search of a software unit.	KC, KU
3	UOM Discovery	This use case describes the selection of a software unit.	KC, KU
4	UOM Adaption	This use case describes the adaptation of information of a software unit.	KC
5	UOM Deletion	This use case describes the removal of a software unit.	KC
6	Activity Creation	This use case describes the creation of a reuse activity.	KC
7	Activity Search	This use case describes the search of a reuse activity.	KC, KU
8	Activity Discovery	This use case describes the selection of a reuse activity.	KC, KU
9	Activity Adaption	This use case describes the adaptation of information of a reuse activity.	KC
10	Activity Deletion	This use case describes the removal of a reuse activity.	KC
11	UOM Download	This use case describes the retrieval of software unit information.	KU
12	Activity Execution	This use case describes the execution of a reuse activity.	KU

Table 5 - Briefly description of the supported use cases

The use cases describe the creation, discovery, adaptation and deletion of software units (Use Case 1-5; 11) and software construction activities (Use Case 6-10; 11). Additionally, the execution of an SCAC is part of the use cases (Use Case 12). Use Cases 1, 2, 3, 4, 5 and 11 are necessary to explain the complete concept, but are not focused on by the research.

4.5. Concept of potential technical environments

In this section, the typical technical environment of the focused concept will be described. For this reason the communication entities, the scalability of the concept, the amount of communication data, as well as the distribution of business logic will be explained.

Note: There may be other possible environments for the shown concept. It is also possible to instantiate the described concept with different technologies and environment setups. A technical realisation of this concept is described in Chapter 5.

4.5.1. Communication concept

The communication concept of the focused approach will be explained in this section. Therefore, communication entities as well as the communication scenarios require explanation.

4.5.1.1. Communication entities

In essence, three main elements exist in a concept-based environment: a concept-based client, a concept-based server, and a concept-based repository system.

Concept-based client system: A client supports the users when interacting with the server. Such a client supports the use cases of a user. In principle, the development project scenarios presented in Section 4.3 and the user profiles Knowledge Creator and Knowledge Consumer as described in Section 4.2 are meant. Typically, the following types of application are used as client systems:

- Desktop applications: A client can be implemented as a typical desktop application. Such applications usually have the advantages of accessibility to an amount of host

system resources. Typically, software engineers work with such applications during the development stages.

- **Web applications:** A client can be implemented as a web application. Such variants usually have the advantage of being (mostly) independent of the available computers, and more so than a desktop application. This distribution architecture allows the execution of a web application from different systems in various locations. Often, such applications do not have the full accessibility to host resources as desktop variants. This is usually for security reasons, for example.
- **Application integration:** A client can be implemented as an extension of an existing application. Development environments such as Eclipse or Visual Studio offers the capability of extending their functionality by using application extensions (so-called plugins or packages; see Eclipse Foundation, 2012; Microsoft, 2012c). A client can execute and be displayed inside such a development environment and carry out its task inside this environment. The advantage of such an application is the capability of communicating easily with the development environment and the fact that users do not need to start an additional application.

Concept-based server system: The main component of the concept-based environment is a server application. The task of the server is to handle user requests (e.g., searching a software unit or performing a reuse activity). Essentially, such an application needs the ability to handle multiple requests simultaneously. A connection to multiple repositories is also required in a scenario where more than one repository exists.

It is also possible to connect multiple servers together. In such a scenario a server is linked as a repository to another server. This is useful for environments that include more than one server or server groups.

Repository system: In the concept-based environment access is required to one or more software unit repositories. This can be realised by adapters. Each adapter handles the

A general approach to realise knowledge-based automated reuse activities

communication between a repository and the server. Repositories typically provide the capability of being accessed using database adapters (e.g., Java Database Connectivity JDBC or Open Database Connectivity ODBC), general services (e.g., Web Services) or by providing specialised services (e.g., ports of a Microsoft Structured Query Language SQL server). It is also possible for repositories to only provide an interface for humans to access software unit information.

Based on these three elements, the following communication scenarios are of interest within the scope of the focused concept (see Figure 50 and Figure 51):

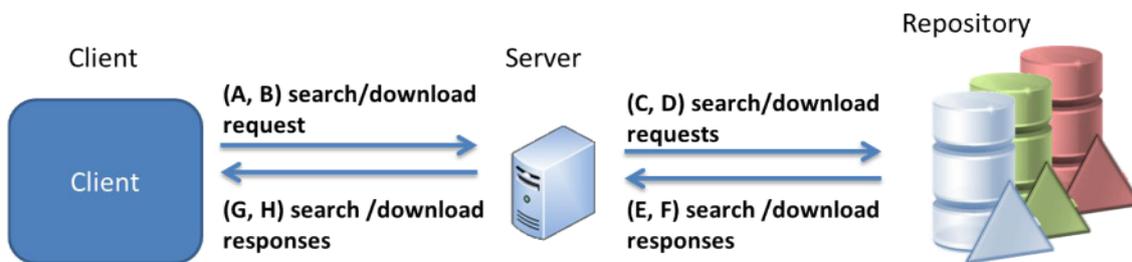


Figure 50 - Knowledge injection scenario

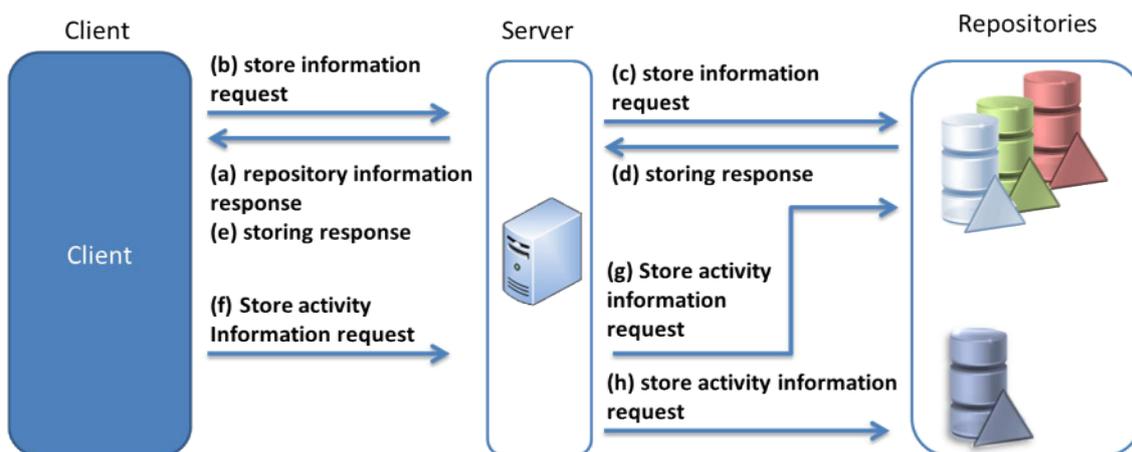


Figure 51 - Knowledge extraction scenario

Knowledge Injection Scenario: In Figure 50 it is possible for the client to send search and download requests (A) and (B). The server receives and performs these requests. During the

A general approach to realise knowledge-based automated reuse activities

execution of these tasks, the server communicates with the different repositories (C) and (D). The different databases' responses (E) and (F) are then used to generate responses for the client by the server. These responses will then be sent to the client system (G) and (H).

Knowledge Extraction Scenario: In Figure 51, an experienced user is able to use the client to store a software unit or additional information about a software unit. The server receives the request (b) and performs this by communicating to the different databases (c). It may be possible to update the client system on the state of the storing request by a response from the server (e) based on a response of a repository (d). It may be necessary to provide the user with information about the connected repository system to begin the knowledge injection process (a). An existing software unit is a necessary requirement for storing or performing reuse activities. Companies may have special procedures for filling their repositories. As a result, the first steps (a-e) shown in Figure 51 are optional. The next step is mandatory; a user has to store their knowledge about a reuse activity so it can be performed by the system. Not all repositories support the storing of reuse activities because of the given data model of the repositories. For that reason, a special reuse activity repository may exist in the concept-based environment (see Figure 51). A user may start a request to store activity information (f), and the server adapts and forwards the request to the specific repository (g). In the case of this software unit repository not being able to store this kind of data, the request is forwarded to the special repository (h). It may also be necessary to provide the user with information about the connected repository system to begin the knowledge injection process (a). It may be possible to relay to the client system the state of the storing request by a response from the server (e) based on the response of a repository (d).

Note: The described scenario does not include restrictions on any of the elements, such as security or the download restrictions of the database. One download restrictions example is discussed later on in Section 5.2.4.)

A general approach to realise knowledge-based automated reuse activities

Figure 50 describes the basic scenario for searching and receiving data. The main focus of the concept is the execution of reuse activities. Based on this restriction, the following element is necessary in the communication scenario: a Reuse Activity System (RAS) which handles reuse activity information. Depending on the activity, such systems carry out different tasks. For example, in an integration scenario, data may be integrated with a software system. In this case, the RAS is able to integrate the software unit into this software system, based on the stored reuse activity knowledge. Another example is the controlling of transformation software. A transformation activity may include the task of executing a software application, transforming a software unit into another form or type (e.g., a compiler transforms classes into binary code). In this case, a Reuse Activity System controls the transformation application. As mentioned in Chapter 2, the amount of reuse activities and controllable tools is high and differs. Such systems may be implemented by the use of different programming languages, operating systems and communication technologies. A typical task of the RAS is to perform reuse activities. The necessary information is given by the software unit model.

Based on the communication scenario in Figure 50 the following extension may be created (see Figure 52).

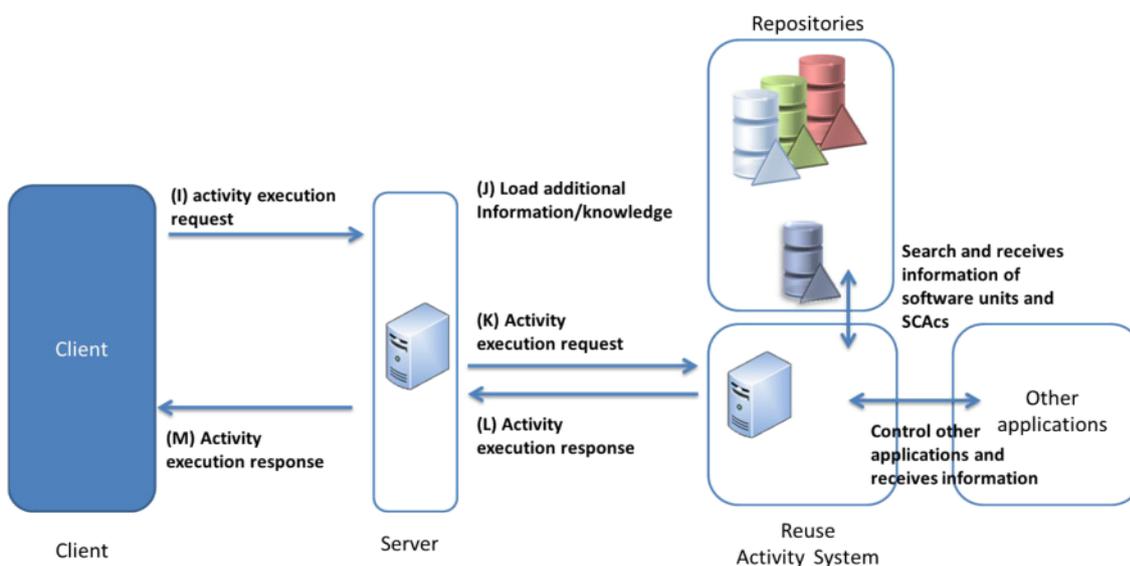


Figure 52 - Request for reuse activity execution

Knowledge Injection Scenario: Figure 52 shows the changes for the knowledge injection scenario. A user is able to request an execution of a reuse activity by using the client system (I). The server is able to perform this request by communicating to the specific RAS (K). It may be required that the server system needs to load additional information about the software unit or the stored reuse activity from one or more repositories to perform the activity (J). The activity result is sent from the RAS to the server (L) which creates a response for the client system (M). As explained in the description of the RAS, such a system may communicate with other applications necessary for the reuse activity (N).

Knowledge Extraction Scenario: In the case of the Knowledge Injection Scenario, this extension changes the idea of the communication context; only the content of a search request may change. A user is now able to search for a reuse activity or the result of an activity.

4.5.1.2. Scalability scenarios of the focused approach

In the previous section, the communication structure was explained. Different scenarios may be possible based on this communication structure.

Note: The following discussion only includes scenarios which are necessary for the discussion of the basic idea (see Section 4.1), the focused concept and the approach discussions in the following chapters. Other possible scenarios may exist but are not discussed here.

For the purpose of this discussion, two border scenarios will be described which differ in the value of their scalability: monolith and total distribution scenarios. A monolith scenario describes a complete environment, based on the discussed concept running on one system. From the system perspective, it is not relevant if this scenario is realised by different applications or if only one application includes all features of such an environment. Figure 53 illustrates this example.

A general approach to realise knowledge-based automated reuse activities

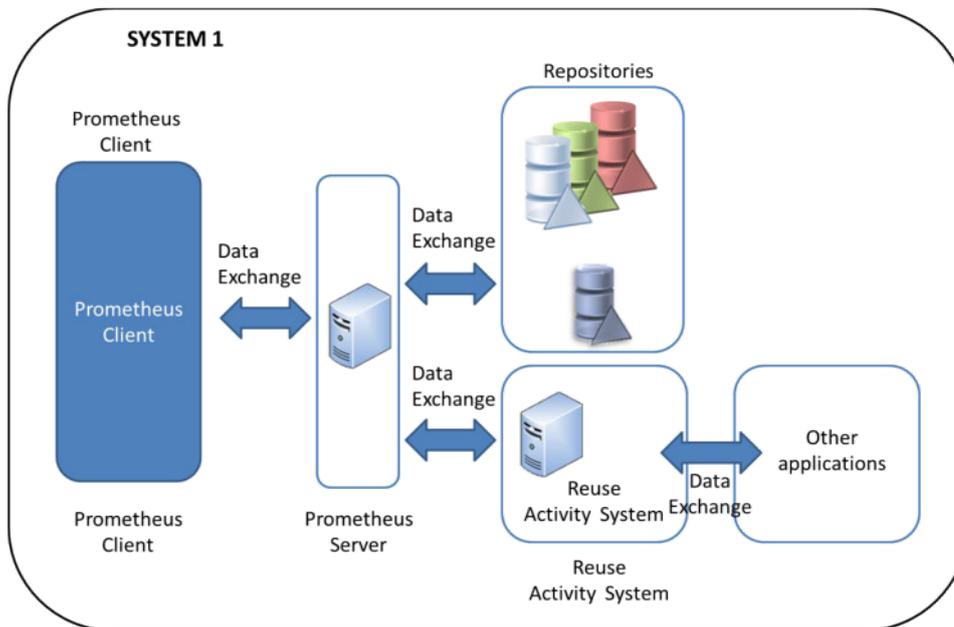


Figure 53 - Monolith scenario

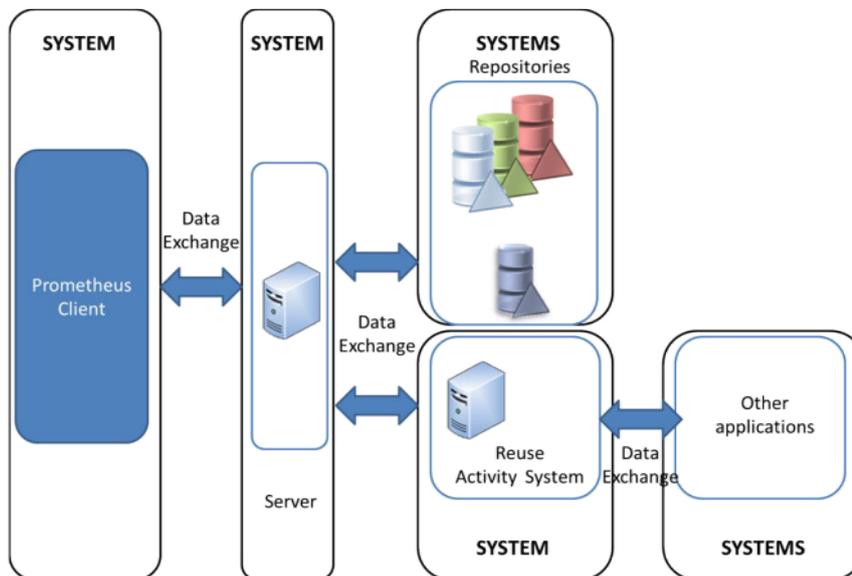


Figure 54 - Completely distributed scenario

In Figure 53 each element of a concept-based environment (server, client, RAS, etc.) is part of the same system. The other scenario describes the high scalability in the described approach. In this scenario each element is a standalone application and runs on a different system. Figure 54

A general approach to realise knowledge-based automated reuse activities

demonstrates this scenario by presenting all elements (server, client, RAS, etc.) as part of a different system. For this thesis, a realistic scenario is between the two scenarios.

A distributed scenario can be found in global working companies (cf. Example of Schneider Electric in Section 3.2.1.2).

4.5.1.3. Amount of data

In the described communication and scalability scenarios, the amount of data may differ. Indeed, this is an relevant point for the definition of communication interfaces. Additionally, two scenarios are very useful for the explanation for the main concept: Data-driven or ID-driven scenarios.

In a data-driven scenario, the communication contains the complete datasets which will be used by the caller. Data values may be changed by any element in the concept-based environment. In an ID-driven scenario, only necessary values will be transmitted. The context of these values which software unit belongs to these values is recognised by a unique identifier. Such an approach reduces the necessary bandwidth of a single call. One example is the execution of a transformation software construction activity: where the client asks for software unit information. After receiving the information about the software unit, the client asks for the execution of the transformation activity. Figure 55 and Figure 56 show both data scenarios for this example.

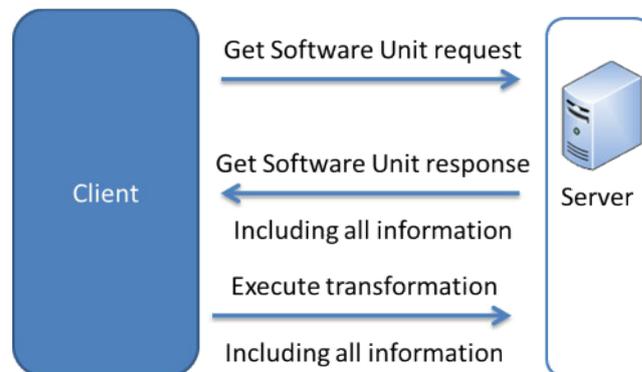


Figure 55 - Data-driven communication

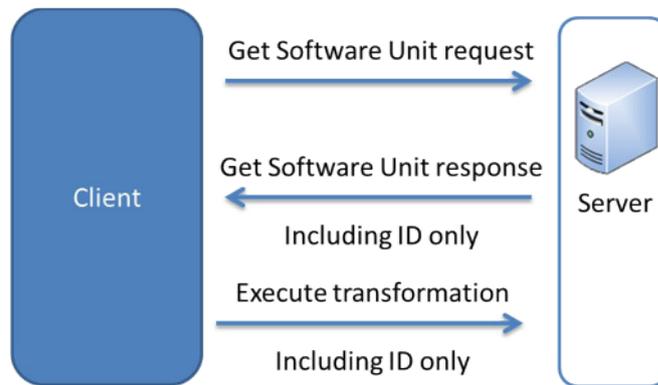


Figure 56 - ID-driven communication

In Figure 55, the request for a software unit is answered with the full amount of values of the software unit even though not all information is necessary for the user's current use case. In Figure 56, only values which are requested are received. This may increase the amount of requests but minimises the amount of data in a single request. Based on the received information, a user may choose a transformation activity and request its execution by the server. In Figure 55, all information from the software unit and of the transformation activity is sent to the server, which initiates the execution. In Figure 56, only the ID of the software unit and the ID of the transformation unit is sent. The server has to load all necessary information.

Note: As with the description of both scalability scenarios, the different variants between both data scenarios are possible. At this point, no decision has been made as to which approach is better. The realisation of the approach in Chapter 5 focuses mainly on the ID version.

4.5.1.4. Distribution of business logic

The business logic may be centralised, totally distributed, or partially distributed. In the case of the concept-based environment, all variants are feasible, but only one direction is focused upon. The concept describes a service for the automation of software reuse activities. One relevant element of the concept is that an inexperienced user is able to start a simple request to perform an activity. Therefore, it is recommended that the logic for handling data and performing reuse

A general approach to realise knowledge-based automated reuse activities

activities is from the perspective of a client system behind the service. Therefore, the ID-driven data approach is preferred, in order to keep the communication between the server and the client system simpler (from the perspective of the communication data model). Behind the service, the distribution of the business logic is not regulated by the concept described.

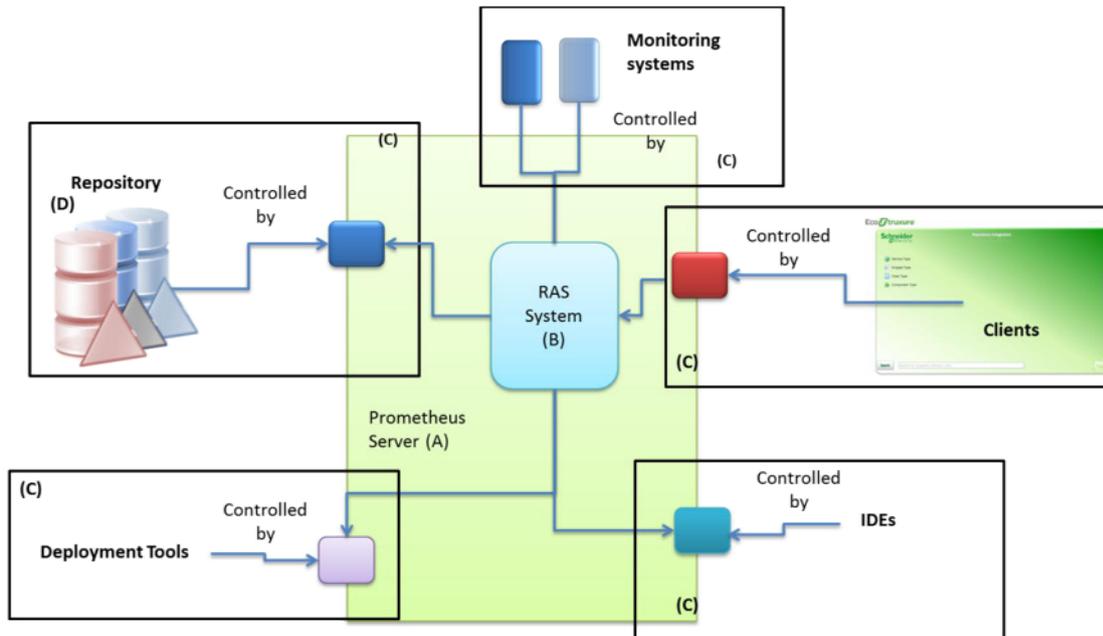


Figure 57 - Example of distribution of business logic for the concept-based environment

Figure 57 shows an example for the distribution of business logic. It shows that the concept-based server handles all requests from a client system. This is the central logic of the complete environment (A). All other elements in this environment are controlled by the server. The logic for handling execution of reuse is placed in the RAS (B) and is started by the server. Usually, an application controlled by a RAS includes the logic for the specific task (C), though it may be possible for a RAS itself to contain this logic and functionality. The logic to read and write knowledge within a specific repository is usually included in the repository system (D), but is initiated by the server.

4.5.2. Software unit model

One relevant element of this concept is the Software Unit Model. The aim of this model is to describe different concrete software unit types as abstract software units with concrete information.

Figure 58 shows this in the example of services, components, and classes. All three units of modelling have concrete properties (shown by using different shapes), but they are stored as a common software unit abstraction (triangle shape). This abstraction creates a common view on the different units and makes it easier to handle.

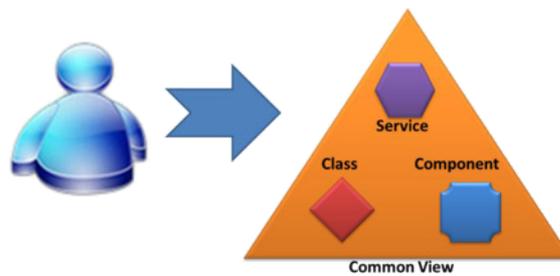


Figure 58 - Standardisation of the view on services, components, and classes

The aim of this model is to describe software units from the perspective of their usage in software unit reuse. Using this perspective, Figure 59 shows 4 different views.

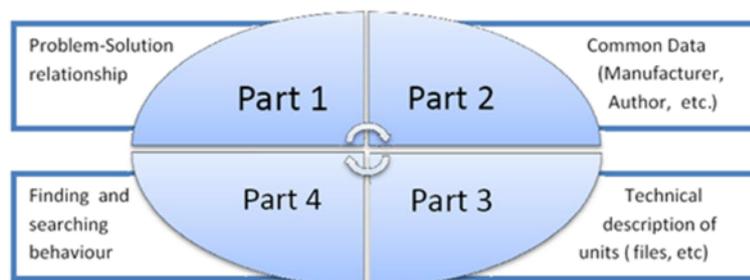


Figure 59 - Areas of the Software Unit Model

The model consists of four parts. Part 1 shows the ‘problem-solution approach’. Part 2 relates to ‘general business information’ about the solution (e.g., manufacturer, name, and author). Part 3 describes the solution as a technical unit (e.g., a type of unit, a technology, a file format, or

A general approach to realise knowledge-based automated reuse activities

files). In Part 4, the technical contents are described, thereby, explaining a (semantic) search approach that is discussed in a previous publication (See Zinn et al., 2010a). Here, the SCA types explained in the previous section are used to classify SCAs. If an instance of the model is generated (e.g., by the registration of a newly developed unit), the user has to specify information that is stored in the appropriate area of the model. The data may also be entered automatically into Part 3 of the model. This is possible as the technical data is generally detectable, such as by file size, file type, file name and technology. Nevertheless, the data from other sections of the data model is not automatically detectable. The model describes services, components, and classes in the same way and abstracts them into units (unit view). Based on this abstraction, the model will be extended by collection requirements of different use cases (views). Figure 60 demonstrates this relationship.

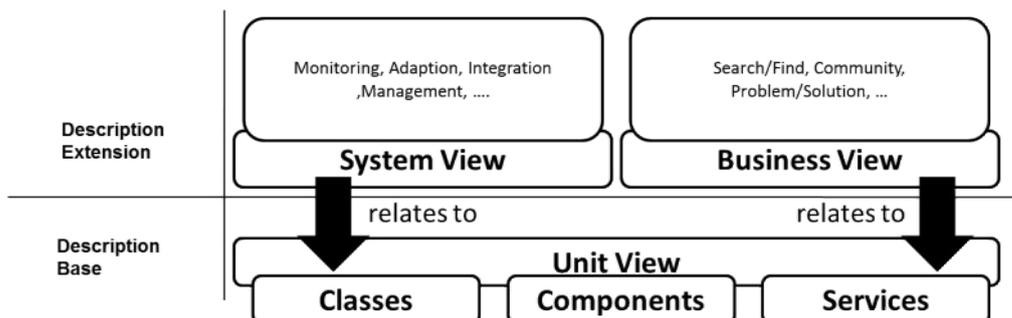


Figure 60 - Relevant views of the Software Unit Model

Figure 60 shows, for example, integration as a system view extension. Therefore, all reuse activity describes an extended system view on the model. This describes the relation of the model for the reuse activity to specific elements of the common view model.

Note: This thesis focuses on Part 3 of the described model. All other parts are also relevant and were analysed during the research study for this thesis, but for the focused approach, the technical perspective of this model is relevant. Therefore, only Part 3 and a special property of Part 4 will be discussed in this thesis. It is relevant to show the relation to the other parts so that

A general approach to realise knowledge-based automated reuse activities

the picture on software units is more complete. In Chapter 5, a possible realisation of these parts of the concept is shown.

4.5.3. Reuse activity models

In the concept description, the reuse activity models are relevant elements of the focused approach. These models are included and handled by the RAS. Figure 60 shows this as a Business View. From the technical perspective these models use information from the fundamental Software Unit Model and will be used by special plugins to perform a SCAC. Figure 61 shows the usage with the focus on the technical part of the Software Unit Model.

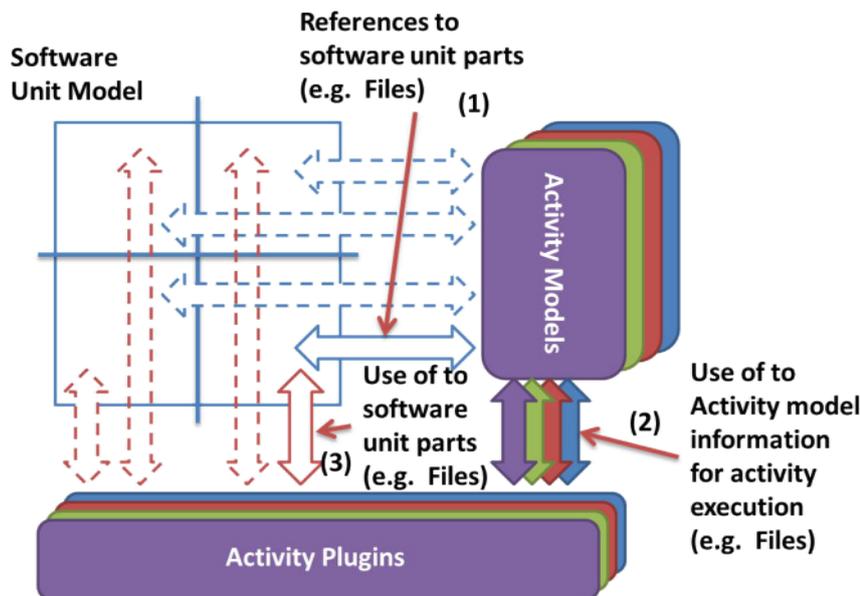


Figure 61 - Software Unit Model as fundamental information base for reuse activities

This plugin behaviour includes several technical advantages (cf. Figure 61):

- (1) The model technology used in instances of software reuse activity models refers information from an instance of the fundamental software unit model. This means the different models and plugin technologies have to be compatible. Typical model descriptions, as for example, UML may be used, also when focusing on the mapping of different activity models and the fundamental software unit models. Semantic models

may be used to extend models with meaning. Such semantic technology has the advantage that the wording of different domains may be connected to each other. Additionally, semantic models can be used for knowledge queries (see W3C, 2009). Also, it may be possible to connect some of the reuse activity models together.

- (2) The different models are used during runtime to perform SCAC. Thereby, the stored information is managed inside an instance of the focused approach. For that purpose, different scenarios exist; on the one hand, the information may be loaded in a runtime representation (e.g., object structure) on the other, it is possible to operate on the data using a database also in the case of (semantic) model reasoning. These approaches and the existing variants differ in their features (e.g., McCarey, Ó Cinnéide and Kushmerick, 2008).

This also leads to the question as to how the models may be stored. Typical examples are databases, where different database technology is required to exist. It is also possible to use description language such as XML. For semantic models, description languages like RDF or Web Ontology Language (OWL) (cf. W3C, 2004; W3C, 2009) are useful.

- (3) Another relevant point is the use of extensible application parts (e.g., plugins) to perform software reuse activities. Such parts use the information stored in the different instances of the activity models and in the related software unit model. The plugins use this information to perform related reuse activities. Plugins can be adapted or replaced by other plugins to extend SCAC functionality of a realised environment based on the concept.

4.5.4. Extensibility

An relevant requirement for the focused concept is extensibility. The growing amount of knowledge based on new technologies, concepts or processes in the area of software

development requires flexibility. Systems and humans have to be flexible to learn and be able to handle such new knowledge.

The focused concept has to be extensible relating several different points:

1. Fundamental Software Unit Model
2. Reuse activities models
3. Support of different existing repositories
4. Support of different development environments and tools
5. Support of different client applications

4.5.4.1. Fundamental software unit model

The Software Unit Model described in Section 4.5.2 aims to describe different software units in a common model. For the focused concept it is relevant that this model can be extended to describe other software units.

This may be reached by focusing on two different concepts for the software unit model. The model describes a software unit from an abstract perspective. This means a concrete software unit for example, a component or a service, is described as a generic software unit which has properties and sub parts. Due to each software unit having the same possible content, the mind-set given by the based software unit type is hidden. The information of the type is stored and may be used by the experienced user during the knowledge extraction or by the inexperienced user during the knowledge injection. The personal perspective that everything is a software unit abstracts and simplifies the handling of software units. Every new unit type, for example, code snippets, are only software units. All units are used in the same way.

This only works if the chosen model technology supports such common description, as well as the extension for new information types which were not part of the prior model.

4.5.4.2. Reuse activities models

Another relevant point from the perspective of extensibility is the extensibility of the system for reuse activities. As described in Section 4.5.3 such models may be built for different reuse activities. The idea behind this concept is that such models are extensions for the Software Unit Model (see Section 4.5.2). Figure 62 shows the extension concept.

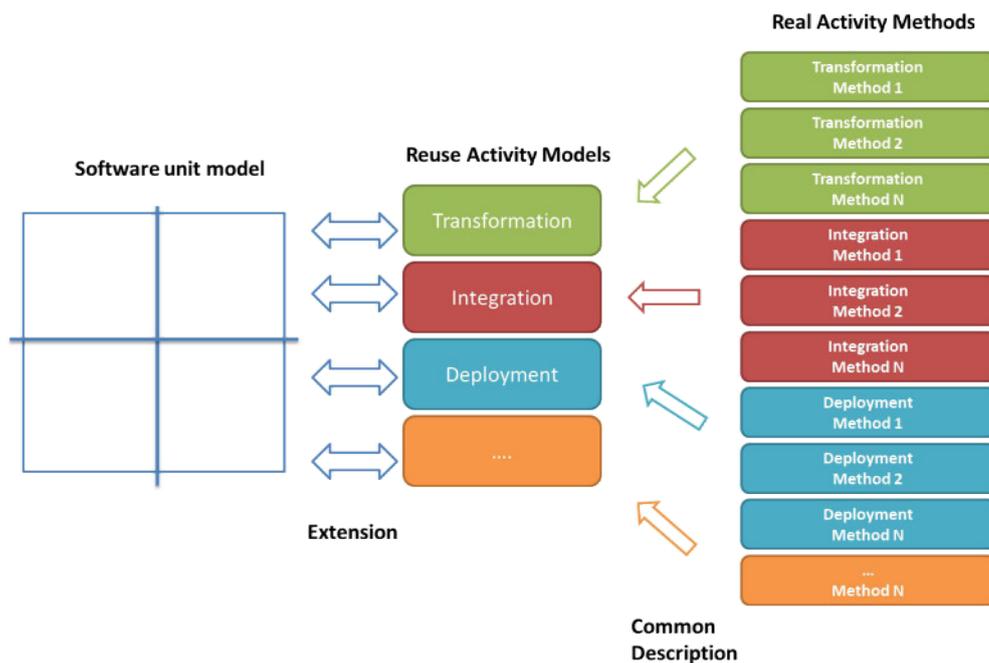


Figure 62 - Extension concept of reuse activity models

Figure 62 shows that the fundamental Software Unit Model will be extended by software reuse activity models. Each activity model describes a set of real activity methods, as for example, the transformation activity model describes several approaches on how to transform a software unit. From another perspective it is possible to say that the fundamental Software Unit Model will be used by activity models. The advantage of this methodology is that anybody should be able to add a new activity model or extend an existing model to handle a special type of activity.

Also, relevant from the perspective of extensibility is the capability to combine reuse activity models to reach a special aim. This is demonstrated by following example:

A general approach to realise knowledge-based automated reuse activities

An experienced user creates a Java-based software unit and stores it by using the focused approach. This experienced user also adds a transformation activity which transforms this Java-based unit into a .Net-based unit. Another user of this approach adds an integration activity rule to this transformed unit. This rule integrates the .NET unit into IDE Visual Studio.

The second user uses the result of a previous reuse activity. This flexibility may be used to build more complex processes based on combined reuse activities. This is seen as extensibility of the concept.

Eventually, it may be possible to create complete development processes for a software unit, but this is not covered by this research.

4.5.4.3. Support of different existing repository

Typically, software engineers are experienced users that work in their normal business and development environment. A problem may occur if this environment changes or if these people have to use other unknown environments. The use of new or other repositories is a particularly relevant part of a work between different teams. For these reasons, an relevant requirement for extensibility is the support of such a repository. The focused approach has to enable the integration of the existing repository.

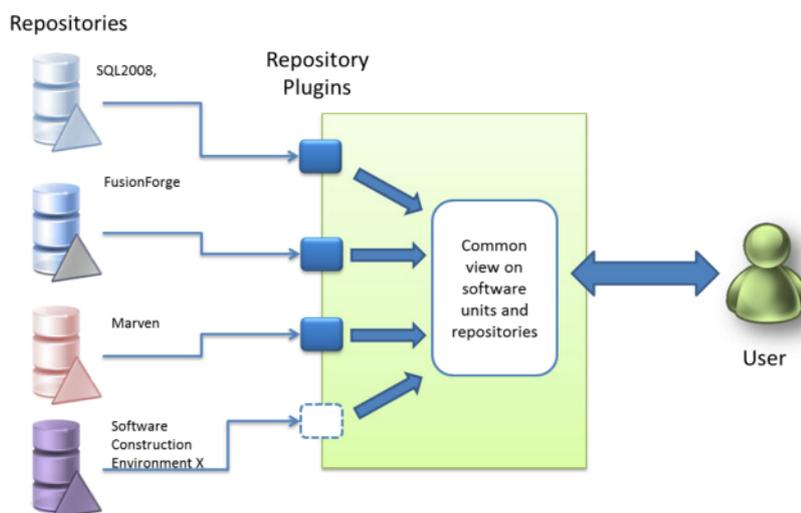


Figure 63 - Reducing view complexity on different repositories

A general approach to realise knowledge-based automated reuse activities

The user is not aware of the different repositories which are used by the focused approach; neither is the user aware of the different usage pattern of these repositories. The usage pattern of the focused approach has to remain the same for the user, and the different usage patterns of the different repositories have to be handled by the approach directly. Plugins take over the task of handling repositories. The concept-based environment is responsible for creating a common view for the user on software units in the repositories and the repositories themselves. Figure 63 shows this relationship.

The problem of different usage patterns does not only come from different technologies used by these tools. It is also possible that companies require different processes for handling repositories. A typical example may be the quality check of a software unit before it is deployed to a repository.

The focused approach has to be able to handle such regulation. This does not mean that the approach has to take into account all existing business processes around a software unit, but it is relevant that a user of the focused approach will at least be informed about the possibilities of software unit management. If a repository does not support the storing of a software unit, or the technical aspect of the focused approach is not able to handle this, the user needs to be informed about this situation. Another example may be the security issues. If the focused approach is not capable of downloading information because special security information is necessary (e.g., user name and password) the user has to be informed.

Repositories without an interface for automation use, pose a special challenge. In this case special methodologies have to be developed (see Section 5.2.4) or the focused approach will not be able to handle such repositories. An example of such would be repositories with a user interface that is only designed to be handled by human users.

Another concept requirement is that the server reacts also as a repository. So an instance of this approach may be used in another case. Figure 64 shows this concept. Here, the 'Environment X' is another environment based on the concept described in this chapter.

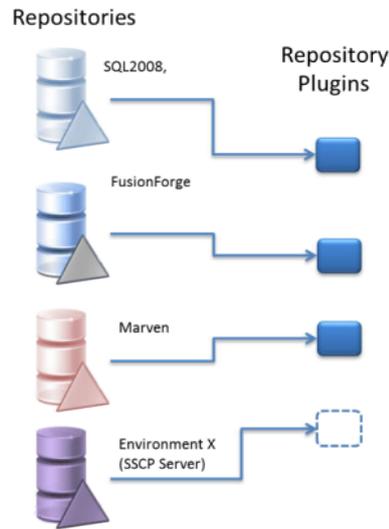


Figure 64 - Concept environment as repository

4.5.4.4. Support of different development environments and tools

The focused approach has to be able to work with different development environments and tools. The kind of work depends on the task described in the reuse activities model, which in turn depends on the software unit model. As discussed in Section 2.2.2.3, the number of different tools is growing. Therefore, the support of the tools becomes problematic. The following discussion describes the challenge of this support for the topic of expandability.

Additionally, if a user also uses Visual Studio and Eclipse, the focused approach has to be able to handle both of these IDEs. In addition, it is possible that these IDEs provide different ways to do one task, for example, the integration of a software unit into a development project. The focused approach has to provide a way or a methodology so that such different IDEs and other possible variants are manageable.

Next to the IDEs - which are typical tools of a software engineer's developing software - there are other tools which exist. Such tools may also be required in a reuse activity and should be made usable by the focused approach (see transformation and deployment SCAC example in Section 3.1).

A general approach to realise knowledge-based automated reuse activities

In a similar way to the support of different existing repositories, this requirement has the problem of different IDEs having different behaviours and technology restrictions. At the very least it is relevant that a user of the focused approach is informed about the possibilities of handling the different tools. Figure 65 demonstrates an environment which would need to be handled by the focused approach.

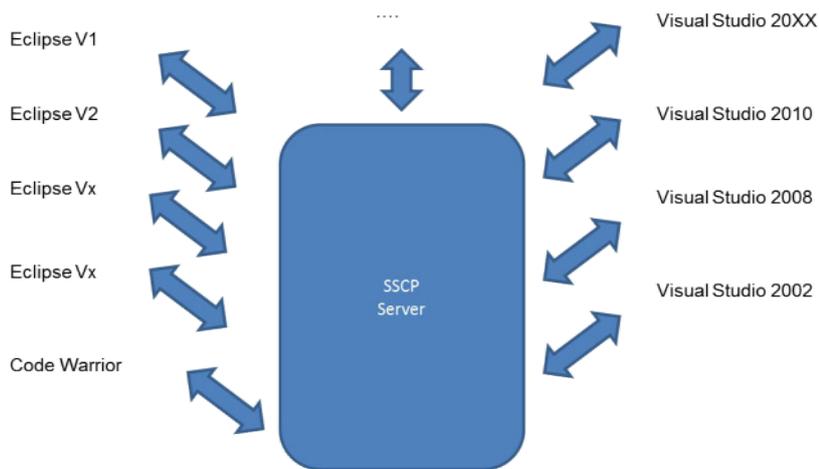


Figure 65 - Typical development environments

From the perspective of expandability, the focused approach has to be able to interact with existing IDEs and other tools used in software reuse activity, as well as being able to interact with new tool approaches. This expandability is handled by the plugin system (cf. Figure 57).

4.5.4.5. Support of different client application

The extensibility is not restricted to the direction of tools used by the focused approach, but also includes the tools using this approach. In basic terms, a user interface should be used to interact within this approach. The user interface being a standalone application or having been integrated into a development environment is not relevant for the purpose of this thesis (cf. Section 4.5.1.1); in general terms, the intent of this thesis is not to create a new tool. From this point of view an integrated user interface should be more focused. It may also become

interesting to use the approach automatically by other systems instead of humans, but this is not covered by this research.

The described concept of a service interface (see Section 4.4.1) is an relevant factor of being more extensible. Different user interfaces or systems may be built using this interface. So the interface is independent from technical conditions, as for example runtime environments or non-technical factors, like corporate identity. As a result, the focused approach is extensible by the number of client systems using the approach. Figure 66 summarises the potential client systems as for example integrated, web, desktop, and mobile clients.

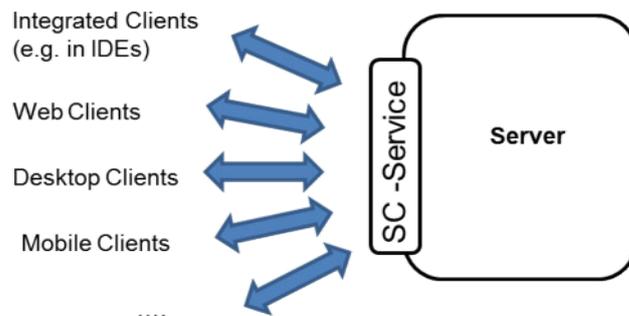


Figure 66 - Multiple client system using the same service of the focused concept

4.6. Summary

First of all, this chapter presents the basic idea of the focused approach of this thesis. The idea of this approach is the storage and reuse of software construction activity knowledge. The aim is to support users who do not have sufficient knowledge to perform a specific reuse activity of specific a software unit.

In general, different user types may utilise this idea. In this work, software engineers are focused on those who want to use smaller software units in their software development projects. At the very least, experienced software engineering users are focused on in the approach which assists the inexperienced in handling a specific software unit.

After putting forward the basic idea, the concept is more fully explained in this chapter. The concepts described follow two elementary parts of the solution approach. The first one is a

A general approach to realise knowledge-based automated reuse activities

common Software Unit Model describing different existing software unit concepts as a general software unit. This model is extended by different software construction activity models, describing the SCAC information required by using the Software Unit Model information in a specific reuse activity. The second one is a service-oriented environment providing a service for (re)use functionality (i.e., storing, distributing, and execution) based on the information in the models.

Using these two parts, the problem area will be handled as follows: The service can be used to store information as knowledge using the different models. This includes the storage of different software units and relevant software construction knowledge as information. This should handle the problem of the variations of technologies.

The service also hides the environment for knowledge distribution. An inexperienced software engineer has not to know this environment (e.g., server location, etc.) and, therefore, a limitation of this problem area is expected by the concept.

The last and most relevant solution approach is the service executing software construction activity. After an experienced software engineer enters the necessary SCAC information into the service-oriented environment, an inexperienced user is able to perform this without the knowledge an experienced user would need for the execution of this SCA without this environment. With this approach, it can be expected that an inexperienced software engineer is able to perform a software construction activity independently outside of their current knowledge level.

In Chapter 5 one concrete instance of the concept is explained which is used for the case study in Chapter 6.

5. Solution realisation

This section describes an instance of a concept described in the previous chapter. Therefore, the realisation of the used architecture, technologies, interfaces, and their usages will be described.

A relevant part of this chapter is the description of the used models to store software units and software construction activity information. This extends the concept description in Chapter 4.

To differ from the concept described in Chapter 4, the realisation is called ‘Prometheus’.

5.1. Development approach

The software shown in this chapter is the result of a development lifecycle conducted during the PhD research. In the following, the scenes are described to outline the development approach.

The first scene is the development of the proof of the concept application. This was done in the first year of research. The aim was to create a simple application showing that the basic idea of the research topic was realisable in a software application. The created application was built in a rapid development procedure model with no focus on stability, full functionality or error handling. It supports only the integration of information into a Visual Studio 2008 instance. In 2007 this application was shown at an internal academic conference (SEIN 2007).

In 2008 the creation of a second version of this tool was started. Following a prototype procedure model, the first prototype was analysed for positive and negative behaviour, but it was dismissed and the second prototype was built from scratch. Based on the experience developed in the research, this prototype includes several topics:

- Integration into IDEs
- Transformation
- Deployment
- Service provision

Development approach

This prototype changed several times, so a waterfall development model was used beginning with the topic of integrations. This feature was created and then tested. Each time errors or new research results were identified the prototype was adapted. The same procedure was made for the other topics (transformation and deployment). Important changes during the development were the integration of the (SCAc-) service and the integration of SCAc data models. The tests for the functionality were different, and most tests were done by the author of this thesis. Additionally field tests were conducted. Thereby, other software engineers used special features and gave feedback.

This prototype was presented in 2009 at a relevant German fair (Cebit 2009), in several PhD meetings at the Darmstadt University of Applied Sciences and in meetings with external companies (i.e. Schneider Electric Automation GmbH and Engineering Methots AG). The development of this prototype took about 2 years. As the first prototype, it was one piece of software monolithic architecture).

In 2010 the last version of the prototype was created. The previous version of the second prototype was analysed for positive and negative effects. Two reasons led to the development of a third prototype.

The first one was the fact that the service provision concept can be realised independently from the UI interface. So the decision was made to create plugin architecture to host different service technologies to connect different UIs. As a result, three different UIs were created: a Desktop client, a Visual Studio/Eclipse integrated UI and a Webpage based UI. The last was used for the case study. The second reason is the negative monolithic behaviour of the second prototype. Changes have side effects on other functionality and the deployment was circular. Hereby, a plugin and interface architecture were used in the third prototype.

During the test phase of the case study, only error fixes were conducted. The deployment SCAc functionality was not used in the case study, but in other investigations (see Zinn *et al.*, 2012a). Therefore it was adapted and errors were fixed.

5.2. Selected technical environment

5.2.1. Distribution model and relevant architecture elements

The used Prometheus environment uses a common architecture with three layers: client, middleware (server), and database (repositories) (see Figure 67). The communication between the individual layers is realised on the basis of a service-oriented architecture. This architecture is used to cover four different user scenarios. The individual layers, the technical implementation, user scenarios, and possible alternative implementation are described as follows.

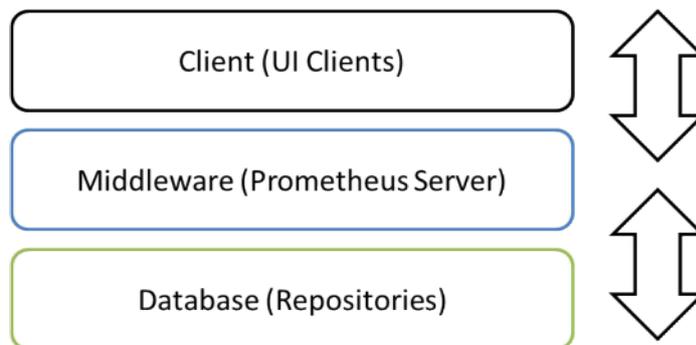


Figure 67 - Prometheus architecture overview

5.2.1.1. Layer 1 - Client

The Prometheus architecture distinguishes between three different types of clients:

User Client: This client type is defined for the two user profiles KU and KC (see Section 4.2).

The user interface described in this chapter corresponds to this client type. The client uses a special interface to the middleware layer to communicate. The following functionalities of the middleware can be used:

- Search: search software units / reuse activities
- Update: update software units / reuse activities
- Add / Remove: adding and deleting software units
- Add / Remove transformation rule: add or delete a transformation rule of a software unit

Selected technical environment

- Execute transformation rule: performing a transformation rule of a software unit
- Add / Remove integration rule: adding or deleting an integration rule of a software unit
- Execute integration rule: implementation of an integration rule of a software unit
- Add / Remove deployment rule: adding or deleting a deployment rule of a software unit
- Execute deployment rule: implementation of a deployment rule of a software unit

As part of this work, the client was implemented using Silverlight technology from Microsoft (2012d/e) and uses a SOAP client to communicate with the middleware (Prometheus server) by using SOAP Web Service.

Note: Because the middleware should be able to offer a variety of communications technologies (to reach the extensibility requirements of Section 4.5.4), the clients may also be able to use various communication technologies. As part of the research for this thesis, four different user clients have been developed: desktop, an add-in for Visual Studio 2008-2010, an add-in for Eclipse, as well as a Silverlight client. The clients were each produced as a further development and presented at different events (see Acknowledgements). Each of the clients uses SOAP-based communication.

In the realised Prometheus environment, three different clients exist: Integration Client, Transformation Client, and Deployment Client (based on the focused SCA described in Chapter 4).

Integration Client: This client type describes applications that are able to integrate software units in development environments. This is only possible for those development environments that offer other programs appropriate interfaces. In contrast to the User Client, this client type is not a sender of messages. Rather, they are contacted by the middleware to integrate software units by performing an integration SCAC. For this application, this client type offers an interface to provide the following function:

- Integrate: integration of a software unit

Selected technical environment

The Integration Client has also been developed using soap-based web services, .NET and Java technologies.

Transformation Client: This client type is capable running batch-based transformation applications. The middleware sends input parameters to clients for the execution of a transformation SCAC. The corresponding client performs this transformation by using these parameters and sends the result (transformed software unit) back to the middleware. The client offers an interface to provide the following function:

- Transformation: transformation of a software unit

In this work, this client type is realised using a SOAP-based Web Service and .NET technologies.

Deployment Client: This client type deploys software units into embedded devices. Therefore, it controls other applications to perform the deployment (cf. SCAC deployment example in Section 3.1.6). This is similar to the transformation client. The middleware sends deployment SCAC information to the client. The corresponding client performs this deployment by using parameters for different deployment applications. Both client types - deployment and transformation - are able to display manual orders as text to the user. In a case of deployment, this could be necessary (e.g., to switch a device on/off for manual restart).

As the Transformation Client, the interface has also been developed using a SOAP-based web service and .NET technologies.

5.2.1.2. Layer 2 - Middleware

Prometheus server: The core of the Prometheus environment is a communications infrastructure that enables communication between the various elements of the middleware by using predefined interfaces. The server includes also the logic for the RAS system. This is explained in Section 4.5.1. The service is realised as a single application using different plugins

Selected technical environment

that are part of the application instance. Figure 68 shows an overview of the server infrastructure.

Note: The complete environment (including plugins) was created by the author of this thesis.

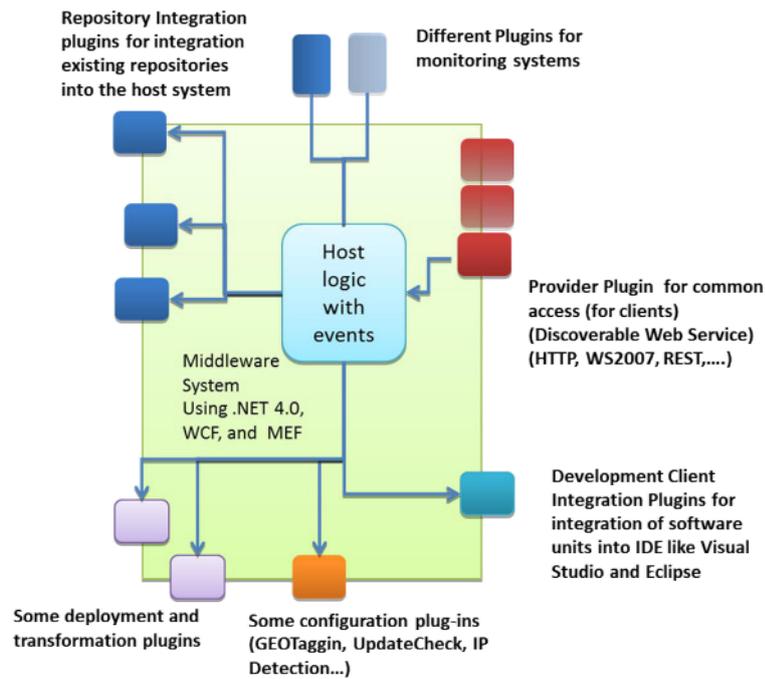


Figure 68 - Overview of Prometheus server architecture

The different plugins as shown in Figure 68 for repository integration, deployment tasks, server configuration, IDE client integration, user client management, and monitoring tasks have the basic task of sending information to the core of the Prometheus server or receiving information from the server and passing it on to external communication partners.

Selected technical environment

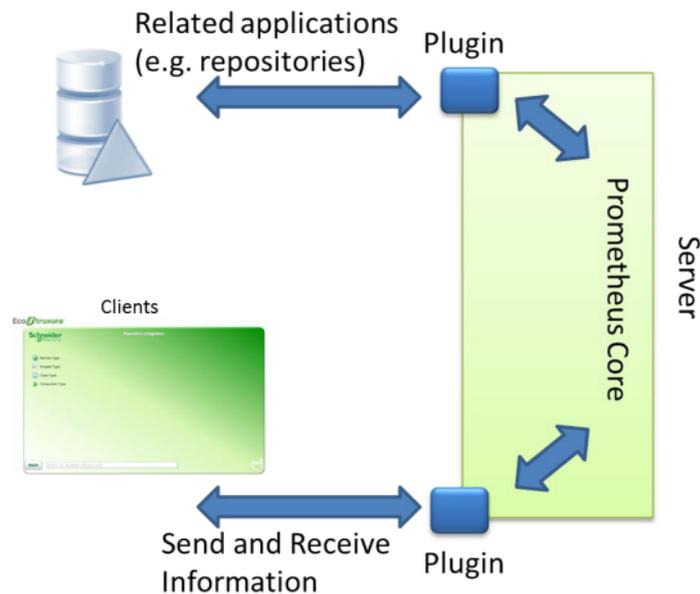


Figure 69 - Information flow of the Prometheus core

Figure 69 indicates this by demonstrating the flow of information described in general terms. The following example illustrates this relationship.

Example ‘Search Example’: A user enters a query into a user client and sends it to a Prometheus server. A User Client plugin receives this search request. The plugin converts the inquiry to ensure that it meets the client user interface requirements of the Prometheus core, and sends them on to the core. The Prometheus core forwards the request to the connected plugin (i.e., repository plugins). The repository plugins convert the query of the core and perform a search in the connected databases. The results of different searches are, by the appropriate database plugins in the format that the core plugin database interface is defined, transformed, and transmitted to the core. The core forwards the result back to the calling User Client Plugin by using the client user interface. The User Client Plugin transforms the result into the format that is defined by the user client and User Client Plugin. Figure 70 shows the communication behaviour of this search example including the user.

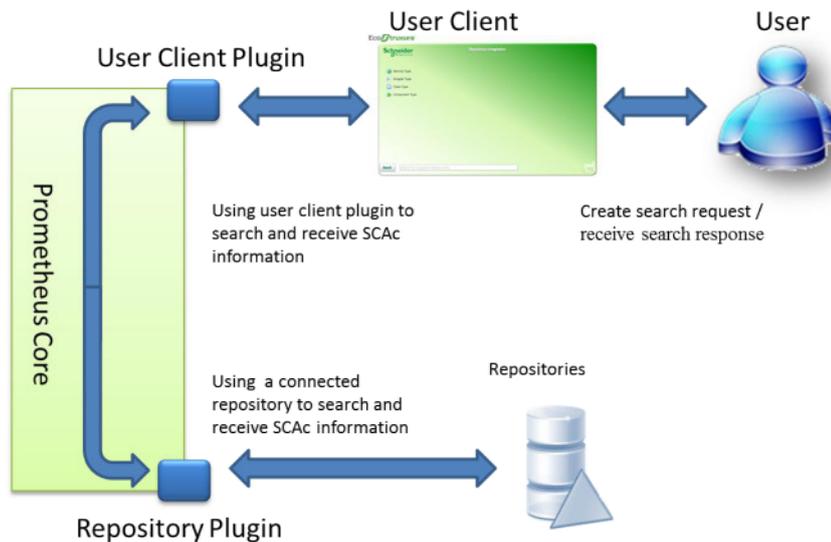


Figure 70 - Communication behaviour of a search request

The individual plugins, interfaces, and the communication relationship of the Prometheus core are presented as follows.

Prometheus Plugins: Plugins can, in this approach, be divided into user client, reuse activity (i.e., transformation, integration, and deployment), and repository. Figure 68 shows additional monitoring and reporting plugins. However, these are not relevant to the investigation and were only used in the context of the experiment (see Chapter 6).

Each plugin follows the communication structure shown in Figure 69. This results in the following distribution scenario for each plugin (cf. Sections 4.5.1.4):

1. **Absolute distribution:** In this scenario, plugins and their communication partners (Prometheus core and plugin system), are different instances. This allows distributed architecture on different physical systems. This is not used in the focused Prometheus environment.
2. **Relative distribution:** In this scenario, plugins and their communication partners (Prometheus core and plugin system) are part of the same application instance. This is not used in the focused Prometheus environment.
3. **Mixed distribution:** In this scenario, plugins and their communication partners (Prometheus core and plugin system) are differently interconnected. This is used in the

Selected technical environment

focused Prometheus environment for most of the core plugins. Two logical variations are possible:

1. Prometheus core and plugin form an instance together. In this scenario, the external tool (e.g., repository) and the Prometheus elements are installed on different systems or on separate applications (see Scenario 1 Figure 71).
2. External tool (e.g., repository) and a related plugin create an instance together. In this scenario the Prometheus core and the plugin are installed on different systems or separate applications (see Scenario 2 in Figure 71).

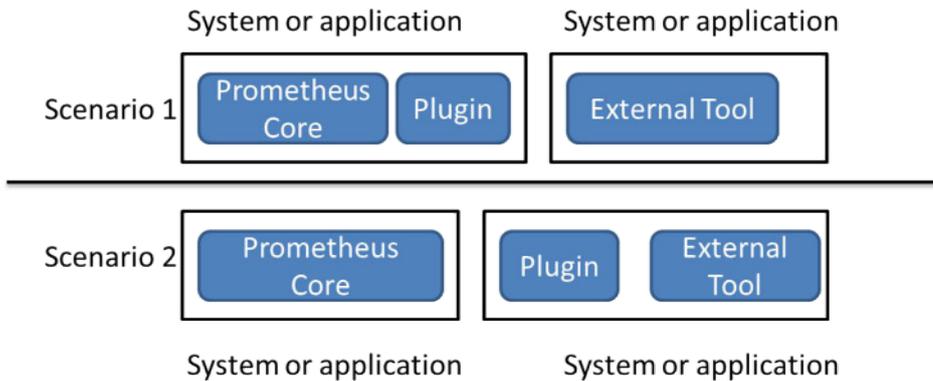


Figure 71 - Distribution possibilities of the used Prometheus architecture

The different distribution scenarios used are illustrated by concrete examples in the following plugin descriptions:

User Client Plugins (UCP) – UCPs have one area of responsibility

1. Receive and transmit information from or to user clients.

Basically, the interface and the communication protocol between a user and a UCP client application are not determined by the SSCP approach. UCPs can use any compatible interface and any compatible communication protocol. This enables the integration of other protocols or technologies. As part of this work a SOAP-based Web Service is defined, and includes the capability of bidirectional (synchronously and asynchronously) information exchange. There

Selected technical environment

are additional Web Service eventing mechanisms integrated into the Web Service. All operations can be used as synchronous and asynchronous web services calls.

The used interfaces between Prometheus core and the used User Client Plugin are defined in Section 5.2.2.3. User client plugins are used for the knowledge injection (i.e., creation/mapping of software units/SCAs) into the Prometheus environment. This refers to the following tasks: Software Construction Artefact Injection, Additional Software Construction Content Injection and Activity Reuse Knowledge Creation (cf. Section 4.4.2).

Transformation Client Plugins (TrCP) – TrCPs have two areas of responsibility

1. Receiving and transmitting information from or to transformation clients.

The interface and the communication protocol between the Transformation Client Plugins (TrCPs) and the transformation clients were not determined by the focused approach. This is to guarantee a higher extensibility by enabling the integration of different protocols and technologies. Also, the number of different applications is high (see Chapter 1). As a part of this work, the different transformation applications are executed directly by the transformation plugins. In contrast to the web service calls of the User Client Plugin, a Transformation Plugin has to be aware of the existing file structure and correct parameters for the transformation application (see Section 3.1.1). In addition, there may be a need to clean this file structure after the transformation is complete.

2. Receiving and transmitting information from or to the Prometheus core

In contrast to the communication with the transformation application, the communication with the Prometheus core is defined in detail. The TrCP receives transformation reuse activity information from the Prometheus core. Based on this information it prepares and executes the transformation. These tasks require the interpretation of the formatted reuse activity information into the special protocol of the transformation application. The transformation

Selected technical environment

result is also transformed into the given communication protocol of the Prometheus core and is sent back to the core.

The used interface definition between the Prometheus core and the used Transformation Client Plugin is shown in Section 5.2.2.5. This type of plugin is used for the knowledge injection into the environment of the inexperienced user. This refers to the task of Activity Reuse Knowledge Injection (cf. Section 4.4.2).

Integration Client Plugins (ICP) – ICPs have two areas of responsibility

1. Receiving and transmitting information from or to integrated clients.

In essence, the Integration Client Plugins (ICPs) follow the same procedure as the Transformation Client Plugins. They differ by handling different applications and information; in the Prometheus environment, the focused applications are Visual Studio 2008, Visual Studio 2010, as well as Eclipse (Juno). The protocol and technology used between the ICPs and these three IDEs are specified by the focused approach; it is based more on the provided communication possibilities of the IDEs. In the case of both Visual Studio versions, the Visual studio COM technology was used. In the case of Eclipse, an Eclipse plugin was written providing a SOAP-based Web Service. This structure is shown in Figure 72.

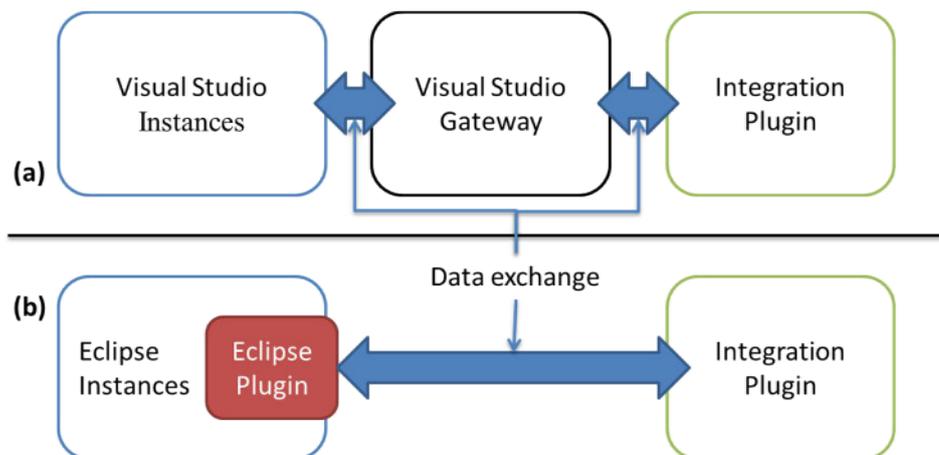


Figure 72 - Integration plugins for Visual Studio and Eclipse

Selected technical environment

Figure 72a includes the ICP for Visual Studio which serves a special gate way tool developed to handle Visual Studio instances. Figure 72a shows a ICP for eclipse serving directly eclipse instances. For the research an Eclipse plugin was developed for communication with the ICP.

2. Receiving and transmitting information from or to the Prometheus core.

The Prometheus core sends integration reuse activity information as well as related software unit information to the specific integration plugin, these plugins then forward this information to the Visual Studio Gateway or to the integrated Eclipse plugin. The result – which in this case is a more or less empty error list – is forwarded back to the Prometheus core.

The used interface definition between Prometheus core and the used Integration Client Plugin is shown in Section 5.2.2.5. This type of plugin is used for the knowledge injection into the environment of the inexperienced user. This refers to the task: Activity Reuse Knowledge Injection (cf. Section 4.4.2).

Deployment Client Plugins (DeployCP) – DeployCPs have two areas of responsibility:

1. Receiving and transmitting information from or to deploy clients.

Deployment reuse activities have two properties of interest; the first is where more than one application is used in the deployment process. For this reason, DeployCP uses multiple deployment applications. The second property is the required manual support by the user during the deployment process (see Zinn et al., 2012a). Similar to other client plugins, the protocol and communication technologies are not defined by the focused approach (for extensibility reasons).

2. Receiving and transmitting information from or to the Prometheus core.

The Prometheus core sends deployment reuse activity information as well as the required software unit artefacts to the Deployment Client Plugin. This plugin creates and executes the deployment process. In contrast to other client plugins, this plugin type is able to send user

orders to the Prometheus core. This may be necessary because some embedded devices have to restart for deployment to proceed, but this might have to be done by turning the power off and on. The result of the whole deployment process – in this case, a more or less empty error list – is forwarded to the Prometheus core.

The used interface definition between the Prometheus core and the used Integration Client Plugin is shown in Section 5.2.2.5. This type of plugin is used for the knowledge injection into the environment of the inexperienced user. This refers to the task: Activity Reuse Knowledge Injection (cf. Section 4.4.2).

5.2.1.3. Layer 3 – Database

Repository Client Plugins (RCP) – RCPs have two areas of responsibility:

1. Receive and transmit information from or to repository clients.

Similar to other plugins, the communication protocol and technology is not regulated by the focused approach. An RCP has to react to the given communication protocols of the different repository systems (see Section 5.2.3). The task of this plugin is to read and write information to the repositories. A special feature in this implementation is that RCPs have to use a special approach-based repository. This includes RCPs that are used to connect to a special reuse activity repository (using the special reuse activity models, see Section 5.3). Also, a database is needed to store software units based on the fundamental software unit model, for example to store the result of a transformation SCAC. This kind of repository is also connected by using an RCP.

2. Receiving and transmitting information from or to the Prometheus core.

The RCPs receive read and write requests sent by the Prometheus core. These requests and the responses are well defined by the Prometheus environment. The RCP uses this information to read/write data from/to a connected repository. The protocol information is, therefore, translated into the special database language (e.g., SQL).

Selected technical environment

Contrary to other client plugins, RCPs use a subscription approach, based on the Web Service Eventing protocol (W3C. 2006). Using that mechanism as a basis, the plugins are able to send requests to the Prometheus server (e.g., to require additional information or send information to other parts of the server).

The used interface definition between Prometheus core and the used Repository Client Plugin is shown in Section 5.2.2.3.

5.2.2. Interface definitions

In the previous section, the different elements of the Prometheus architecture were shown. In the following, the interfaces which are used by this architecture will be defined.

The Prometheus environment uses 11 interfaces (see Figure 73). Three interfaces (I1 – I3 a, b, c) are the relevant core interfaces, and 8 interfaces are used between the plugins and other systems. As mentioned in the previous section, these 8 interfaces are not defined by the focused approach. The interface operations use the ID-driven approach described in Section 4.5.1.3.

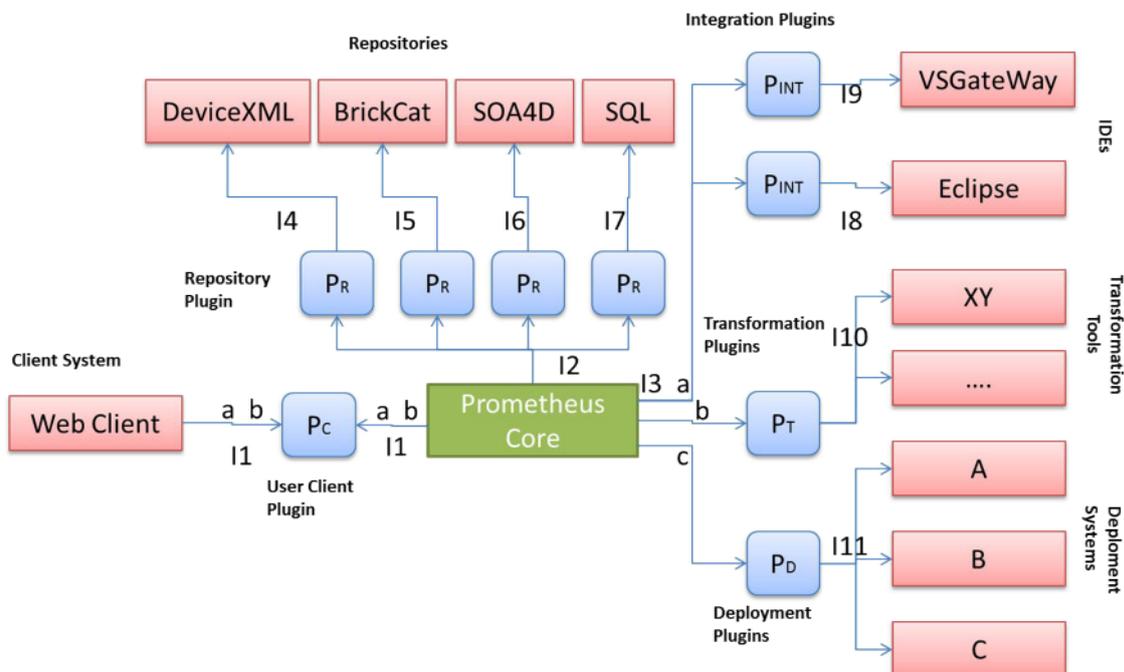


Figure 73 - Relevant interfaces in the Prometheus architecture

Selected technical environment

In the following sections, the different operations of used interfaces will be described.

Therefore, the interfaces are grouped (see Table 6).

Interface Group	Contained Interfaces
User Client Plugins interfaces	I1 a/b
Reuse Activity Interfaces	I1 a/b
Repository Client Plugin Interface	I2
Reuse Activity Plugin Interfaces	I3 a/b/c
Repository Client Interfaces	I6, I4
Integration Client Plugin	I8, I9
Non-Interface Types	I5, I7, I10, I11

Table 6 - Used interface groups

5.2.2.1. Knowledge user related interfaces (I1a)

```
public interface IService1 : IServiceMethods
{
    List<Artefact> Search(List<Guid> guidPath, Guid[] repositoryIDs, string searchword, ArtefactType[] artefactType, UOMType[] uomType,
        SearchFieldType[] searchFieldType, bool useAttributeSearch, List<string> artefactCategories, List<string> uomCategories);

    byte[][] GetItems(List<Guid> guidPath, List<Guid> fileIDs);

    byte[] GetItemsAsZip(List<Guid> guidPath, List<Guid> fileIDs);

    bool PerformTransformation(List<Guid> guidPath, Guid transformationActivityID);

    bool PerformDeployment(List<Guid> guidPath, Guid deploymentActivityID);

    bool PerformIntegration(List<Guid> guidPath, Guid integrationActivity, Guid receiveClient);
}
```

Figure 74 - Relevant interfaces of the User Client (UC)-Plugin

Figure 74 shows the relevant 6 interfaces in the area of user interfaces: Search, GetItems, GetItemsAsZip, PerformTransformation, PerformIntegration and PerformDeployment. These interfaces form operations that are necessary for the Knowledge User Profile (see Section 4.2). There are also other operations to manage (create, update and remove) software construction artefacts, software units, and reuse activities. These methods are listed in the next section.

The search operation allows the client to provide user search queries. The parameters listed in Table 7 are required:

Selected technical environment

Search		
<i>Type</i>	<i>Name</i>	<i>Description</i>
List<Guid> = Globally Unique Identifier	guidPath	Includes the path (serverID, repositoryID) to be searched for a Software Construction Artefact, Software Unit, or SCAc.
Guids[]	repositoryIDs	Additional IDs of repositories to be searched for by a Software Construction Artefact, Software Unit, or Reuse Activity.
String	searchword	The entered search term. (May contain multiple words separated by commas)
SearchFieldType[]	searchfieldTypes	Includes the search fields, (e.g., search in file names, metadata, description, topics, etc.).
UOMType[]	uomTypes	The types of UOM (Data, GUI, Function, Structure).
List<String>	uomCategories	Freely defined categories for software units (need support by the databases).
List<String>	artefactCategories	Free defined categories for software construction artefacts.
SearchOperation[]	Searchoperation	Type of search types (Free text, Attribute or semantic search (need support of the repository plugins and/or the repository itself).
List<Artefacts>	-	The return value are artefacts including UOMs fitting the search attributes.

Table 7 - Parameters of the search operation

The call returns a set of artefacts that meet the given requirements. The next operation in this area is the *GetItems/GetItemsAsZip* operation. These allow the caller to get complete software units (including all available information) or parts of it from the UCP. The parameters listed in Table 8 are required:

GetItemsAsZip/ GetItems		
<i>Type</i>	<i>Name</i>	<i>Description</i>
List<Guid>	guidPath	Includes the path (serverID, repositoryID, artefactID, uomID) to identify the correct UOM.
List<Guid>	fileIDs	The IDs of the files, which should be included in the operation response.
List<FileInformation>		Return value includes one or more file information (e.g., name, byte code, creation date.). In the case of <i>GetItemsAsZip</i> this includes one zip file

Table 8 - Parameters of the *GetItems/GetItemsAsZip* operation

Both operations return the binary information of a software unit. The operation *GetItems* returns this as a list of binary values as well as returning the information of the files; each value representing a file. The operation *GetItemsAsZip* also delivers the same information, but packed in a single ZIP file.

Selected technical environment

The next relevant operations are the reuse activity operations *PerformTransformation*, *PerformIntegration* and *PerformDeployment*. All three operations need typical and specialised information. Typically, the ID of the server (*serverID*), the repository (*repositoryID*) and the artefact (*artefactID*) - where the software unit is based on – is given. In the case of the *DoTransformation* operation the ID of the transformation rule is also given. The parameters listed in Table 9 are required:

PerformTransformation		
Type	Name	Description
List<Guid>	guidPath	Includes the path (serverID, repositoryID, artefactID, uomID) to identify the correct UOM which includes the transformation activity.
Guid	transformationActivityID	The ID of the transformation activity which should be performed.
List<TransferTypes>	-	The return value includes a list of UOM artefacts.

Table 9 - Parameters of the *PerformTransformation* operation

The result is a set of *transfertypes* which include the new software unit (see Section 5.3.2). In the case of *PerformIntegration*, the three IDs are also given. In addition, the ID of an integration rule has to be set as well as the information about the service endpoint which defines the external integration tool (i.e., Visual Studio Gateway or Eclipse Plugin (see ICP in Section 5.2.2)). The parameters listed in Table 10 are required:

PerformIntegration		
Type	Name	Description
List<Guid>	guidPath	Includes the path (serverID, repositoryID, artefactID, uomID) to identify the correct UOM which includes the integration activity.
Guid	integrationRuleID	The ID of the transformation activity which should be performed.
InetgrationClient	Client	Includes service information to connect to the IDE.
List<Errormessages>	-	The return value includes a list of error message for the user.

Table 10 - Parameters of the *PerformIntegration* operation

Selected technical environment

The next reuse activity operation contains the two typical IDs: the ID of a deployment rule as well as the necessary communication address. The method returns a list of errors (*List<ErrorMessage> Errormessages*). The parameter and return values are listed in Table 11:

PerformDeployment		
<i>Type</i>	<i>Name</i>	<i>Description</i>
List<Guid>	guidPath	Includes the path (serverID, repositoryID, artefactID, uomID) to identify the correct UOM which includes the deployment activity.
Guid	deploymentRuleID	The ID of the transformation activity which should be performed.
String	communication	The address where the device can be found (e.g., a service endpoint address).
List<Errormessages>	-	The return value includes a list of error message for the user.

Table 11 - Parameters of the *PerformDeployment* operation

5.2.2.2. Knowledge creator related interfaces (I1B)

These interfaces are used by experienced users (Knowledge creator; KC profile, see Section 4.2) who wish to add information or knowledge. Figure 75 shows the interface used in this work, which includes 14 operations.

```

Artefact CreateArtefact(List<Guid> guidPath, int artefactType, string artefactName, string artefactDescription, MetaInformation artefactmetainformation);
Artefact UpdateArtefact(List<Guid> guidPath, int artefactType, string artefactName, string artefactDescription, MetaInformation artefactmetainformation);

UOM CreateUOM(List<Guid> guidPath, int uomType, string uomDescription, string uomPath, MetaInformation artefactmetainformation);
UOM UpdateUOM(List<Guid> guidPath, int uomType, string uomDescription, string uomPath, MetaInformation artefactmetainformation);

FileElement AddData(List<Guid> guidPath, Guid packageID, string name, string fullName, string extension, string directoryName, DateTime dateTime, long size,
byte byteContent, int humanSalecolor, int contentSelector, bool IsUpdate, string relPath, string orgPath);
List<Artefact> Search(Guid repositoryIDs, string searchword, ArtefactType artefactType, UOMType uomType, SearchFieltype searchFielType,
bool useAttributeSearch, List<string> artefactCategories, List<string> uomCategories);

List<IntegrationActivity> LoadIntegrationActivity(List<Guid> integrationActivityIDs);
bool RemoveIntegration(Guid ID);
bool CreateIntegration(List<Guid> GuidPath, List<IntegrationActivity> integrationactivities);
bool UpdateIntegration(List<Guid> GuidPath, List<IntegrationActivity> integrationactivities);

List<TransformationActivity> LoadTransformationActivity(List<Guid> transformationActivityIDs);
bool RemoveTransformation(Guid ID);
bool CreateTransformation(List<Guid> GuidPath, List<TransformationActivity> transformationactivities);
bool UpdateTransformation(List<Guid> GuidPath, List<TransformationActivity> transformationactivities);

List<DeploymentActivity> LoadDeploymentActivity(List<Guid> deploymentActivityIDs);
bool RemoveDeployment(Guid ID);
bool CreateDeployment(List<Guid> GuidPath, List<DeploymentActivity> deploymentactivities);
bool UpdateDeployment(List<Guid> GuidPath, List<DeploymentActivity> deploymentactivities);

List<TransformationApplication> GetAllAvailableTransformationApplication(Guid ID);
List<TransformationApplication> GetTransformationApplication(Guid id, Guid transformationApplicationID);
bool RemoveTransformationApplication(Guid id, Guid transformationApplicationID);
bool CreateTransformationApplication(Guid id, TransformationApplication transformationApplication);
bool UpdateTransformationApplication(Guid id, TransformationApplication transformationApplication);

```

Figure 75 - Advanced interface of the UC-Plugins

Selected technical environment

Therefore, different operations groups are used: the Software Unit Handling Group, the Integration Activity Group, the Transformation Activity Group, and the Deployment Activity Group. These groups are discussed in the following sections.

Software unit handling group operations

The first two operations are named `CreateArtefact` and `UpdateArtefact`. They enable an experienced user to create or update a Software Construction Artefact which can contain different software units. Therefore, alternative information is necessary. First of all, the information demand (ID) of the Prometheus instance and the repository in which the new SCA should be saved, have to be set (`serverID` and `repositoryID`). The next is the type of artefact (`artefactType`) which states if the artefact contains UOM which includes data, structure, graphical, or function elements (see Section 4.4.1.1). A name (`artefactName`) and the description (`description`) also have to be set. The parameter and return values are listed in Table 12:

Create/UpdateArtefact		
Type	Name	Description
List<Guid>	guidPath	Includes the path (<code>serverID</code> , <code>repositoryID</code>) were the artefact should be created.
Int	artefactType	Type of the artefact. Therefore, the UOM types Data, Function, GUI, Structure is used.
String	artefactName	A customised (user friendly) name for the artefact.
String	description	Description of the professional content of the containing software unit.
Artefact	-	The return value is a new SCA.

Table 12 - Parameters of the `CreateArtefact` operation

The next two operations are called `CreateUOM` and `UpdateUOM`. These enable the storing of a software unit into a selected repository or the updating of an existing one. This operation also needs special information. First of all, the path of the UOM has to be set. This is done by using the IDs of the service (`serverID`), the repository (`reposID`) as well as the artefact (`artefactID`). Additionally, it is necessary to define the UOM type (for `uomType`, see section 4.4.1.1), a customised name (`uomName`), and a small description (`uomDescription`).

Selected technical environment

The parameter and return values are listed in Table 13:

Create/UpdateUOM		
Type	Name	Description
List<Guid>	guidPath	Includes the path (serverID, repositoryID, artefactID) were the artefact should be created.
Int	uomType	Type of the UOM. Possible values are Data, Function, GUI, and Structure.
string	uomName	A customised (user friendly) name for the uom.
string	uomDescription	Description of the professional content of the uom.
Bool	-	The return value indicates if the creation/update task was successful (true) or unsuccessful (false).

Table 13 - Parameters of the *CreateUOM* operation

This operation creates an instance of an empty software unit model. To add existing files to this software unit, the *AddData* operation has to be used. At the end, the Prometheus server needs three different types of information:

- The path to the Software Unit Model which is given by the ID of the server (serverID), repository (reposID), artefact (artefactID), software unit (uomID), and a reference to a package (packageID) as optional information. A package ID indicates that information belongs together.
- The file information includes typical information, as for example, Path (directory, fullname), name (name, fullname), file extension (extension), file length (size), the byte content (bytecontent), as well as the creation date (datetime).
- Additional information for the focused approach, for example a package ID for grouping different files in a download package (e.g., the different web pages of a Hypertext Markup Language (HTML) documentation), the definition of this file as human or machine readable (humansselector), and based on that, the real content type.

The parameter and return values are listed in Table 14:

AddData		
Type	Name	Description
List<Guid>	guidPath	Includes the path (serverID, repositoryID, artefactID, uomID, packageID) to identify the artefact.
string	name	Name of a single file.
String	fullname	Name and path of a single file.

Selected technical environment

AddData		
<i>Type</i>	<i>Name</i>	<i>Description</i>
String	extension	The file extension of a single file.
String	directoryName	The directory name of a single file.
Datetime	dateTime	The creation time of a single file.
Long	size	The size in kilobyte of single file.
Byte[]	byteContent	The byte content of a single file.
Int (Human or machine data)	humanSelector	The content type of the file. Possible values are 'machine' or 'human'
Int (Dynamic enum for human or machine readable data)	contentSelector	The type content of the file to add. Values depends on the humanSelector.
FileElement	-	The return value is an object containing all information set by this operation.

Table 14 - Parameters of the *AddData* operation

The last relevant operation is used to delete existing items (e.g., software units, single files or reuse activities). This operation is called *RemoveItem* and has a simple structure. First the server (*serverID*) and the repository (*reposID*) have to be set. Then the ID (*ID*) of the item that order to be removed. Finally, the type (e.g., UOM, activity, file, etc.) has to be set. The operation returns a true/false for a successful/unsuccessful execution. The parameter and return values are listed in Table 15:

RemoveItem		
<i>Type</i>	<i>Name</i>	<i>Description</i>
List<Guid>	guidPath	Includes the path (<i>serverID</i> , <i>repositoryID</i>) were the artefact should be created.
Guid	id	The ID of the element which should be removed.
Bool	-	The return value indicates if the operation execution was successful or not.

Table 15 - Parameters of the *RemoveItem* operation

Integration activity group

The integration part of the instantiated SCS offers one group of operations: *Integration Activity Group*. In this group, four operations are defined *LoadIntegrationActivity*, *RemoveIntegrationActivity*, *CreateIntegrationActivity*, and *UpdateIntegrationActivity*.

The first operation is the *LoadIntegrationActivity*, which is used to load information about integration activities from the server into the client. Therefore, a list of IDs

Selected technical environment

(integrationActivityIDs) has to be set. The operation returns a list of IntegrationActivity objects (List<IntegrationActivities>), which contains all necessary information. The parameter and return values are listed in Table 16:

LoadIntegrationActivity		
Type	Name	Description
List<Guid>	IntegrationActivityIDs	The IDs of integration activities which should be loaded.
List<IntegrationActivity>	-	The return value includes a list of IntegrationActivities. Each describes a complete integration activity.

Table 16 - Parameters of the *LoadIntegrationActivity* operation

The operation RemoveIntegration removes a single integration activity based on the given ID (ID). The parameter and return values are listed in Table 30:

RemoveIntegration		
Type	Name	Description
Guid	ID	The ID of the integration activity.
bool	-	The return value indicates if the remove task was successful (true) or unsuccessful (false).

Table 17 - Parameters of the *RemoveIntegration* operation

The operation CreateIntegration stores a list of new integration activities (List<Integrations> integrationActivities) to a given software unit, which is indicated by a list of IDs (List <Guid> guidPath). The UpdateIntegration uses the same structure. The parameter and return values are listed in Table 18:

CreateIntegration / UpdateIntegration		
Type	Name	Description
List <Guid>	GuidPath	Includes the path (serverID, repository, artefactId, UOMId) to select the software unit (UOM) related to the created/updated integration activity.
List<IntegrationActivity>	integrationactivities	A list of complete descriptions of integration activities.
Bool	-	The return value indicates if the creation/update task was successful (true) or unsuccessful (false).

Table 18 - Parameters of the *Create/UpdatesIntegration* operation

Transformation activity group

For the creation of a transformation reuse activity, different operations are necessary which can be divided into two groups: a Transformation Application Group and a Transformation Activity Group. Both groups are described as follows: the Transformation Application Group contains an operation for managing transformation applications: `GetAvailable-Transformation-Application`, `GetTransformationApplication`, `RemoveTransformationApplication`, `Create-TransformationApplication` and `UpdateTransformationApplication`.

The operation `GetAllAvailableTransformationApplication` responds with a list of transformation application descriptions (`List<TransformationApplication>`) which can be used and handled by the Prometheus server. The parameter and return values are listed in Table 19:

GetAvailableTransformationApplication		
Type	Name	Description
Guid	ID	The ID of the Prometheus server containing the transformation application.
List<TransformationApplication>	-	The return value includes a list of transformation application descriptions containing all information about the configuration of transformation applications.

Table 19 - Parameters of the *GetAvailableTransformationApplication* operation

The operation `GetTransformationApplication` responds with a single transformation application (`TransformationApplication`) description, based on the given ID (`id`). The parameter and return values are listed in Table 20:

GetTransformationApplication		
Type	Name	Description
Guid	ID	The ID of the Prometheus server containing the transformation application.
Guid	transformationApplicationID	The ID of the transformation application.
List<TransformationApplication>	-	The return value includes a transformation application description containing all information about the configuration of a transformation application.

Table 20 - Parameters of the *GetTransformationApplication* operation

Selected technical environment

The operation `RemoveTransformationApplication` removes a single transformation application, based on the given ID (id). The parameter and return values are listed in Table 21:

RemoveTransformationApplication		
Type	Name	Description
Guid	ID	The ID of the Prometheus server containing the transformation application.
Guid	transformationApplicationID	The ID of the searched transformation application.
bool	-	The return value indicates if the remove task was successful (true) or unsuccessful (false).

Table 21 - Parameters of the *RemoveTransformationApplication* operation

The operation `CreateTransformation` creates a transformation activity description which may be used by transformation reuse activities. The description is included in only one parameter (Tapp) of type `TransformationApplication`. The `UpdateTransformationApplication` operation has the same structure and updates existing transformation applications. The parameter and return values for both operations are listed in Table 22:

Create / UpdateTransformationApplication		
Type	Name	Description
Guid	ID	The ID of the focused Prometheus.
TransformationApplication	transformationApplication	The complete description of a transformation application.
Bool	-	The return value indicates if the remove task was successful (true) or unsuccessful (false).

Table 22 - Parameters of the *Create/TransformationApplication* operation

The Transformation Activity Group contains the following operations for handling transformation activities: `LoadTransformationActivity`, `RemoveTransformationActivity`, `CreateTransformationActivity` and `UpdateTransformationActivity`.

The first operation is the `LoadTransformationActivity` operation. This operation is used to load information about transformation activities. Therefore, a list of IDs (`transformationActivityIDs`) has to be set.

Selected technical environment

The operation returns a list of TransformationActivity objects (List<TransformationActivities) which contains all the necessary information. The parameter and return values are listed in Table 23:

LoadTransformationActivity		
Type	Name	Description
List<Guid>	TransformationActivityIDs	The IDs of transformation activities which should be loaded.
List<TransformationActivity>	-	The return value includes a list of <i>TransformationActivities</i> . Each describes a complete transformation activity.

Table 23 - Parameters of the LoadTransformationActivity operation

The operation RemoveTransformation removes a single transformation activity description, based on the given ID (id). The parameter and return values are listed Table 24:

RemoveTransformationActivity		
Type	Name	Description
Guid	ID	The ID of the transformation activity description
bool	-	The return value indicates if the remove task was successful (true) or unsuccessful (false)

Table 24 - Parameters of the RemoveTransformation operation

The operation CreateTransformationActivity creates a transformation activity description. The description is included in only one parameter of type, TransformationApplication, and requires a second parameter to identify the path to the related software units (List<Guid> id). The UpdateTransformationApplication operation has the same structure, and updates the existing transformation activity description. The parameter and return values are listed in Table 25:

CreateTransformationActivity / UpdateTransformationActivity		
Type	Name	Description
List<Guid>	GuidPath	Includes the path (serverID, repositoryID, artefactID, uomID) to select the software unit (UOM) related to the created/updated transformation activity.
Transformation	transformation	The complete description of a transformation application.
bool	-	The return value Indicates if the creation/update task was successful (true) or unsuccessful (false).

Table 25 - Parameters of the Create/UpdateTransformation operation

Deployment activity group

For the deployment activity the five different operations are defined: CreateDeploymentActivity, GetAllDeploymentForASingleUOM, GetSingleDeployment, RemoveDeploymentActivity and UpdateDeploymentActivity

The first operation is the LoadDeploymentActivity operation, which is used to load information about transformation activities. Therefore, a list of IDs (deploymentActivityIDs) has to be set. The operation returns a list of DeploymentActivity (List<DeploymentActivities>) objects which contains all the necessary information. The parameter and return values are listed in Table 26:

LoadDeploymentActivity		
Type	Name	Description
List<Guid>	DeploymentActivityIDs	The IDs of deployment activities which should be loaded.
List<DeploymentActivity>	-	The return value includes a list of DeploymentActivities. Each describes a complete deployment activity.

Table 26 - Parameters of the LoadDeploymentActivity operation

The operation RemoveDeployment removes a single deployment activity description based on the given ID (ID). The parameter and return values are listed in Table 27:

RemoveDeploymentActivity		
Type	Name	Description
Guid	ID	The ID of the deployment activity.
Bool	-	The return value indicates if the remove task was successful (true) or unsuccessful (false).

Table 27 - Parameters of the RemoveDeploymentActivity operation

This operation CreateDeployment stores a list of new deployment activities (i.e., List<DeploymentActivity> deploymentActivities) to a given software unit, which is indicated by a list of IDs (List<Guid> guidPath). The UpdateDeployment uses the same structure. The parameter and return values are listed in Table 28:

Selected technical environment

CreateDeployment / UpdateDeployment		
Type	Name	Description
List<Deployment>	Deploymentactivities	A list of complete descriptions of deployment activities.
List <Guid>	GuidPath	Includes the path (serverID, reposed, artefactId, UOMId) to select the software unit (UOM) related to the created/updated deployment activity.
bool	-	Indicates if the creation/update task was successful (true) or unsuccessful (false).

Table 28 - Parameters of the Create/UpdateDeployment operation

Support interfaces

These interfaces are required to use KU and KC related operations with the correct information about the current Prometheus environment.

```
Service GetServiceInformation(Guid id);
Repository[] GetAvailableRepositories();
```

Figure 76 - Additional support Interface of the UC-Plugin

Figure 76 shows the two operations of the support interfaces, which are necessary for KU and KC related operations. GetServiceInformation is an operation that returns the information to a Prometheus server, such as a version number and contained information about repositories. Because of this, the Prometheus server can be connected in clusters; a Prometheus server ID will be integrated into the call. Therefore, clients are able to use these IDs from another Prometheus server or repository in their different requests. The parameter and return values are listed in Table 29 (cf. 4.5.4.3):

GetServiceInformation		
Type	Name	Description
Service	-	The return value contains an information object of all available repositories in a Prometheus server, including (e.g., name, ID, functional limitations of each repository, etc.)

Table 29 - Parameters of the GetServiceInformation operation

The second operation in this section is the *GetAvailableRepository* operation, which gives information about the active (actually available) repositories of the Prometheus server. Because of the cascading feature, a Prometheus server ID is also used. This method is intended for informational purposes and to validate whether communication is necessary or possible with a desired repository. The parameter and return values are listed in Table 30:

GetAvailableRepositoryInformation		
Type	Name	Description
Guid	ID	The ID of the Prometheus server containing the software unit.
Repository[]	-	The return value includes an information object of a Prometheus server, including (e.g., name of the Prometheus server, ID of the different repository, etc.)

Table 30 - Parameters of the GetAvailableRepositoryInformation operation

5.2.2.3. User Client Plugins interfaces and reuse activity interfaces (I1 A/B)

The User Client Plugin Interface is used for the communication between the user client and the Prometheus server. Therefore, the Prometheus Service implements this interface and the User Client Plugin uses the interface to communicate to the Prometheus core. The interface is divided into different areas: Knowledge User and Knowledge Creator related interfaces and some minor support operations.

5.2.2.4. Repository client plugin interface (I2)

The Repository Client Plugin interface defines the communication between the Prometheus core and the different repository plugins. Therefore, different method groups are used: Software Unit Handling Group, Integration Activity Group, Transformation Activity Group, Deployment Activity Group and Repository Control Group. From the perspective of software development, each group is defined in a specific interface and the interface shown in Figure 73 inherits these interfaces.

Selected technical environment

For simplification, this interface uses most of the operations already defined in other interfaces. The purpose of all interface operations will be explained, but only new or adapted operations will be shown in a table view.

The Software Unit Handling Group contains nine operations that handle software units as content. CreateArtefact, UpdateArtefact, CreateUOM, UpdateUOM, AddData, Search, GetItems, GetItemsAsZip and RemoveItems.

The operations GetItems and GetItemsAsZip do not differ from the definition of the user interface. Both operations are used to get byte data from a repository. Also, the operations: CreateArtefact, UpdateArtefact, CreateUOM and UpdateUOM do not differ in their purpose or in terms of the necessary information. The creation methods save data into the repositories. The update operations update such information inside the repository. The operations AddData, RemoveItem and Search do not differ in their interface structure, but they are implemented differently based on the related repository.

The Integration Activity Group contains three interfaces related to manage integration activity information: LoadIntegrationActivity, CreateIntegration and SetTransferTypes. The operation RemoveIntegration is realised by using the RemoveItem operation of the Repository Control Group. None of these operations differ in their structure or purpose from the previous descriptions.

The Transformation Activity Group contains four different operations: LoadTransformationActivity, CreateTransformation, UpdateTransformation and RemoveTransformation. Note: The operation RemoveIntegration is realised by using the RemoveItem operation of the Repository Control Group. None of these operations differ in their structure or purpose.

The Deployment Activity Group contains five different operations: CreateDeploymentActivity, GetAllDeploymentForASingleUOM, GetSingleDeployment, RemoveDeploymentActivity and UpdateDeploymentActivity. Note: the operation RemoveIntegration is realised by using the

Selected technical environment

RemoveItem operation of the Repository Control Group. None of these operations differ in their structure or purpose.

The Repository Control Group contains two relevant operations: RemoveItem and GetAllRepositories. Neither operation differs in its structure or purpose as defined in the previous section. The RemoveItems operation deletes data which belongs to an item which should be removed, (e.g., the files of a UOM). The other operation returns information about the repository, including repository ID and which operations of the interface are supported.

Note: The repository interfaces include additional operations for handling the repositories, (e.g., initialisation of communication). Also, some methods are included for handling the plugin in the focused Microsoft Extensible Framework (MEF) system.

5.2.2.5. Reuse activity plugin interfaces (I3 a/b/c)

Integration Plugin Interfaces (I3a): The interface I3a is provided by integration plugins. In the case of the focused Prometheus environment, two plugins use this interface. The first one is for Visual Studio integrations. The second one is for Eclipse integrations. Both plugins use the same interface for communication from the core to the plugins. Figure 77 shows the user interface including an event and the previously explained DoIntegrationActivity operation.

```
public interface IPIntegration
{
    [Operation Contract]
    List<ErrorMessage> DoIntegration(List<Guid> guidPath, Guid integrationActivity, ReceiveClient ide);
}
```

Figure 77 - C# notation of the integration plugin interface

DoIntegration		
Type	Name	Description
List<Guid>	guidPath	The IDs of the repository, Software Construction Artefact, and UOM to be searched for a transformation activity.
Guid	integrationActivity	The ID of the integration SCA to be performed.
ReceivClient	ide	Information about the service endpoint.
List<ErrorMessage>	-	The return value includes a list of error messages for the user.

Table 31 - Parameters of the DoIntegration operation

Selected technical environment

The method DoIntegration includes different parameters. The first one (List<Guid> guidPath) identifies the Prometheus Service, the repository, the SCA, as well as the UOM that relies to the integration SCA. The second parameter (Guid integrationActivity) is the ID of the integration SCA to be performed. The last parameter includes the service information where the SCA should be performed (ReceiveClient ide). The method is able to respond a list of custom errormessage.

Transformation Clients Interface: Figure 78 shows the relevant service operations used.

```
public interface ITransformation
{
    [OperationContract]
    List<TransferType> DoTransformation(Guid serviceID, Guid repositoryID, Guid artefactID, Guid uomID, string transformationRuleName);
}
```

Figure 78 - C# notation of the transformation client Interface

The method DoTransformation requires several pieces of information. The ID for the corresponding Prometheus Service (Guid serviceID), the ID of the database that contains the software unit to be transformed (Guid RepositoryID), the ID of the software unit containing the artefact (Guid artefactID), the ID of the UOM, and the transformation rule have to be executed (string transformationruleName). The method returns a list of transfer types as the return value. TransferType consists of a set of file elements and additional descriptions of the result of a transformation. In the case of the Prometheus environment the result of the transformation is a new software unit (see Section 5.3.2). The parameter and return values are shown in Table 32.

DoTransformation		
Type	Name	Description
Guid	serviced	The ID of the Prometheus server.
Guid	repositoryID	The ID of the repository.
Guid	artefactID	The ID of the Software Construction Artefact.
Guid	uomID	The ID of the UOM including the transformation activity.
string	transformationRule Name	The name of the transformation activity which should be executed.
List<TransferType>	-	The return value includes a list of UOM artefacts.

Table 32 - Parameters of the DoTransformation operation

Deployment Clients Interface:

```
public interface IDeployment
{
    [OperationContract]
    List<ErrorMessage> DoDeployment(Guid serviceID, Guid repositoryID, Guid artefactID, Guid uomID, string deploymentruleName);
}
```

Figure 79 - C# Notation of the deployment web service

Figure 79 shows the relevant service operations used. The DoDeployment operation requires several pieces of information, as with the other client methods. It is necessary to use different IDs to select the correct Software unit, for example, the ID for the corresponding Prometheus server (Guid serviceID), the ID of the repository that contains the software unit to be deployed (Guid RepositoryID), the ID of the software unit containing the artefact (Guid artefactID), the ID of the UOM, and the deployment rule to be executed (string deploymentruleName). The operation returns a list of errors (List<ErrorMessages> Errormessages).

The parameter and return values are listed in Table 33. As discussed in Section 5.3.3, an operation is also available for the subscription to manual instruction (DoSubscription). This operation is not discussed further because it is a common subscription operation with eventing mechanisms.

DoDeployment		
<i>Type</i>	<i>Name</i>	<i>Description</i>
Guid	serviceID	The ID of the Prometheus server is to be searched.
Guid	repositoryID	The ID of the repository to be searched for a transformation activity.
Guid	artefactID	The ID of the Software Construction Artefact.
Guid	uomID	The ID of the UOM including the deploymentDeployment activity.
string	deploymentRuleName	The name of the transformation activity which should be executed.
List<ErrorMessages>	-	The return value includes a list of error messages for the user.

Table 33 - Parameters of the DoDeployment operation

5.2.2.6. Repository client interfaces

The connection to the different repository systems was handled using the provided interfaces.

In three cases, these interfaces were real system interfaces:

Selected technical environment

I4: The DeviceXML repository provides a Web Service based on ASP.NET. This service includes an interface with three operations. The first one is the initial login into the system ('authentication') which requires a username and password. The second one is a search function which delivers device xml classes. The third operation delivers pictures for a given device xml class.

I6: The SOA4D repository provides a Web Service interface based on the GSoap library (FSU, 2007). SOA4D provides operations to iterate over the SOA4D project structure to get information about, for example, stored software units.

I12: The last repository in the used Prometheus environment was a second Prometheus environment using the described I1 interface (cf. 4.5.4.3).

5.2.2.7. Integration client plugin

For the connection to the Visual Studio Gateway and the Eclipse Plugin, a Web Service interface was used. Even though both systems were developed with different technology, the SOAP-based Web Service includes the same interface. Figure 80 shows the relevant service operations used:

```
public interface IReceiveService
{
    [OperationContract]
    void SetTransferTypes(List<IntegrationObject> transferTypes);

    [OperationContract]
    void ReceiveZip(byte[] p);
}
```

Figure 80 - C# Notation of the Integration client plugin Interface

SetTransferType		
Type	Name	Description
List<IntegrationObject>	transferTypes	List of integration objects. Each object describes how a given UOM element has to be integrated in a given development environment.
void	-	Return value does not exist.

Table 34 - Parameters of the SetTransferType method

The method SetTransferType receives a list of IntegrationObjects. These objects contain the binary data and a description of how these are to be integrated (see Section 4.5.3). An

Selected technical environment

integration client plugin interprets these data sets and the integration of the specific type of development environment. The parameter and return values are shown in Table 34.

The method `ReceiveZip` receives a ZIP package that allows the user to save the zip file. This file includes all files and folders necessary for an activity. This is needed in the download scenario (see Section 4.5.1). Also the result of a transformation activity includes one or more files. To avoid several download operations these are included in one zip file. The parameter and return values are shown in Table 35:

ReceiveZip		
<i>Type</i>	<i>Name</i>	<i>Description</i>
byte[]	Files	A list of files which can be stored as zip files.
Void	-	Return value does not exist for this operation.

Table 35 - Parameters of the `ReceiveZip` method

5.2.2.8. Non-service interfaces

Connections to other existing system (e.g., the repositories) are called non-service interfaces in this realisation. The Prometheus environment uses different repository technologies and applications without any service interfaces. These are listed as follows:

I5: The Brick Catalogue only provides a web page for humans. As a result, there is no system interface which can be handled by a repository plugin. The problem was solved by an automated reading of the web page and by creating an internal database inside the plugin which was created for this repository.

I7: When connecting to the SQL based repository, the SQL connection technology of Microsoft .Net combined with the Entity Framework of .NET was used. Therefore, the connection to the server was handled by the .NET Environment. The special repository plugin for this repository only handled the objects and sent read/write requests.

I13: For the communication between the Eclipse Integrated Development Environment (IDE) plugin and Eclipse, the Eclipse IDE Application Programming Interface (API) was used.

Selected technical environment

I14: For the communication between the Visual Studio Gateway and a Visual Studio Instance the COM Interface of visual studio was used.

I10: The transformation application calls are realised by using normal process calls.

I11: The transformation application calls are realised by using normal process calls and File transfer protocol (FTP)-based connections.

5.2.3. Used technologies and communication protocols

For the realisation of the different architecture elements, different technologies are used. For the relevant parts of the realisation, the used technologies are now briefly described. Technologies or architecture parts which are not relevant are not discussed or listed in this thesis. Figure 81 shows an overview of used technologies in the Prometheus environment.

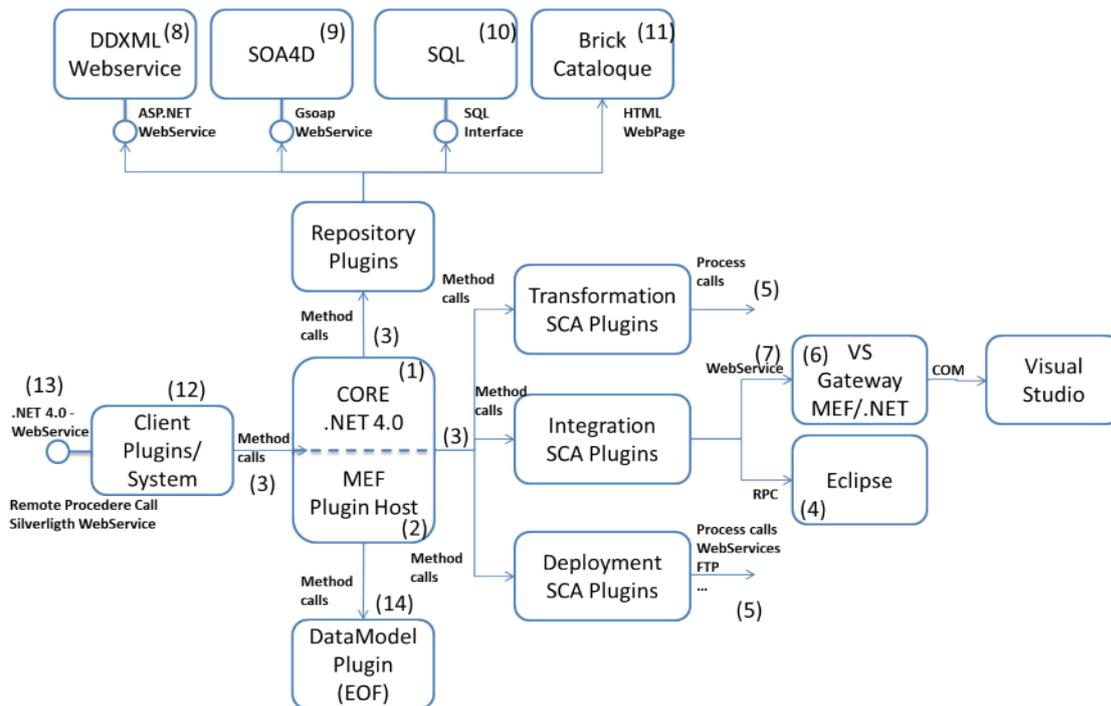


Figure 81 - Used technologies

The Prometheus core was developed by using Microsoft .NET Version 4.0 (see 1 in Figure 81). The used programming language is C#. The plugins were also developed using this platform as well as the used programming language. As a plugin system or architecture, the MEF was used

Selected technical environment

(see 2 in Figure 81). This enables lazy loading and the rescheduling of plugins into a plugin host, which is in the current scenario Prometheus core. The plugin differs in its communication technology. The plugins use normal method calls or .NET eventing mechanisms to communicate with the core (see 3 in Figure 81). Some plugins use or provide Web Services based on SOAP 1.2. This interface technology is realised based on Windows Communication Foundation (WCF), which is an API extension of .NET 4.0, including different technologies for communication. One relevant feature of WCF used in this implementation is the hosting of multiple Web Services in one application, without the use of a classical web server (e.g., Internet Information Server). This enables the Prometheus core or the plugins to host different Web Services directly without the need for an external web server. Some of the plugins control other applications by using process calls (see 5 in Figure 81). Thereby, typical functionality of the .NET platform was used.

The Eclipse integration plugin is developed using the Java programming language and a special API of the Eclipse IDE. It also uses a Web Service which is based on a Java SOAP extension. The Visual Studio Gateway is a standalone application based on C# and .NET 4. It also uses a WCF-based web service and the Microsoft COM technologies (see 6 and 7 in Figure 81). Visual Studio provides a COM object to perform IDE activities.

For the connection to the different repositories, the following technologies were used: SQL (see 8 in Figure 81), GSoap based Web Service (FSU, 2007; see 9), the ASP.NET based Web Service (see 10 in Figure 81), and simple HTML API of .NET 11.

The user interface of the client system used is realised in Microsoft Silverlight technology (see 12 in Figure 81; Microsoft, 2012d). The version 4.0 was used. For communication purposes, the client system reacts as Web Service consumer using WCF (see 13 in Figure 81).

The datamodels implemented are created by using a code first approach based on the Entity Object Framework (EOF) which is a Microsoft-supported open source project for object-oriented data handling of databases (see 14 in Figure 81; Microsoft, 2010).

5.2.4. Extensibility approaches

Section 4.5.4 discusses the requirement of the focused extensibility (based on variants of technologies and changes in future) for the following points: Fundamental Software Unit Model, Software Construction Activities models, support of different existing repositories, support of different development environments and tools and support of different client applications.

The support of different client applications as extensibility attributes is realised by using the plugin concept which enables the integration of clients using different communication concepts or technologies. In this case, a SOAP based Web Service is used which fits to the W3C web service standard (W3C, 2007). Also, the used data model in the communication protocol is fully serialisable and describable by using XML technology. The advantages of this approach are:

- Different platforms or programming languages support this kind of service technology,
- (Web) Services can be created to enable a distributed environment (e.g., Service-oriented Architecture; SOA).

The plugin system is also used for extensibility to support of different development environments. In this scenario, two different IDE types were integrated using different technologies. One was also integrated using a Web Service approach, which shows the possibilities for distribution. The integration of the repository also uses the same approach. One repository (Brick Catalogue), which is only accessible by scanning the provided web pages, shows the advantages (See 10 in Figure 81). This repository is not able to accept write requests or provide a system interface. It provides a simple web page (HTML) for human readers. To get information, the different web pages have to be scanned by a self-written HTML client. The scan results are stored using a second software unit datamodel instance in the plugin. This specialised repository connector is part of one repository plugin and shows the flexibility of the

plugin approach. Even if ‘special’ handling or technologies are necessary, this can be handled by a plugin and does not affect the environment.

The Software Unit Model and the three different reuse activity models were developed as semi semantic models. One relevant requirement was to create one technical model that can be referred to by others. Therefore, following characteristic of semantic models is relevant: semantic models can be extended by relating the different semantic terms of two models. This characteristic is used to combine the three software construction activity models with the software unit model (cf. Parreiras, 2012).

5.3. Realised models

In this section, the realisation for the different models of software unit and reuse activities (see Section 4.5.2) is described. Therefore, four different model instances were created:

- Software Unit Model – Realised model of the basic Software Unit Model (see Section 4.5.2)
- Transformation Activity Model – Realised model of an activity model (see Section 4.5.3)
- Integration Activity Model – Realised model of an activity model (see Section 4.5.3)
- Deployment Activity Model – Realised model of an activity model (see Section 4.5.3)

The different models are realised using a single model concept. This concept consists of several layers which together give the description of the focused software units and activities. In basic terms, it contains two layers (see Figure 60 in Section 4.5.2):

The ‘Unit’ or U-Layer describes software units with the necessary information for this approach. The ‘Application’ or A-Layer describes the different reuse activity view.

5.3.1. Software Unit Model instance

In this section, the U-Layer is described. This layer consists of four different areas described in Section 4.5.2 (cf. Figure 59).

Realised models

In region 1: the "Stakeholder View", authors and responsible persons are described. Region 2: the "Problem-Solution View" describes the relationship between the problem and solution. Region 3: the "Technical View" describes a device from a technical perspective. In region 4: the "Content View", on the other hand, is a description from the technical point of view for searching behaviors. The four areas are described in the following sub-sections. There, each element in the different model region will be described, as well as the relational aspects of each element.

5.3.1.1. Restriction rules for the datamodel

Restriction 1: In general, each element of the datamodel has no relation to other elements or semantics. The exceptions are explained in the next sub sections.

Restriction 2: Each element is a 'Thing' and has an ID. This is relevant for the handling of model instances inside the Prometheus environment (cf. Figure 82).

Restriction 3: Also, relevant is the general view on persons (see Figure 82). A person can be a natural person or a synthetic person in the scope of the model.

Restriction 4: A system is a computer application.

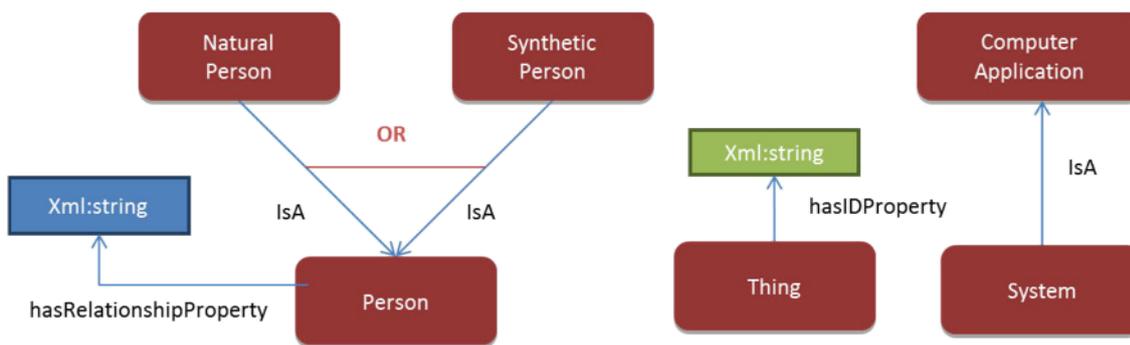


Figure 82 - Model restrictions

5.3.1.2. Region 1 "Stakeholder View"

The stakeholder view is relevant for this thesis because information about possible stakeholders (e.g., project owner or unit creator) may be relevant for users for reuse. Therefore, this region is

Realised models

This region is named U-R1 and is demonstrated in Figure 83. The element relations can be described as follows: A ‘Person’ is an ‘Author’ (U1(1→3)) if this person creates the ‘UOM’ (U1 (3→U3(1))) or is the current UOM owner. A ‘Person’ is a ‘User’ (U1(1→11)) if this person:

- (U1 (11→5)) makes a ‘Ranking’ that is a sub type of a ‘Comment’ (U1 (5 → 12)) for a UOM or an related SCAC (U1 (5→U3(1))) or
- Creates a ‘Comment’ (U1 (11→12)a) for a UOM or related SCACs (U1 (7→12)a) or a ‘Feedback’(U1 (11→7)) that is also a ‘Comment’ (U1 (12→ U3(1))) or a comment of a comment (U1 (7→12)b) or
- Comments on an existing comment (U1 (11-12)b) which is equal to (U1 (11→7)) and (U1 (11→12)a).

Each ‘Person’ has ‘Contact Information’ (U1 (1→4)). Ranking is a special type of ‘Statistical Data’ (U1 (5→6)) and is directly related to the ‘UOM’ U3(1→U1(6)). A ‘Monitor’ is a special ‘Service’ (U1 (9→8)) and sets and reads ‘Statistical Data’ (U1 (6→8)a and b). Both ‘Monitor’ and ‘Service’ are ‘System’ elements (U1 (9→2)) (U1 (8→2)).

5.3.1.3. Region 2 “Problem-Solution View”

The research area of this study does not focus on problem-solutions in terms of software units and has no contribution to these issues. However, this is an relevant research field, (cf. solution and relation comparison discussed by Jeong and Kim, 2012). Therefore, the used data model includes a relation to this topic which may be extended in the future by other semantic models. But this is not part of the focused research. The problem solution context is related to the professional content of a UOM and not to the SCACs.

No.	Element	Description
U2-1	Problem	This element represents a problem that can be solved by a UOM.
U2-2	Solution	This element represents a solution included in a UOM for a particular problem.
U2-3	Software Engineer	This element represents a special user who is a problem owner and is searching for a solution.
U2-3	MetaData	This element represents additional information describing a solution.

Table 37 - Defintion of elements of the problem solution view (U2)

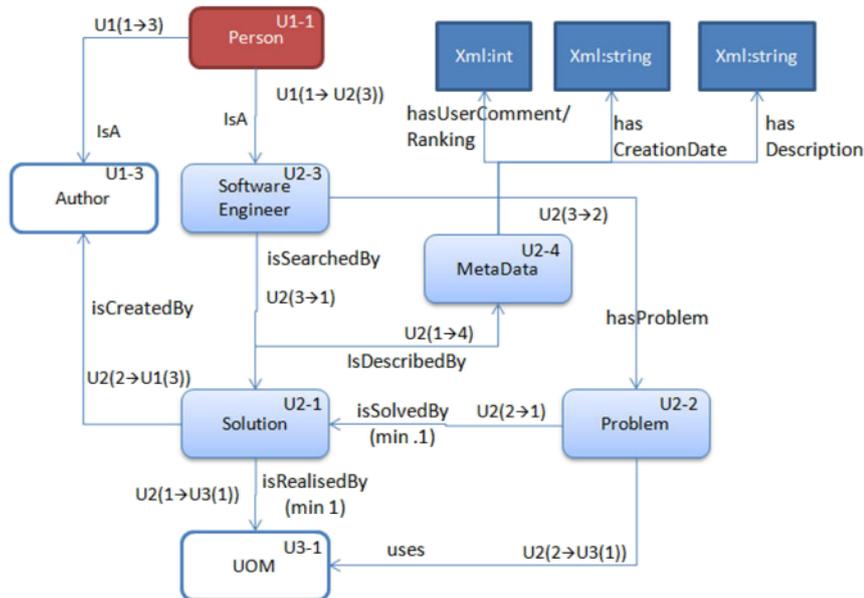


Figure 84 - Relevant elements of the area 2 – problem and solution view” (U-R2)

Figure 84 shows the elements in this area. The main elements are ‘Solution’ and ‘Problem’. Thereby, a ‘Problem’ is solved by a ‘Solution’ (U2(2→1)). A ‘Solution’ is realised by a ‘UOM’ (U2(1→U3(1))) that is used to solve a ‘Problem’ (U2(2 → U3(1))). Additionally, a ‘Solution’ can be described by additional ‘Metadata’ (U2(1→4)). A ‘Software Engineer’ which is a ‘Person’ (U1(1→U2(3))) searches for solutions (U2(3→1)) because of engineer has one or more problems (U2(3→2)). Finally this model shows that a solution is created by an ‘Author’ of an ‘UOM’ (U2(2→U1(3))).

5.3.1.4. Region 3 “Technical View”

The technical view is the relevant view for the primary research of this thesis. It describes software units (based on different technologies and types) in a common view. This is part of the concept described in Chapter 4. All SCAC-based models relate to this section. Table 38 shows the 22 elements that are relevant in this area.

Realised models

No.	Element	Description
U3-1	UOM	This element is the central definition of a software unit as a reusable element. It represents the view of this approach in this thesis when it is "spoken" of as a reusable software unit. Therefore, the element is meant as an alternate member for the entire data model. All other elements are descriptive elements for the 'UOM' element.
U3-2	Unit	This element represents a real software component in the context of this thesis that is equivalent to a unit of a class, component or service.
U3-3	Human Readable Content	This element describes the contents of a 'Unit', a 'Data', as suitable for human element format.
U3-4	Machine Readable Content	This element describes the contents of a 'Unit', a 'Data' element as for systems suitable format.
U3-5	Service Information	This element defines a 'Machine Readable Content' 'Data' element which is a service type. Therefore, it is an interface description (e.g., endpoint and WSDL file). This element behaves disjointly to 'Class' and 'Binary' elements.
U3-6	BinaryData	This element defines a 'Machine Readable Content' 'Data' element as a binary type. This is the definition of binary files that do not comply with the other descriptive elements ('Class' and 'Service'). This element behaves disjunct from the elements 'Class' and 'Service'.
U3-7	ClassData	This element defines a 'Machine Readable Content' 'Data' element as a class type. This serves the definition of class files. This element behaves disjunct from the elements 'Binary' and 'Service'.
U3-8	Data	This element is the most common element which is used to describe the content of an available element.
U3-9	Audio	This element is a 'Human Readable defined content' 'Data' element as the audio type. This is the definition of audible 'Audio' data. This element behaves disjunct from the elements 'Video', 'Document', 'Link' and 'Picture'.
U3-10	Document	This element is a 'Human Readable defined content' 'Data' element as the document type. This is the definition of readable documents. This element behaves disjunct from the elements 'Video', 'Audio' and 'Picture'.
U3-11	Video	This element is a 'Human Readable defined content' 'Data' element video type. This serves to define foreseeable movie data. This element behaves disjunct from the elements 'Document', 'Audio', 'Link' and 'Picture'.
U3-12	File-Links	This element represents physical data of a file.
U3-13	Technology	This element describes the basic technology, which is required by a 'Technical Environment'.
U3-14	Programming Language Charactersitic	This element defines the language that has to be available in the runtime environment.

Realised models

No.	Element	Description
U3-15	Component	This element defines a 'unit' element as a component software unit.
U3-16	Class	This element defines a 'unit' element as a class software unit.
U3-17	Service	This element defines a 'unit' element as a service software unit.
U3-18	Snippet	This element is a "unit" and is defined as a textual element software unit. In this work, this type of software unit is out of focus.
U3-19	File Structure	This element describes a file in a folder location
U3-20	Tec Environment	This element is a descriptive element. It describes the technical environment, which a "unit" element requires to be able to run.
U3-21	Picture	This element is a 'Human Readable defined content' 'Data' element as a picture type. This is the definition of foreseeable image files. This element behaves disjunct from the elements 'Video', 'Audio', 'Link' and 'Document'.
U3-22	Folder Structure	This element describes a folder structure
U3-23	Dependencies	This element defines dependencies (e.g., other files, system files, or settings)
	Link	This element defines a 'Human Readable Content' 'Data' element as a link type. This is used for the definition of document links to other documents Note: This element is not used in the realised environment and, therefore, not part of Figure 106.

Table 38 - Defintion of elemens of the technical view (U3)

Figure 85 shows the elements and their relations. These are defined as follows: the main element of the technical view is the 'UOM' element. The main task is to hold the relations to the other less technical views (see paragraph 'Region Relations' at the end of this chapter). It is the entry point of the model. A 'UOM' represents the 'Unit' (U3(1→2)) in this part of the model. This separation was made to simplify the model. The 'Unit' element represents the technical viewpoint. It includes a unit type (equal to SCA types in Section 4.4.1.1) that is represented by the elements 'Snippet', 'Service', 'Class' and 'Component' (U3(2→15,16,17,18)). Also, a 'Unit' has a description for the runtime 'Technology' (U3(2→13)) that includes a description for the programming language 'Characteristic' (U3(13→14)). The technology description is also used for the description of the 'Technical

Realised models

Environment' (U3(20→13)) that is used as a 'Dependency' (U3(23→13)) of 'Files' (U3(12→23)). Also other files can be dependencies of existings files (U3(23→12)). The file association to the 'Unit' element is described by two different content types. The first one is the 'Human Readable' content (U3(2→3)). This content is represented by 'Video', Audio' and a 'Document' element (U3(9,10,1,21→3)). The second one is the 'Machine Readable' content (U3(2→4)). This is defined as a 'Class', 'Binary Data' and 'Service Information' element (U3(5,6,7→4)). Basically, both content elements 'Data' elements (U3(8→3) and U3(8→4)) can be represented by real files (U3(8→12)) with typical file properties. For the relation of files to different SCAC models, the typical 'File Structure' and 'Folder Structure' is used if 'Files' has to be described (U3(19→12) and U3(22→12)). These two elements are also related because of the structure of files is part of the folder structure in this model (U3(19→22)).

The last relation to be described is that a 'Unit' has a content definition described in the content view region.

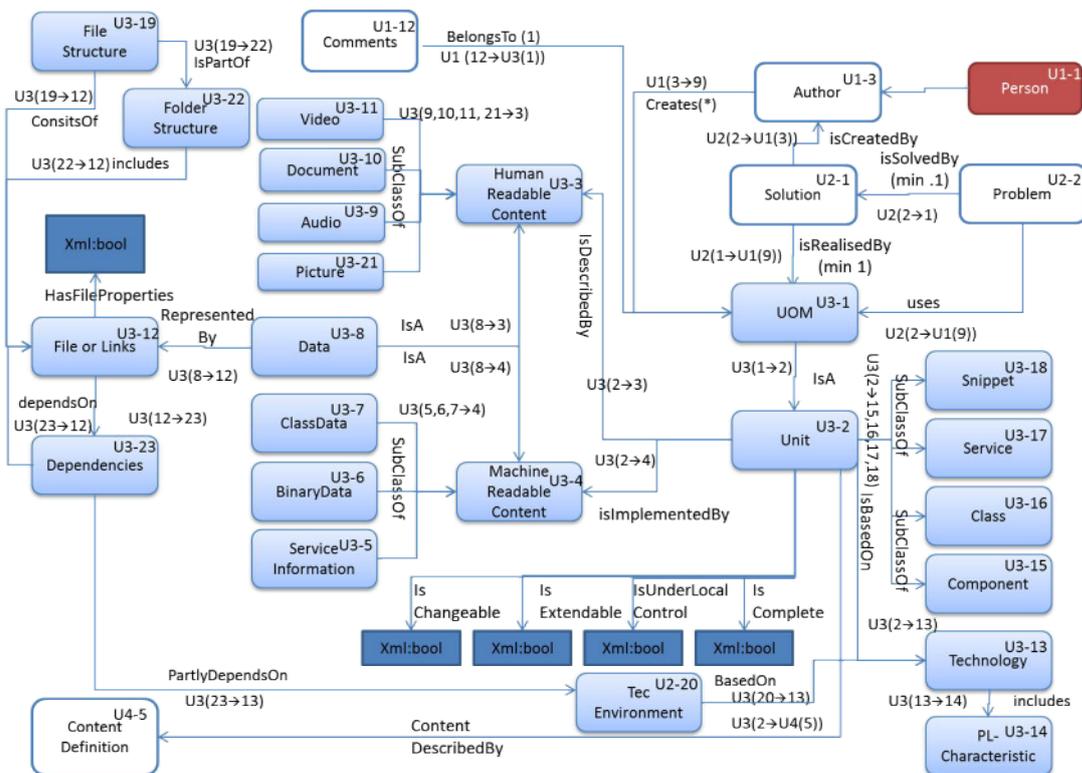


Figure 85 - Relevant elements of the area 3 – technical view (U-R3)

Realised models

To classify the different unit types, special attributes are part of the ‘Unit’ element. A ‘Snippet’ for example is changeable, extendable, under local control and not complete. A ‘Service’ is not changeable, not extendable, not under local control and complete. A ‘Class’ is changeable, extendable, under local control and complete. A ‘Component’ is not changeable, not extendable, under local control and complete. Table 39 summarises these classifications:

		Attributes			
		IsChangeable	IsExtendable	IsUnderLocalControl	IsComplete
Unit Types	Snippet	X	X	X	-
	Class	X	X	X	X
	Service	-	-	-	X
	Component	-	-	X	X

Table 39 - Classification of unit types

5.3.1.5. Region 4 “Content View”

The research area does not focus on a search for UOM. A software construction reuse process requires the search for UOMs or SCACs. Therefore, the realised model describes the description of the content from a business and professional view. The description of more technical perspectives (e.g., interfaces) is not provided by the model. Such information can be added by using research results of other research (cf. Combination of semantic models in Parreiras, 2012). Table 40 describes the 15 elements that are used in the content view area.

No.	Element	Description
U4-5	Content Definition	This element is the main element of region U4-R4 (content view). It represents the professional content of the ‘Unit’ element.
U4-11	GUI	This element defines a “Contentdefinition” element as a GUI type. The content of the related unit is a user interface or contains user interface information.
U4-14	Function	This element defines a “Contentdefinition” element as a function type. The content of the related unit is a set of functions.
U4-12	Structure	This element defines a “Contentdefinition” element as a structure type. The content of the related unit is structured information, (e.g., an interface).
U4-15	Data	This element defines a “Contentdefinition” element as a data type. The content of the related unit is data.
U4-2	Subject	This element is a substantive.
U4-9	Verb	This element is a verb.
U4-3	Optional Information (Tags)	This element extends the content element with a set of subjects (Tags) and provides keywords used for the search of a “Unit” element.

Realised models

No.	Element	Description
9	Synonym	This element contains synonyms for substantives or verbs.
U4-4	Subject - Verb Combination	This element is a relation between “Subjects” and “Verb” elements. This is used to search for key substantive and verb pairs related to a specific software unit.
U4-7	Synonym Relation	This is an extension point to other semantic models dealing with subject synonyms.
U4-8	Verb Relation	This is an extension point to other semantic models dealing with verb synonyms.
U4-10	Synonym	This element is the main element and represents synonyms in this model. All other synonyms are related to this base class.
U4-1	Subject Synonym	This element represents synonyms for subject elements.
U4-6	Verb Synonym	This element represents synonyms for verb elements.

Table 40 - Defintion of elements of the content view (U4)

In the following, the relation between these elements will be explained and are shown in Figure 86.

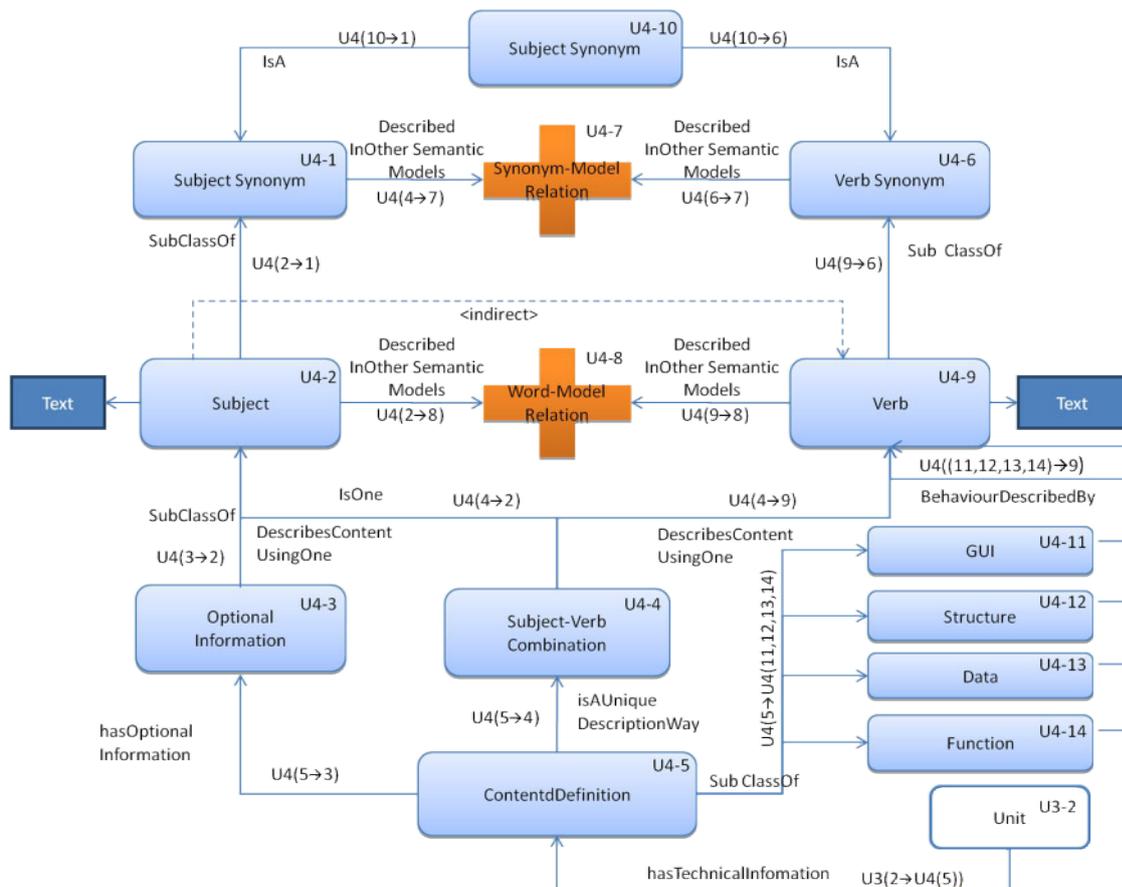


Figure 86 - Relevant elements of the area 4 – content view (U-R4)

Realised models

The main element in this area is the ‘ContentDefinition’ element. It represents the professional description of software unit content (U3(2→U4(5))). A ‘ContentDefinition’ includes a simple ‘Subject-Verb’ combination (U4(5→4)) that includes a ‘Subject’ (U4(4→2)) and a ‘Verb’ (U4(4→9)). This relation is used to describe the content using two words. The ‘ContentDefinition’ element can be a ‘GUI’, ‘Structure’, ‘Data’, or ‘Function’ element (see Section 4.4.1; U4(5→11,12,13,14)). This relation is combined with the ‘Verb’ (U4(11,12,13,14)→9) to give a user the possibility to search for the content by using a ‘Substantive-Verb-ContentType’ relation. Also, a ‘Subject’ element can be described more precisely using ‘Optional Information’ (U4(3→2)). Such textual information is part of the ‘ContentDefinition’ element (U4(5→3)). Also, the ‘SubjectSynonym’ and the ‘VerbSynonym’ have a relation to a relation element to other semantic models (‘Synonym-Model Relation’ element, (U4(6→7) and U4(4→7)). ‘Subject’ and ‘Verb’ elements can be expressed using ‘Subject synonyms’ elements (U4(2→1)) or ‘Verb synonyms’ (U4(9→6)). Both synonym elements are based on the ‘Synonym Element’ (U4(10→1) and U4(10→6)).

This area includes some extension points to relate the model with other semantic models. The ‘Subject-Verb’ relation for example can be expressed by using other word-relations models (U4(2→8) and U4(9→8)).

5.3.1.6. Region relations

The previous descriptions of the different regions of the model also include relations between these regions. To create a non complex view on these relations, this paragraph shows them separately.

The four areas shown in Figure 87 are related by using the ‘UOM’ and ‘Unit’ element. The ‘UOM’ element represents a non-technical element and, therefore, is connected to Region 1 and Region 2. An ‘UOM’ has a ‘Problem’ description (U2(2→U3(1))) and realises (by using the ‘Unit’ element) a solution for this problem (U2(1→U3(1))). The ‘UOM’ element also has two relations to region 1. An ‘Author’ as the creator of the software unit (‘UOM’,

Realised models

(U1(3→U3(1))). The second relation is between the ‘UOM’ and the ‘Comment’ element. A ‘UOM’ can include comments (U1(12→U3(1))). The ‘Unit’ element is a more technical description of an ‘UOM’. It includes a relation to the ‘Content Definition’ element in region 4 (U3(2→U4(5))). This extends a unit with the description of the professional content.

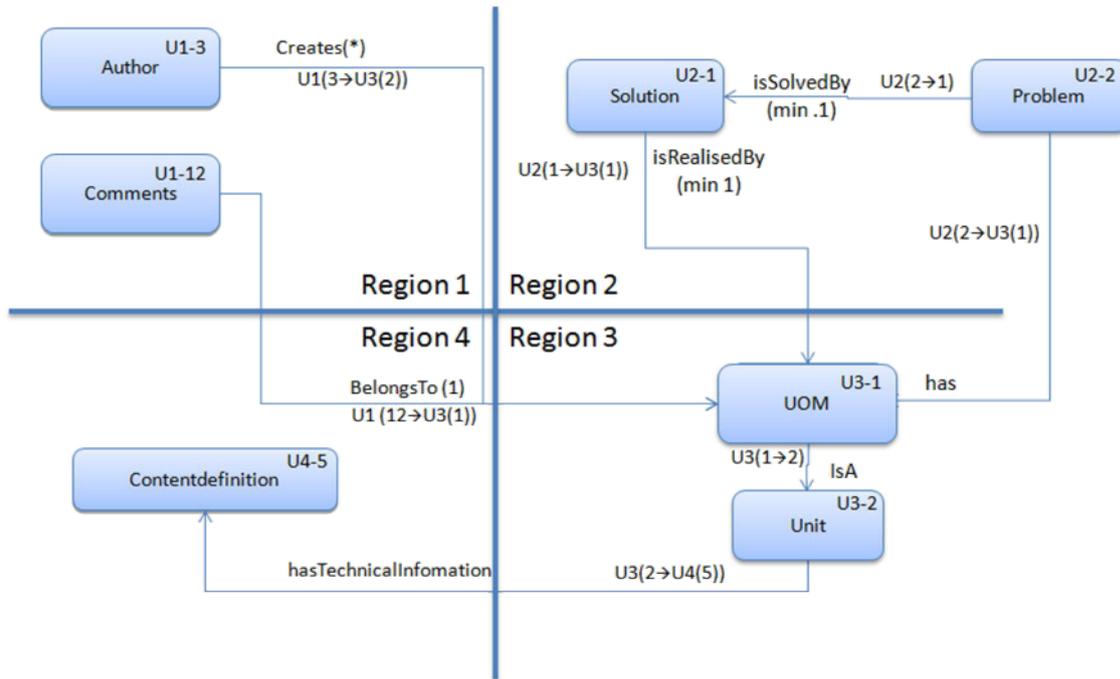


Figure 87 - Relevant relations between the different areas of the U-Model

5.3.2. Transformation model instance

The concept description in Section 4.5.4.2 shows that SCAC models extend the fundamental U-Model with SCAC specific information. In this section, the realised Transformation SCAC model is explained. This includes the explanation of the elements, their relationship, as well as the relationship to the existing Unit Model. Figure 88 shows these elements and their connections:

Realised models

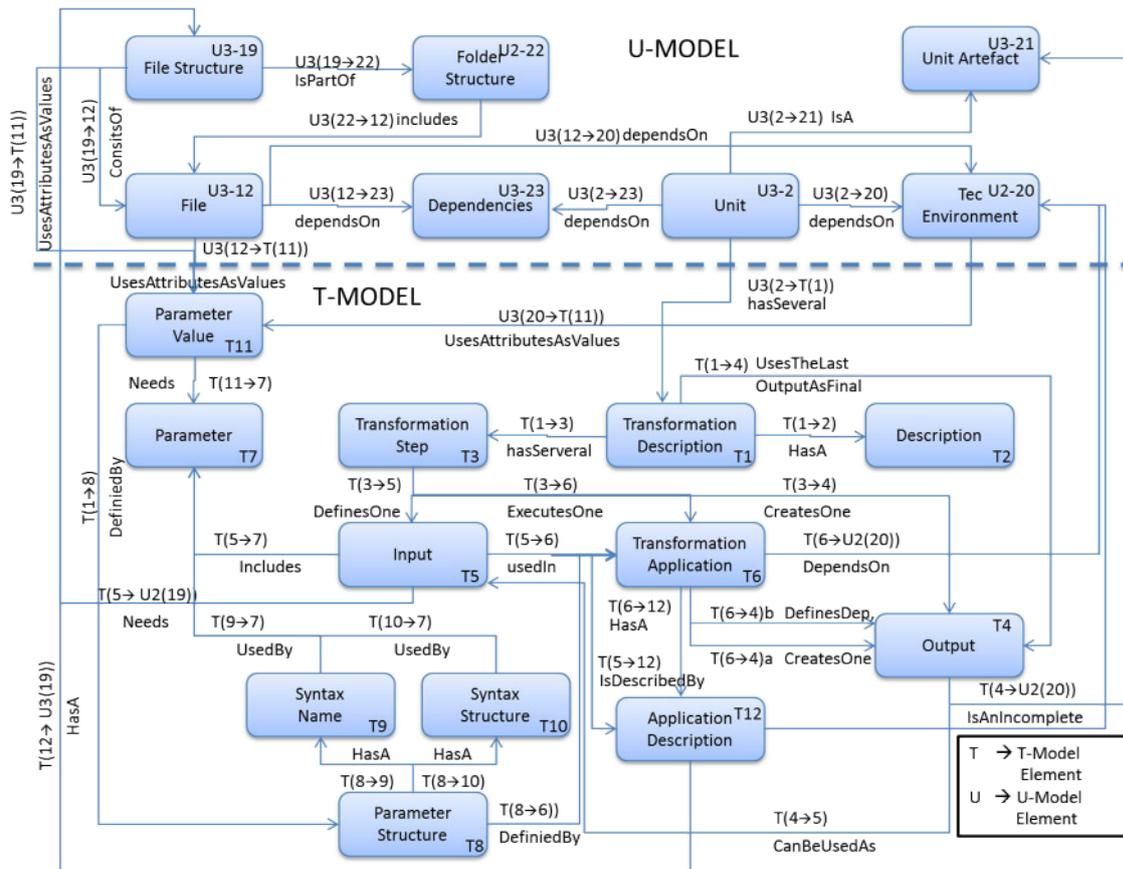


Figure 88 - Data model extension for transformation activities maintenance

In this area, the main link between the transformation model extension (called T-Model) and Software Unit Model (called U-model) is defined. The central element of the T-model is the ‘Transformation Description (T1)’. It represents a complete transformation. The connection is that one unit of the U-model has one or more T1 (U3(2→T(1))). T1 describes a transformation which again consists of (T(1→3)) several ‘Transformation Steps’ (T3). Furthermore, T1 has a textual ‘Description’ element meant for human readers that describes the overall transformation process (T2, T(1→2)). The last link of T1 is an ‘Output’ element of the last transformation step (T4, T(1→4)), which is the result of the entire transformation.

T3 provides a transformation description, which means the execution of an application transformation. T3 is, therefore, associated with the other three areas of the T-models (T(3→5),

T(3→4), and T(3→6)). These represent the input (T5), the transformation application itself (T6) and the result of T3 (T4).

5.3.2.1. Input of a transformation application

The 'Input' (T5) of a transformation application contains (T(5→7)) all necessary input parameters (T7) and is used by (T(5→6)) this application (T6). An instance of T7 is described by three relevant aspects. The first is the parameter structure (T8), whose (T(8→9,10)) syntax name (T9) and syntax structure (T10) are defined by the transformation application (T6) (T(8→6)). T8 and T10 are used by T7 (T(9→7), T(10→7)). The second relevant area concerns the values to be entered in the parameters. In the context of this thesis, these are text values, which are either freely definable values or defined by certain properties of files (U3-12, U3(12→T(11))) or structure information of files and folders (U3-19, U2-22, U3(19→(11))), which come from the software unit to be transformed. In addition, these values come from properties of the technical environment (U2-20) which are required by the software unit (U3-2) or the transformation (U3(20→T(11)) and U3(2→23)).

5.3.2.2. The transformation application

The actual transformation application (T6) is described by means of three relevant elements. The first element is comprised of the input parameters (T5, T(5→6)). The next element is the result of the performed transformation application (T(6→4 a and b), T4). The last element is the application description (T12, T(6→12)). T12 includes the files and folders (T(12→U3(19))), which the application consists of, the dependencies from runtime environment (U3(12→20)), and the definition of the start file and the execution environment as attributes. This dependency relation can be expressed by T(12→U2(19)) also.

5.3.2.3. Transformation result

The result of a transformation application is represented by the area T4 in Figure 88. Essential in context with the U-model is that T4 represents a software unit (U3-21) with missing

Realised models

additional files in the sense of the U-model ($T(4 \rightarrow U2(21))$). Users can add other artefacts to the result of a transformation, (e.g., documentation to create a new reusable software unit in the sense of the U-model). This is done by reference to other software unit artefacts. T4 may have dependencies, (e.g., files, system variables, etc.), defined by the application.

Each transformation step can have an output ($T(3 \rightarrow 4)$). This output can be used for later executed transformation steps as input ($T(4 \rightarrow 5)$).

5.3.3. Deployment Model instance

The deployment model was created by analysing three different Web Service deployment approaches. The study's test subjects consisted of the following three embedded device engines: Advantys STB using the Sonata engine, Advantys STB using the Dynamic Deployment engine and GX300 Gateway using OSGI Deployment (see Zinn et al., 2012a). In the following text, the model shown in Figure 89 and Figure 90 is explained.

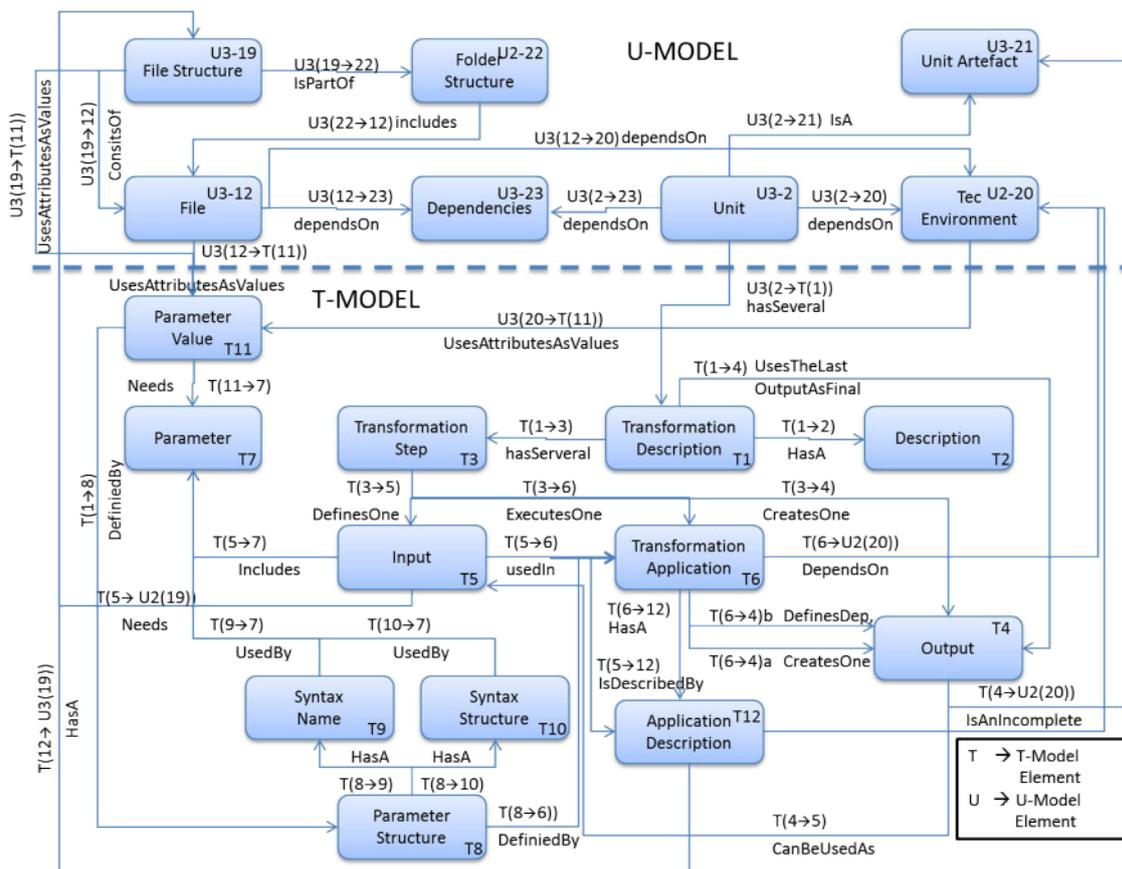


Figure 89 - Data model extension for deployment activities Part 1

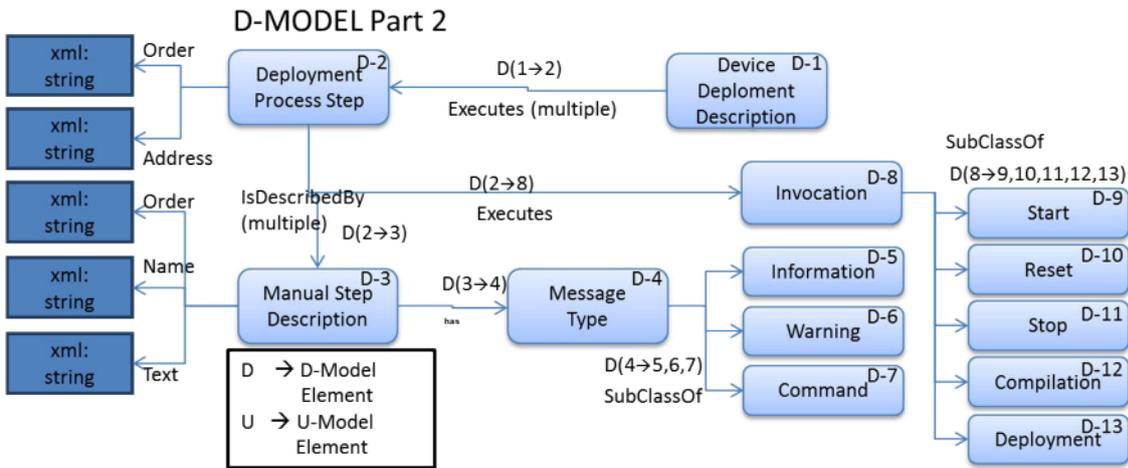


Figure 90 - Data model extension for deployment activities Part 2

5.3.3.1. Maintenance

In this area, the main link between the model extension (D-model) and the model to extend (the U-model) is defined. The central element of the D-model is the ‘Device Deployment Description’ (D1). It represents a complete focused device deployment. The connection is that one unit of the U-model has one or more ‘Device Deployment Description’ elements. The relation is realised by creating a ‘Device Deployment Extension’ that is related to all ‘Device Deployment Description’ elements (U25, U3(25→D(1))). This extension is related to the ‘Extension’ element U3(24→25) of a unit that relates to all SCAC related extensions (U23, U3(2→24)). Additionally, D1 has a simple ‘Description’ element that describes the complete deployment SCAC for the user search (D14, D(1→14)).

D1 describes a complete device deployment activity which again consists of several deployment steps (D2, D(1→2)). Each of these steps has a ‘Manual Step Description’ meant for human readers and describes the overall device deployment process (D3, D(2→3)). A D3 has a ‘Message Type’ (D(3→4)) that can be an ‘Information’, ‘Warning’ or a ‘Command’ element (D(4→5,6,7)). A D3 has the attributes: ‘Order’, ‘Name’, ‘Text’. The order describes the order of all messages. The name is used as a simple topic of the message and the text attribute includes the text part of the message.

An experienced user can describe the invocation step for each D2 D(2→8). This is an order for the automation software or for the user in the manual step description. An ‘Invocation’ can be a ‘Start’, ‘Reset’, ‘Stop’, ‘Compilation’ or ‘Deployment’ element (D(8→9,10,11,12,13)).

5.3.3.2. Input of a deployment application

Like a transformation SCAC, the focused deployment SCAC, also uses console-based applications to deploy the software units to the devices. As a result, the input part of a deployment model is similar to the transformation model. The Input (D15) is used by ‘Deployment Application’ (D16, D(15→16)) and includes, for this application, several parameters (D17, D(15→17)). The ‘Parameter Values’ needed by the ‘Parameters’ (D18, D(17→18)) have a ‘Parameter Structure’ (D19, D(18→19)). This structure is simply defined as ‘Syntax Name’ (D20, D(19,20)) and (D(D21, D(19, 21))). Multiples of these simple parameter pairs can be used by the ‘Parameter’ element (D(9,10 →7)).

The input of a deployment step can be external files and deployment information. This information can be represented by the parameters. External file information is expressed by the relation (D15→U3(19)) and (U3(12→D(18))).

5.3.3.3. Deployment application

The output of a D2 is defined and created by the deployment application (D16, D(16→23ab)). This application depends on a ‘Technical Environment (U2(20), D16→U2(20)). An application has an ‘Application Description’ element (D22, D(16→22)) that includes the technical environment requirement of the application (D(22→U2(20))) that can be used by the input of a (D2 (D15→D22)). The application description also defines external files and their file structure that are necessary for the deployment process or step (D22→U3(19)).

5.3.3.4. Result of a deployment SCAC

The result of a single D2 (D(2→23)) can be used as input of the next D2 (D(23→15)). From the scope of the research, the deployment SCACs output is not relevant. For future research in this

area, it may be interesting to see software units executed on embedded devices also as reusable software units.

5.3.4. Integration model instance

The idea of this integration model was published by Zinn et al. (2011b). For the creation of this model, the integration features of two IDEs (Visual Studio 2010 and Eclipse ‘Juno’) were analysed. Additionally, it was proven how the result information can be related to the U-Model. In the following, the realised integration model shown in Figure 91 is described.

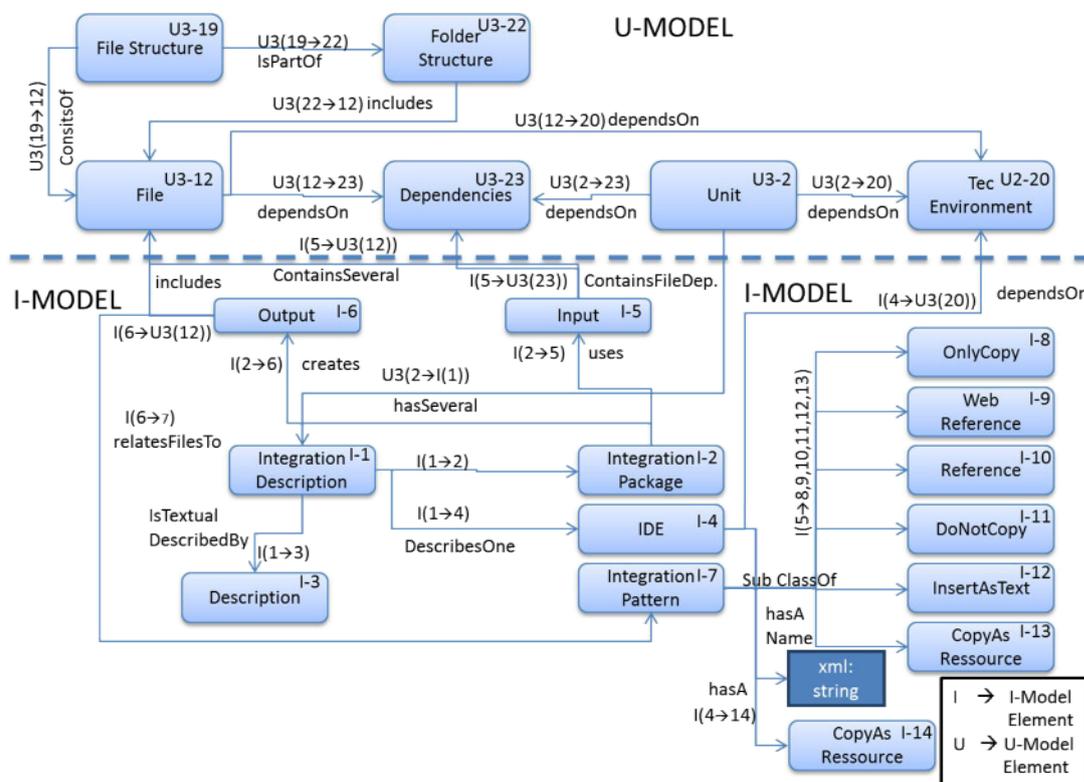


Figure 91 - Data model extension for Integration activities

5.3.4.1. Maintenance

In this area, the main link between the model extension (I-model) and the model which the authors wish to extend (the U-model), is defined. The central element of the I-model is the ‘Integration Description’ (I1) that represents a complete integration. The connection is that one ‘Unit’ of the U-model has one or more I1 (U3(2→I(1))). I1 describes an integration which again

consists (I(1→2)) of several ‘Integration Packages’ (I2). Furthermore, I1 has a textual description meant for human readers, and describes the overall transformation process (I3, I(1→3)). The last link of I1 is the description of the focused IDE (I4, I(1→4)) into which the different files have to be integrated.

I2 provides an integration of several files of a software unit. It is, therefore, associated with the other two relevant elements of the I-model (I(2→5) and I(2→6)). These represent the input (I5) and the integration output description (I6).

5.3.4.2. Input of an integration activity

The input (I5) of an integration package contains (I(5→U3(12))) all necessary files from a software unit. This also includes the dependencies of these files (I(5→U3(23))).

5.3.4.3. Result of an integration

The result (I6) of an integration package includes all files and dependencies defined in the input (I5, I(5→U3(12,23))). In addition this element may contain files and folders (described by U3-12, I(6→U3(12))) which are not part of the focused software unit. Each file will be described by an integration pattern (I7, I(6→7)). This pattern includes different values:

- OnlyCopy (I8, I(7→8)): This copies a file without referencing it in the solution tree of the project. This is necessary for second level dependencies that are not controlled by the IDE environment.
- WebReference (I9, I(7→9)): This marks a file as a web reference. Different IDEs utilise different methods for managing this information. For example, Visual Studio can use a WSDL file to create a reference to a web service that is based on the corresponding WSDL description.
- Reference (I10, I(7→10)): This copies a file and includes it in the solution tree of the project. This is a traditional reference that can be included or imported. This is necessary for managing the dependencies of a unit.

- DoNotCopy (I11, I(7→11)): This prevents a file from being transferred into a project's environment. Not all the files that are included in a unit are necessarily required by the IDE (e.g., documentation).
- InsertAsText (I12, I(7→12)): This flags the content of a file that is to be treated as text when loaded into the IDE. This is useful for code references (using or import) and code snippets.
- CopyAsResource (I13, I(7→13)): This flags a file to be used as a resource and includes it in the project. (e.g., configuration files).

5.3.4.4. Integrated development environments

The 'Output' information of the 'Integration Package' is used in an IDE. It is, therefore, necessary to define the focused IDE. An 'IDE' (I4) has a simple 'Product Name', an 'Endpoint Description' (I15, I(4→14)) and a 'Technical Environment' (U2-20, I(4→U2(20))).

The 'Endpoint Description' is all the information needed to connect the Prometheus environment to a specific IDE. The 'Technical Environment' is also used in the description of files. From a semantic point of view, this node can be used to validate the compatibility between a software unit and the platform of the IDE. The 'Technical View' part of the U-Model includes the description of technology that can be used to describe the possible technologies an IDE need to support for the integration (using the 'Technical Environment' element). Based on this information, the Prometheus environment may prove if such an IDE is connected. This is possible by the relations of technology and platform characteristics described in Figure 88.

5.4. Usage concepts

In this section, the usage concept of the realised Prometheus environment is shown. The aim is to describe the different user profiles based on this realisation. This supports the understanding of the experiment in the next chapter where this realised environment is used. Therefore, the

Uses Cases (based on Section 4.4.3) will be discussed in more detail in the different related sections. The usage concept is based on the information represented by the models described in Section 5.3.

5.4.1. Focused use cases

Figure 92 and Figure 93 extend Figure 49 with a complete Use Case scenario for the Prometheus environment. This extension is related to the realised Prometheus environment. In the following section, each Use Case (see Figure 93) and its relation will be explained. Therefore, each case will be described by an activity diagram and the explanation of the user interface used (if this was needed for the specific use case). Both (diagram and interface) describe the behaviour and the UI of the used Prometheus environment. The used graphical user interface was created in the scope of the research for the company Schneider-Electric. The internal name for this project was ‘Corporate Repository’. The selected UI parts shown in this chapter serve to aid the reader to understand technical descriptions and examples in the following sections.

Note: Note: This graphical user interface is not part of the investigation of this work. The research focuses on Prometheus as a service platform; therefore, the user interface shown is one possible representation of the focused interaction.

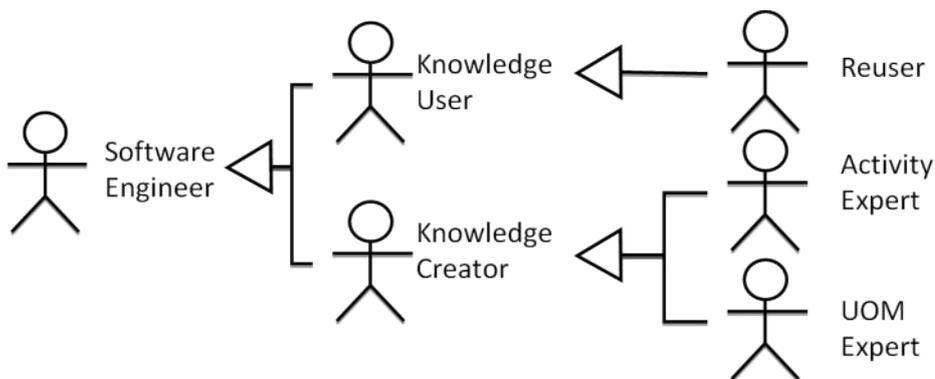


Figure 92 - Focused stakeholder of the Prometheus environment

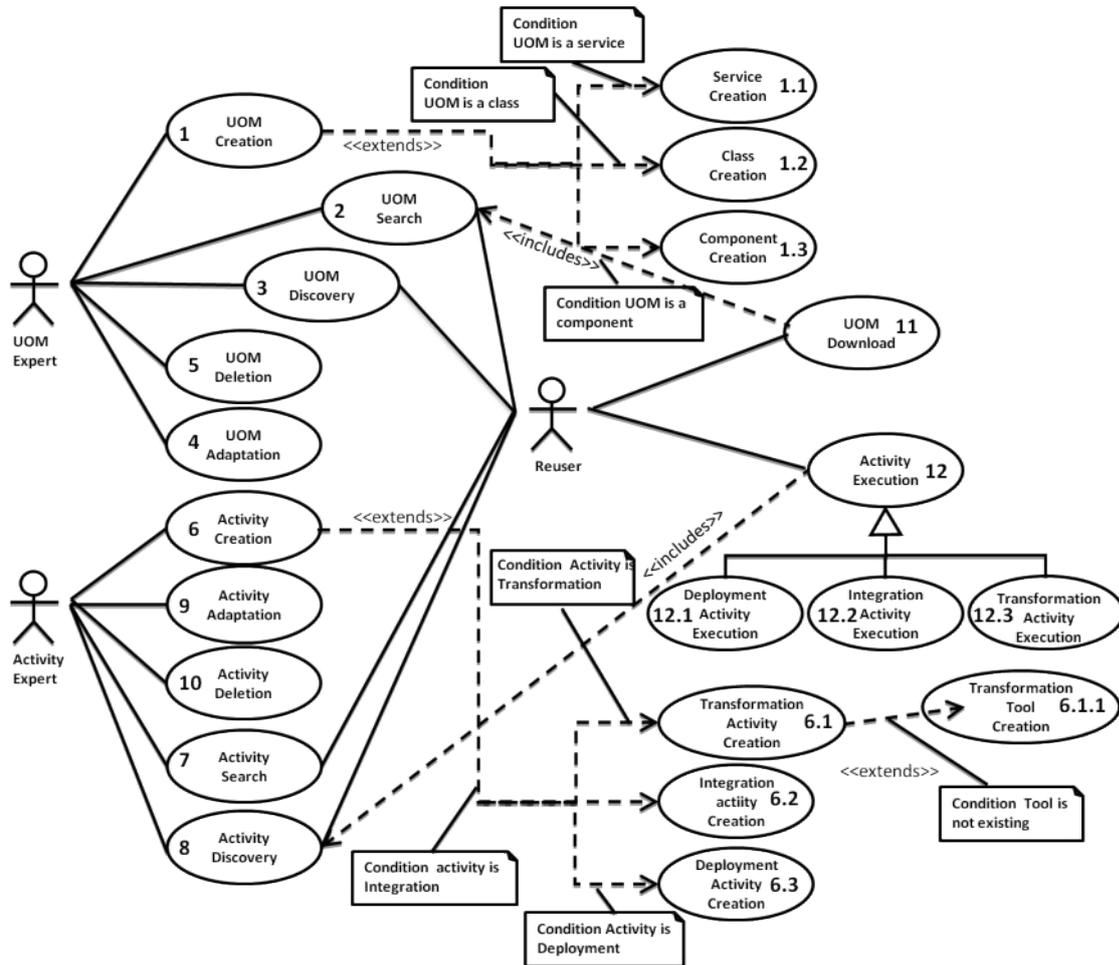


Figure 93 - Use Case diagram for the focused Prometheus environment

Figure 92 shows the previously mentioned Software Engineer types, Knowledge User and Knowledge Creator. In the realisation of the concept (cf. Chapter 4) in this chapter, the Knowledge User is called ‘Reuser’ and the Knowledge Creator is separated into two stakeholders. The first one is the Activity Experienced user. This stakeholder profile describes an experienced user for one or more SCACs. The second one is the UOM Experienced user who knows one or more software units very well.

5.4.2. Knowledge creator profile use cases

In Figure 93, the knowledge creator profile is used for two different profile characteristics. The first is the UOM Expert user who describes a software engineering experienced user for a

specific UOM. Such a person wants to provide his experienced user knowledge in the given scenario. The second characteristic is the Activity Experienced user; such a person is experienced user in one or more reuse activities (SCAc) and wants to share their experienced user knowledge in the given scenario.

Note: For each use case, a user interface is required an example of the used user interface is given. Because of it is a simple remove of data in a database. The use cases for deletion of UOMs and activities are not described.

5.4.2.1. Use case 1 – create UOM

A software unit is created by a UOM Experienced user. For this, the Prometheus server searches for connected repositories. The server determines whether the addition of information into the repository is allowed. In the next step, the user can create a software unit. The system asks for the following metadata. Note: See Section 5.3 for the underlying data model.

- Authors and responsible persons (e.g., name, surname, date of birth, contact information): This metadata describes people (stakeholder) related to the software unit (see Section 5.3.1).
- Name, type, content type and initial descriptions of the software unit. These data are metadata that describe the component directly: specific descriptions of the types (GUI, Function, Data and Structure) and the content type (class, component and service) (see Section 5.3.1). Selecting a type realise Use Case 1.1, 1.2 and 1.3. These Use Cases are not explained separately in this thesis.
- Artefact affiliation: The artefact affiliation is used to determine which area of responsibility the real software unit is in (e.g., logging). This is done by a simple description by the experienced user.

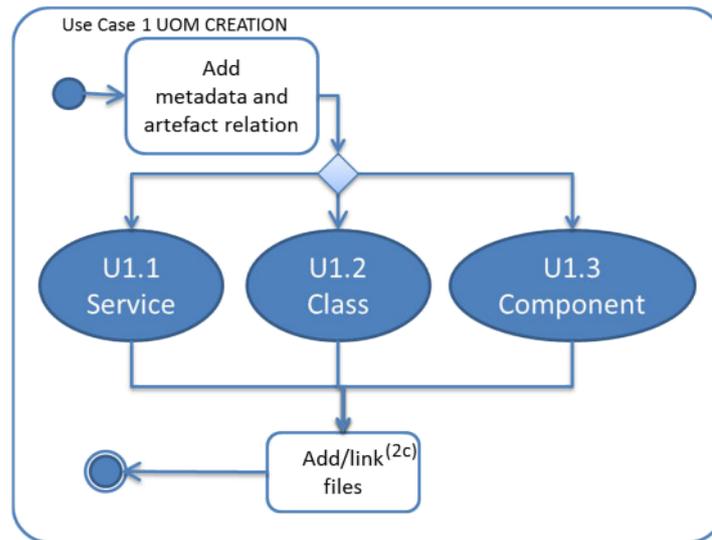


Figure 94 - Activity diagram for Use Case 'UOM Creation'

After this initial creation, the software unit parts (files) can be inserted into the system. This requires the following information for each UOM to be made (see UI in Figure 95):

- File information, (e.g., file name, file size, file extension (type), creation and modification date).
- Prometheus metadata, (e.g., content type, packet affiliation; see Section 5.3.1).
- Additional metadata, (e.g., creation time, description; see Section 5.3.1).

This step is repeated for all files of a software unit. If the files already exist in another repository, these can be linked by entering the download link or file transfer link into the file element description. The artefact concept explained in Chapter 4 is used to create packages of UOMs based on business content. Therefore, an artefact includes information that is equal to the metadata of a UOM. Additionally, UOMs can be added to the artefact. Figure 94 shows the use case as an activity diagram.

Note: Creating, editing or deleting an artefact is not part of this description. These handlings are equal to the creation, editing and deletion of UOMS.

Usage concepts

Repositories (1) Artefacts (2) UOM FileElements Inetgrations Transformations

Prometheus SQL S DPWS DPWS4J activation.jar Transformation1

Artefact Create Artefact Edit UOM Create UOM Edit FileElement Create FileElements Create FileElement Edit Integration Create Transformation Activities Transformation Appli

Artefact Base Information

Id a44d0a98-d7b9-4682-91d4-c4f47cbbdead Name UTest

Path test Type Service

(3)

Figure 95 - UOM creation UI

Figure 95 shows the user interface for generating UOMs.

Note: All UI elements of the wizard are shown in this figure. The wizard guides the user through the different processes (e.g., UOM creation) and shows only necessary UI elements.

This is done to simplify the UI explanation.

First, the user has to select a repository to store the UOM information (1). In the next step, the user selects an existing SCA or creates a new one (2). For the creation of a UOM, a user has to enter the metadata for UOMs (3).

Repositories Artefacts UOM FileElements Inetgrations Transformations

Prometheus SQL S DPWS DPWS4J activation.jar Transformation1 (1)

Artefact Create Artefact Edit UOM Create UOM Edit FileElement Create FileElements Create FileElement Edit Integration Create Transformation Activities Transformation Application (2)

File Base Information

Id 4ea7ef92-c59a-4ee0-a3cd-338c50655def PackageID b95915cb-387b-4467-aa81-eda57285d8b1

DateOfPublication 12/29/2012 8:42:05 PM FileElementType

FileName Test.txt Size 12222

(3)

Figure 96 - File element creation UI

Figure 96 shows the file element creation UI. This field is accessible (2) after selecting a UOM in the selection area (1). Then all information shown about file elements in the U-model can be entered (3).

5.4.2.2. Use case 2 – UOM search

This use case is used by two stakeholders. The UOM-Experienced user and the Reusers are rarely able to search for a UOM (see Figure 97). Therefore, the user (1) defines keywords and (2) starts the search by sending a search request:

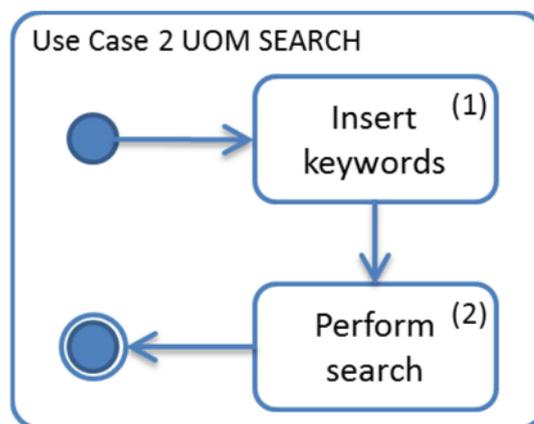


Figure 97 - Activity diagram of use case 'UOM SEARCH'

The use case is provided to users through a user interface. This focused interface is a web application based on Silverlight (See Section 5.2.1.1) and can be accessed with a web browser using a Unified Resource Identifier (URL). The user is then able to specify their search terms in a search box (1). By confirming the search button (2), the query is sent to the Prometheus server. Figure 98 shows the elements of the search user interface.

Note: The other graphical elements in Figure 98 are typical of a content management system and not relevant to this thesis. They are not explained.

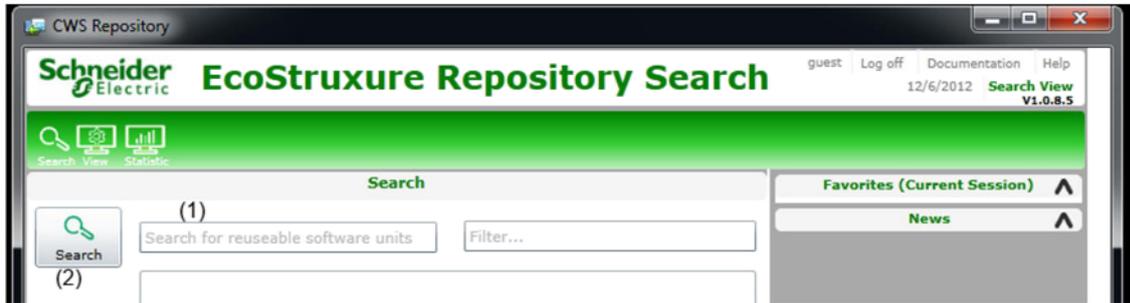


Figure 98 - User Interface for UOM search

5.4.2.3. Use case 3 – UOM discovery

This use case is used by an experienced user. This user has to confirm if the result of a search contains a UOM which is suitable for his requirements (see Figure 99), Therefore, this use case includes Use Case 2 “UOM Search” with the condition that the search request result contains at least one UOM.

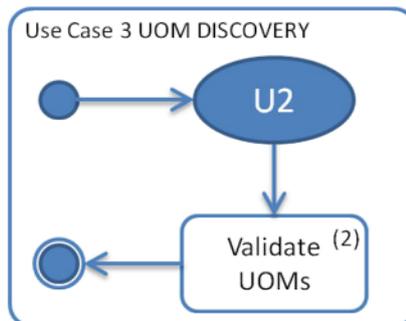


Figure 99 - Activity diagram of use case ‘UOM DISCOVERY’

The search results (found software units) are displayed in a compact form and can be shown in a more detailed presentation as required. Figure 100 shows an example of the more detailed representation. This includes both metadata about a unit, such as author and description, as well as information on possible executable activities (e.g., download of software unit information and SCA information).

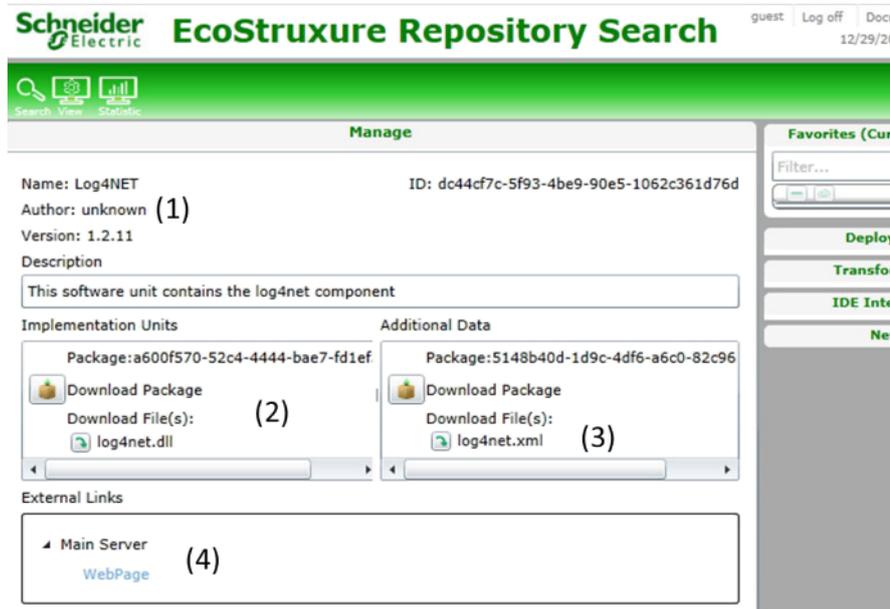


Figure 100 - Detailed presentation of a software unit

Figure 100 shows a detailed presentation of a software unit, where varied information is presented. Based on this user interface, a user is able to:

- find out who created it and who is responsible for this software unit (1),
- read additional descriptions (3),
- analyse and download unit artefacts or additional documents (2) (4), and
- explore links to other kinds of content (e.g., web pages).

Based on this information, the user may be able to decide whether this element is useable or not. The UI shown here is not used for UOMs or activity experienced users. These users use the UI shown in Use Case 1 for creating and changing information. Therefore, they use the wizard to select elements such as SCAs and UOMs.

5.4.2.4. Use case 4 – UOM adaptation

This use case is used by the experienced user. After the successful discovery of a UOM (Use Case 3), a user (2) is able to change the metadata of a UOM or change other values (e.g., files,

Usage concepts

dependencies, etc.). The user (3) can decide to save these changes or (4) or to cancel the operation (see Figure 101).

Note: The selection of a UOM is not defined as a standalone use case.

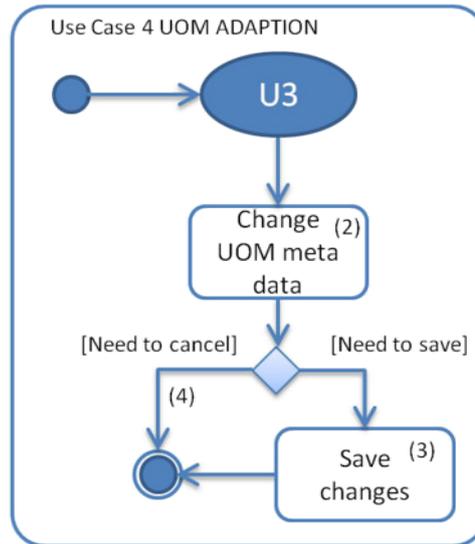


Figure 101 - Activity diagram of use case 'UOM Adaptation'

The user interface described in Use Case 2 is also used here (see Figure 100). By adding new files or dependencies, a wizard begins and this supports the user in adding information.

Repositories	Artefacts	UOM	FileElements	Integrations	Transformations
Prometheus SQL S	DPWS	DPWS4J	activation.jar		Transformation1 (1)

Artefact Create	Artefact Edit	UOM Create	UOM Edit	FileElement Create	FileElements Create	FileElement Edit	Integration Create	Transformation Activi
Artefact Base Information (2)								
Id	4ce698fa-1194-4e4e-81d6-0c790cb1ce91					Name	DPWS4J	
Path	test (3)					Type	Service	
UOM Meta Information								
Id	0352c379-63b3-47a2-9a64-707d65f4b702					Website	www.soa4d.org	
Server	Prometheus2012					Category		

Figure 102 - Wizard to add new UOM file information

Figure 102 shows the UI for editing a UOM. After selecting the UOM (1), the editing field can be selected (2). After this, the metadata can be changed and other elements (i.e., file elements, or SCACs) can be created, removed or changed.

5.4.2.5. Use case 6 – activity creation

This use case is used by the Activity Experienced user. This user wants to create a reuse activity.

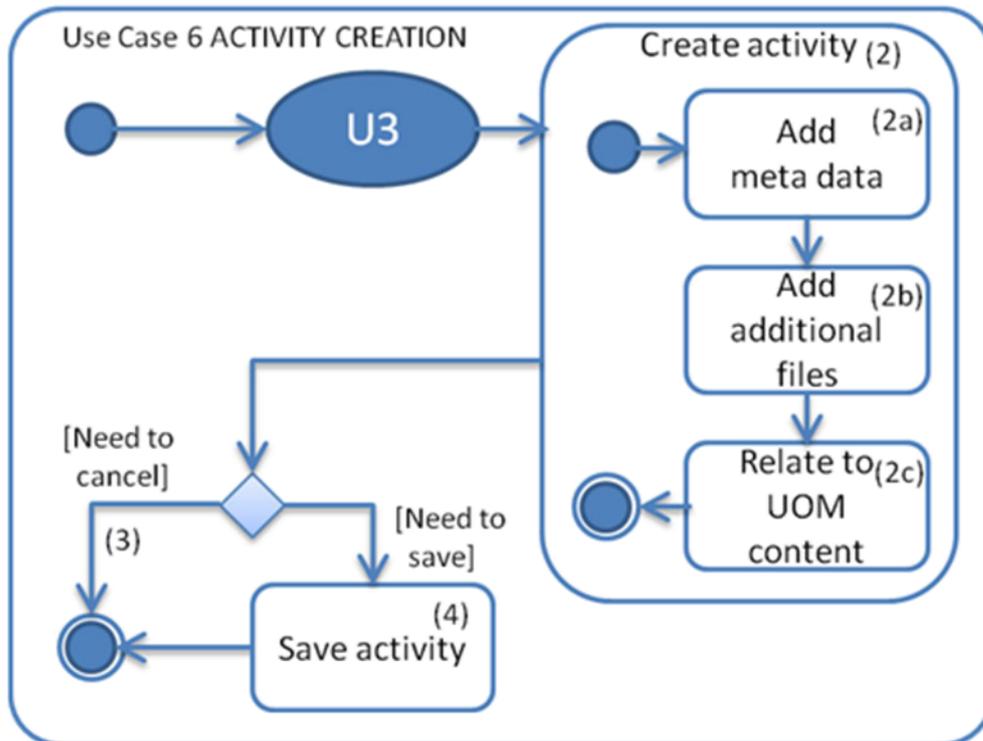


Figure 103 - Activity diagram of use case 'ACTIVITY CREATION'

Therefore, the user has to discover a UOM (Use Case 3), (2) create an activity by adding meta information and additional file information and relating existing information of the UOM to the new activity. The user (3) can decide at any time to cancel this use case or, (4) to save the new activity to the UOM (see Figure 103). A wizard is, therefore, used to create an activity. After the discovery of a UOM (see Figure 100), the user is able to click and add an activity. For each activity model (see Section 5.3), one use case and a UI wizard is created.

5.4.2.6. Use case 6.1 – integration activity creation

For this use case, a wizard is used to generate an integration reuse activity; therefore, three different UIs are used. In the first UI, different metadata can be entered by the user. In the second UI, additional files, which are not part of the UOM, may be added. This UI is also used to define how single files can be integrated into an IDE. The last UI is used to define the compatible IDE environment.

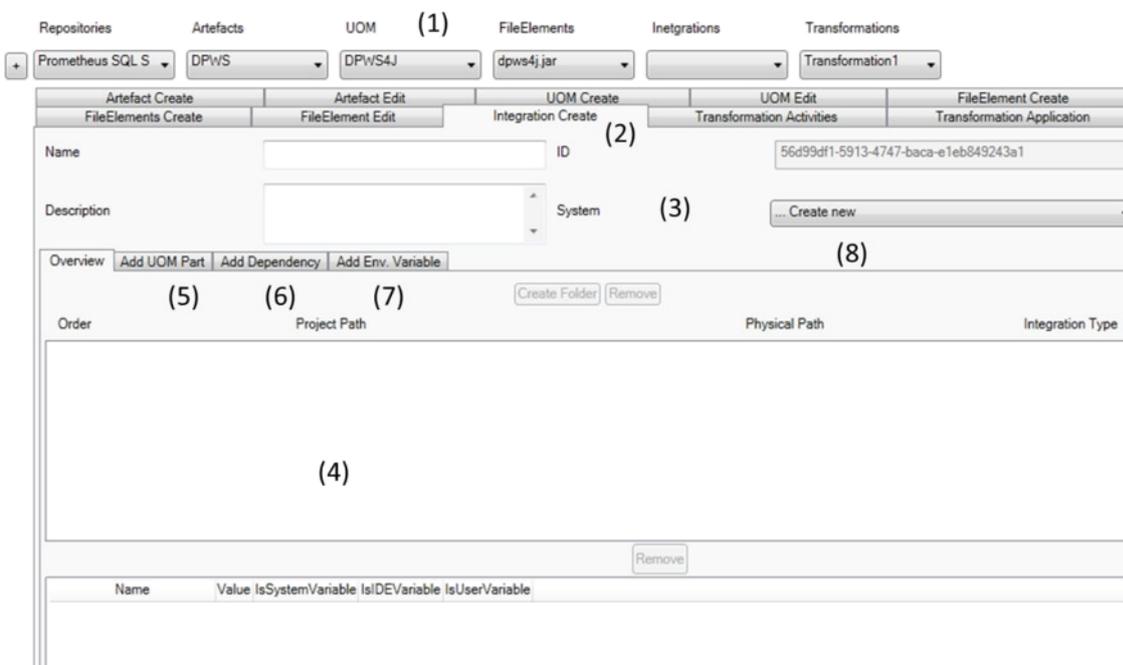


Figure 104 - Main UI for integration activity creation

For the creation of an integration rule, a corresponding menu item (2) is provided including a detailed user interface for software units (4). This is accessible after selecting an UOM (1). Figure 104 shows the settings page for an integration activity. Besides specifying general information (3), the user needs to specify the following information:

- The development environment where the data should be integrated (8)
- The UOM files for integration (5)
- The type of integration for each file (i.e., copy and specify reference) (5)
- The environment variables needed for the integration (7)

Usage concepts

- Optionally, one or more dependencies (e.g., external files) (6)

Note: Figure 104 shows the overview page.

In this study, the development environments Visual Studio and Eclipse were used as experimental subjects. Therefore, appropriate extensions for the Prometheus environment were developed. Accordingly, the user can select at this point between Visual Studio 2005, 2008 and 2010 and the Eclipse versions G4 and E5. The Prometheus environment compares the technical description of the selected development environment with the appropriate requirements of the selected software unit or integration rule and alerts the user of possible incompatibilities.

Note: The user is able to store integrate rules even if they are incompatible. This may be mainly used for experimental trials within a project. KU users have to be informed about such characteristics when they are selecting an integration.

After completion of the specification of integration rule or complete SCAc, it can be stored. The user has to perform a test. The Prometheus environment does not recognise the error and the user has to test and comment on the activities as successful or proven; the integration of user profile for KU is then released.

This use case extends Use Case 6 by adding integration activity-specific steps. First of all, the user (2a) has to set some metadata about the activity. In the next step (2b), the user can add some additional files necessary for the integration. After adding this information, the user (2c) is able to define the integration types and environment settings for each integrateable file. In the last two steps, the user (2d) can specify the integration environment that is able to handle the integration result (2e) and describe the integration result with additional descriptions.

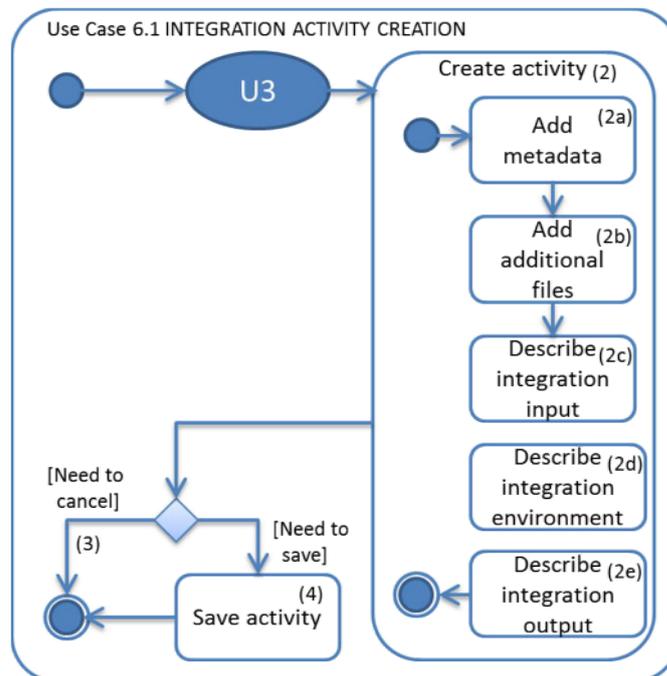


Figure 105 - Integration steps

In creating an integration activity, the user may be supported by the Prometheus Predictive Knowledge System. On the basis of existing knowledge of other integration-scale integrations, the system is able to create suggestions for the additional integration rules (case based reasoning). This system is not part of the thesis, however, a first publication was made as result of the research of this Ph.D. thesis (cf. Zinn, Fischer-Hellmann and Schoop, 2012a).

5.4.2.7. Use case 6.2 – transformation activity creation

This use case is realised by five different user interfaces sections. First of all, a Prometheus server needs to know information about possible transformation tools. This is a step not covered by the use cases shown in this thesis. The installation of the transformation tool is done by a system administrator. Figure 106 shows the UI used for the research. In the first area general information (e.g., name and platform technology) are entered by the user (1). In the second area, the parameter value key (defined by the transformation activity model) used to store all possible parameters and their value types (2). All entered parameters are shown in the overview section (3),

Usage concepts

The screenshot shows a software interface for setting up a transformation tool. It features several input fields and buttons. The 'Meta data' section includes fields for Name, Platform, ID, and FileName. The 'Parameters' section includes fields for ParameterSyntax, Value Type, Value, and Name. A table below the parameters section has columns for ParameterSyntax, ValueType, Values, Description, and Name. A 'Remove Parameter' button is located at the bottom of the table. A 'Save' button is on the right side of the interface.

Figure 106 - Example of an UI for transformation tool setup

Figure 107 shows the main UI for the creation of transformation SCAC. Next to the general information (1), the user can add transformation rules (2). Also, the user can set information about the output of the transformation (3) which the user can define in an additional wizard (4).

The screenshot shows a software interface for creating a transformation activity. It features several sections: 'Meta Information' with fields for Name, ID, From Tec, and To Tec; 'Transformation Rule Definition' with a table listing rules and buttons for Add, Remove, Edit; and 'Transformation Output' with fields for Type, Files, Tec, and Memo. A 'Define' button is on the right side of the interface.

Figure 107 - Main UI for transformation activity creation

For the creation of transformation rules, the detailed user interface for software units provides a corresponding menu item. Figure 108 shows the main settings UI for a transformation rule. Besides specifying the name, the user needs to enter the:

Usage concepts

- Definition of the target technology, the required runtime environment, and general metadata (1)
- Selection of appropriate transformation tools (2)
- Parameterisation of the transformation tools (3) based on the selected transformation tool (2)
- Define the file output of the rule (4). This includes the description if the rule output is used as input for other rules or as output for the transformation activity.

The screenshot displays the 'Main User Interface for transformation rule creation' with several key sections:

- Meta Information:** Includes fields for Name, ID (50a13862-82de-4967-8d0b-d58fc610e420), From Tec, To Tec, Call, and Memo. A dropdown menu for 'T-App' is visible.
- Parameter and I/O Definition:** Contains a 'Parameter Syntax' section with 'Choose Parameter' and 'Syntax' dropdowns, and a 'Parameter Value' section with 'Value' and 'Current Parameter' fields.
- Folder Options:** Features a 'Create Folder' section with a 'Label' field and a '+FC' button, and a 'Remove Folder' section with a '-' button.
- File Options:** Includes 'Add File (Future Creation Mode)', 'Add File (Copy Mode)', and 'Add File (Output Mode)' sections, each with a 'Name' field and a '+F' button.

Annotations (1) through (5) are placed on the interface to correspond with the usage concepts listed above.

Figure 108 - Main User Interface for transformation rule creation

In the first step, the user gives it to the target technology and the required runtime environment. Here, the user can choose from previously created target technologies and runtime environments, or define the transformation environments by themselves. The system indicates the potential for incompatibilities.

Usage concepts

Note: The user is able to store transformation rules even if these are incompatible. This may be mainly be used for experimental trials within a project. KU users have to be informed about such characteristics when they selecting an integration.

In the second step, the user selects a transformation application. It is also possible to insert new transformation application information into the system. Therefore, a user has to specify:

- All the necessary parameters including different values of a single parameter
- The location and application path of a transformation application
- Additional input files (optional)

In the third step, the user sets the connection between the data (files) of the software unit and the transformation applications. Also, the parameters needed by the tool for this transformation process were configured.

In the fourth and final step, the user defines the result of the transformation rule which is, in the case of a new software unit, based on the software unit model.

The screenshot displays two panels of the transformation output definition wizard. The top panel, titled 'Transformation Application', includes a 'Meta Information' section with fields for Name (Default), ID (ca8ac48a-4074-41ed-b74d-0e9c8a0bd548), Type (Service), Memo (Test), Website, Server (default), Version (default), Authors, and ChangeLog. The bottom panel, titled 'File Definitions', features a file tree on the left with '(3)' next to it, and a 'File Options' section on the right. This section includes 'Folder Options' (Add Folder, Remove Folder), 'File Options' (Name, ID, Memo, Type, C-Type), an 'isDependency' checkbox, and 'Add normal File' buttons (Add File, Remove File) with '(5)' next to them. A '(4)' is also present next to the Name field in the File Options section.

Figure 109 - Relevant UI areas of the transformation output definition wizard

Usage concepts

The UI for the definition of the final output of the transformation activity is summarised in Figure 109. A user can set the metadata of the new UOM (1). This includes the UOM type (2). Additionally, the user has to specify the files (4) (i.e., normal files or dependencies) (4) and defines them as human readable or machine readable (3). This is necessary information based on the U-Model.

Figure 110 shows the creation of a transformation activity as an activity diagram. Therefore, a user (2a) adds meta information about the complete transformation activity. In the next step (2b), additional files can be added that are necessary for a transformation rule or the transformation result. After this a user can create one or more transformation rules that are used by the transformation activity. For each each rule, (2c1) metadata and (2c2) additional files can be added. A user then selects the transformation tool (2c3).

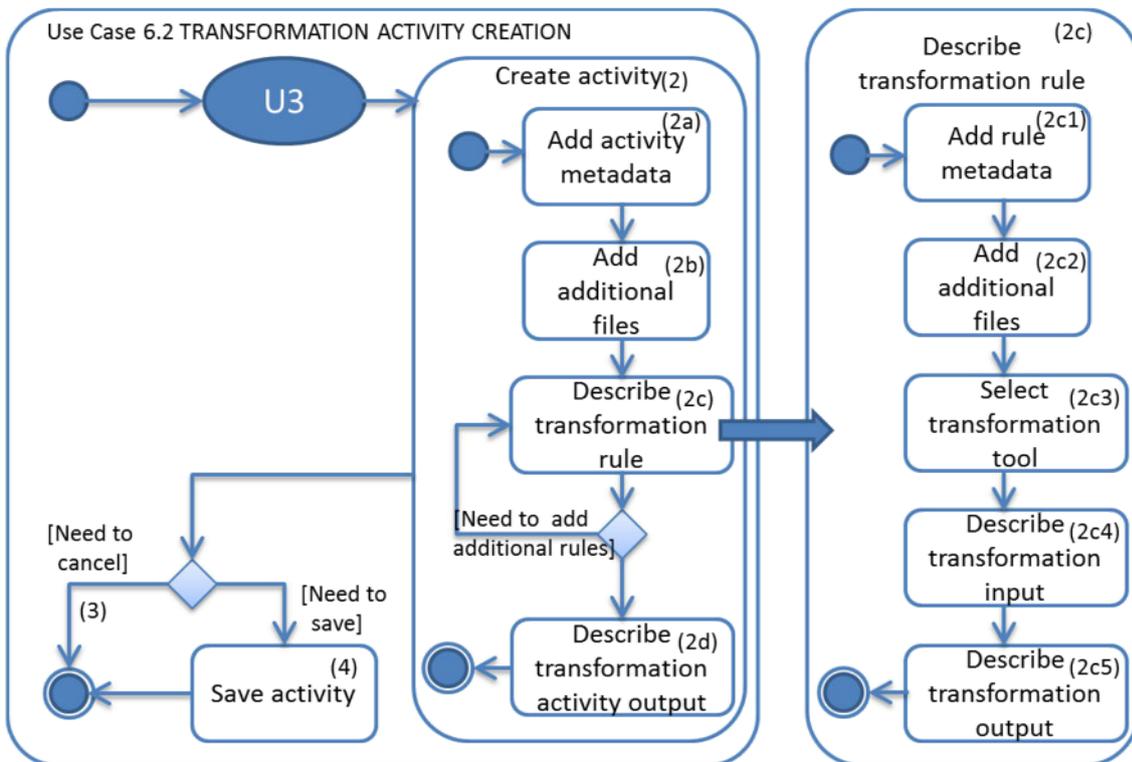


Figure 110 - Transformation steps

After this selection, the user can describe the transformation input by linking the parameters of the transformation tool with values of UOM, additional files, previously transformation rules or

selfdefined (2c4). For each transformation rule, (2c5) the transformation output (a new UOM) has to be defined. After finishing all transformation rules, the user (2d) also has to define the transformation activity output which is a new UOM. After completion, a transformation activity can be saved. The user has to perform a test. As in the scope of integration here, the user has to tell the system if this test was successful or not. After saving the transformation, it is then available for KU profile users.

In creating a transformation, the user may be supported by the Prometheus Predictive Knowledge System. On the basis of existing knowledge of existing transformation, the system is able to create suggestions for the additional transformation rules. This system is not part of the thesis, however, it is explained in a first publication (cf. Zinn, Fischer-Hellmann and Schoop, 2012b).

5.4.2.8. Use case 6.3 – deployment activity creation

This use case is realised by using two nearly the same interface as seen in Figure 106 and includes two different steps. The first (1) step includes the input of meta information of a deployment process (e.g., the name of the process). In the next step (2), all deployment steps for this process have to be defined. Therefore, each deployment step requires the following types of information (see also Section 5.3.3):

- Relating the files of the existing software unit to the single process step
- Adding additional files to the current process step
- Defining the deployment command (e.g., start, stop, etc.)
- Relating results of previously deployment process steps to the current process step (optional)
- Definition of manual steps (optional)
- Defining the communication information for the specific device

Figure 111 shows the relation between the use case and the two sub steps.

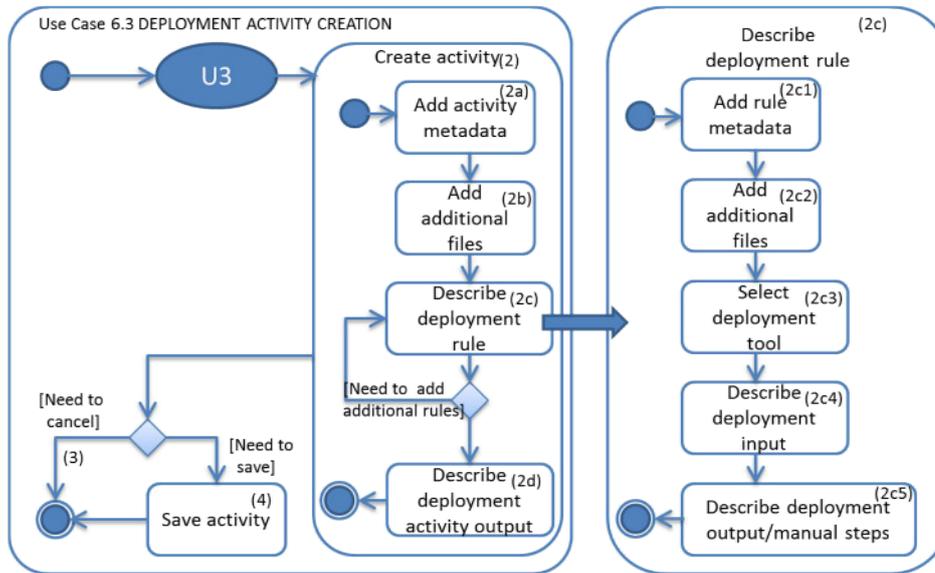


Figure 111 - Relevant steps for deployment activity creation

Figure 111 shows the creation of a deployment activity as an activity diagram. A user (2a) adds meta information about the complete deployment activity. In the next step (2b), additional files can be added that are necessary for a deployment rule or the deployment result. After this, a user can create one or more deployment rules that are used by the deployment activity. For each rule (2c1) metadata and (2c2) additional files can be added. A user then selects the deployment tool (2c3). After this, selection the user (2c4) can describe the deployment input by linking the parameters of the deployment tool with values of an UOM, additional files, previous deployment rules, or self defined rules. For each deployment rule, the transformation output (a new UOM) has to be defined which may need some manual orders by (2c5) the user (e.g., manual restart of a device).

After finishing all transformation rules, the user (2d) also has to define the deployment activity output which is a new UOM.

5.4.2.9. Use case 7 – activity search

This use case is used by the Activity Experienced user to search for an activity (see Figure 112). Therefore, the user can enter key words for searching (1) and start a search request of the system (2; cf. Use Case 2). After discovering a UOM, the different activity information relating to the selected UOM are useable.

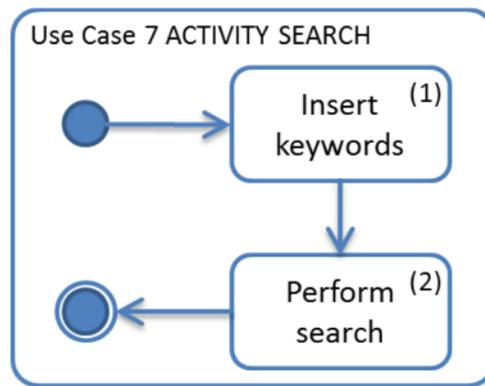


Figure 112 - Activity diagram of the use case 'ACTIVITY SEARCH'

The interface shown in Figure 98 is used for searching UOMs. After discovering a UOM, the user can analyse the metadata of the existing activities. Figure 113 shows an example of search key words. A user can enter key words into the search field (1) for search and (optional) enter keywords for filtering the result. The system uses the search key words and also identifies matching words in the SCAC. The system uses the information stored in the main U-model and the related SCAC models as a search area.

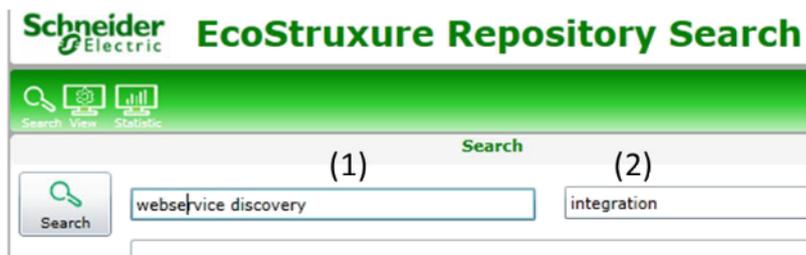


Figure 113 - User Interface for activity search

Both experienced user types can also use the wizard in the different creation and modification use cases to ‘search’ for activities. Therefore, a UOM has to be selected and then the related activities are displayed.

5.4.2.10. Use case 8 – activity discovery

An experienced user has to confirm that a result of a search contains an activity which is suitable for his requirements. Therefore, this use case includes Use Case 7, Activity Search, with the condition that the search request responds to at least one activity (see Figure 114).

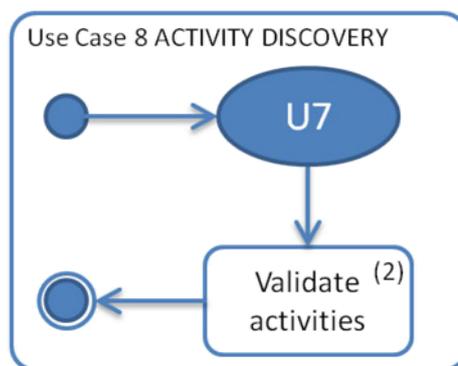


Figure 114 - Activity diagram of the use case ‘ACTIVITY DISCOVERY’

The search results (found activities (cf. Figure 120)) are presented in a compact form, but can be shown in more detail if required. This includes both the metadata of reuse activities such as author and the description of activity results (e.g., the new software unit of a transformation activity).

Figure 122 and Figure 125 show a detailed example of integration and transformation SCA type. Based on these user interfaces, a user is able to:

- Find out who created this activity,
- Find out who is responsible for this activity,
- Read additional descriptions,
- Analyse and download unit artefacts or additional documents, and
- Explore links to other content (e.g., web page).

Based on this information, the user may be able to decide whether this element is usable or not. Experienced user who wants to modify data uses the other described UI.

5.4.2.11. Use case 9 – activity adaptation

This use case is used by the activity experienced user. After the successful discovery of an activity (Use Case 8), a user is able to change the data of a reuse activity (2) or change other values (i.e., files, dependencies, etc.). The user (3) can decide whether to save these changes or (4) not to (see Figure 115).

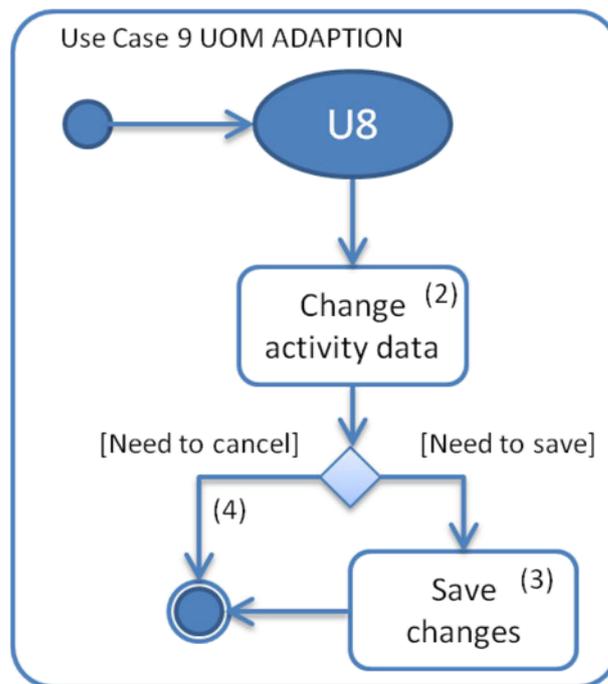


Figure 115 - Activity diagram of the use case 'ACTIVITY Adaptation'

For modification, the experienced user interface has to be used. By adding new files or dependencies, a wizard starts and supports the user to add information. Figure 116 shows a transformation activity. After selecting the repository, artefact and UOM, an existing SCA can be selected (1). Based on the type of SCA, the relevant UI will be displayed (2). The user can now change information (3). The UI is also used in Use Case 6.1, 6.2 and 6.3.

Usage concepts

Repositories: Prometheus SQL S | Artefacts: DPWS | UOM: DPWS4J | FileElements: commons-logging-1 | Inetgrations: | Transformations: Transformation1 (1)

Artefact Create | Artefact Edit | UOM Create | UOM Edit | FileElement Create
FileElements Create | FileElement Edit | Integration Create | Transformation Activities (2) | Transformation Application

Transformation

Meta Information

Name: Transformation1 | ID: c87d2ea5-2500-47e7-a58d-d0da3435c68a | From Tec: Java

Memo: This activity transforms the DPWS4J into .NET libraries (3) | To Tec: .NET

Transformation Rule Definition

Name	Order	FromTec	ToTec
Rule6	5	Java	.NET
Rule10	9	Java	.NET
Rule4	3	Java	.NET
Rule23	22	Java	.NET
Rule2	1	Java	.NET
Rule9	8	Java	.NET
Rule24	23	Java	.NET
Rule17	16	Java	.NET

Transformation Output

Type: 1 | Tec: 1 | Define

Files: 27 | Memo: None

Figure 116 - UI Wizard for UOM adaptation

5.4.3. Knowledge user profile use cases

These use cases handle action for the Reuser stakeholder. This inexperienced user aims to gain information about a UOM or activities. In addition, this user may want to execute a reuse activity.

5.4.3.1. Use case 11 – UOM information retrieval

After reading the information of the UI, downloading data is the second means of gaining information. Therefore, the actual software unit data (see Figure 117, left side ‘Implementation Units’), is distinguished from the data that describes the software unit (Figure 117, right side ‘Additional Data’). The user is able to download the individual files or is able to file together a package to download. How a user then uses this downloaded data is not part of the Prometheus environment. Additionally, the user can read shown information about a software unit.

Usage concepts

Name: Log4NET ID: dc44cf7c-5f93-4be9-90e5-1062
Author: unknown
Version: 1.2.11
Description
This software unit contains the log4net component

Implementation Units	Additional Data
Package:a600f570-52c4-4444-bae7-fd1ef Download Package Download File(s): log4net.dll	Package:5148b40d-1d9c-4df6-a6 Download Package Download File(s): log4net.xml

External Links
Main Server

Figure 117 - UI for downloading UOM information

This use case depends on Use Case 3 and includes two sub-steps. The first (2) sub-step is the 'reading' of information. If the user (4) decides to download this information, this can be done by requesting it (see Figure 117 and Figure 118). The selection of a download folder is not part of this use case view.

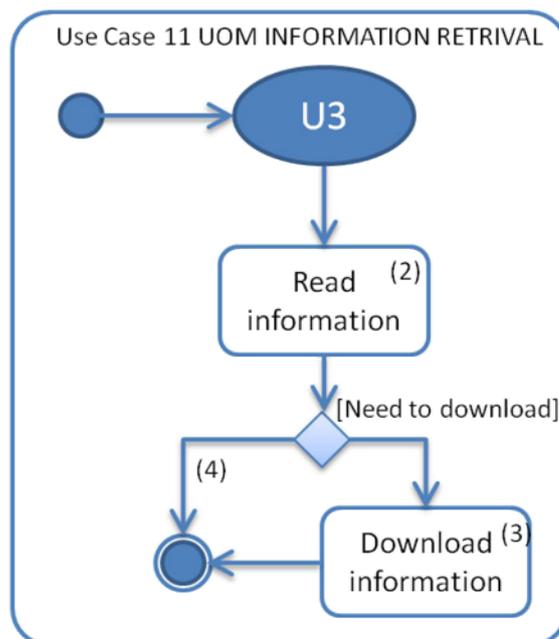


Figure 118 - Activity diagram of the use case 'UOM INFORMATION RETRIVAL'

5.4.3.2. Use case 12 – activity execution

The execution of reuse activities is relevant in the scope of this work. This use case requires the Use Case 8 to have been performed before. The user can execute a reuse activity selecting the focused activity and pressing the execution button (2; see Figure 119). This sub-step depends on the type of reuse activity. Therefore, this sub-step is realised by the Use Cases 12.1, 12.2 and 12.3.

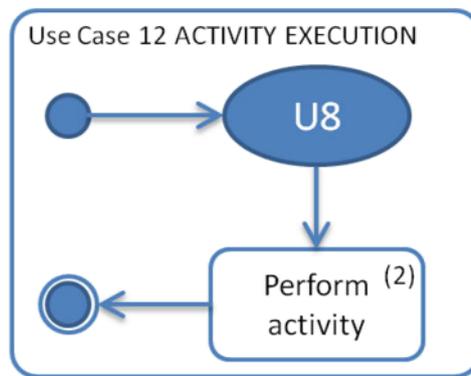


Figure 119 - Activity diagram of the use case 'ACTIVITY EXECUTION'

An activity can be executed by using the UOM overview user interface, including the activity overview. Figure 120 shows the area where to find the stored SCAC (1). A selected SCAC can be executed by pressing the execution button (2). It is also possible to execute an activity on the detailed activity view in the UI (see Use Cases 12.1, 12.2, and 12.3).

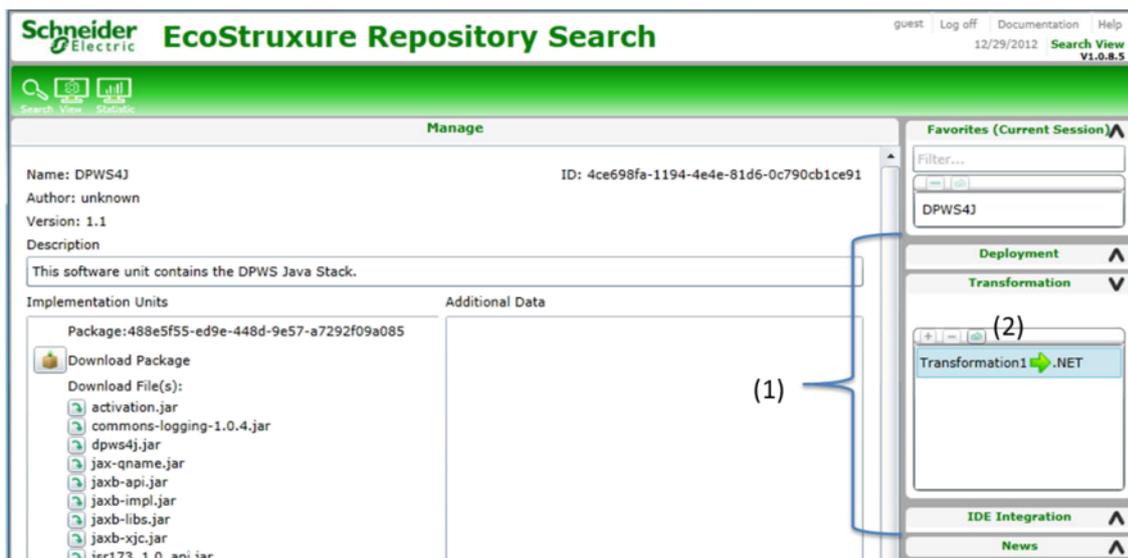


Figure 120 - UI for activity execution (a) in UOM overview (b) in activity detail

In the following section, the execution processes are explained as use cases, including a brief discussion of the user interfaces used.

5.4.3.3. Use case 12.1 –integration activity execution

This use case enables a user to execute an integration reuse activity. Therefore, the user (2) has to specify the service endpoint address. The system asks the user for this information. The user (3) is also able to change the given configuration of the integration rules, but (4) this step is optional. The integration will be executed automatically if the user (5) requests it by pressing the execution button. If the Prometheus environment (6) discovers an inconsistency (e.g., the given IDs and the specified IDs in the activity are not the same) it displays an alarm to the user. Figure 121 shows the Activity diagram for this use case. The user interface used is demonstrated in Figure 122. The result of this use case is an integrated software unit in a specified IDE.

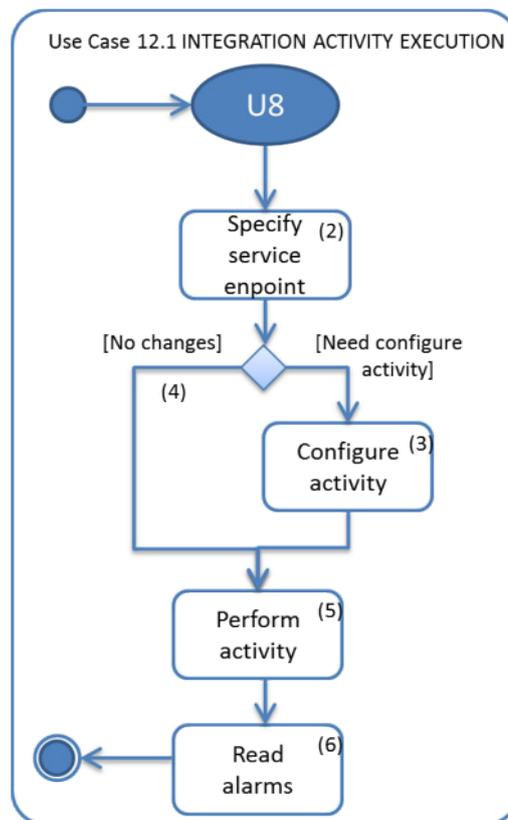


Figure 121 - Activity diagram of the use case 'INTEGRATION ACTIVITY EXECUTION'

Usage concepts

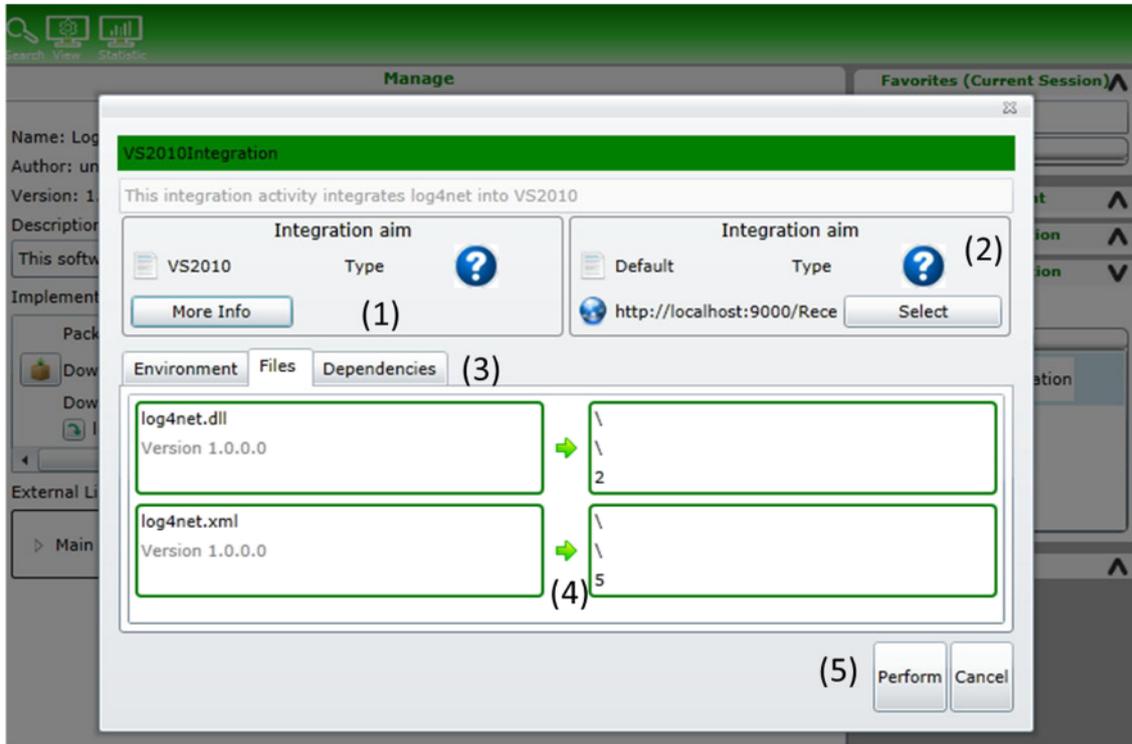


Figure 122 - UI for integration activity execution

Figure 122 shows the information screen about an integration SCAC where it can be executed (5). A user can find general information about the SCAC (1). The relevant information as for example the required environment, the files, and the related dependencies can be reviewed (3).

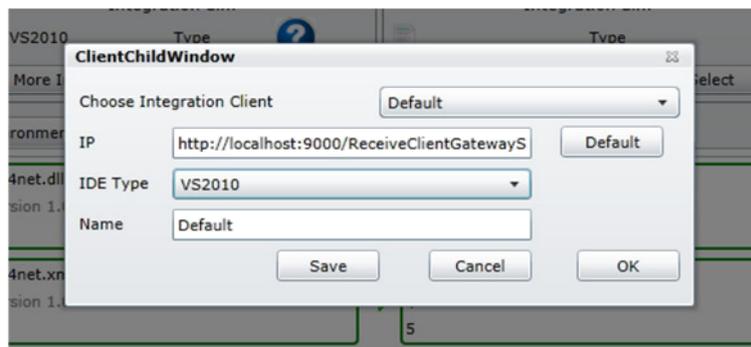


Figure 123 - Configuration UI for IDE service endpoints

Figure 122 shows especially the file section (4). The user can see the project and folder structure for each file in the destination IDE. Additionally, the IDE for integration can be chosen (2). Figure 123 shows the configuration UI for the IDE service endpoint. A user has to specify a service endpoint and the IDE type. If the focused IDE is not the specified IDE in this UI, the system will warn the user.

5.4.3.4. Use case 12.2 – transformation activity execution

This use case enables an inexperienced user to perform transformation activities. Therefore, the user has to select a transformation activity (Use Case 8). The user (2) is also able to change the given configuration, for example, the transformation rules, but (3) this step is based on the knowledge level of the user. Finally, the user (4) can perform the selected activity (the perform button).

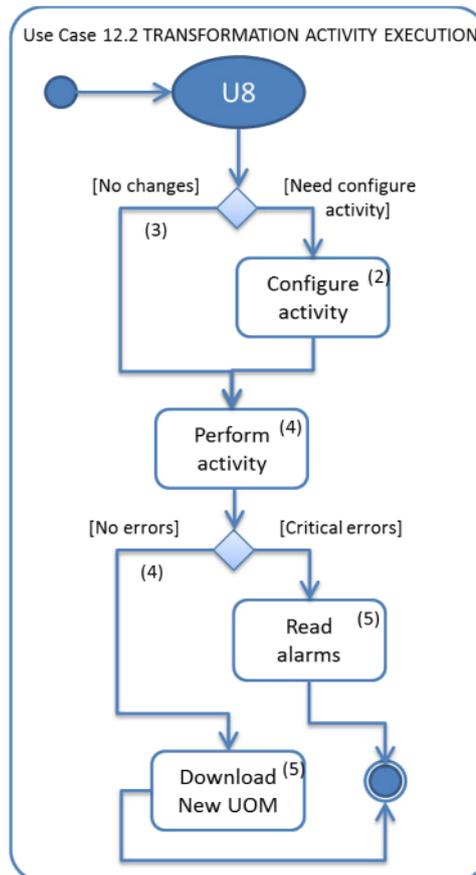


Figure 124 - Activity diagram of the use case 'TRANSFORMATION ACTIVITY EXECUTION'

Usage concepts

The result of the transformation is a new software unit, one based on the software unit model. If the Prometheus environment (5) detects an inconsistency (e.g., the given IDs and the specified IDs in the activity are not the same), it displays an alarm to the user. The user (6) can download the new unit after the transformation activity is completed successfully. Figure 124 shows the Activity diagram for this use case. The user interface is demonstrated in Figure 125.

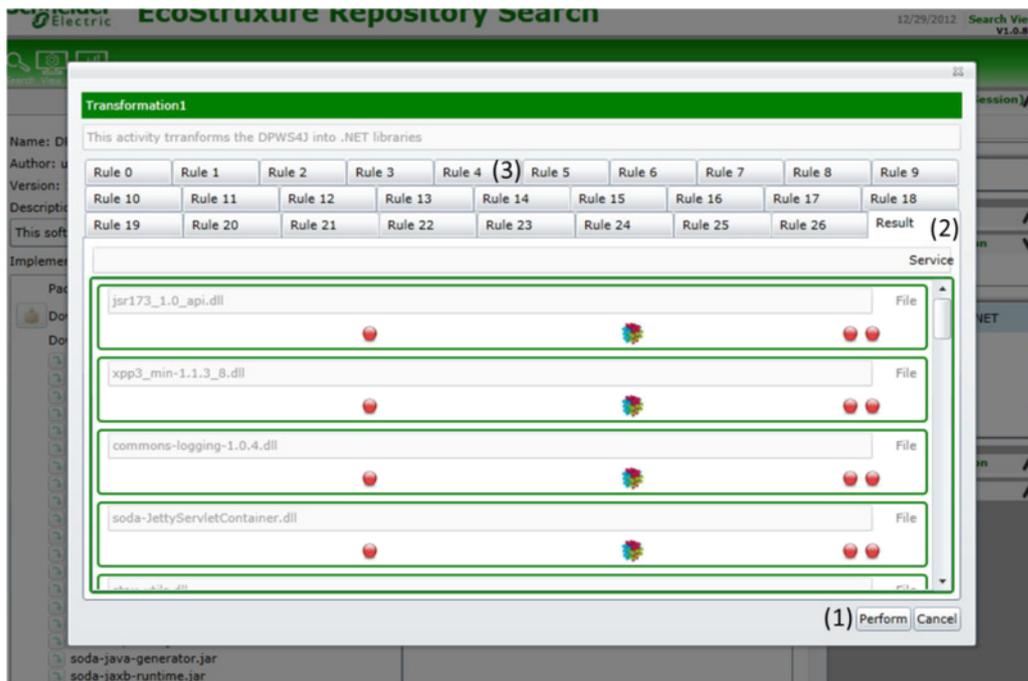


Figure 125 - UI for transformation activity execution

In the user interface shown in Figure 125, each rule of a transformation activity can be seen separately (3). This includes the information for each input file or parameter for each transformation rule. Also, the result and additional information of the transformation can be seen (2). The activity can be executed by pressing the 'Perform' button (1).

5.4.3.5. Use case 12.3 - device deployment activity execution

This use case enables a user to execute a device deployment reuse activity. After identifying the correct activity, the user (2) has to specify the service endpoint address of a device. The system asks the user for this information. The user (3) is also able to change the given configuration of

Summary

the deployment rules, but (4) this step is optional. The activity (5) will be executed automatically if the user requests it by pressing the execution button. If the Prometheus environment (6) identifies an inconsistency (e.g., the given IDE and the specified IDE in the activity are not the same), it displays an alarm to the user. Figure 126 shows the activity diagram for this use case. The result of this use case is a deployed software unit in a device.

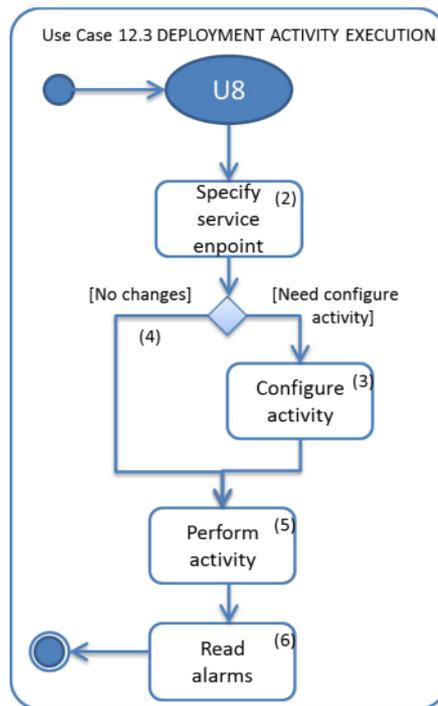


Figure 126 - Activity diagram of the use case 'DEVICE DEPLOYMENT ACTIVITY EXECUTION'

Note: The deployment plugin explained in this chapter is not used in the experiment described in Chapter 6. The possible realisation with the presented Prometheus environment is published by Zinn, Fischer-Hellmann and Schoop (2012a).

5.5. Summary

Chapter 5 discusses a realised Prometheus environment. The environment uses an application called Prometheus server. This server uses plugins to communicate to different elements of the environment. Such elements are: a user client system, different repository system, and different applications used to perform the focused software construction activities.

Summary

The communication technologies used between these elements are services, methods, and process calls. This chapter explains the interfaces used. The plugin system is the major part of the discussion about the extensibility of this system as well as the service communication.

The data types used in the service and other interfaces are related to the realised data models for software units, transformation software construction activities, integration software construction activities and device deployment software construction activities. These models are relevant because the common view creates the focused approach. For the realisation of the focused approach, typical existing technologies are used. The created environment is used in Chapter 6 to perform a case study with different software engineers as participants.

6. Evaluation and research result analysis

This chapter describes the case study used for the thesis primary research. The aim is to identify the effect of the realisation of the focused approach which supports software engineers by performing SCAC. This description includes the research methods used to evaluate the focused approach (cf. Chapter 4), using the realised Prometheus environment (cf. Chapter 5). The first section describes the research theory and the related research methods. This includes the scientific viewpoint and is focused on a practical case study. After the explanation of the case study setup, including the case studies procedure and measurement model in the second section, the analysis methods are explained in the third section. This chapter concludes with an overview of the results of the case study and the consideration of the scientific viewpoint of the first section.

6.1. Focused case study and scientific viewpoint

This chapter focusses on a case study supporting the primary research. To support the understanding of the structure and methods used for the case study, this study is now briefly described.

6.1.1. General overview about the case study

In general, the case study observes software engineers performing software construction activities. Six experienced software engineers are measured by performing 12 SCACs (2 SCACs for each engineer). The measured variables (i.e., time, task and knowledge resources) are compared to the measured values of a group of 48 inexperienced software engineers (4 participants perform the same SCAC). This comparison shows the difference between experienced and inexperienced software engineers in a normal working environment.

In the next step, the measured values of an additional group of 48 inexperienced software engineers (4 participants perform the same SCAC) are compared to the other values measured previously. In this case, these inexperienced software engineers used the realised approach (cf.

Chapter 5). Comparing these values to the others shows the differences between engineers using the realisation of the focused approach and engineers in a normal working environment.

6.1.2. Scientific case study theory

The case study scientific basis is shown by using the discussion of Baxter and Jack (2008) regarding qualitative case study methodology. In these discussions two different case study approaches that guide case study methodology are analysed. The first one is proposed by Stake (1995) and the second is proposed by Yin (2003 and 2006). Baxter and Jack (2008) use the following conceptual structure in their analysis: Determining the Case/Unit of Analysis, Binding the Case, Determining the Type of Case Study, Single or Multiple Case Study Designs, Proposition, Issues, Conceptual Framework, Datasource and Database.

Note: In the following, the points determining the Case/Unit of Analysis, Binding the Case and Determining the Type of Case Study will be discussed. The concrete realisation of the case study including Single or Multiple Case Study Designs, Proposition, Issues, Conceptual Framework, Datasource and Database is explained in Section 6.3.

Determining the Case/Unit of Analysis: The case study aims to analyse the difference between supported and unsupported software developers who want to perform software unit reuse. As a result the cases are scenarios including software developers with different knowledge levels performing reuse of the same software unit. The unit of analysis is the behaviour of software engineers in specific reuse scenarios.

Binding the Case: Following the analysis of Baxter and Jack (2008), it is relevant to determine the boundaries for the case study. In this case, the place (Creswell, 2003), time and activity (Stake 1995), and the context is relevant. The case study will be performed in normal environments of software developers (see Section 6.3.1.1). The focus of the case study is set on the reuse activities which should be done by the developers (see Section 6.3.1.4). The context of the case study is software reuse of different sized software units of different contents and types (see Section 6.3.1.4).

Determining the Type of Case Study: Based on Yin (2003) and Stake (1995), Baxter and Jack (2008) explain typical types of case studies. Follow this explanation, the case study of this thesis is a ‘multiple-case study’ (Yin 2003) which is also defined by Stake (1995) as ‘collective’. Such a case study type focuses on multiple case studies with different settings. Hereby, the differences within and between the cases are relevant. In the focused case study different scenarios are handled. The knowledge level of the software engineers, the reuse task to do and the type of software unit will be changed. By comparing the results of the different cases to the case study hypothesis, this hypothesis can be carefully discussed and proven.

6.1.3. Case study hypothesis

The following theoretical statements are used as a hypothesis for the case study research:

1. SCAC knowledge/information can be stored in an environment and can be reused by inexperienced users.;
2. Such a reuse produces a comparable (working) result as an experienced user in a normal application area, but with a reduction in learning the required knowledge for the specific SCAC or a comparable knowledge transfer task; and,
3. The inexperienced status of the user which relates with the specific SCAC does not change.

The hypothesis focuses on a positive effect for software unit reuse by software developers if developers are supported. After the case study result, a discussion of these 3 hypothetical statements has to be proven.

In the following sections the methods and the structure of the case study is explained in detail.

6.2. Research theory and methods

The primary research focuses on the creation of a concept to enable inexperienced user to perform knowledge based software construction activities. As primary source for evaluation an experimental case study is used. In the beginning of this section the research type and theory is

explained. This includes the definition of context related terms and the definition in relation to the research question. Tittenfick

6.2.1. Relating focused problems and research question

Firstly, a recap of the research question shown in Chapter 1: *‘How does one provide successful reuse of different software units in the area of software construction considering the possibilities of reusing and performing related software construction activities even if software engineers do not have the required knowledge?’* This will be related to the discussed problems of missing knowledge of software construction activities. Therefore, the relevant terms in this question are explained in relation to the current context.

The research question aims to deal with the execution of reuse activities of different software units (*‘...successful reuse of different software units...’*) by inexperienced software engineers (*‘even if software engineers do not have the required knowledge’*). This means that a user has an inadequate knowledge level in one or more of the focused software construction activity areas. This research focuses on *‘software units’* (i.e., objects, components and services) which should be successfully reused by undertaking such activities. In order to successfully reuse (*‘successful reuse’*) a software unit, the problem of an inadequate knowledge level of SCAs has to be dealt with. As a result, all problem areas (i.e., knowledge required for specific technologies, inadequate knowledge level of software engineers, and knowledge required by distribution environment) discussed in Section 2.2.3.3 have to be limited or solved. This includes the related sub problems discussed in Section 3.2. The focused *‘software construction activities’* in this research are: integration, transformation, and device deployment of software units (cf. Section 3.1).

6.2.2. Selected research type

One objective of the research is to demonstrate the practical applicability of the focused approach. To demonstrate the success or failure of this objective, a case study is used. Following Creswell's (2009) discussion about qualitative, quantitative and mixed procedure models, the case study has quantitative and qualitative elements, which will now be explained. The measuring of data in the case study is quantitative. Thereby, one or more of certain (predefined and classified) characteristics are measured and the results are combined and discussed with the research aim. The number of participants and the analysis of the case study results are quantitative. Based on limitations regarding participants (people with suitable software engineering backgrounds), the number of measured participants is applicable to perform the case study and to come to a conclusion. Therefore, the number of participants is also seen as qualitative. However, this number is not to be considered applicable in general. These means that the result of the case study cannot be used as a statement for each existing software engineer or SCAC.

6.2.3. Used research theory, methods and application area

Clarke (2005) states that a research model consists of a theory, related methods, and an application area. In this thesis, the definitions made by Clarke (2005, Slide 21) are used for these terms:

- *“A theory is a set of interrelated constructs (concepts), definitions and propositions (statements) that presents a systematic view of phenomena by specifying relations among variables.”*
- *“Methods (a.k.a. techniques) are used to reveal the existence of, identify the ‘value’ significance or extent of, or represent semantic relationships between one or more concepts identified in a model from which statements can be made.”*
- *“Application domains are defined as those substantive areas, examples, cases, that theory and methods are applied.”*

In the following, the above defined three elements will be explained focusing on the realisation this thesis.

6.2.4. Theoretical viewpoint

The following theoretical statements are used as hypotheses for the case study research:

- SCAC knowledge/information can be stored in an environment and
- can be reused by inexperienced users.
- Such a reuse produces a comparable (working) result as an experienced user in a normal application area,
- but with reduction of learning the required knowledge for the specific SCAC or a comparable knowledge transfer tasks.
- The inexperienced status of the user which relates with the specific SCAC does not change.

6.2.5. Methods

To create a systematic view on the primary research, several methods have to be defined. These methods are classified by the author as follows:

- Preparation methods: Methods supporting the preparation and setup of the study (e.g., identification of an experienced user in the application area).
- Measurement methods: Methods supporting the measurement and storing of values and results.
- Analysis methods: Methods defining the rules for verification of results in context to fulfil requirements and theory.

Following, the case study preparation and measurement methods will be defined and explained. These methods are necessary to understand the complete measurement of values in the primary research. The definition and explanation of the analysis methods follows in Section 6.4.1

Note: Only the transformation and integration SCAC are discussed for the methods used in the case study. Due to limitations of the security the deployment SCAC is not investigated in the practical part of the case study (cf. Section 6.4.2.3). But it is discussed in the result analysis using the measured results of transformation and integration SCACs for comparison. (cf. 6.4.2.3)

6.2.5.1. Preparation methods

For the theory and preparation of the case study two relevant preparation questions have to be answered:

- How to identify a SCACs and related software units? A method is required to show how SCACs and related software units were chosen.
- How to identify inexperienced and experienced users? A method is required to show the processes and variables to separate users: the inexperienced and experienced user.

Following, the questions will be answered.

Method 1 (M1): The first method to explain is the identification method to identify reusable software units. Software unit are necessary for the experiment. They have to be reused by the use of different SCAC. Additionally, the SCAC depends on these units. The application area is set in the global environment of the company: Schneider Electric. Inside this environment, special software units exist and have to be reused in several projects and products. A software unit typically belongs to one business unit area (e.g., power and industry) of Schneider Electric. In the case study, software construction activities of such software units will be performed. Two of the three focused SCACs will be used for the case study. Therefore, software units were identified by evaluating if the reuse of the provided software units included:

- at least one integration task for Eclipse or Visual Studio, and
- at least one transformation task for console-based transformation tools.

Both activities have to be repeated in the case study.

Additionally, following requirements have to be fulfilled:

Evaluation and research result analysis

- learning to perform integration or transformation should be made within several hours (max 4 hours). This is based on time limits of the participants and the observer.
- documentation or examples for the SCAC have to exist inside the Schneider Electric environment or in other accessible locations. This is necessary to give all participants a chance to perform the activities.
- the software units have to be used in different vertical or horizontal projects. The requirement refers to the development project scenarios discussed in Section 4.3.

The procedure to identify software units was to contact different business unit's product manager for software units to identify relevant software units and experienced users. The experienced users were asked to explain different integration and transformation reuse activities for their software unit. A software unit that fulfilled the selection requirements explained above was inserted in a list of useable software units for the research.

Additionally, it was relevant to identify an experienced user for each software unit able to perform the transformation or integration task manually (see next method M2).

Note: Four of the six software units were identified by proving internal repository information about the software units. The overview information about the units contains the software unit owner. These were asked to be the experienced users (cf. method M2) or if these people knew of a potential experienced user. These users were asked for software units which fulfilled the requirements. The other two software units were identified by asking the different product owners (managers of software products of Schneider Electric) for software unit experts who could identify potential software units.

Method 2 (M2): The next methods to explain are the identification methods to identify inexperienced and experienced users, these methods are described below. Experience user a necessary to

- (1) identify SCAC for a software unit (see method M1),
- (2) be measured as comparison to the values of inexperienced engineers

(3) to insert SCAC information into the Prometheus environment

For each software unit in the Schneider Electric environment, at least one responsible and experienced software engineer exists. These engineers were asked whether they have already reused the specific software unit several times, or could identify a potential experienced user. The required experience has to include the use of a transformation and integration SCAC. If an engineer answered the questions (for software unit, transformation and integration experience) with yes this user was selected as an experienced user for one software unit the related SCAC. It doesn't matter whether the experienced user is the inventor of the software unit. For the case study it is relevant to have a trained software engineer as a comparison to untrained engineers. This can be used to show the difference between experienced and inexperienced software engineers.

Method 3 (M3): Inexperienced users were identified by searching for software engineers in actual software development projects inside all business units of Schneider Electric. The primary research focuses on the support of software engineers with less experience of specific SCAC reuse. Therefore, it was ascertained that each inexperienced engineer was not an experienced user for one specific software unit, the related SCAC, the related technology and related transformation tools or IDEs. For the identified software units, an SCACs (cf. Section 6.3.1.4). It was relevant to identify engineers without experience or with less experience in Java or .Net based software development. Depending on their answer it was relevant to identify if these people also have less experience in Eclipse or Java software development kit for web services (for inexperience in Java). For participants with less experience in .NET it was relevant to identify users with less experience in IKVM, Visual Studio or .NET software development kit for web services.

6.2.5.2. Measurement methods

The aim is to measure information that can be used to analyse if the realisation of focused approach have an impact on users to perform SCACs. To discuss this, it is necessary to measure

experienced users and inexperienced users performing SCAC reuse with and without (i.e., normal way of working) the realisation of the focused approach. The results are values that can be compared to prove the theoretical statement in Section 6.2.4. Therefore, the measurement methods will be described. For each identified problem area, the related knowledge problem view discussion (cf. Section 3.2) is used to define the measurement methods (i.e., focusing knowledge storing, knowledge learning, searching and receiving knowledge, knowledge exchange and knowledge execution). The following question must be answered: How can the impact of the focused approach for the focused problem areas be measured (including the identified knowledge problems)? In the following the used method will be explained.

Method 4 (M4): The problem of knowledge storing includes the problem of identification, access, and use of a repository. The previously identified experienced users (see method M2) are observed in configuring an SCAC (i.e., insert of SCAC related information into the Prometheus environment). This includes the measurement of identification, access, and use of the Prometheus environment. This method measures only the time a user needs for this activity. The information a user inserts in the system and the tasks the user completes are not relevant for later analysis of the research. The method (M4) is performed by an observer and a system (cf. the case study setup description in Section 6.3.1.2).

Method 5 (M5): This method is used to measure the effort of knowledge learning. This means the time spent on knowledge resources and the number of used knowledge resources. It focusses the problem of multiple technology variations, multiple existing tools used in SCACs, and of knowledge interpretation. To be more precise, the method focuses on knowledge resources used for learning.

This method is based on a comparison between users using the focused approach and not using the approach. Both types of user are inexperienced. Therefore, for both user types, the time spent for learning (i.e., time spent to use a knowledge resource), the success of the task (i.e., the personal opinion of the participant of their success), and the number of used knowledge

resources will be measured. These measurements can be used to discuss the effect of the realised approach on the problem of knowledge interpretation. This method is used while the participants perform different SCAC with different software units and tool technologies to measure values.

Method 6 (M6): This method measures the same values than the method M5 but has a different focus. To measure the problem of searching and receiving knowledge this method focuses on the related sub problems: the problem of localisation (for identification and access to a repository) and the problem of the use of a search engine (different technologies and component worlds, variations of SCAC related actions, existing tools used in SCACs), and search result analysis.

This method measures how long a user spent on knowledge resources on repositories. To search for software units and SCAC related information.

Method 7 (M7): To measure the values focusing the problem of knowledge exchange no special behaviour is necessary. It has to be measured if the inexperienced software engineer created a valid value. Therefore, the experienced software engineer is used to determine if the result created by the inexperienced software engineer is useful. Also it is measured how many knowledge resources a participant has used.

Method 8 (M8): The problem of knowledge execution is focused on by using this measuring method. Here, the complete task of the execution of an SCAC is relevant. Therefore, this method measures the time needed, knowledge resources and tasks used to perform the SCAC. Additionally the validation of method M7 is used to.

6.3. Case study setup, procedure and measurement model

To explain the analysis methods it is first necessary to explain the case study setup. This includes the application domain, the used infrastructure and the case study procedure model.

6.3.1. Application domain

The application domain is the environment of the company Schneider Electric. Schneider Electric is a global company with ~160,000 employees and 7 business units (i.e., power, datacenter, building, industry, life space, infrastructure and water). Actually, ~15,000 software engineers are working in software development projects. These include device, desktop and server level development projects. Inside this environment the case study is done. The participants of the case study are software engineers from different business units. Therefore, the typical office working environment of software engineers provided by Schneider Electric is used. It is relevant for the study not to change the normal working environment to have comparable initial situation.

Often, software development projects consist of different software engineers located on different sites all over the world (cf. Qu, Ji and Nsakanda, 2012). The method M1 describes one way to identify software units and experienced users inside this global scenario.

The technical setup of the case study is divided into three distinct areas: description of the environment, description of the technical structure of the case study and the necessary elements (i.e., software unit and SCAC) for the case study.

6.3.1.1. Description of the environment

The experiment was conducted at a German location of the company Schneider Electric Automation GmbH (Address: Steinheimer Strasse 116, 63500 Seligenstadt, Germany) which is part of the global Schneider Electric network. The company participated by providing personnel from the site and from other locations, and the company intranet and the software units/SCAC for the case study. The study itself was conducted in normal offices, which provide a connection to the intranet sources (e.g., internal repositories) and internet sources (e.g., external repositories).

6.3.1.2. *Description of the technical structure of the experiment*

The technical design of the experiment is mainly a hardware and software infrastructure. Figure 127 shows this structure inside the Schneider Electric intranet. Six relevant elements are involved.

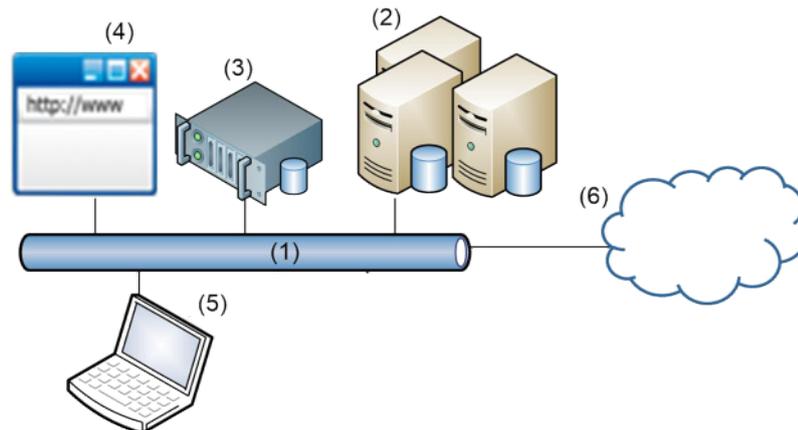


Figure 127 - Experimental environment and setup

The first element is the intranet (1), which is used to connect the various elements of the technical structure together. Additionally, it enables the communication to the internet (6). The second element (2) is the connected databases including the software units and complete information about the reuse activities. For the experiment, three databases are relevant:

- SOA4D - this is an open source repository including software units and further information about device profiles for web services. This repository is based on Forge technology and offers a web interface. This repository is not part of the Schneider Electric intranet; it is placed on the internet.
- Prometheus Database - this is a specially developed repository. It belongs to the SCAC approach of this thesis and uses a Microsoft SQL database and Microsoft SQL database interface.
- Brick Catalogue and DDXML- These are Schneider Electric internal repository that includes different reusable software artefacts.

The third element (3) in the case study setup is the server inside the Prometheus environment. The server maintains information about software units and software construction activities in the connected databases and makes this information available to the user. Finally the server performs requested activities and presents the available results to users (see Section 5.2.1.2). The fourth element (4) is a web page including a user interface through which the user can communicate with the server. This web page is executed on a web server. This web server contains a web application and provides the user an ability to query information from the server or to perform SCACs on the server. The basic technology of the Web application is Microsoft Silverlight version 5.0. The web page is located and available via the company's intranet. The participants use a typical Schneider Electric laptop (5) including different IDEs and runtime environments (i.e., Java and .NET) for software engineers. In addition to the computer network environment, there is the possibility to use a telephone, the internet, voice, conversation or literature. This is also part of the company's infrastructure.

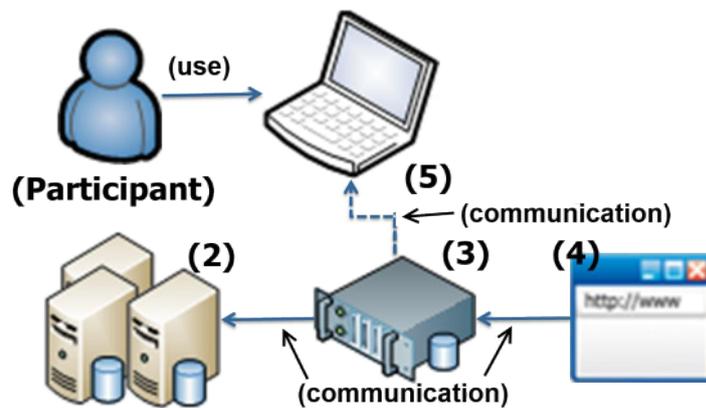


Figure 128 - Basic experiment scenario

This experimental setup allows for the working scenario shown in Figure 128. The participants use the element (5) (office laptop) to view the normal working environment. Within this environment, all necessary software applications are found in order to search for information on the internet, information, or perform activities on the intranet as well as various communication technologies (e.g., FTP, Skype, Telnet, etc.). Furthermore, participants can click the element (4)

unit. For the selection of experienced users, method M2 is used. Altogether 6 experienced users are necessary which are able to do one integration and one transformation activity (per user). The second experimental group (Group 2) consists of 48 inexperienced software engineers selected by using the method M2. The third group (Group 3) also includes 48 inexperienced participants. For this group, the same rules for selection are used as for Group 2 (see preparation method M1 Section 6.2.5.1).

Note: In the beginning of the case study video recording was planned to be used as support of the measurement. But this was not permitted by the work council of Schneider Electric.

6.3.1.4. Software units and software construction activities

In the case study, six different software units are used. One integration and one transformation software construction activity for each software unit is used in the case study. In this section the software unit and the related software construction activities are briefly described:

Software Unit 1 – Device Profile for Web Service (DPWS; Java Stack): DPWS is a protocol to extend the basic web services definition with the information required by electronic devices (like footprint, performances, security and event driven messaging). The DPWS Java stack can be found in the SOA4D repository (cf. SOA4D, 2012) and contains a set of 23 Java components which are necessary for the different activities. The transformation software construction activity uses IKVM as a transformation tool. Challenges are apparent when an inexperienced user has to find the repository, access it, download the correct DPWS Java Stack, find the IKVM repository, access it and download the IKVM. Next to the fact that an inexperienced user has to find out the correct parameter for each Java component, this user has to do this in the correct sequence. The sequence depends on the dependency hierarchy of the Java components which has to be discovered beforehand. The second software construction activity is the integration of the transformation result into a Visual Studio IDE. The challenge for an inexperienced user is the dependencies created by the IKVM transformation. Special

files have to be copied in the project folder and, next to the transformed components; special files have to be referenced by the Visual Studio project.

Software Unit 2 – Device Profile For Web Service (C++ Stack): The C stack of the DPWS software unit contains several classes written in programming language C. It fulfils the same functionality as the DPWS Java Stack. The challenges for integrating this software unit as a source-code are the dependencies between the different classes and the setting in the project environment. Regarding the transformation, the problem is to setup a special compiler with the correct settings and prepare the correct file and folder structure. Based on the number of classes and dependencies, these SCAs are classified as ‘advanced’ compared to the Ecostructure Web Service (EWS) and Log4Net/J software unit. The integration should be made with the same source-code into the Visual Studio IDE. This software unit can also be found in the SOA4D repository.

Software Unit 3 – EcoStructure Web Service: The EcoStructure Web Service is a common data exchange web service enabling enterprise systems of different domains (e.g., industry and building) to exchange information. The service is used between all business units of Schneider Electric and external partners or customers. The challenge of this software unit is to create out of the existing WSDL file and the specification running web service server and clients. Based on the strict specification, knowledge is required to create a valid and consistent web service server and client. If these implementations are not consistent, they are not able to communicate to other EWS implementations. The SCAs here are the transformation of the WSDL file to a .NET client using the SVCUtil tool provided by Microsoft. The result of the transformation has to be integrated regarding the correct dependencies of the .NET framework. As a test, the client should be able to call one web operation of an existing EWS server. Based on the dependencies of the WSDL file and the specification, the complexity of the SCAs are classified as ‘middle’. The EWS software unit can be found in the Schneider Electric intranet and the Prometheus SQL repository.

Software Unit 4 – EcoStructure We bService: This software unit is the identical EWS web service. The SCACs here are to create an EWS client in a Java software development kit technology (transformation SCAC). The results have to be integrated into an Eclipse Java project. The SCAC has the same complexity as the realisation in .NET.

Software Unit 5 – Log4Net: Log4Net is an open-source software unit providing logging functionality. The source-code project contains only a few classes and one configuration file is necessary. The SCAC for this unit is the integration of the classes into visual studio and the compilation of the classes using the .NET compiler in a console call (transformation SCAC). Based on the fewer dependencies of this software unit to other classes, libraries and settings, these SCACs are classified as ‘simple’. This unit is available on the Internet and the Schneider Electric Brick Catalogue repository.

Software Unit 6 – Log4J: The Log4J is a Java based software unit and includes the same functionality as the .NET variant. The SCAC here is the transformation of the single binary file using IKVM. The result (including all IKVM dependencies) should be integrated in the Visual Studio environment.

Table 41 lists all units used in the experiment.

Name / ID	Description	Tec/ Unit Type / Repository	Integration effort / Transformation effort
DPWS / SU1	Enable devices for WS* profiles	Java / Component / SOA4D	Advanced → Visual Studio Advanced → IKVM
DPWS / SU2	Enable devices for WS* profiles	C++ / Code/ SOA4D	Advanced → Visual Studio Advanced → C-compiler
EWS / SU3	Web sService for data exchange of business between units	Soap-C# / Web Service / Prometheus	Middle → Visual Studio Middle → SVCUtil
EWS / SU4	Web sService for data exchange of business between units	Java-Android / Web Service / Prometheus	Middle → Eclipse Middle → Java2SOAP
Log4J / SU5	Logging functionality for Java	.NET / Code / Brick Catalogue	Simple → Visual Studio Simple → IKVM
Log4Net / SU6	Logging functionality for .NET	Java / Component / Brick Catalogue	Simple → Visual Studio Simple → .NET Compiler

Table 41 - Case study software units

6.3.2. General case study sequence

The case study is used to prove the ability of the focused approach to support the inexperienced user. Following, the case study procedure model used for the different directions is explained briefly.

The basis for the case study demonstrated in this chapter is the explanation of the idea and the concept of the new approach focused by this thesis in Chapter 4. In Chapter 5 one concrete instance of the concept is explained and used for the case study in this chapter.

In principle, there are three different experimental groups performing seven different scenarios for each SCAC. Table 42 describes each scenario.

Scenario	Description	Group
(1) Observation of the experienced users	The experienced users from group (1) perform transformation and integration activities with the software unit and without using the focused approach.	(1)
(2) Collection of software units and activities	Each of the experienced users inserts the relevant information in the Prometheus Environment and performs the SCAC using the focused approach.	(1)
(3) Validation of Prometheus	The experienced users validate the results of the activities performed by using the focused approach.	(1)
(4) Reuse activities with Prometheus	In this scenario the participants of group (2) are asked to perform a transformation or integration activity. Therefore, they use the focused approach.	(2)
(5) Reuse activities without Prometheus	In this scenario the participants of group (3) are asked to perform a transformation or integration activity. This group is not supported by the focused approach.	(3)
(6)/(7) Validation of the results	The experienced users validate the results of the activities performed by group (2) and (3).	(1)

Table 42 - Case study scenario summary

In principle, the case study uses the procedure explained in Table 42. After identifying and selecting an inexperienced user, experienced user, software units and related SCAC (method MP1, MP2, and MP3), the experienced user is observed by performing a specific transformation and/or integration activity without support of the Prometheus environment (Scenario 1). After this, the experienced users are asked to enter the information into the Prometheus environment and perform the same SCAC using the Prometheus environment (Scenario 2). The experienced user validates the results of the SCAC performed by the Prometheus environment (Scenario 3). If the result is valid, Group 2 is asked to perform the

same SCAC as the experienced user (Scenario 5). This is done without the support of the Prometheus environment. Additionally, Group 3 is asked to perform the same SCAC using the Prometheus environment (Scenario 4). The experienced user has to validate the results of Group 2 and Group 4 (Scenario 6 and 7).

After performing all scenarios for all selected SCACs, the measurement is done and the analysis of the measured values and hypothetical values can be analysed to formulate a conclusion for the primary research.

6.3.3. Measurement and experiment results overview

In this section the measurement procedure of the case study will be explained. This includes the definition of the measurable variables and the process of measuring.

6.3.3.1. Definition of variables for comparing methods

The results of the case study measurements are stored into variables. In addition, each variable is assigned to name within the experiment and is used in one or more of the measurement methods. In this section all variables are named and briefly presented. Table 43 explains the different measurable variables in the different scenarios. The scenarios and the variables are numbered. In Section 6.4.1.2, the different variables are used to explain the analysis rules and, therefore, the relation between the variables.

Scenario / variable Number	Value type	Name: Description
(1)/(1)	Time	ActivityDuration: How long does it take an experienced user to perform a complete activity? This variable contains a value that expresses how long the experienced user needs for preparation and execution of an SCAC.
(1)/(2)	Time	TimeForKnowledgeResources: How long does an experienced user spend on external knowledge resources? This variable describes the time needed to handle different knowledge resources throughout the activity.
(1)/(3)	Boolean	ActivityCarriedOutSuccessfully: Has the experienced user completed the activity successfully? This variable represents whether an activity was successful or not.
(1)/(4)	List of resources	UsedKnowledgeSources: What type of knowledge sources, the experienced user handles to perform the activity? This variable describes the sources consulted such as Google, a phone or an experienced user to perform the activity.
(1)/(6)	List of tasks	MadeSubTasks: What sub tasks have the experienced user made to perform an activity? This variable collects all sub tasks done (e.g., open web page or download software unit)
(2)/(1) (input)	Time	TotalInputDuration: How long does the experienced user require to enter all the information into the Prometheus system? This variable contains a value of the

Evaluation and research result analysis

Scenario / variable Number	Value type	Name: Description
		testimony of an experienced user on how long the whole process of entering all their data needs.
(2)/(2) (input)	Boolean	SuccessfulEntry: Could the experienced user enter all the relevant information? This variable tells us whether an experienced user could enter all the information about a software module and complete activities in the system.
(2)/(3)	Time	ActivityDuration: How long does it take an experienced user to perform a complete activity using the Prometheus environment? This variable contains a value that expresses how long the experienced user needs for preparation and execution of a SCAC.
(2)/(4)	Time	TimeForKnowledgeResources: How long an experienced user spent on external knowledge resources using the Prometheus environment? This variable describes the time needed to handle different knowledge resources throughout the activity.
(2)/(5)	Boolean	ActivityCarriedOutSuccessfully: The experienced user has completed the activity successfully? This variable represents whether an activity was successful or not.
(2)/(6)	List of resources	UsedKnowledgeSources: How many knowledge resource are used to perform the activity? This variable describes the sources consulted, such as Google, phone or an experienced user to perform the activity.
(2)/(7)	List of tasks	MadeSubTasks: What sub tasks have the experienced user make to perform an activity? This variable collects all sub tasks done (e.g., open web page or download software unit)
(3)/(1)	Boolean	ResultsValid: Is the result of an activity conducted by Prometheus equivalent to the result of the same activity conducted by an experienced user? This variable indicates whether the experienced identifies the result of his Prometheus activity as valid.
(4)/(1)	Time	ActivityDuration: How long does it take an experienced user to perform an activity? This variable contains a value that expresses how long the experienced user needs for the preservation of the task to perform the activity.
(4)/(2)	Time	TimeForKnowledgeResources: How long does an inexperienced user spend on external knowledge resources? This variable describes the time needed to handle different knowledge resources throughout the activity.
(4)/(3)	List of resources	UsedKnowledgeResources: How many knowledge resource are used to perform the activity? This variable describes the sources consulted, such as Google, phone or an experienced user to perform the activity.
(4)/(4)	Boolean	ActivityCarriedOutSuccessfully: Has the inexperienced user has completed the activity successfully? This variable represents whether an activity was successful or not.
(4)/(5)	List of tasks	MadeSubTasks: What sub tasks have the experienced user completed to perform an activity?
(5)/(1)	Time	ActivityDuration: How long does it take an experienced user to perform an activity? This variable contains a value that expresses how long the experienced user needs for the preservation of the task to perform the activity.
(5)/(2)	Time	TimeForKnowledgeResources: How long an inexperienced user spent on external knowledge resources? This variable describes the time needed to handle different knowledge resources throughout the activity.
(5)/(3)	Boolean	UsedKnowledgeResources: How many knowledge resource are used to perform the activity? This variable describes the sources consulted, such as Google, phone or an experienced user to perform the activity.
(5)/(4)	List of resources	ActivityCarriedOutSuccessfully: Has the inexperienced user completed the activity successfully? This variable represents whether an activity was successful or not.
(5)/(5)	List of tasks	MadeSubTasks: What sub tasks have the experienced user completed to perform an activity?
(6)/(1) (7)/(1)	Boolean	ResultsValid: Was the result created by the participants an valid result? The experienced software engineer proof the results of Scenario 4 and 5 to be valid.

Table 43 - Overview variables of comparison methods used in case study scenarios

In general, the variables of Scenario 1 are used for the measuring methods MM5, MM6, MM7 and MM8. Therefore this scenario focuses on the problem of learning, search and receipt, and execution of SCAC knowledge. Scenario 2 focuses on the problem of SCAC knowledge storing and, therefore, the measurement method MM4. The second part of Scenario 2 focuses on the measuring methods MM5, MM6, MM7 and MM8. Scenario 3 is a support scenario for Scenario 5 and therefore supports the measuring methods MM5, MM6, MM7, and MM8 indirectly by proving the propriety of the Prometheus environment. Scenario 4 and 5 are equal to Scenario 1 and support the same measuring methods. Scenario 6 and 7 validates the SCAC results of scenarios 4 and 5. As a result, it supports the measuring methods MM5, MM6, MM7 and MM8 indirectly. Table 44 summarises this relationship between measuring methods and scenarios.

	MM4	MM5	MM6	MM7	MM8	General Support
Scenario 1		X	X	X	X	
Scenario 2	X	X	X	X	X	
Scenario 3						X
Scenario 4		X	X	X	X	
Scenario 5		X	X	X	X	
Scenario 6						X
Scenario 7						X

Table 44 - Case study scenario related to measurement methods

6.3.3.2. Measurement execution process

Section 6.3.1.3 shows different techniques which can be used to measure the previous discussed variables. In the following description, the relation between variables and the measurement technique is described.

In Scenario (1) seven variables are measured for each SCAC. The variable ActivityDuration is measured by the human observer. Here, the observer measures the complete time the experienced user needs to perform an SCAC manually. The experienced user defines the end point of the task. The time is recorded in whole seconds. For the variable

TimeForKnowledgeResource the human observer notes the time a participant spent on knowledge resources. The time is recorded in seconds. The variable UseKnowledgeSources is measured by a human observer. The resources used are listed by source name and a type if possible, e.g., co-worker, telephone and web page, Google (Web browser). The variable ActivityCarriedOutSuccessfully is measured by the human observer. The participant is asked after the completion of the activity if the work was finished successfully. The variable can only be set to yes or no. The variable MadeSubTasks is measured by the human observer. Here, the observer notes the progress of the entire task. The aim is to recognise different tasks and their duration, (e.g., “00:10:41h user open web page”).

Note: The same variables are used in Scenario (2), Scenario (4) and Scenario (5).

In Scenario (2) three additional measurements are made: The variable TotalInputDuration is measured by a human observer and measures the time an experienced user needs to enter all information into the Prometheus environment. The observer records the start time point at which they hand over the task to the experienced user. The end time is determined by the finishing signal of the user. The observer notes this point in time. Time is measured in whole seconds.

The variable SuccessfullEntry is measured with the human observer and the Prometheus environment. First, the experienced user has to inform the observer about the successful use of the environment. Secondly, the server in the Prometheus environment writes log files about entries. The variable can only be set to yes or no and present the personal opinion of the experienced user of completing the task.

In Scenario (3), (6) and (7) one measurement is made: The variable ResultsValid is captured by the type of measurement (1). The experienced user examines the results of the performed SCAC from Scenario (3), (6) and (7) with the same activity carried out in scenario (1). It tells the observer whether the result has the same value and is usable. The variable can only be set to yes or no.

Note: The maximum time for a single activity is set to 4 hours. This is based on the fact that all participants are volunteers and, therefore, perform this case study during their normal working time.

6.4. Result analysis

This section discusses the results created by the case study and the primary research. The aim is to measure the impact of the realised approach on the reuse of SCAC. Therefore, the case study results will be compared using the different defined variables and scenarios.

The basis of these discussions is the results of the case study shown in Section 6.4.2, the related properties of the fundamental concept discussed in Chapter 4, and the realised environment in Chapter 5. This section also discusses the possible positive effect of the approach for deployment software construction activities which are not part of the case study.

6.4.1. Analysis methods definition

In this section the analysis methods will be discussed. This includes the definition of variables, values and the statements for each variable measured by the different measuring methods. After this, the statements for the comparison of different values of the same variable will be discussed. At the end, the relationship between the variables and the resulting analysis statements will be defined.

6.4.1.1. General analysis concept

The case study creates a special analysis environment. An experienced software engineer and a group of inexperienced software engineers are asked to perform software reuse in their normal environment without the support of the Prometheus environment. This is done in the first scenario (by the experienced user) and in the fourth scenario (by the inexperienced users). Independent of the different variables and their value types, Figure 130 shows the expected behaviour of the measured values. It is expected that the experienced user (red line) needs less time, knowledge sources, and sub tasks to perform a SCAC than an inexperienced user (blue

Evaluation and research result analysis

line). By performing the same SCAC with the Prometheus environment (Scenario 2 and 5) this will support experienced as well as inexperienced users and may be create values in the area A, B and C (grey area).

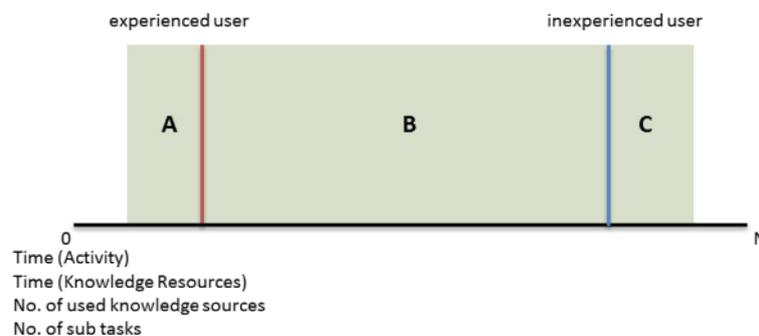


Figure 130 - Estimated results of experienced and inexperienced user

Using this behaviour, the problem of exchange and users' knowledge level can be discussed for each reused SCAC type. Using the Prometheus environment may create better results than an experienced user (A) or an inexperienced user (A, B). Also it is possible that a worse result than an inexperienced user (C) or an experienced user (B, C). Using this analysis approach it can be determined how the focused approach is working for one special SCAC regarding the same knowledge to perform. This is called single view in the case study analysis.

The different scenarios are performed using different software units and different tools. Therefore, the required SCAC knowledge differs between each single view. By comparison of the measured values of scenario 1, 2, 4, and 5 for each single view, it can be estimated whether the focused approach is feasible for different SCAC knowledge (e.g., information for setup or parameters). This is called multiple views in the case study analysis.

In general, three results are expected. The first one is that all measured values are placed in the combined area of A and B in Figure 130. In this case the approach shows that it can support inexperienced software engineers by performing different SCACs even if the engineers do not have the required knowledge. From the perspective of McCarey, Ó Cinnéide and Kushmerick (2008) a technique is identified to store and subsequently distribute reuse activity relevant

software unit knowledge among software engineers. The second is opposite to the first one. All measured values are in the area of C in Figure 130. In this case, the focused approach creates no added value for inexperienced software engineers. For the research itself, it only creates the statement that the realised environment is not an adequate technique to limit the lack of techniques mentioned by McCarey, Ó Cinnéide and Kushmerick (2008). The third result variant is a mix of the first and second result type. In this case the different results have to be analysed further. From the single view perspective it is necessary to prove if the focused approach is supporting only one or some types of values (e.g., time, knowledge resources, or sub task). Additionally, it has to be checked if the approach is supporting only one or a few different SCAC knowledge (i.e., only transformation or integration) by analysing the multiple view results.

6.4.1.2. Variable value definition

The first variables to discuss are used in Scenarios 1, 2, 4, and 5. The time measuring variables `ActivityDuration` and `TimeForKnowledgeResources` have no other special information than the time. Their relationship is that `TimeForKnowledgeResources` is part of `ActivityDuration`. The variable `ActivityCarriedOutSuccessfully` stated whether a single participant was able to finish the task. This is not the validation of the task. The `UsedKnowledgeSources` variable states the number of external (not the personal memory) knowledge resources. Together with the variables `MadeSubTasks` and `TimeForKnowledgeResources` it can be declared how often and how long a single knowledge resource was used. Additionally, `MadeSubTask` also shows how many sub tasks were done for the whole activity.

In Scenario 3, 6, and 7 the variable `Is Valid` is used. Looking on this variable for its own statement is relevant: the participant passed or failed to perform the SCAC.

For the input of values in Scenario 2 the variables `TotalInputDuration`, `SuccessfulEntry` and `MadeSubTasks` are used. Looking only at one single measurement, the values produce simple statements: A user needs a specific time to insert all information into the Prometheus

environment. This includes a number of specific tasks. An experienced user finished or did not finish the activity, which is expressed by the variable `SuccessfulEntry`.

6.4.1.3. Comparison of different measurements

The next step in the analysis is the comparison of multiple instances of the same scenarios. In the first step, the Scenarios 1, 2, 4, and 5 can be analysed together. Basically, it can be determined how long each participant needs to perform a given activity. Also the number of used knowledge resources, number of performed tasks, and the time spent on knowledge resources can be measured. Also average values can be calculated for each variable. For each participant, a statement can be created that identifies the needed time and for tasks to be completed. The same classification pattern can be used for `MadeSubTask` and `UsedKnowledgeSources`. Based on this, a statement for each participant can be created identifying the needed for sub tasks and knowledge resources for the given tasks. The content for the `ActivityCarriedOutSuccessfully` variable is the number of participants who finished a special SCAC and how many did not finish it.

Scenario 1 has the problem that only one experienced user for each SCAC performed this scenario. As a result, there is no comparable participant for measurement. The other experienced users performed other SCACs. Scenario 4 and 5 can use this comparison. In general, results under same conditions can be measured.

The problem of no comparability based on different SCACs is also true for Scenario 2 (i.e., the first three variables). All experienced users perform different a SCAC with different conditions (e.g., the software unit has to enter the system or is already available). The only variable which is useful in the current perspective is `SuccessfulEntry`. Thereby, a statement can be made as to how many experienced users were able to insert the necessary information. This is also valid for Scenario 3, 6 and 7. The statement here is only concerned about how many participants created a valid result. This statement can be made from an SCAC independent or focus view for each SCAC.

Evaluation and research result analysis

Table 45 shows the measurement results of a single participant performing one software construction activity. Here, the participant needed 01:28:42h to perform an activity. Exactly 00:39:50h (i.e., 44, 91%) of this time is spent on the task to gain knowledge and information about it. At least this participant uses 9 different knowledge resources and undertook 35 subtasks. The validation made by the experienced software engineer was made after the participant successfully finished the activity.

Time needed	KR Time	KR used	Sub Task done	Success	Valid
1:28:42	0:39:50	9	35	1	1

Table 45 - Summary of measured values of an SCA performed by a participant

In Table 46 an example for the preparation of an experienced software engineer in Scenario 2 is shown. As explained before the time and number of made sub tasks are measured.

Time needed	Success
0:40:31	1

Table 46 - Summary of measured values of a Prometheus preparation

Table 4647 shows the average values of one software construction activity performed by the experienced user and both inexperienced user groups (cf. single view in Section 6.4.2.1). Here, for each measured variable, the minimum, average and maximum value is calculated. The maximum and minimum value is only discussed further in the analysis if a special value is reached compared to other participants.

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	1:00:00	0:05:46	0:33:00	0:06:18
Average time	1:27:27	0:06:12	0:33:00	0:06:18
Max time	1:38:17	0:06:18	0:33:00	0:06:18
Min KR	9,00	1	2	1
Average KR	9,80	1	2	1
Max KR	10,00	1	2	1

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min tasks done	33,00	5,00	33,00	6,00
Average tasks done	33,40	5,80	33,00	6,00
Max tasks done	35,00	6,00	33,00	6,00
Min KR time	0:39:50	0:04:39	0:09:00	0:04:39
Average KR time	0:44:34	0:04:39	0:09:00	0:04:39
Max KR time	0:46:16	0:04:39	0:09:00	0:04:39
Success/Valid	1/1	1/1	1/1	1/1

Table 47 - Average values example of a software construction activity

The multiple views can be created by using the average values of all single view results of the same software construction activity type (e.g., transformation or integration). For this case study this results in two different tables (i.e., average results for integration and transformation software construction activities) which have the same structure as shown in Table 47.

6.4.2. Case study result analysis

In this section, the relevant results of the case study are shown and discussed based on the analysis methods shown in the previous section. In the first part, the average values for each software unit and the related SCACs are explained (single view). After this an average view is described focusing on integration and transformation SCACs (multiple views). Finally the relevant analysis results are discussed.

Note: A complete list of all measured values can be found in Appendix Section F

6.4.2.1. Single view analysis

In this section the measured results for each software construction activity are shown and discussed. The first example is discussed in more detail. For the others only average values are shown. In the Appendix Section F the all measurements are summarised for each single SCAC as shown in Table 47 and Table 48.

Evaluation and research result analysis

The DPWS Java stack was measured with a transformation and integration software construction activity. For the transformation, the following values are measured.

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	1:28:42	0:39:50	9	35	1	1
User M (2)	1:32:01	0:44:11	10	33	1	1
User M (3)	2:10:32	0:59:38	19	70	1	1
User M (4)	0:49:02	0:28:22	9	41	1	1
User P (1)	0:03:28	0:02:13	2	8	1	1
User P (2)	0:03:01	0:01:37	2	5	1	1
User P (3)	0:03:05	0:02:21	2	7	1	1
User P (4)	0:02:45	0:02:22	2	5	1	1
Expert M	0:13:21	0:03:43	3	25	1	1
Expert P	0:02:34	0:01:54	2	6	1	1

Table 48 - Measured values of each participant (DPWS4J transformation SCAc KR - Knowledge Resource)

Regarding the values in Table 48, the following statements can be made. For manual steps the inexperienced software engineers needed between 0:49:02h and 2:10:32h to perform the SCA while the expert required 0:13:21h. The difference is smaller between the software engineers using the focused approach. Here, the time is between 0:02:45h and 0:03:28h. The inexperienced users needed between 0:28:22h and 0:59:38h of the whole time to handle between 9 and 19 different knowledge resources. Between 33 and 70 tasks were done by each participant. The expert software engineer spent 0:03:43 minutes using 3 knowledge resources and performing 25 tasks. Compared to the software engineers who used the focused approach, the difference is apparent. Between 5 and 8 tasks were done by these participants and only 3 knowledge resource was used for 0:01:37h and 0:02:22h. The experienced software engineers using the focused approach have comparable values. The values shown in Table 48 are summarised to average values for further analysis (cf. Section 6.4.1.3) as shown in Table 49.

Evaluation and research result analysis

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	0:49:02	0:02:33	0:13:21	0:02:01
average time	1:30:04	0:02:51	0:13:21	0:02:01
max time	2:10:32	0:03:05	0:13:21	0:02:01
min KR	7,00	1	2	1
average KR	11,25	1	2	1
max KR	19,00	1	2	1
Min tasks done	33,00	5,00	25,00	4,00
average tasks done	44,75	5,75	25,00	4,00
max tasks done	70,00	7,00	25,00	4,00
Min KR time	0:29:22	0:01:48	0:03:00	0:01:54
average KR time	0:43:15	0:02:17	0:03:00	0:01:54
max KR time	0:59:38	0:02:37	0:03:00	0:01:54
Success/Valid	1/1	1/1	1/1	1/1

Table 49 - Average values of the DPWS Java transformation SCAC (KR Knowledge Resource)

As explained in Section 6.4.1.1 the expected value is between the experienced user and the inexperienced user. Based on the average values of Table 49, Figure 131a shows that the values of the experienced software engineer who performed this task manually as 100% (green line). The values of the inexperienced software engineers who did the task manually are represented as the maximum line (blue line). The values measured for the inexperienced engineers (red line) and the experienced engineers (purple line) in Figure 131(b) (enlarged view of Figure 131a to identify values smaller 100%) show that the focused approach required fewer knowledge resources, time and tasks to perform this SCAC.

Evaluation and research result analysis

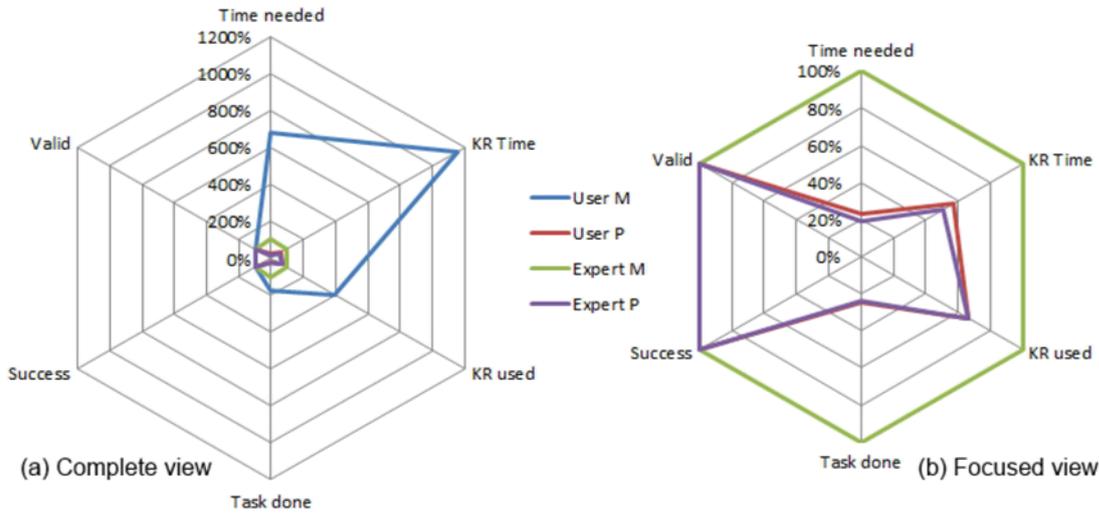


Figure 131 - Average results for the DPWS Java transformation SCAC

Based on Figure 131, it can be stated that for the measured transformation SCAC of the DPWS Java software unit unsupported inexperienced software engineers require more time (675%) to perform the SCAC, spent more time (1157%) to use more knowledge resources (392%), and do more tasks (179%) than an experienced user. On the other side inexperienced software engineers supported by the Prometheus environment use less time (23%) to perform the SCAC, spent less time (58%) to use less knowledge resources (67%), and do less tasks (25%) than an inexperienced user without such support. A supported experienced software engineer uses the same number of knowledge resources (2; 67%) but required subtly less time for the SCAC (19%), time for knowledge resources (51%) and tasks (24%).

The integration of the transformed DPWS Java stack software unit results in the average values is shown in Table 50.

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	0:18:15	0:02:45	0:07:45	0:02:46
average time	0:28:09	0:02:59	0:07:45	0:02:46
max time	0:45:05	0:03:14	0:07:45	0:02:46
Min KR	6,00	1	4	3
average KR	7,25	2	4	3

Evaluation and research result analysis

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
max KR	9,00	3	4	3
Min tasks done	46,00	5,00	23,00	6,00
average tasks done	50,75	6,25	23,00	6,00
max tasks done	56,00	7,00	23,00	6,00
Min KR time	0:06:21	0:01:55	0:02:45	0:01:56
average KR time	0:12:30	0:02:23	0:02:45	0:01:56
max KR time	0:20:34	0:02:43	0:02:45	0:01:56
Success/Valid	1/1	1/1	1/1	1/1

Table 50 - Average values of the DPWS Java integration SCAc (KR Knowledge Resource)

The inexperienced software engineers needed on average 0:28:09h (363%) to perform the activity. The unsupported experienced software engineer required 0:07:45h (100%) for the same activity, and the engineers using the Prometheus environment needed significantly less time. Exactly 0:02:59h (39%) was required by the inexperienced software engineer and 0:2:46h (36%) by the experienced engineer. A similar picture can be identified by the time spent on knowledge resources. The unsupported experienced software engineers spent 0:12:30h (455%) on average while the experienced engineer with the same condition spent 0:02:45h (100%). Compared to this the supported engineers spent less time. The inexperienced engineers took 0:02:23h (87%) and the experienced engineers took 0:1:56h (70%). The number of knowledge resources used differs significantly between the unsupported engineers. Here, the inexperienced engineers used 7.25 (181%) and the experienced engineer used 4 (100%). The supported software engineers used only 2 (50%; inexperienced user) and 3 (75%; experienced user) knowledge resources. The number of tasks done differs between the supported and unsupported groups significantly. Inside the groups the differences are not so high. The unsupported experienced engineer performed 23 (100%) tasks while the inexperienced software engineers, 50.75 (221%) on average. The supported groups use fewer tasks. The inexperienced groups use 6.25 (27%) tasks while the experienced engineer uses only 6 (26%) tasks.

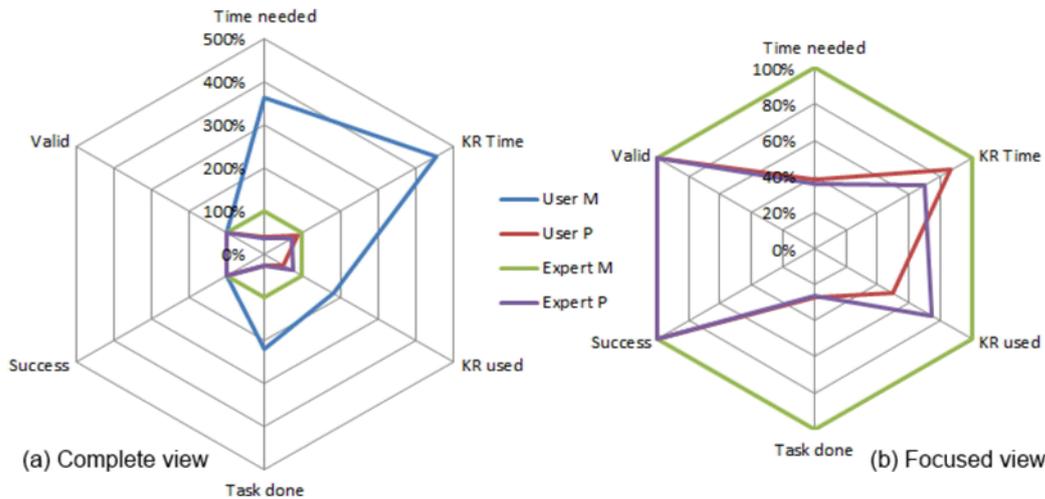


Figure 132 - Average results for the DPWS Java integration SCAC

Figure 132a focuses on the relation between both unsupported groups. The inexperienced engineer (blue line) needed more time, resources and tasks than the experienced software engineer (green line), Figure 132b focuses on the difference between the unsupported experienced software engineer (green line), the values if this person is supported (purple line), and the inexperienced software engineers (red line) supported by the focused approach. Here, it is shown that the focused approach supports all engineers by integrating the transformed DPWS Java Stack. All participants fulfilled the activity successfully and created a valid result.

From the transformation SCAC point of view Table 51 shows the average values measured for the DPWS C stack.

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	0:25:36	0:02:26	0:21:34	0:02:01
Average time	0:33:14	0:02:57	0:21:34	0:02:01
Max time	0:41:13	0:03:17	0:21:34	0:02:01
Min KR	5,00	2	5	2
Average KR	12,00	2,5	5	2
Max KR	18,00	3	5	2
Min tasks done	19,00	5,00	10,00	7,00
Average tasks done	22,75	7,25	10,00	7,00
Max tasks done	26,00	8,00	10,00	7,00

Evaluation and research result analysis

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min KR time	0:16:23	0:02:05	0:07:21	0:01:54
Average KR time	0:23:48	0:02:27	0:07:21	0:01:54
Max KR time	0:29:52	0:02:47	0:07:21	0:01:54
Success/Valid	1/1	1/1	1/1	1/1

Table 51 - Average values of the DPWS C transformation SCAC (KR Knowledge Resource)

For the transformation task of the DPWS C stack, inexperienced software engineers needed 00:33:14h on average to perform the SCAC without the support of the Prometheus environment. The experienced software engineer under the same conditions required 00:21:34h. Also the values of the used knowledge resources differ. While the inexperienced software engineers use 12 knowledge resources on average, the experienced uses only 5. While the inexperienced software engineers spent 00:23:48h of their time to use the knowledge resources, the experienced user spent only 00:07:21h. The number of tasks done for the SCAC differs (average view). Inexperienced software engineers needed 22.75 tasks and the experienced software engineer needed 10 tasks for the SCAC. All participants without the support of the focused approach were able to fulfil the SCAC and create a valid result. Figure 133a shows the differences between the inexperienced (blue line) and the experienced (green line; 100%) software engineers. On average, inexperienced software engineers require more time (154%) to perform the SCAC, spent more time (324%) to use more knowledge resources (240%) and do more tasks (228%) compared to the experienced user.

The integration activity of the DPWS C stack shows other values than the transformation activity.

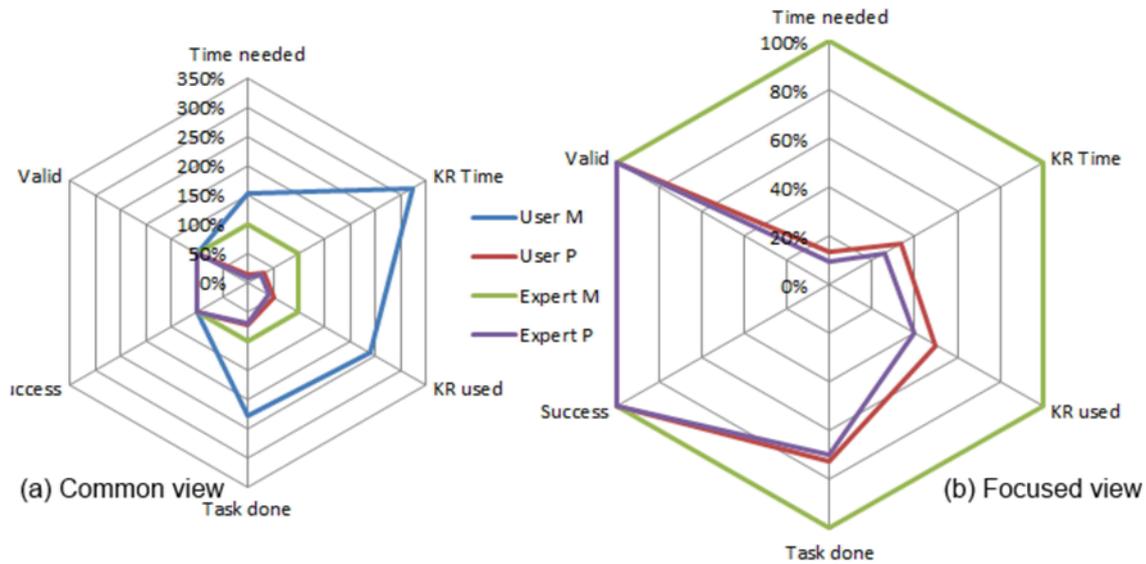


Figure 133 - Average results for the DPWS C stack transformation SCAC

By comparing the values of the experienced software engineer without the support of the Prometheus environment to the inexperienced engineers with support, the following statements can be made. The inexperienced software engineers require less time (00:02:57h; 14%), spent less time (00:02:27; 33%) on more number of knowledge resources (2,5; 50%), and did similar number of tasks (7.25; 73%). Figure 133b shows the experienced software engineer as 100% line (green line) and the average values of inexperienced engineers with support (red line).

Comparing the values of the experienced software engineer using the Prometheus environment and the average values of inexperienced engineers with the same support, the following statements can be made. The experienced software engineer needed less time (00:02:01h; 9%) for the complete SCAC and the use of knowledge resources (2; 40%). The number of tasks performed is 7 (70%) than the inexperienced software engineers on average. This engineers spent less time on knowledge resources (00:01:54h; 26%).

The measurement of integration SCAC for the DPWS C stack delivers the average values shown in Table 52. These are discussed as follows.

Evaluation and research result analysis

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	1:10:03	0:02:45	0:37:56	0:02:23
Average time	1:25:39	0:03:07	0:37:56	0:02:23
Max time	1:42:04	0:03:20	0:37:56	0:02:23
Min KR	9,00	2	4	4
Average KR	14,75	2,25	4	4
Max KR	22,00	3	4	4
Min tasks done	15,00	7,00	20,00	7,00
Average tasks done	31,75	7,75	20,00	7,00
Max tasks done	53,00	9,00	20,00	7,00
Min KR time	0:37:17	0:01:30	0:04:24	0:01:16
Average KR time	0:51:01	0:02:06	0:04:24	0:01:16
Max KR time	1:06:10	0:02:22	0:04:24	0:01:16
Success/Valid	1/1	1/1	1/1	1/1

Table 52 - Average values of the DPWS C integration SCAC (KR Knowledge Resource)

The integration of the DPWS C stack shows a similar picture as the other measurements. The inexperienced software engineers with the same conditions needed 01:25:39h (226%) on average to fulfil the SCAC. The experienced software engineer needed 0:37:56h (100%). With support, the experienced software engineer needed 0:02:23h (6%) and the inexperienced software engineers needed 0:03:07h (8%) of time. The number of knowledge resources is the same (4; 100%) for the experienced software engineer. The inexperienced software engineers without support needed 14,75 (369%) knowledge resource on average. The inexperienced software engineers with support needed 2,25 (56%) knowledge resources. The time spent on the knowledge resources differs between all participant groups. While the unsupported experienced software engineers needed 0:04:24h (100%) the inexperienced engineers with the same conditions spent significantly more time (i.e., 0:51:01; 1160%) on knowledge resources. The supported engineers require less time. The experienced engineers spent 0:01:16h (29%) and the inexperienced software engineers spent 0:02:06h (48%) of time. All participants fulfil their work and differ in the number of tasks. The inexperienced groups required 31.75 (159%;

Evaluation and research result analysis

without support) and 7.75 (39%; with support). The experienced engineer needed 20 tasks (100%; without support) and 7 tasks (35%; with support).

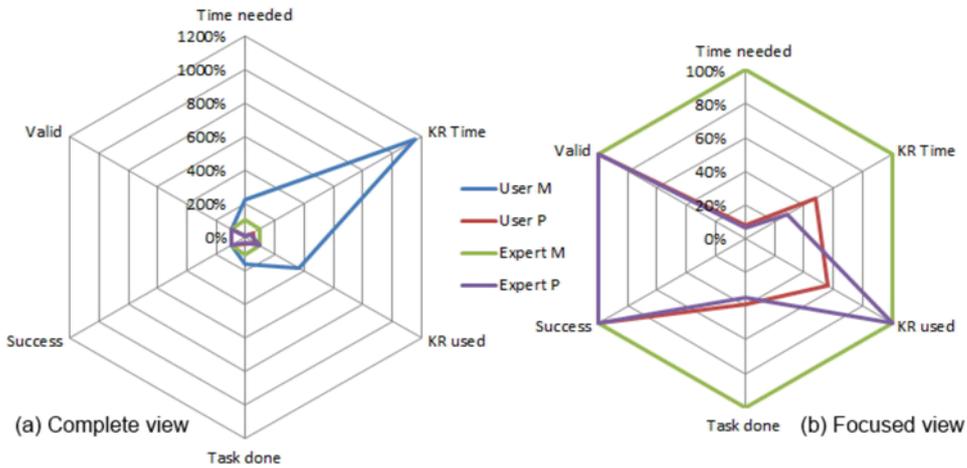


Figure 134 - Average results for the DPWS C integration SCAC

The final statement for this integration SCAC is similar to the previously analysed transformation SCAC. The approach supports inexperienced software engineers especially in reducing the number of knowledge resources and the time spent on these resources.

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	0:17:45	0:02:45	0:03:43	0:02:45
average time	0:22:02	0:03:12	0:03:43	0:02:45
max time	0:29:18	0:03:45	0:03:43	0:02:45
Min knowledge resource	5,00	2	3	2
average knowledge resource	7,75	2	3	2
max knowledge resource	9,00	2	3	2
Min tasks done	12,00	5,00	7,00	4,00
average tasks done	22,75	6,50	7,00	4,00
max tasks done	36,00	9,00	7,00	4,00
Min KR time	0:09:31	0:01:48	0:02:07	0:02:03
average KR time	0:13:25	0:02:20	0:02:07	0:02:03
max KR time	0:16:58	0:03:02	0:02:07	0:02:03
Success/Valid	1/1	1/1	1/1	1/1

Table 53 - Average values of the Log4J transformation SCAC (KR Knowledge Resource)

Evaluation and research result analysis

The measured value of the Log4J software unit for the transformation SCAC values are shown in Table 53. The unsupported inexperienced software engineers needed 0:22:02h (593%) and fulfil 22.75 (325%) tasks. This includes 00:13:25h (634%) for 7,75 (258%) knowledge resources on average. Figure 135b shows this (blue line) compared to the values of the experienced software engineer (green line). These engineers needed 0:03:43h (100%) to perform 7 (100%) tasks. 00:02:07h (100%) is used to handle 3 (100%) knowledge resources. This is shown in Figure 135a. Figure 135b shows the measured values of this engineer and the engineers supported by the focused approach. Regarding the time used for the activity and the time spent on knowledge resources the difference between the three groups is not huge. This is a difference to other transformation SCACs. Another difference to other measured SCACs is that, on average, the inexperienced software engineers with support spent more time (0:02:20h; 110%) on knowledge resources (2; 67%) than the unsupported experienced software engineers. The supported inexperienced software engineer needed 00:03:12h (86%) to perform 6,50 (93%) task. The supported experienced software engineer performed 4 (57%) task but required only 0:02:45h (74%). This engineers spent 00:02:03h (97%) on 2 (67%) knowledge resources.

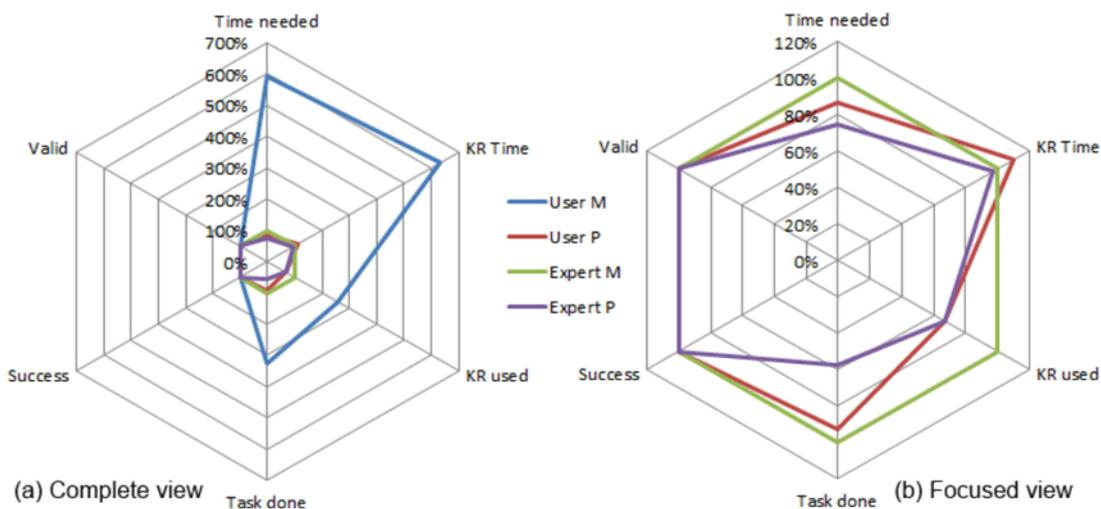


Figure 135 - Average results for the Log4J transformation SCAC

Evaluation and research result analysis

The results show that the focused approach supports inexperienced software engineers significantly. In this example the differences between supported engineers and the unsupported experienced software engineer are minimal.

A similar picture is shown by the integration SCAC of the Log4J software unit. Table 54 shows the average values.

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	0:12:34	0:02:32	0:03:56	0:02:15
Average time	0:31:22	0:02:47	0:03:56	0:02:15
Max time	0:49:34	0:03:07	0:03:56	0:02:15
Min KR	3,00	2	2	2
Average KR	11,00	2,75	2	2
Max KR	18,00	4	2	2
Min tasks done	7,00	4,00	7,00	5,00
Average tasks done	21,00	4,50	7,00	5,00
Max tasks done	39,00	5,00	7,00	5,00
Min KR time	0:06:44	0:02:11	0:02:15	0:01:45
Average KR time	0:15:49	0:02:34	0:02:15	0:01:45
Max KR time	0:21:03	0:03:02	0:02:15	0:01:45
Success/Valid	1/1	1/1	1/1	1/1

Table 54 - Average values of the Log4J integration SCAC (KR Knowledge Resource)

The differences between the unsupported groups are similar to other measured integration SCACs. The inexperienced software engineers required 0:31:22h (797%) to perform 21 (300%) tasks. They spent 0:15:49h (703%) for 11 (550%) knowledge resources. The unsupported experienced software engineer required 0:03:56h (100%) including 00:02:15h (100%) used for 2 (100%) knowledge resources. This engineer performed 7 (100%) tasks. The differences of both unsupported groups are shown in Figure 136a. Here, the experienced engineer (green line) required less time, knowledge resources, and performed fewer tasks than the inexperienced engineer group (blue line).

Evaluation and research result analysis

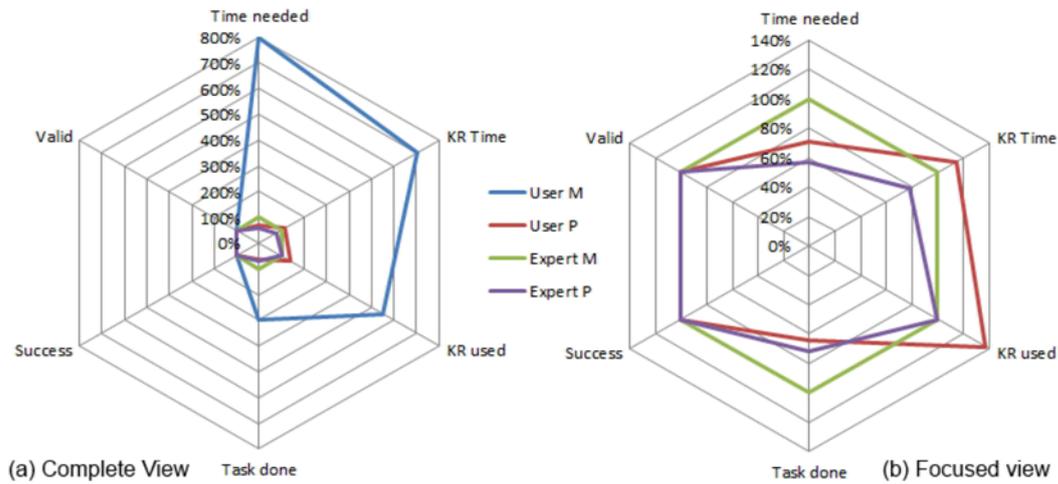


Figure 136 - Average results for the Log4J integration SCAC

Figure 136b shows the differences between the unsupported experienced software (green line) engineer and the supported engineers. The experienced software engineer supported by the focused approach needed 0:02:15h (57%) and the supported inexperienced engineer groups needed 0:02:47h (71%) on average. The unsupported experienced engineer needed 0:03:56h (100%) which is more time than the supported engineers. In other measured values this differs. The number of knowledge resources used is the same for experienced engineers; 2 (100%). The inexperienced engineers without the support of the focused approach used 2,75 (138%) of knowledge resources on average. Regarding the time spent on knowledge resources, the picture also differs. The unsupported inexperienced software engineer spent 0:15:49h (703%). This is significantly more than the other groups spent on knowledge resources. The unsupported experienced software engineer spent 0:02:15h (100%). Also, 0:02:34h (114%) was spent on such resources by the inexperienced software engineers with support by the focused approach. These engineers needed more time than the experienced engineer. The time spent 0:02:34h (114%) is more the time spent by the unsupported experienced software engineer. The supported experienced software engineer needed less time (0:01:45h; 78%). The task done by the supported experienced user is 5 (71%). For the experienced user without support the number of tasks done is 7 (100%). Figure 136b shows 4.5 (64%) and 5 (71%) tasks done by

Evaluation and research result analysis

inexperienced and experienced software engineers with support. This is the first time the inexperienced engineers used fewer tasks on average than the supported experienced engineer. Figure 136a show that the unsupported inexperienced software engineers perform 21 tasks in average. All participants were successful in finishing the activity and created a valid result.

The next SCAC to analyse is the Log4NET transformation SCAC. In Table 55 the average values measured of this SCAC scenario are listed.

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	0:21:02	0:01:59	0:12:35	0:02:02
Average time	0:25:06	0:02:29	0:12:35	0:02:02
Max time	0:28:31	0:03:05	0:12:35	0:02:02
Min KR	19,00	4	3	3
Average KR	26,25	5	3	3
Max KR	34,00	6	3	3
Min tasks done	34,00	6,00	9,00	4,00
Average tasks done	43,00	7,00	9,00	4,00
Max tasks done	49,00	8,00	9,00	4,00
Min KR time	0:16:36	0:01:34	0:05:56	0:01:50
Average KR time	0:18:56	0:01:54	0:05:56	0:01:50
Max KR time	0:21:02	0:02:21	0:05:56	0:01:50
Success/Valid	1/1	1/1	1/1	1/1

Table 55 - Average values of the Log4Net transformation SCAC (KR Knowledge Resource)

The most interesting values are related to the number of knowledge resources used. While the unsupported inexperienced software engineers used 26.25 (875%) knowledge resources and spent 00:18:56h (319%) of time, the unsupported experienced software engineer used only 3 (100%) knowledge resources in 00:05:56h (100%). The supported inexperienced software engineer required more knowledge resources (5; 167%) but invested only 00:01:54h (32%) of time. This is close to the experienced user supported by the Prometheus environment. This person spent 00:01:50h (31%) and used 3 (100%) knowledge resources. The other values (i.e., task done and time needed) are similar to other transformation SCACs. The unsupported

Evaluation and research result analysis

inexperienced software engineer required the most time (00:25:06h; 199%) and did the most tasks (43; 478%). The unsupported experienced software engineer performed only 9 (100%) tasks and required 00:12:35h (100%). The values of the supported engineers are not as high as the values of the unsupported experienced engineer. For the time required for the whole tasks, both are similar. The supported inexperienced software engineer required 00:02:29h (20%) to perform 7 (78%) tasks while the supported experienced user required 00:02:02h (16%) and performed 4 (44%) tasks. The unsupported engineers are shown in Figure 137 with the blue line (inexperienced software engineers) and green line (experienced software engineers). The supported engineers are represented by the purple line (experienced software engineers) and the red line (inexperienced software engineers).

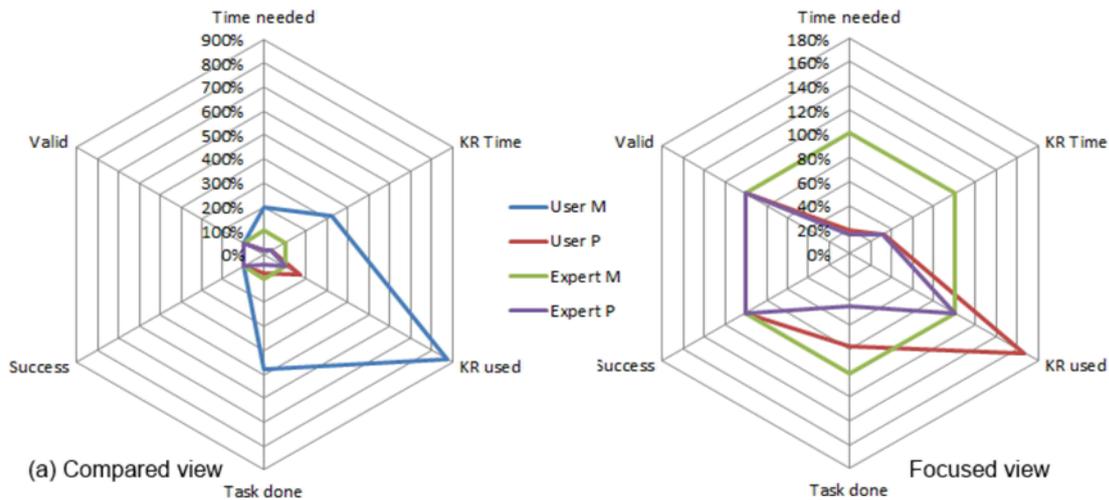


Figure 137 - Average results for the Log4Net transformation SCAC

The integration of the Log4Net software unit delivers a similar picture for the integration SCAC. Figure 138 shows the inexperienced software engineers without the support (blue line), the experienced software engineer with the same conditions (green line), the group of supported software engineers (red line) and the supported experienced engineer (purple line).

Evaluation and research result analysis

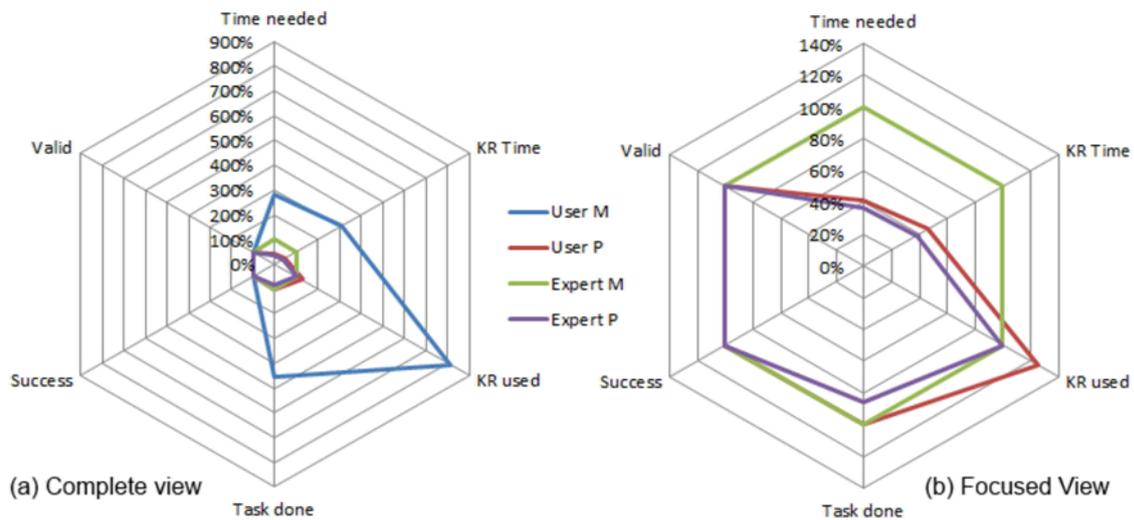


Figure 138 -Average results for the Log4Net integration SCAC

The figure is based on the values shown in Table 56.

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	0:14:45	0:02:31	0:06:54	0:02:29
Average time	0:19:13	0:02:52	0:06:54	0:02:29
Max time	0:24:09	0:03:05	0:06:54	0:02:29
Min KR	12,00	2	2	2
Average KR	16,25	2,5	2	2
Max KR	21,00	3	2	2
Min tasks done	24,00	6,00	7,00	6,00
Average tasks done	31,75	7,00	7,00	6,00
Max tasks done	39,00	8,00	7,00	6,00
Min KR time	0:12:48	0:02:02	0:05:06	0:01:56
Average KR time	0:15:39	0:02:21	0:05:06	0:01:56
Max KR time	0:17:23	0:02:46	0:05:06	0:01:56
Success/Valid	1/1	1/1	1/1	1/1

Table 56 - Average values of the Log4Net integration SCAC (KR Knowledge Resource)

The unsupported inexperienced software engineers needed on average 0:19:13h (279%) to perform the activity. The unsupported experienced software engineer required 0:06:54h (100%) for the same activity, and the engineers using the Prometheus environment needed less time.

Evaluation and research result analysis

Exactly 0:02:52h (41%) was required by the inexperienced software engineer and 0:02:29h (36%) by the experienced engineer. A similar picture can be identified by the time spent on knowledge resources. The unsupported experienced software engineers spent 0:15:39h (307%) on average while the experienced engineer with the same condition spent 0:05:06h (100%). Compared to this the supported engineers spent less time. The experienced engineers took 0:01:56h (38%) and the inexperienced engineers took 0:2:21h (46%). The number of knowledge resources used differs significantly between the unsupported engineers. Here, the inexperienced engineers used 16.25 (813%) and the experienced engineer used 2 (100%). The supported software engineers used only 2.5 (125%; inexperienced user) and 2 (100%; experienced user) knowledge resources. The number of tasks done differs between the supported and unsupported groups not so much. The unsupported experienced engineer performed 7 (100%) tasks while the inexperienced software engineers, 31.75 (454%) on average. The supported groups use similar number of tasks as the unsupported experienced engineer. The inexperienced groups use 7 (100%) tasks while the experienced engineer uses only 6 (86%) tasks.

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	0:18:09	0:02:10	0:09:08	0:02:46
Average time	0:22:37	0:02:39	0:09:08	0:02:46
Max time	0:26:12	0:03:01	0:09:08	0:02:46
Min KR	12,00	2	2	3
Average KR	14,75	2,5	2	3
Max KR	19,00	3	2	3
Min tasks done	19,00	6,00	7,00	7,00
Average tasks done	23,00	6,75	7,00	7,00
Max tasks done	28,00	7,00	7,00	7,00
Min KR time	0:15:03	0:01:50	0:05:06	0:02:16
Average KR time	0:17:36	0:02:02	0:05:06	0:02:16
Max KR time	0:21:11	0:02:13	0:05:06	0:02:16
Success/Valid	1/1	1/1	1/1	1/1

Table 57 - Average values of the EWSJ transformation SCAC (KR Knowledge Resource)

Evaluation and research result analysis

In Table 58 the measured values for the transformation SCAC of the EWS J transformation are shown. An interesting point is that the supported experienced user required more knowledge resources (3; 150%) than the supported inexperienced software engineer (2.5; 125%). The unsupported experienced engineer required fewer knowledge resources (2; 100%). Focusing on the time spent on knowledge resources, the picture changes. While the supported inexperienced engineers spent only 00:02:02h (40%), the experienced software engineer with the same constraints spent 00:02:16h (44%). The unsupported experienced engineer spent more time on knowledge resources (00:05:06h; 100%). The unsupported inexperienced software engineer spent more time (00:17:36h; 345%) and used more knowledge resources (23; 738%) than the unsupported software engineer. The number of tasks is in the following order. The unsupported inexperienced software engineers performed 23 (329%) tasks and performed them in 00:22:37h (248%). The unsupported software engineer spent 00:09:08h (100%) to perform 7 (100%) tasks. The supported engineers required less time. The inexperienced group required 00:02:39h (29%) and performed 6.75 (96%) tasks. The experienced software engineer spent 00:02:46h (30%) (which is more than the average of the supported inexperienced software engineer) to perform the same number of tasks as the unsupported experienced engineer (7; 100%).

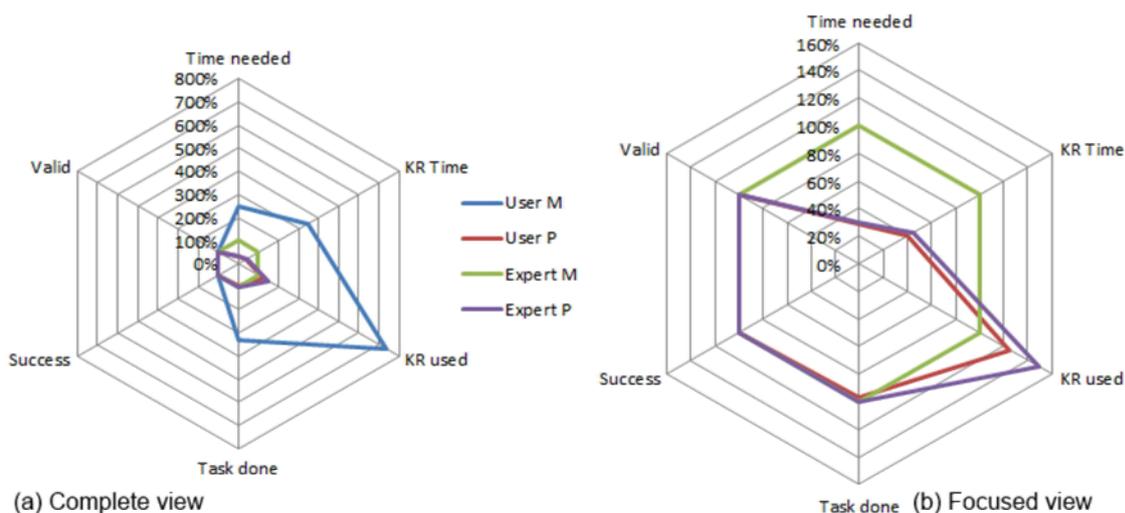


Figure 139 - Average results for the EWSJ transformation SCAC

Evaluation and research result analysis

Figure 139 shows the blue line (inexperienced software engineers) and the green line (experienced software engineers). The supported engineers are represented by the purple line (experienced software engineers) and the red line (inexperienced software engineers).

Table 58 shows the results measured for the integration of the EWS software unit (Java technology).

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	0:26:54	0:02:10	0:15:03	0:02:51
Average time	0:39:36	0:02:31	0:15:03	0:02:51
Max time	0:53:23	0:02:54	0:15:03	0:02:51
Min KR	9,00	2	4	4
Average KR	14,25	2,5	4	4
Max KR	23,00	3	4	4
Min tasks done	7,00	6,00	7,00	5,00
Average tasks done	34,00	6,50	7,00	5,00
Max tasks done	52,00	7,00	7,00	5,00
Min KR time	0:18:03	0:01:45	0:05:06	0:01:54
Average KR time	0:24:05	0:02:08	0:05:06	0:01:54
Max KR time	0:29:05	0:02:31	0:05:06	0:01:54
Success/Valid	1/1	1/1	1/1	1/1

Table 58 - Average values of the EWSJ integration SCAc (KR Knowledge Resource)

The group of inexperienced software engineers required 00:39:36h (263%) on average. When compared to the unsupported experienced software engineer, they needed more time and required 00:15:03h (100%). The supported experienced engineer spent 00:02:51h (19%) and the supported inexperienced software engineer spent 00:02:31h (17%). Compared to the unsupported experienced software engineer, this is significantly less time.

The unsupported inexperienced software engineer performed 34 (486%) tasks and the experienced software engineer with the same conditions performed 7 (100%) tasks. Compared to this time required, the supported software engineer performed fewer tasks to perform the SCAc. The inexperienced engineers also performed 6.5 (93%) tasks while the experienced

Evaluation and research result analysis

engineer performed 5 (71%). The same order, but with fewer intervals, can be identified for time spent on knowledge resources. The inexperienced software engineer without support spent 00:24:05h (472%) on 14.25 (356%) knowledge resources (on average). The unsupported experienced software engineer spent only 00:05:06h (100%) on 4 (100%) knowledge resources. The supported inexperienced software engineer uses the same number of knowledge resources (4; 100%) but spent only 00:01:54h (37%). The supported inexperienced software engineer used only 2.5 (63%) of knowledge resources and spent 00:02:08h (42%) in total. Figure 140 shows the described values in graphical form. The unsupported engineers are shown with the blue line, the inexperienced software engineers and the green line for experienced software engineers. The supported engineers are represented by the purple line, the experienced software engineers, and red line for inexperienced software engineers.

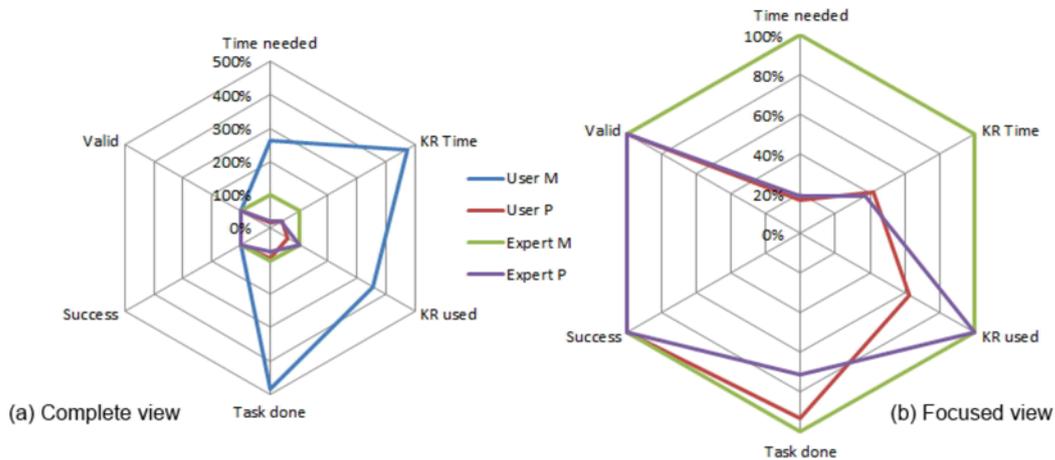


Figure 140 - Average results for the EWS J integration SCAC

The final software unit is EWS (.NET). In the following, the integration and transformation SCAC of this unit will be described. Table 59 shows the average values for the integration SCAC.

Evaluation and research result analysis

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	0:26:54	0:02:10	0:09:08	0:02:01
Average time	0:33:21	0:02:32	0:09:08	0:02:01
Max time	0:41:56	0:02:54	0:09:08	0:02:01
Min KR	4,00	2	4	2
Average KR	5,00	2,5	4	2
Max KR	6,00	3	4	2
Min tasks done	7,00	5,00	7,00	4,00
Average tasks done	19,50	5,75	7,00	4,00
Max tasks done	52,00	7,00	7,00	4,00
Min KR time	0:04:08	0:01:45	0:05:06	0:01:54
Average KR time	0:14:47	0:02:08	0:05:06	0:01:54
Max KR time	0:29:01	0:02:31	0:05:06	0:01:54
Success/Valid	1/1	1/1	1/1	1/1

Table 59 - Average values of the EWS .NET integration SCAC (KR Knowledge Resource)

The differences between the unsupported groups are similar to other measured integration SCACs. The inexperienced software engineers required 00:33:21h (365%) to perform 19.5 (279%) tasks. They spent 00:14:47h (290%) for 5 (125%) knowledge resources. The unsupported experienced software engineer required 00:09:08h (100%) including 00:05:06h (100%) used for 4 (100%) knowledge resources. This engineer performed 7 (100%) tasks.

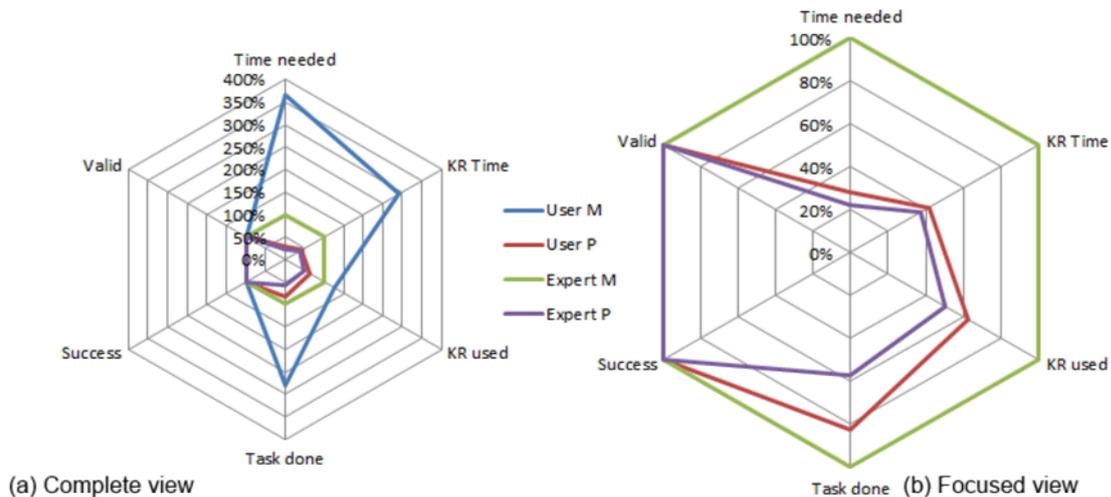


Figure 141 - Average results for the EWS .NET integration SCAC

Evaluation and research result analysis

The differences of both unsupported groups are shown in Figure 141a. Here, the experienced engineer (green line) required less time, knowledge resources and performed fewer tasks than the inexperienced engineer group (blue line). Figure 141b shows the differences between the unsupported experienced software engineer (green line) and the supported engineers. The experienced software engineer (purple line) supported by the focused approach needed 00:02:01h (22%) and the supported inexperienced engineer groups (red line) needed 00:02:32h (28%) on average. The unsupported engineer needed 00:09:08h (100%) which is more time than the supported engineers. In other measured values this differs.

The number of knowledge resources used, is for experienced engineers 2 (50%). The inexperienced engineers without the support of the focused approach used 2.5 (63%) of knowledge resources on average. Regarding the time spent on knowledge resources, the picture is similar. The unsupported experienced software engineer spent 00:14:47h (290%). This is significantly more than the other groups spent on knowledge resources. The unsupported experienced software engineer spent 00:01:54h (100%). A total of 00:02:08h (42%) was spent on such resources by the inexperienced software engineers with the support of the focused approach. These engineers needed more time than the experienced engineer. The supported experienced software engineer needed less time (0:01:54h; 37%).

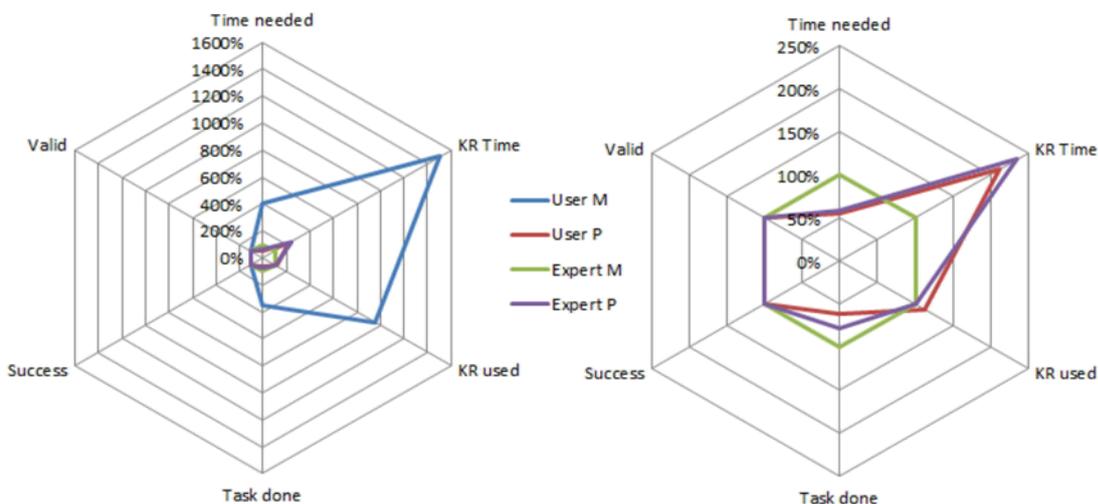


Figure 142 - Average results for the EWS .NET transformation SCAC

Evaluation and research result analysis

The completed tasks done by the experienced user with support was 4 (57%) and 5.75 (82%) for the inexperienced engineers).

The last SCAC is the transformation of the EWS (.NET). Figure 142 shows the measured values. The most interesting point is that the unsupported experienced software engineer spent less time 00:01:10h (100%) on knowledge resources and used fewer knowledge resources (2; 100%) as the supported inexperienced engineer. This person used 2.25 (113%) knowledge resources and spent 00:02:29h (213%). The supported experienced software engineer spent 00:02:45h (236%) and used 2 knowledge resources (100%). Regarding the total time spent and the task performed, the picture changes. Here, the supported engineers required less time and did fewer tasks. While the unsupported experienced software engineer (100% line) needed 00:05:13h to performed 9 (100%) tasks the same person with support required only 00:03:04h (59%) to perform 7 (78%) tasks. The supported inexperienced software engineer required (on average) the minimum time (00:02:52h; 55%) to perform 5.5 (61%) tasks. As in the other SCACs the inexperienced software engineer without support needed the most time (00:21:07h; 405%) and performed 31.75 (353%) tasks. This includes 00:17:34h (1505%) of time spent on knowledge resources and 31.75 (950%) on tasks. Table 60 includes these average values.

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min time	0:16:00	0:02:43	0:05:13	0:03:04
Average time	0:21:07	0:02:52	0:05:13	0:03:04
Max time	0:26:45	0:03:05	0:05:13	0:03:04
Min KR	15,00	2	2	2
Average KR	19,00	2,25	2	2
Max KR	22,00	3	2	2
Min tasks done	25,00	2,00	9,00	7,00
Average tasks done	31,75	5,50	9,00	7,00
Max tasks done	42,00	8,00	9,00	7,00

	Inexperienced User Manual	Inexperienced User Prometheus	Experienced User Manual	Experienced User Prometheus
Min KR time	0:14:23	0:02:17	0:01:10	0:02:45
Average KR time	0:17:34	0:02:29	0:01:10	0:02:45
Max KR time	0:21:34	0:02:45	0:01:10	0:02:45
Success/Valid	1/1	1/1	1/1	1/1

Table 60 - Average values of the EWS .NET transformation SCAC (KR Knowledge Resource)

6.4.2.2. Multiple view analysis

In the multiple view analyses, all complete software construction activities for transformation and integration are compared. As a result, statements for the two types of SCACs can be made. Additionally, both SCAC types can be compared based on analysis.

The measured values of the six software units and their transformation SCAC can be summarised as follows (cf. Figure 143a). From the average point of view in transformation software construction activities, inexperienced software engineers without support needed 379% of time compared to an experienced software engineer (100%). An inexperienced software engineer using the Prometheus environment needed 38% of the time.

The time spent on knowledge resources by supported inexperienced software engineers is 81% of the time that the experienced software engineer without support spent. The inexperienced software engineers spent 714% of time compared to the supported experienced engineer. Also the number of used knowledge resources is different. While supported engineers used fewer knowledge resources (98%), the experienced software engineer without support used significantly more knowledge resources (575%). Figure 143a shows this with the blue line (not supported inexperienced software engineers), red line (supported in inexperienced software engineer), green line (supported experienced software engineers), and the purple line (supported experienced software engineer).

Even though it is not in the scope of the research, results shows that an experienced software engineer can also be supported by the focused approach in performing transformation activities.

Evaluation and research result analysis

However, these differences are not as significant as the difference between inexperienced software engineers with and without support. Regarding the tasks done the differences are not so high. Inexperienced software engineers without support perform 173% of tasks. Here, the unsupported experienced engineer is at the 100% mark. The supported inexperienced engineers perform 76% of tasks in comparison to the 100%.

Based on the results, the inexperienced software engineers are supported by the focused approach significantly. These engineers were able to reuse a transformation activity in less time and with less knowledge resource than the inexperienced control group. Figure 143b shows this with the red line (supported inexperienced software engineer), green line (supported experienced software engineers), and the purple line (supported experienced software engineer).

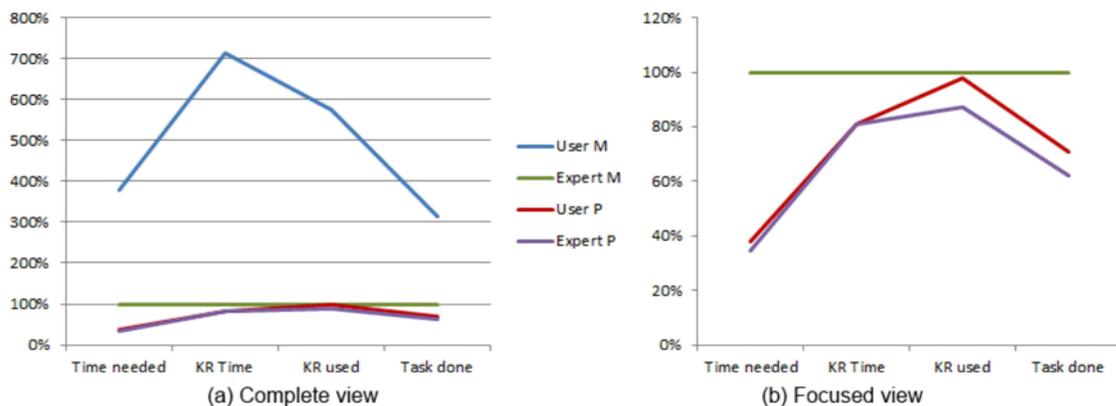


Figure 143 - Average results all measured transformation SCAC

The integration SCAC shows a similar picture to the transformation SCAC. The inexperienced software engineers without support needed more time (382%) to perform 316% of tasks compared to the unsupported experienced software engineers (100%). By using the focused approach, inexperienced software engineers were able to perform the integration activities by using less time (34%) and performing fewer tasks (68%). The time spent on knowledge resources by unsupported inexperienced software engineers (564%) shows the highest difference compared to the inexperienced software engineer (100%) regarding all measured

Evaluation and research result analysis

values for this SCAC. The supported engineers without experience spent less time (63%) on such resources. This is related to the fact that these engineers also used only less knowledge resource (82%). But the unsupported engineers with the support needed significantly more of such resources (399%). Figure 144a shows the difference between the four different scenarios measured including the unsupported inexperienced engineer (blue line). Figure 144b focuses on the explained differences between supported engineers (green and purple line) and the unsupported experienced software engineer (red line).

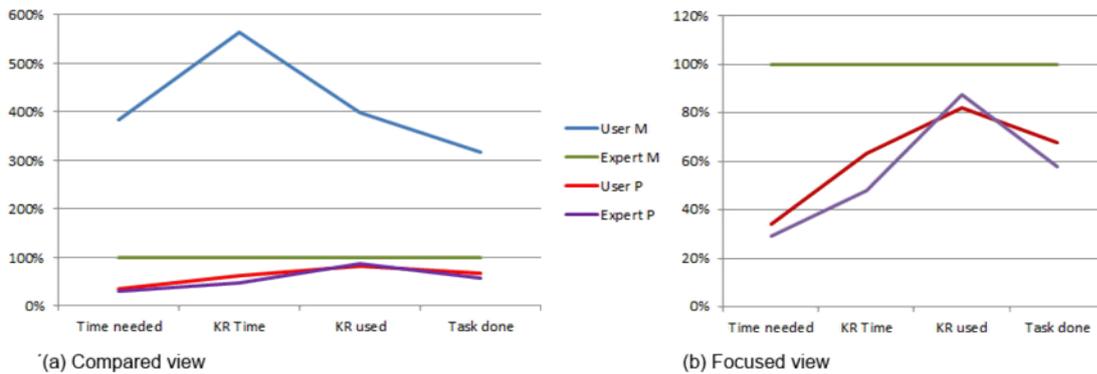


Figure 144 - Average results all measured integration SCAC

Figure 144b also shows the difference between the experienced and the inexperienced software engineer using the focused approach. The differences are minimal. The experienced software engineer required less time (29%), performed fewer tasks (58%), and spent less time (64%) with knowledge resources (48%). The number of resources is about 88% for the experienced software engineer and 82% for the group of inexperienced engineers.

Based on the results the final statements can be made in that inexperienced software engineers are significantly supported by performing integration SCAC using the focused approach. These engineers were able to perform integration activities in less time and with less knowledge resource than the inexperienced control group.

By comparison, the measured transformation and integration SCACs can be reasoned as follows:

In both SCACs the added value of the focused approach is clearly recognisable. The realised environment supports inexperienced software engineers. It performed both SCAC types with different sub activities. For both types it is shown that time, the number of used knowledge resources and the number of tasks performed can be reduced. Particularly for the complete activity time, the number of knowledge resources used, and the time spent on these resources, this statement is valid. The measured values also show differences. In general, the described reduction of effort is more distinctive in transformation activities. Integration activities are performed faster with fewer knowledge resources. A possible reason for this could be that integration activities are easier to understand and to perform. The IDEs used could be another factor in this comparison. Visual Studio and Eclipse both use intuitive and similar ways for integration. Such behaviour makes it easier to identify a way for integration. On the other side, for transformations a lot of different and specialised tools exist. These may have different behaviours and require domain-specific knowledge for use. This makes it difficult to use. In the case study all participants were able to perform the SCAC successfully.

6.4.2.3. Reflecting case study results on device deployment software construction activity

The deployment software construction activity was explained in the concept (cf. Section 4.5) and the realisation of the related software construction model was shown (cf. Section 5.3.1). The device deployment activity was not part of the case study because of security issues. The work on device deployment required (in the application area of Schneider Electric) special training and also, to be trained in first aid. Therefore, the analysis of the deployment SCAC is of a theoretical nature. The published deployment SCAC experiment (cf. Zinn, Fischer-Hellmann and Schoop, 2012a) was done with trained staff.

Based on the described software construction activity models a comparison analysis can be made to reason how the focused approach takes effect on the focused deployment.

It may be supposed that the focused approach is able to facilitate the performance of deployment software construction activities. This is based on two facts.

The first one is the typical procedure used for device deployment. As described in the example in Section 3.1.6 a lot of vendor or device specific tools are used. Often, these tools are console-based tools performing some transformation or deployment activities. Regarding the three problem areas such tools represent technology which required knowledge for use (cf. example in Section 3.1.6). A software engineer should know or be prepared to learn this knowledge to use such tools. The third problem area describes this problem because of missing knowledge in the environment in which SCAC knowledge should be performed. Based on the fact that a lot of console-based tools are necessary for deployment, this scenario is similar to a transformation activity. Additionally, the deployment into a device has similarities to the integration of a software unit into an IDE. These analogies are an indication that the focused approach can support this type of activity.

The second fact is the similarity of the software construction models. . Based on the fact that console-based tools are used for deployment which is, in the scope of the research, equal to the start (device vendor specific) console application, the first part of the deployment activity model (cf. Figure 89) has a similar structure to the transformation SCACs model. It describes all necessary information and relates to a console application. The inexperienced software engineers do not know this model.

Working with devices shows another problem. The example in Section 3.1.6 shows that manual sub tasks (e.g., to switch a device on or off) have to be made. Until this point, the focused approach is able to support an inexperienced user. At this point the support by the focused approach is limited. The approach is able to show a textual description of sub steps (created by the experienced user), but this depends on the experienced user creating such information and how an inexperienced user understand such descriptions.

6.4.2.4. Other results of the case study

Next to the focused case study results, three outside results are acknowledged. The first one is the aforementioned result where experienced users are also supported by the focused approach. Often a supported experienced user needed less time than the inexperienced user without this support. Because of the saved time for the setup of an SCAc, this is not a surprising result.

More surprising were the differences between both supported user types (i.e., inexperienced and experienced). The experienced user needed less time and in some cases fewer knowledge resources. As a possible reason for this effect, it can be considered that these users performed the storing of SCAc related knowledge into the Prometheus Environment. Based on this task, these users know the system better than the inexperienced users.

The last acknowledged unforeseen result of the case study is the time experienced users required to insert information into the Prometheus system. Table 61 shows this setup time for each SCAc.

Software unit	SCA type	Setup time	Average time with/without support	Breakeven point
SU1 DPWS C	Integration	00:17:04	00:33:14/00:02:45	0,56
SU1 DPWS C	Transformation	00:46:32	01:25:39/00:02:57	0,56
SU2 DPWS J	Integration	00:25:12	00:28:09/00:02:59	1,00
SU2 DPWS J	Transformation	00:51:01	01:30:04/00:03:05	0,59
SU3 EWS .NET	Integration	00:34:23	00:33:21/00:02:32	1,12
SU3 EWS .NET	Transformation	00:23:44	00:21:07/00:02:32	1,28
SU4 EWS J	Integration	00:12:03	00:39:36/00:02:31	0,32
SU4 EWS J	Transformation	00:09:44	00:22:37/00:02:39	0,49
SU5 Log4Net	Integration	00:09:23	00:19:13/00:02:52	0,57
SU5 Log4Net	Transformation	00:05:34	00:25:06/00:02:29	0,25
SU6 Log4J	Integration	00:11:24	00:31:22/00:02:47	0,40
SU6 Log4J	Transformation	00:13:54	00:22:02/00:03:12	0,74

Table 61 - Setup time for focuses SCAcs

Taking the average time saved by inexperienced users (i.e., difference of time needed for the inexperienced users with and without support) and compare it to the setup time of the

experienced user, it shows that a breakeven point (regarding the time) is reached after 1 or 2 reuses (depending on the SCAC).

6.4.2.5. Final result of the case study

The previous section compared unsupported inexperienced software engineers with supported inexperienced software engineers by using the unsupported experienced engineer as a 100% comparison line. The aim of the case study was to identify the impact of the focused approach on inexperienced software engineers. Therefore, a direct comparison of the inexperienced software engineer performing the SCACs with and without the focused approach was necessary.

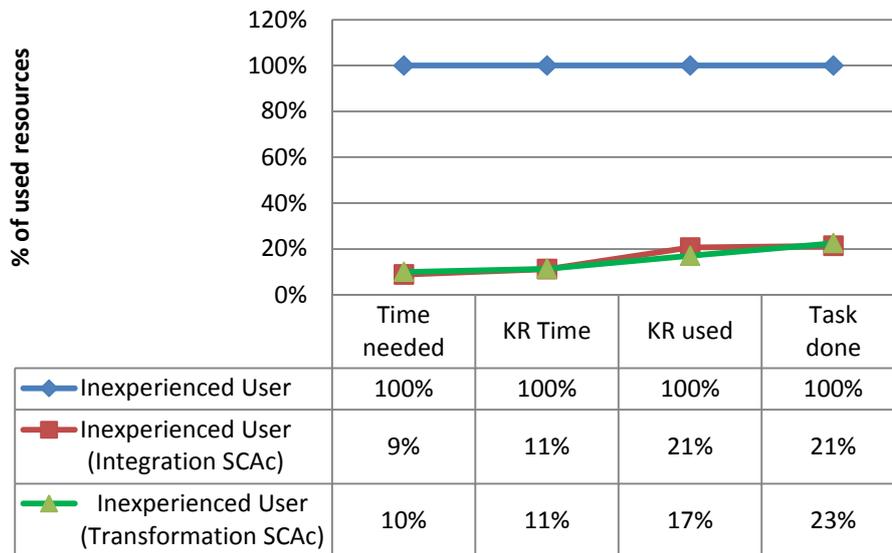


Figure 145 - Final comparison of inexperienced software engineers

Figure 145 shows a comparison between the inexperienced software engineers measured in the case study using the average values (in %) of the case study. The engineers not using the focused approach of the both SCAC types are represented by the blue line (100%). Compared to these engineers the supported engineers performing transformation SCACs required only 10% of the time. For the integration SCAC, the value is about 9%. The reduction of time for SCAC is between 90%-91% on average. A similar picture can be found for the number of knowledge resources here; the supported engineers performed 17% (transformation SCAC) to 21%

(integration SCAC) of the tasks of an unsupported engineer. This results in a reduction of time to about 79-83%. Both reductions may be explained by the reduction of effort for setup (e.g., installation, configuration, and so on) of the SCAC and related tools. The other relevant point of this thesis is the impact of the use of knowledge. Here, Figure 145 shows interesting results. Compared to the unsupported inexperienced software engineer, supported engineers spent only 11% (for transformation SCAC) and 11% (for integration SCAC) of the time on knowledge resources. This reduction (about 89%) is based on the fact that the number of knowledge resources is reduced from about 79% to 83%. For the transformation SCAC, the supported engineers required only 21% of the tasks the unsupported engineers required. For the integration SCAC this values is about 23%.

Finally, the case study shows that the focused approach is able to support inexperienced software engineers to perform two types of software construction activities. The supported engineers performed these activities faster and produced the same result as expected by the experienced engineer. The engineers used fewer knowledge resources than the engineers without support. The conclusion out of this fact is that the focused approach is one way to enable inexperienced software engineers to perform the software construction activities researched.

6.5. Case study hypothesis review

The case study hypothesis in Section 6.1.3 and the related theoretical viewpoints discussed in Section 6.2.4 focus on following points:

4. SCAC knowledge/information can be stored in an environment and can be reused by inexperienced users.
5. Such reuse produces a comparable (working) result to an experienced user in a normal application area, but without the need of learning the required knowledge for the specific SCAC or a comparable knowledge transfer.

6. The inexperienced status of the user, which relates with the specific SCAC, does not change.

Relating these three points, the case study shows that SCAC related knowledge can be stored in such an environment. The experienced software engineer validates and approves that the created results are comparable working results. The supported inexperienced software engineers used fewer knowledge resources and spent less time on these resources (compared to the unsupported inexperienced software engineers) to produce valid results. This can be interpreted as follows. The results show that the need of learning (i.e., use of knowledge resources and spent time on these resources) or handling the required knowledge for the specific SCAC or a comparable knowledge transfer is lower compared to supported inexperienced software engineers. However, both groups created working results. Even if the inexperienced status of the user can be changed (e.g., the person now knows that reuse of a software unit is possible, which is relevant for reuse) (Garcia, 2006) it can be stated that the user did not use the same resources and spent less time on knowledge resources. Therefore, it is probable that the inexperienced status did not change in the same way and to the same level. The focused approach does not describe how an SCAC is performed or how it is prepared, although different technologies of software units and SCACs have been used in six parallel experimental settings. The measured values (i.e., time and knowledge resources) are similar and result in a positive effect (i.e., reduction of time and knowledge resources used).

6.6. Summary

This chapter describes the specific research methods to perform a case study and analyses the case study results. Thereby, the setup of the case study is shown. This includes a description of the environment, participants, variables, and methods for analysing and measuring of values. The aim of the case study is to measure the impact of the focused approach on inexperienced software engineers.

The case study is performed in cooperation with different software unit experts (experienced software engineers) and two groups of inexperienced software engineers. The experienced as well as the inexperienced software engineers perform two different software construction activities (i.e., different integration and transformation activities) for each of six different software units. Each activity was performed with and without the use of the Prometheus environment based on the focused approach of this research. The number of used knowledge resources, the time spent on these resources, the time needed for performing the activity, the success, and the number of tasks done was measured for each software construction activity performed.

The chapter analyses the results from two perspectives. The first one is the comparison of the different values. The second one is the discussion of the measured values regarding the aim of the thesis. The comparison of the values results in the following statement.

Regarding the time, knowledge resources and tasks, the experienced software engineers needed less time, knowledge resources and accomplished fewer tasks than the inexperienced software engineers. Using the Prometheus environment, inexperienced engineers needed less time, knowledge resources and did fewer tasks than inexperienced software engineers without this environment. Often, a Prometheus environment user needed less time, knowledge resources and did fewer tasks than experienced software engineers performing software constructions manually. This was an unexpected result of the case study.

The aim of the research is to enable inexperienced software engineers to perform software reuse activities with a reduced need of handling the necessary knowledge. The measured results show that a reduction of used knowledge resource and time spent on these resources is possible. Therefore, the results of the case study support the research aim. In general, the three knowledge problem areas were handled by the approach as expected by the concept of the approach. Different technology information and the related software construction activity knowledge were stored in the environment. Additionally, the environment reduces the task to

Evaluation and research result analysis

be completed (i.e., time, number of knowledge resources and setup task). The inexperienced software engineers perform software construction activities without the full activity knowledge required. Therefore, this chapter concludes that the aim of the research was fulfilled. The next chapter concludes the primary research regarding the research contribution and summarises the research. This includes also the discussion of the limitations of research.

7. Conclusion

To conclude the thesis, this chapter discusses the contribution made by the research. Therefore, it starts with a summary of the research and its achievements. After the discussion of the contribution it also considers the limitations and possible directions of further research based on the results of this study.

7.1. Summary and achievements of the research

In Chapter 2, the literature was discussed showing the following picture in software unit reuse:

The increasing need of knowledge for software unit reuse is a challenge for software engineers.

The manifold environments where software development projects are used are a typical reason for different and specialised knowledge. Typically, this knowledge relates to a software unit itself and the activities a software engineer wants to perform a reuse process. Often, problems occur because of missing or inadequate knowledge level of software engineers. The impact created by missing or insufficient knowledge differs. With a simple increase of time or cost, a project may fail.

To mitigate or even eliminate these problems, the research project of the thesis aimed to develop a concept focusing on the execution of software construction activities (i.e., transformation, integration, and deployment) without a sufficient amount of knowledge. Being able to perform such activities without the specialised knowledge enables software engineers to fulfil tasks (i.e., reuse activities) that, usually, require an investment of time in learning.

In order to identify the aforementioned challenges, the literature was used to demonstrate the problem of missing knowledge in Chapter 2. In the literature, a lack of techniques to store and distribute reuse activities and relevant knowledge among software engineers was identified. A more detailed analysis shows four related problem areas (cf. Section 2.2.3):

- 1) Insufficient knowledge level of software engineers.
- 2) A variant of existing technologies and its related reuse activity knowledge.
- 3) Knowledge required for the distribution environments.

Conclusion

4) Missing definition of reuse activity knowledge

These problem areas are based on knowledge problems. They include challenges for creating an adequate technique to handle the identified lack in supporting inexperienced users in performing software unit reuse activities.

Chapter 3 analyses the focused software construction activities (i.e., integration, transformation, and deployment) and the knowledge problem identified by the literature review in more detail. Thereby, knowledge problem areas (knowledge storing, searching/retrieving, learning, distribution, and execution) of software construction activities are discussed and related to the three problem areas. Chapter 3 discusses that these knowledge problems occur in the three different problem areas. Additionally, existing approaches are discussed. Chapter 3 concludes that these do not solve the identified problem areas and, therefore, support the statement about a missing adequate technique to store and subsequently distribute software construction activity relevant knowledge among software engineers as identified in the literature review.

The approach this thesis is focusing on is introduced in Chapter 4. The idea of this approach is the storage and execution of software construction activity knowledge. The aim is to support users who do not have enough knowledge to perform a specific reuse activity on a specific software unit.

The approach describes two elementary parts of the focused solution. The first one is a common Software Unit Model describing different existing software unit concepts (i.e., classes, components, and services) as a general software unit. This model is extended by different software construction activity models, describing the information required by using the information from the Software Unit Model in a specific software construction activity. The second one is a service-oriented environment providing a service for reuse functionality (i.e., storing, distribution and execution of software construction activity knowledge) based on the information in the given models.

Conclusion

The service provides functionality to users for storing software construction activity information which can be used as knowledge. Thereby, different models are used. This includes the storage of different software units and relevant software construction activity knowledge.

The abstract models used for software units and software construction activities should handle the problem of variations in technologies (solution approach for problem area 1: Insufficient knowledge level of software engineers). The service hides the environment for knowledge distribution. An inexperienced software engineer does not need to know this environment. Therefore, a limitation of necessary knowledge is expected by the use of the concept (solution approach for problem area 3: Knowledge required for the distribution environments).

The last solution approach is the service executing software construction activity. After an experienced software engineer entered the necessary information about software construction (e.g., a software unit and the necessary information about its integration into an IDE) into the service-oriented environment, an inexperienced user was able to perform it with less the necessary knowledge for execution. Using this approach the concept expects that inexperienced software engineers are able to perform a software construction activity independently of their current knowledge level (solution approach for problem area 2: A variant of existing technologies and its related reuse activity knowledge).

The three different software construction activity models (i.e., transformation, integration, and deployment SCAC models) constitutes a kind of reuse activity knowledge and, therefore, a solution approach for the problem area 4 (i.e., Missing definition of reuse activity knowledge).

To analyse the focused approach, the models and the service-oriented environment has been implemented as the essential elements of the approach and integrated into the working area of a global company producing software in different businesses. One possible realisation of the approach was described in Chapter 5. This realisation of the focused approach was used in Chapter 6 in a case study. Two of the three focused software construction activity types (i.e., transformation and integration) are tested in several reuse scenarios in this working area. A

Conclusion

total of 102 participants performed 120 software construction activities to reuse 6 different software units (i.e., classes, components and services) with two software construction activities for each software unit. Thereby, the activities were performed with and without the focused approach. The relevant values measured are: number of used knowledge resources, number of tasks done, time spent on knowledge resources and the time needed to perform the software construction activities.

Chapter 6 also analysed the results of the case study. The main result is that the focused approach enables inexperienced software engineers to perform software construction activities with a reduction in spent time and knowledge resources (compared to inexperienced software engineers not using the approach). Using the focused approach, inexperienced software engineers were able to perform unknown knowledge intensive software reuse activities to reuse different software unit technologies and its related knowledge. Combining the measured values of the transformation and integration software construction activities and using the experienced software engineer (i.e., an expert for a specific SCAC of a specific software unit) line for comparison led to following conclusions. Figure 146 shows the inexperienced users not using the focused approach as 100% (blue line).

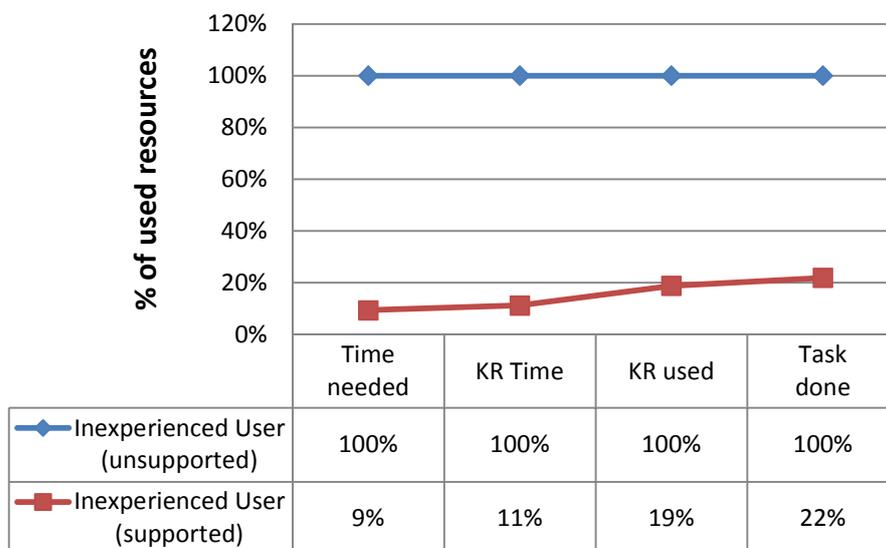


Figure 146 - Comparison between supported and unsupported software engineers

Conclusion

Comparing the the inexperienced engineers using the focused approach (red line) following statement can be made. The reduction of time, knowledge resources, and tasks for inexperienced software engineer are 91% of the activity time, 89% of time spent on knowledge resource, 81% of knowledge resources and 78% of tasks could be reduced (cf. Figure 146).

The case study also shows positive effects for experienced software engineer. But this effect is not as significant as it is for inexperienced engineers. The effects for experienced software engineers are identifiable but lower that the effects for inexperienced software engineers: On average 70% of time, 36% of time spent on knowledge resources, 17% of knowledge resources, and 66% task done are saved (cf. Figure 146).

The third focused software construction activity (i.e., deployment software construction activity) is discussed theoretically based on the measured values of the other types. Based on the similar model and behaviour of deployment and the other software construction activities, It is showed that a similar positive effect can be expected. Chapter 6 discussed that deployment software construction activities require manual steps performed by the user (in some cases; e.g., restart a device using the power switch). For such cases the focused approach has no positive effects.

Based on this case study result, Chapter 6 concluded that the principle aim of enabling inexperienced software engineers to perform software construction activities even if these people do not have the required knowledge is demonstrated by the case study.

7.2. Research contribution discussion

The literature shows that knowledge is relevant for software unit reuse. For example, Ajila (2005) and Cummings and Teng (2003) identify knowledge as a critical success factor in software unit reuse. Isoda (1992), Bughija (2001), and McCarey, Ó Cinnéide and Kushmerick (2008) state that tasks of reuse are based on knowledge.

From a scientific point of view the research result will contribute to the area of software unit reuse. This contribution will be discussed using the relevant statements identified in Section

Conclusion

2.2.2.3. McCarey, Ó Cinnéide and Kushmerick (2008) conclude that a lack of techniques to store and subsequently distribute relevant software construction activity software unit knowledge among software engineers exists. This research identifies four problem areas in the literature which make the creation of adequate techniques challenging (cf. Section 2.2.3.3). In the following section, the contribution created by this research for the four problem areas is discussed.

7.2.1. Contribution to the problem of different technologies

Frakes and Isoada (1994) state that reuse is difficult because of different technologies. The knowledge of reuse activities, which McCarey, Ó Cinnéide and Kushmerick (2008) call task relevant component knowledge is based on the technology of the software unit it is related to. Next to the multitude of existing technologies, Ajila and Zeng (2004) state that the rapid changes of technologies and required knowledge have to be maintained. As a result, the first challenge to limit the impact of the lack McCarey, Ó Cinnéide and Kushmerick (2008) focuses on, is to handle the problem of different technologies and the related knowledge.

Regarding Frakes and Isoada (1994) the difficulty of technologies can be reduced using an abstract way of storing and representing. This idea was used in the research by creating a set of models which were able to store different software unit technology information (i.e., object-oriented, component-based and service-oriented technology information) and software construction information (i.e., transformation, integration, and deployment activities). The models simplify the view on the different software unit technologies by using abstraction (e.g., classes, components and services presented as a simple set of files). This creates a common view on software units. The activity models describe the information of the activities and relate them to the simple software unit model (e.g., the file name as a parameter of a transformation activity). In the case study, different software units of different component worlds were also used (e.g., .NET and Java-based software units). Additionally, the models were able to store information which describes the use and configuration of other technologies. These

Conclusion

technologies are necessary to perform software construction activities. The created service-oriented environment was able to maintain this information and the related knowledge. By simplifying the view on different technologies (which changes rapidly; cf. Ajila and Zheng, 2004) one possibility of simplifying the maintenance of such technology was created. In the case study, the participant used different software unit technologies and technologies necessary for different software construction activities. The case study shows that independent of these different technologies, the inexperienced software engineers required less knowledge and spent less time on knowledge resources to perform each of the software construction activities. This research contributes to the field by providing a solution in reducing the use of different technology knowledge of software units and software construction activities by using abstraction models to unify the view on these technologies. Additionally, the research shows that an automation environment based on such models was able to maintain the variations of technologies.

7.2.2. Contribution to the problem of different knowledge levels

Among others, the literature review identifies the challenge in finding a way to distribute reuse activity knowledge based on the software engineers' knowledge level. Ye (2001) discusses the software engineers' knowledge related reuse types (i.e., well known, vaguely known, beliefs and unknown component). The analysis using the knowledge analysis of Zinn et al. (2011a) and Ye and Fischer (2005) show that software engineers can have different knowledge levels. The conclusion of Ye and Fischer (2005) that software engineers may be not able to perform reuse because of a lack of knowledge seems to be valid. Also, McCarey, Ó Cinnéide and Kushmerick (2008) state this.

On the one side, this knowledge is required to perform a software construction activity. The case study shows that inexperienced users spent the most time in consuming external knowledge resources (e.g., reading web pages or talking to experts). From the viewpoint of the research, this is a task of learning the required information (i.e., creating knowledge) for a

Conclusion

software construction activity. In this case the previously discussed problem of knowledge interpretation occurs. Qu, Ji and Nsakanda (2012) and Choi, Lee and Yoo (2010) identifies this especially in an environment where multiple teams are exchanging knowledge. This may lead to variations in the reuse activity result or to the failure of the reuse activity. Another point of interpretation is that software engineers use different ways of working to perform the same activity even if the underlying information is equal (Visser 1990; Sen 1997). In the case study, this is implied by the number of tasks the user performed. These numbers differ for each software engineer. This may be explained based on the different ways of working. Also it may be based on the knowledge software engineers already have to know to use a knowledge resource repository. (cf. discussion about knowledge distribution in distributed teams of Qu, Ji and Nsakanda 2012; Choi, Lee and Yoo 2010)

The research of this thesis identifies a concept that enables inexperienced software engineers to perform software construction activities even if the engineers do not have the required knowledge. The approach used in this research shows that an inexperienced software engineer (L4-unknown component; cf. Ye and Fischer, 2005) as well as an experienced software engineer (L1-well known component; cf. Ye and Fischer, 2005) can be supported in performing a software construction approach with reduced knowledge and reuse effort. As a result, the research shows that an inexperienced user's knowledge level using the focused approach is not a reason why reuse should fail. The automation approach shown in this thesis is one possible way to challenge the problem area. This is demonstrated by the case study.

The case study shows that inexperienced software engineers were able to produce invariant results by using the focused approach. Comparing this to the group of inexperienced software engineers, without the support of the focused approach, comparable learning activities are not identified in this group. This means, the inexperienced users spent significantly less time on knowledge resources. Additionally, they used fewer knowledge resources than the unsupported inexperienced software engineers. As a result, the research identifies a way to handle the

Conclusion

knowledge level problems. As discussed, the time spent by inexperienced software engineers to gain (i.e., learn and interpretation) required knowledge was decreased. It may be possible that the focused approach supports users in learning. But this was not the focus of the approach and is in fact improbable. The focused approach does not show explanatory information to users. Additionally, the time supported engineers require does not underline this statement. Figure 146 shows that the time and knowledge resources producing an similar and valid result is about 10 to 20% of the values of an unsupported software engineer. All supported software engineers produced the same valid results. This leads to the conclusion that the focused approach avoids the problem of knowledge learning and interpretation. Additionally, the produced results avoid lengthy learning processes and possible invariants. Regarding the problem of different knowledge levels, this approach identifies that an insufficient knowledge level does not produce invalid software construction activity results. Inexperienced software engineers are able to perform SCACs with and without the focused approach. But by using the approach long learning processes and possible invariants are avoidable.

7.2.3. Contribution to the problem of knowledge distribution

McCarey, Ó Cinnéide and Kushmerick (2008) state that the distribution of knowledge about technology between engineers and teams is inadequate. Next to the problem of interpretation and use of knowledge, an experienced software engineer has to distribute knowledge in a way that other engineers are able to understand (see Taweel et al., 2009; Boden and Avram 2009). This implicates an infrastructure which provides the functionality to upload activity information and knowledge. Additionally, it has to provide the possibility to find and access this infrastructure for searching and receiving uploaded knowledge. Frakes and Kang (2005), Ajila (2006), and Slyngstad et al. (2006) discuss the need of repositories, Usually, software engineers have repositories, but these are different in type and distribution. This can range from personal project files to a team or department repository. As a result, a software engineer has to know where to find a repository, how to access it and how to use it. The last point relates to the

Conclusion

previously mentioned problem of mind-set and capability of formulating a request. As a result, an inexperienced user has to know how to find and access this knowledge source or to know someone who can support him (Qu, Ji, and Nsakanda 2012). As shown by Ajila (2006), large companies are able to store knowledge for reuse but based on organisational problems the expected reuse is limited. To limit the lack described by McCarey, Ó Cinnéide and Kushmerick (2008) one challenge is to create such an infrastructure. Visser (1990) and Morad and Kuflik (2005) stated the use of special teams or single experienced users for a single software unit as support for other software engineers or development teams in bigger companies. Ha, Sun and Xie (2012) and Thörn (2010) also mentioned that this is not usually possible in SMEs.

The research used a service-oriented environment as infrastructure to integrate existing repositories and execute software construction activities. The inexperienced software engineer may be aware about this infrastructure but does not use the different repositories and relevant activity tools directly. The service provided by this environment creates an abstraction layer to the technical environment and possible SCAC applications. The inexperienced user does not need to know the structure of the environment (e.g., installation folder) or special behaviours (e.g., for security or special configurations) of elements in this environment. An inexperienced user does not need to learn such knowledge. This simplifies the knowledge for distribution. The realised environment extends the distribution of knowledge by the feature of knowledge execution. Chapter 3 discusses the knowledge problems related to the problem of knowledge distribution. Identified problems are the search, access and use of knowledge. This research contributes to the field by creating a solution for the discussed problems. The case study shows that inexperienced software engineers are able to perform software construction activities. This includes the search of software unit information, SCAC information, access of the different repositories, download of all information, preparation of the SCAC and performing the SCAC. In the case study the participants were informed that inexperienced users have less knowledge on technology, software units and SCACs. They performed the knowledge intensive SCACs by

Conclusion

activating the button. The inexperienced software engineers did not recognise all sub tasks and had no idea where all the necessary information was stored or where the SCAC was actually performed. However, the inexperienced engineer was able to execute the SCAC knowledge which was stored by experienced engineers within the environment. A result of the study was that supported engineers perform fewer tasks than the unsupported engineers when performing an SCAC. This leads to the conclusion that knowledge can be shared and executed between software engineers without the specific knowledge that is necessary for the distribution, the setup and execution of an SCAC. In short, this knowledge can be used without an adequate knowledge level.

7.2.4. Contribution to definition of software reuse knowledge

The last challenge discussed in this section is the definition of software reuse activity knowledge. In the focused problem statement, task relevant component knowledge should be exchanged between software engineers (see McCarey, Ó Cinnéide and Kushmerick, 2008). However, a definition of this knowledge was not identified in the used literature. Based on the amount of possible knowledge, for an example, based on the technology variations this seems to be challenging. On the other side, activities are recognised by the literature as typical activities of reuse. Bosch and Bosch-Sijtsema (2010) and Shiva and Shala (2007), for example, indicate the need for integrating a reusable software unit into the development environment. Also, for Vliet (2008) and Mens and Vangorp (2006) it is necessary to adapt an existing unit before reuse. Especially in the area of embedded devices (see Carlson et al., 2010; O'Connor et al., 2009) the deployment is an relevant part and often depends on previous created software units.

This research does not contribute to a reuse knowledge definition with a new type of reuse definition. It only demonstrates that a more abstract view is sufficient to enable an environment to store different software construction activity information which can be used as knowledge to

Conclusion

perform software construction activities. This supports inexperienced software engineers in their work. A concrete knowledge definition is not created by the research. A concrete definition might be contrary if it limits the number of storable software construction activities.

7.2.5. Final statement

As a contribution this research describes a concept including different models and a service environment which enables the execution of software construction activities by inexperienced software engineers. Thereby, the contribution to the area of software unit reuse is the identification of a realisable concept to store and subsequently distribute reuse activities (i.e., software construction activities) relevant knowledge among software engineers. The principle aim means to enable inexperienced software engineers to perform software construction activities with less effort of learning (i.e., handling knowledge resources) has been reached.

As a result, this research underline that the low knowledge level of a software engineer, the changing field of required knowledge for technologies and the knowledge required for distribution do represent challenges in creating a technique to support software engineers in software unit reuse. However, the research also shows that in the focused on software construction activities these challenges are manageable.

7.3. Objectives and limitations of research

The objectives of the research have been reached. The first objective was the analysis of existing problems. Here, the lack of techniques and problem areas representing challenges to create such techniques was identified (cf. Section 2.2.3.3). The second and third objective was fulfilled by describing the necessary software unit models and the software construction activity models (cf. Section 4.5). In Chapter 5 the realisation of the concept was done. By realising and combining the different models described in Chapter 4, the fourth objective was fulfilled. As discussed, the case study shows that the concept can be used and supports inexperienced software engineers. Therefore, the fifth objective is fulfilled.

Conclusion

Despite having met the objectives of the research project, some decisions had to be taken which resulted in imposed limitations. The decisions were caused by practical reasons, or to limit the effort spent in areas where no new insights could be expected. These limitations are summarised as follows.

The first limitation is the focus on only three types of software construction activities with a special sub focus for each type. The aim of the research was to demonstrate that inexperienced software engineers could be enabled to perform software construction activities related to software unit reuse. To reach this aim the chosen types and foci were capable to the Ph.D. research. Additionally, with this restriction the effort of software development area was limited to only these three types of SCAC. The focus of the research was not the development of a new software system to prove any existing reuse activity. This limits the research for general statements about all possible software construction activities including all sub foci.

The focus on three different types of software units is a limitation of the research. Also, other software units and related technologies exist. As a result, a general statement that the approach of this thesis focuses on all existing types of software units or its technologies cannot be made.

The next limitation to discuss is the number of participants (102). The software engineers participated as volunteers during normal working hours. Even the time for one measurement task was limited by using smaller software units and the participants had to look at how they make up the missing hours for their normal work. In some cases the related team or project leader accepted this. But sometimes this was not supported by the management. Such reasons limit the number of volunteers. A higher number of participants would lead to a result that can be used to make more general statements about the software construction activities focused by the research.

The research about deployment software construction activities is limited to a discussion even though a small case study was shown by Zinn, Fischer-Hellmann and Schoop (2012a). This limitation is mainly related to the fact that the working rules of Schneider Electric allow the

Conclusion

work on low, medium or high voltage devices only with special training. This is also valid for the first aid personnel. Additionally, experiments with such devices are only allowed in special laboratories that have the access restriction to trained personnel. The author of this thesis is not an electronic engineer or a first aid supporter for this specialised field. This research's previous study about device deployment of (Zinn, Fischer-Hellmann, and Schoop 2012a) was done with trained personal and first aid support who was specialised on accidents in electronic environments. This precludes the result of the primary research focusing on deployment from being proven as whole. Only two of three software construction activities are proven by a case study. The deployment SCAc was discussed using the results of the other SCAcs.

Chapter 3 discusses different problems of software unit reuse. Here, only problems are chosen which are seen as relevant from the literature point of view. Also the capability of the author of this thesis to create an added value for the chosen problems was a relevant requirement. Also other problems may exist. In the scope of the research, the significance of the chosen problems was demonstrated. Appendix Section G shows a discussion including additional problems identified by the author. This discussion was created as part of the Ph.D. research.

Despite these limitations, the research project has led to valid contributions to knowledge and provided sufficient proof of the concept for the approach proposed.

7.4. Future work

This section defines possible future work and research based on this thesis. In general, two basic directions are interesting from the perspective of the author. The first one is the neutralisation of the limitations of this research. The second direction is further research based on the achieved results.

7.4.1. Neutralisation of the limitations

The limitations should be removed by further research. The first one is the limitation created by the number of participants of the case study. To demonstrate the mode of operation of the

Conclusion

focused approach, the used number of participants and software construction activities was useful. By increasing the numbers, the statement of this thesis can be used as a more general statement. This may lead to new findings for the topic.

The next limitation is the theoretical evaluation of the deployment software construction activities. This thesis discusses the possibility of performing such software construction activities. The experiment shown by Zinn, Fischer-Hellmann and Schoop (2012a) also demonstrates the feasibility. Further research could focus on case studies for deployment SCAs to identify practical measured results. In particular, the fact that manual steps may be necessary in some SCAs. Here, an automation concept may create an added value.

The last limitation seen as relevant is the number of different software construction activity types and the focus of each type. The research used integration, transformation, and deployment of software construction activities. As discussed in the previous section these activities are focused on special fields (i.e., console-based transformation activities, IDE integration activities and device deployment). Further research may focus on other special or more general fields inside these types (e.g., UI based transformation, other IDEs for integration and deployment not only focused on devices). Additionally, further research may include more than the three focused SCAs (i.e., transformation, integration, and deployment) and cover other reuse areas (e.g., validation or testing). This may lead to new findings for the topic.

7.4.2. Extended research

The first example of possible extended research is the use of stored software construction activities in case-based reasoning. Case-based reasoning is using existing information about cases to identify new information and circumstances in other cases. In the case of software construction activities, stored information can be (re)used to identify new relations between information. An example is the settings of software construction activities based on existing software units which are adapted automatically to other software units, even though this unit is

Conclusion

not stored in an environment. Zinn, Fischer-Hellmann and Schopp (2012b) discuss some case-based reasoning scenarios based on the focused approach.

The thesis shows the impact of reuse related to cost in Section 2.2.1.5. A discussion about the advantages or disadvantages of SCAs for software unit reuse relating to costs is not part of this thesis. The result of the case study shows that time and use of necessary knowledge is saved by using the focused approach. Even if some studies (cf. Section 2.2.1.5) identify a saving of costs as a result of reuse, this does not automatically demonstrate that costs are saved by the approach of this research. Further research in this area may focus on the cost behaviour of the (re)use of software construction activities and the approach demonstrated in this thesis.

Another extension may improve the research on software construction activities. The focused approach used plugins to perform a software construction activity. These plugins contain knowledge (i.e., coded rules) used for interpretation of the models used in the approach. An aim should be to add this knowledge to the software construction activity models and remove it from the plugins. This may result in more knowledge sensitive models and generalised plugin syntax (i.e., more relation between information stored in the knowledge model as in the source code of the plugins). This could also impact the case-based reasoning (more knowledge can be used for reasoning).

Another part in future research which was not discussed in the thesis was software construction activities and the combination of such activities. A method to combine instances of different software construction activity models may reduce the effort for performing reuse. Related to the research of this thesis, the combination of existing software construction activity models may create an added value. A transformation activity, for example, could be followed by an integration activity. A single ‘transform and integrate’ activity would reduce the number of interactions with such a system. The abstraction of different software unit types simplifies the reuse of these software units (as shown in the research by the use of services, components and classes). Maybe an abstraction of different software construction activity models which show a

Conclusion

similarity would also result in a simplification of reuse. The deployment activity model, for example, has a common structure to the transformation activity model. Perhaps it is possible to create a common model for transformation and deployment by the use of meta models. Also it could be interesting to create a common model with special transformation and deployment extensions which may reduce redundancy.

In a previous publication, the concept of software construction artefacts and their types is described (cf. Section 4.4.2). The idea was to find a classification for the content of a software unit and, therefore, relate it to SCAC. As a result, it may be possible to develop search behaviour for software construction activities.

The last extension of the research seen as an interesting research topic is the use of semantic models. It was shown that semantics can be used to extend description. For example, Seedorf, (2010) uses such models to describe software assets from business perspective for search behaviours. In the future, semantic models may support the search of software construction activities and the creation of such activities.

7.5. Technology review and epilogue

The personal opinion of the author is that reuse of software units is still, and will be for a longer time, a relevant topic for software development; but the meaning could change. Today, technology dependencies are relevant by reuse of software units. This work also focuses on this and shows that this is a potential problem, but a trend is noticeable. Regarding the mobile device platforms Android and Internet Operation System, developing reusable software units requires specialist knowledge about the different platforms (i.e., technology and IDEs). New technologies allow for creating a software unit and deploying it to the different platforms. The same behaviour can be found in the game industry. Tools like Unity3D allows one to write software units and deploy them to different gaming platforms and the research shown in this thesis follows this trend. A result is that a software unit will be created which solves a problem (domain view). But different deployment activities are necessary (software construction activity

Conclusion

view) to deploy it to different platforms. The author's opinion is that software construction activity will become more relevant in the future and the need for automation concepts for such activities will grow. However, this may be could have a shady side. While the need of domain experts (i.e., software engineers producing the domain related content) may be decreasing, the need for software unit construction activities experts (i.e., for transformation or deployment) may increase rapidly.

As final consideration a loop back is made to the greek mythology this thesis starts with. *"But the noble son of lapetus outwitted him and stole the far-seen gleam of unwearying fire in a hollow fennel stalk. And Zeus who thunders in high was stun in spirit, and his dear heart was angered when he saw amongst men the far-ssen ray of fire."* (Hesoid and Evelyn-White, 1914 p. 545)

Regarding this part of the theogony another future dilemma can be identified. Prometheus handed over the fire again and, as a result, mankind was able to perform the relevant activities again. In Greek mythology the sustainability is given by Prometheus. As long as Prometheus can bring back the fire every time it gets lost, mankind can continue. Regarding the focused approach this is similar. As long as such automation approaches are available, software engineers can reuse software units with less experience every time they need it. If such approaches are not available, the problems focused by this thesis are not handled. The Chinese philosopher Confucius supposedly said *"Give a man a fish, feed him for a day. Teach a man to fish, feed for a lifetime."* Regarding the research of this thesis, software engineers have to decide whether they should use such an approach or instead learn specific knowledge to have it more or less sustainable. This research shows a way to enable short-term reuse of specific software construction activities with less investment in learning and with less the risk of failing an activity. In future, knowledge for reuse activities has to be sustainable (long-term view) for each software engineer.

8. References

- Ackoff, R. L. (1989) 'From Data to Wisdom'. *Journal of Applied Systems Analysis*, 16, pp. 3-9.
- Ajila, S. A. (2006) 'The Impact of Firm Size on Knowledge Reuse and Exploration During Software Product Development: An Empirical Study', *Information Reuse and Integration, 2006 IEEE International Conference on*, Waikoloa, Hawaii, 16-18. Sept. 2006. Institute of Electrical and Electronics Engineers (IEEE), pp.:160-165. DOI:10.1109/IRI.2006.252406.
- Ajila, S. A. (2005) 'Reusing Base-product Features to Develop Product Line Architecture', *Information Reuse and Integration, 2005 IEEE International Conference on, Las Vegas, USA*, 15-17 Aug. 2005. Institute of Electrical and Electronics Engineers (IEEE), pp. 288-293. DOI:10.1109/IRI-05.2005.1506488.
- Ajila, S. A. and Zheng, S. (2004) 'Knowledge Management: Impact of Knowledge Delivery Factors on Software Product Development Efficiency', *Information Reuse and Integration, 2004 IEEE International Conference on, Las Vegas, USA*, 8-10 Nov. 2004. Institute of Electrical and Electronics Engineers (IEEE), pp. 320-325. DOI:10.1109/IRI.2004.1431481.
- Alferez, G. H. and Pelechano, V. (2011) 'Systematic Reuse of Web Services Through Software Product Line Engineering', Lugano, Switzerland, 14-16 Sept. 2011. Institute of Electrical and Electronics Engineers (IEEE), pp. 192-199. DOI:10.1109/ECOWS.2011.13.
- Allen, B. P. (1994) 'Case-based Reasoning: Business Applications'. *Communications of the ACM*, 37 (3; March 1), pp. 40-42. DOI:10.1145/175247.175250.
- Allen, P. and Frost, S. (1998) *Component-Based Development for Enterprise Systems Applying the Select Perspective*. Cambridge, UK: Cambridge University Press.
- Almeida, E. S. de, Alvaro, A., Lucredio, D., Garcia, V. C., Lemos Meira, S. R. de (2005) 'A survey on software reuse processes', *Information Reuse and Integration, 2005 IEEE International Conference on, Las Vegas, USA*, 15-17 Aug. 2005. Institute of Electrical and Electronics Engineers (IEEE), pp. 66-71. DOI: 10.1109/IRI-05.2005.1506451
- Altera (2012) Altera. Available at: <http://www.altera.com/products/devices/dev-index.jsp>. (Accessed Oct. 2012).
- Ambler, S. W. (2003) UML 2 Component Diagrams. Available at: <http://www.agilemodeling.com/artifacts/componentDiagram.htm> (Accessed 2008).
- Ampatzoglou, A., Kritikos, A., Kakarontzas, G. and Stamelos, I. (2011) 'An Empirical Investigation on the Reusability of Design Patterns and Software Packages'. *Journal of*

References

- Systems and Software*, 84 (12; December), pp. 2265–2283.
DOI:10.1016/j.jss.2011.06.047.
- Apperly, H. (2003) *Service- and Component-based Development Using Select Perspective and UML*. 1st edition. London, UK: Addison-Wesley. ISBN: 0321159853
- Arkin, A. et al. (2007) Web Services Business Process Execution Web Services Business Process Execution Language Version 2.0. Available at: <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf> (Accessed 12 Jan. 2008)
- Atkinson, C. and Muthig, D. (2002) 'Component-Based Product-Line Engineering with the UML', Proceedings of the 7th International Conference on Software Reuse (ICSR): Methods, Techniques, and Tools. Austin, USA, 15-19 April 2002. London, UK: Springer Verlag, pp. 343-344.
- Bagnasco, A., Chirico, M., Scapolla, A. M. and Amodei, E. (2001) 'Improving Automation and Reuse in TLC Testing Through COTS-based Architecture'. *Aerospace and Electronic Systems Magazine*, 17 (3), pp. 17-21. DOI:10.1109/AUTEST.2001.948968.
- Baudoin, C. and Hollowell, G. (1996) *Realizing the object-oriented lifecycle*. 1st edition. Upper Saddle River, USA: Prentice Hall PTR. ISBN:013124454X
- Bauer, B. and Huget, M.-P. (2005) 'Modelling web service composition with UML 2.0'. *International Journal of Web Engineering and Technology*, 1 (4; February.): pp. 484-501. DOI: 10.1504/IJWET.2004.006272
- Baxter, P. and Jack, S. (2008), Qualitative case study methodology: Study design and implementation for novice researchers. *The Qualitative Report*, 13(4), 544-559
- Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T. and DeBaud, J.-M. (1999) 'PuLSE', *SSR '99 Proceedings of the 1999 symposium on Software reusability*, Los Angeles, USA, 21-23 May 1999. New York, USA: ACM Press., pp. 122–131. DOI:10.1145/303008.303063.
- Bellinger, G., Castro, D. and Mills, A. (2004) Data, Information, Knowledge, and Wisdom. Available at: <http://www.systems-thinking.org/dikw/dikw.htm> (Accessed June 2007).
- Benatallah, B., R. M. Dijkman, M. Dumas, and Z. Maamar. (2004) 'Service Composition: Concepts, Techniques, Tools and Trends'. In Stojanovic, Z. and Dahanayake, A eds. *Service oriented Software System Engineering: Challenges and Practises*. London, GB: Idea Group Publishing, pp. 48-67. ISBN 1591404266
- Bieber, G. and Carpenter, J. (2001) Introduction to Service-Oriented Programming (Rev 2.1). Available at: <http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf> (Accessed 18 Feb. 2007).

References

- Breivold, H. P. and Larsson, M. (2007) 'Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles', *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*. Lübeck, Germany, 28-31 Aug. 2007. Sankt Augustin, Germany: Institute of Electrical and Electronics Engineers (IEEE), pp. 13-20. DOI:10.1109/EUROMICRO.2007.25
- Bjørnson, F. O. and Dingsøy, T. (2008) 'Knowledge Management in Software Engineering: A Systematic Review of Studied Concepts, Findings and Research Methods Used'. *Information and Software Technology*, 50 (11; October), pp. 1055–1068. DOI:10.1016/j.infsof.2008.03.006.
- Blok, M.C. and Cybulski, J. L. (1998) 'Reusing UML Specifications in a Constrained Application Domain', *Software Engineering Conference, 1998. Proceedings*. Taipei, Taiwan, 2-4 Dec. 1998. Washington, USA: Institute of Electrical and Electronics Engineers (IEEE), pp. 196–202. DOI:10.1109/APSEC.1998.733719.
- Bobillo, F., Delgado, M. and Gómez-Romero, J. (2008) 'Representation of Context-dependant Knowledge in Ontologies: A Model and an Application'. *Expert Systems with Applications*, 35 (4; November), pp. 1899–1908. DOI:10.1016/j.eswa.2007.08.090.
- Böckle, G., Pohl, K. and Van der Linden, F. (2005) *Software product line engineering: foundations, principles, and techniques*. Berlin [u.a.], Germany: Springer. ISBN: 3540243720
- Boddy, D., Boonstra, A. and Kennedy, G. (2004) *Managing information systems: an organisational perspective*. 2nd edition. Harlow, UK and New York, USA: Financial Times and Prentice Hall. ISBN: 0273686356
- Boden, A. and Avram, G. (2009) 'Bridging Knowledge Distribution - The Role of Knowledge Brokers in Distributed Software Development Teams' *Cooperative and Human Aspects on Software Engineering, 2009. CHASE '09. ICSE Workshop on*, Vancouver, Canada, 17 May 2009. Washington, USA: IEEE Computer Society, pp. 8–11. DOI:10.1109/CHASE.2009.5071402.
- Boh, W. F. (2008) 'Reuse of Knowledge Assets from Repositories: A Mixed Methods Study'. *Information & Management*, 45 (6; September), pp. 365–375. DOI:10.1016/j.im.2008.06.001.
- Bohn, H., Bobek, A. and Golatowski, F. (2006) 'SIRENA - Service Infrastructure for Real-time Embedded Networked Devices: A Service Oriented Framework for Different Domains', *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies*. Morne, Mauritius, 23-29 Apr. 2006. Institute of Electrical and Electronics Engineers (IEEE), pp. 43–43. DOI:10.1109/ICNICONSMCL.2006.196.
- Bosch, J. (2004) *Software Architecture: The Next Step .3047*. Lecture Notes in Computer Science. Available at:

References

- <http://www.cs.rug.nl/search/uploads/Publications/bosch2004san.pdf> (Accessed Dez. 2007)
- Bosch, J. and Bosch-Sijtsema, P. (2010) 'From Integration to Composition: On the Impact of Software Product Lines, Global Development and Ecosystems'. *Journal of Systems and Software*, 83 (1; January), pp. 67–76. DOI:10.1016/j.jss.2009.06.051.
- Breivold, H. P. and Larsson, M. (2007) 'Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles', *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*. Lübeck, Germany, 28-31 Aug. 2007. Sankt Augustin, Germany: Institute of Electrical and Electronics Engineers (IEEE), pp. 13–20. DOI:10.1109/EUROMICRO.2007.25.
- Budhija N. and Ahuja S. P. (2011) 'Review of Software Reusability' Computer Science and Information Technology (ICCSIT) International Conference on. Pattaya, Thailand, Dec. 2011, Planetary Scientific Research Center (PSRC)
- Bunse, C. and Knethen, A. v. (2008) *Vorgehensmodelle kompakt*. 2nd Edition. Heidelberg, Germany: Spektrum. ISBN: 3827419506
- Burg, S. Van der, de Jonge, M., Dolstra, E. and Visser, E. (2009) 'Software Deployment in a Dynamic Cloud: From Device to Service Orientation in a Hospital Environment', *Software Engineering Challenges of Cloud Computing (ICSE): Proceedings of the 2009 ICSE Workshop on*. Vancouver, Canada, 23 May 2009. Washington, DC, USA: IEEE Computer Society, pp. 61–66. DOI:10.1109/CLOUD.2009.5071534.
- Busi, N., Gorrieri, R., Guidi, C., Lucchi, R. and Zavattaro, G. (2005) 'Towards a formal framework for Choreography', *Enabling Technologies: Infrastructure for Collaborative Enterprise, 2005. 14th IEEE International Workshops on (WETICE)*. Linköping, Sweden, 13-15 June 2005. Washington, DC: Institute of Electrical and Electronics Engineers (IEEE), pp. 107-112. DOI: 10.1109/WETICE.2005.57
- Card, D. and Comer, E. (1994) 'Why Do so Many Reuse Programs Fail?'. *IEEE Software*, 11 (5; September), pp. 114–115. DOI:10.1109/52.311078.
- Carlson, J., Feljan, J., Maki-Turja, J. and Sjodin, M. (2010) 'Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems', *Software Engineering and Advanced Applications (SEAA), 36th EUROMICRO Conference on*. Lille, France, 1-3 Sept. 2010. IEEE Conference Publications, pp. 74–82. Doi:10.1109/SEAA.2010.43.
- Casati, F. and Shan, M. C. (2001)'Dynamic and adaptive composition of e-services', *Advanced information systems engineering The 12th international conference on*. Stockholm, Sweden, May 2001. Oxford UK: Elsevier Science Ltd., pp. 143-163. DOI:10.1016/S0306-4379(01)00014-X.

References

- Chaffey, D. and Wood, S. (2005) *Business information management: improving performance using information systems*. 2nd edition. Harlow, UK and New York, USA: Prentice Hall and Financial Times. ISBN: 0273711792
- Cheesman, J. and Daniels, I. (2000) *UML Components: A Simple Process for Specifying Component-Software*. Boston. 1st edition. Amsterdam, Netherlands: Addison-Wesley, ISBN: 0201708515.
- Childs, B. and Sametinger, J. (2012) 'Literate Programming and Documentation Reuse', *Software Reuse, 1996, Proceedings Fourth International Conference on*. Orlando, Florida: IEEE Comput. Soc. Press., pp. 205–214. DOI:10.1109/ICSR.1996.496128.
- Choi, S. Y., Lee, H. and Yoo, Y. (2010) 'The Impact of Information Technology and Transactive Memory Systems on Knowledge Sharing, Application, and Team Performance: A Field Study'. *Management Information Systems Quarterly*, 34 (4), pp. 855-870.
- Clark, J., et al. (2001) ebXML Business Process Specification Schema Version 1.01. Specification, OASIS. Available at: www.ebxml.org/specs/ebBPSS.pdf (Accessed Dez. 2008)
- Clarke, R. J. (2005) *Research Models and Methodologies Presentation*. Available at: <http://www.uow.edu.au/content/groups/public/@web/@commerce/documents/doc/uow012042.pdf>. (Accessed Feb 2010)
- Componentsource. (2012) Component Source Web page. Available at: <http://www.componentsource.com/features/index-de.html> (Accessed Oct. 2010)
- Computerbase (2008) .NET. Available at: <http://www.computerbase.de/lexikon/.NET> (Accessed Oct. 2008)
- Cooper, H.M. (1988) 'Organizing Knowledge Syntheses: a Taxonomy of Literature Reviews'. *Knowledge in Society*, 1, pp. 104-126. DOI: 10.1007/BF03177550
- Cooper, H. M. (1982) 'Scientific Guidelines for Conducting Integrative Research Reviews'. *Review of Educational Research*, 52 (2), pp. 291-302. DOI:10.3102/00346543052002291
- Creswell, J. W. (2009) *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. 3rd edition. Thousand Oaks, USA: Sage Publications. ISBN: 1412965578
- Cummings, J. L. and Teng, B. S. (2003) 'Transferring R&D Knowledge: The Key Factors Affecting Knowledge Transfer Success'. *Journal of Engineering and Technology Management*, 20 (1-2; June), pp. 39–68. DOI:10.1016/S0923-4748(03)00004-3.
- Czarnecki, K. and Eisenecker, U. (2000) *Generative Programming: Methods, Tools, and Applications*. 6th edition. Indianapolis, USA: Addison Wesley. ISBN: 0-201-30977-7

References

- Dahanayake, A., Sol, H. and Stojanović, Z. (2003) 'Framework for Component-Based System Development Methodology Evaluation', *Journal of Database Management (JDM)*, 14 (1; March): pp. 1-26.
- Davenport, T. (2000) *Working Knowledge: How Organizations Manage What They Know*. 2nd edition. Boston, USA: Harvard Business School Press. ISBN: 1578513014
- Deming, W. E. (2000) *Out of the crisis*. Cambridge, USA: MIT Press. ISBN: 0262541157
- Desouza, K., Awazu, Y. and Baloh, P. (2006) 'Managing Knowledge in Global Software Development Efforts: Issues and Practices'. *Software, IEEE*, 23 (5; Sept.-Oct.), pp. 30-37, Doi:10.1109/MS.2006.135
- Dikel, D., Kane, D., Ornburn, S., Loftus, W. and Wilson, J. (1997) 'Applying Software Product-line Architecture'. *Computer*, 30 (8; August), pp. 49-55. DOI:10.1109/2.607064.
- D'Souza, D. F. and Wills, A. C. (1998) *Objects, Components and Frameworks with UML: The Catalys Approach*. Reading, USA: Addison-Wesley. ISBN: 0201310120
- Eclipse Foundation. (2012) Eclipse Resources. Available at: <http://www.eclipse.org/resources/?category=Extension%20points> (Accessed Jan 2010).
- Edward, A., Ali, M. and Sherif, Y. (1999) 'A Case Study in Software Reuse'. *Software Quality Journal*, 8 (3; November), pp. 169-195. DOI:10.1023/A:1008963424886
- Fayad, M. E. and Johnson, R. E. (2000) *Domain-specific application frameworks: frameworks experience by industry*. New York, USA: John Wiley & Sons. ISBN:0-471-33280-1
- Fayad, M. E., Laitinen, M. and Ward, R. P. (2000) 'Thinking Objectively: Software Engineering in the Small'. *Communications of the ACM*, 43 (3; March 1), pp. 115-118. DOI:10.1145/330534.330555.
- Fettke, P., Intorsureanu, I. and Loos, P. (2002) 'Komponentenorientierte Vorgehensmodelle im Vergleich', 4 *Workshop komponentenorientierte betriebliche Anwendungssysteme (WKBA)*. Augsburg, Germany 11. Juni 2002. Augsburg, Germany: TU Chemnitz, pp. 19-43.
- Fichman, R. G. and Kemerer, C. F. (2001) 'Incentive Compatibility and Systematic Software Reuse'. *Journal of Systems and Software*, 57 (1; April), pp. 45-60. DOI:10.1016/S0164-1212(00)00116-3.
- Fitzgerald, B., et al. (2006) 'The Software and Services Challenge' *Contribution to the preparation of the Technology Pillar on Software, Grids, Security and Dependability of the 7th Framework Programme*. Available at: ftp://ftp.cordis.europa.eu/pub/ist/docs/directorate_d/st-ds/fp7-report_en.pdf, (Accessed Aug. 2007)

References

- Frakes, W. B. and Fox, C. J. (1996) 'Quality Improvement Using a Software Reuse Failure Modes Model'. *IEEE Transactions on Software Engineering*, 22 (4; April), pp. 274–279. DOI:10.1109/32.491652.
- Frakes, W. B. and Isoda, S. (1994) 'Success Factors of Systematic Reuse'. *IEEE Software*, 11 (5; September), pp. 14–19. DOI:10.1109/52.311045.
- Frakes, W. B. and Kang, K. (2005) 'Software Reuse Research: Status and Future'. *IEEE Transactions on Software Engineering*, 31 (7; July), pp. 529–536. DOI:10.1109/TSE.2005.85.
- Frijters, J. (2011) IKVM.NET Home Page. Available at: <http://www.ikvm.net/> (Accessed Oct. 2009)
- FSU. (2007) gSOAP: SOAP C++ Web Services. Available at: <http://www.cs.fsu.edu/~engelen/soap.html> (Accessed Oct. 2009).
- Gamma, E. (1995) *Design patterns: elements of reusable object-oriented software*. 1st edition. Reading, USA: Addison-Wesley. ISBN: 0201633612
- Garcia, V. C., de Almeida, E. S., Lisboa, L. B., Martins, A. C., Meira, S. R. L., Lucredio, D. and de M. Fortes, R. P. (2006) 'Toward a Code Search Engine Based on the State-of-Art and Practice', *Software Engineering Conference 13th Asia Pacific (APSEC)*. Kanpur, India, 6-8 Dec. 2006. IEEE Conference Publications, pp. 61–70. DOI:10.1109/APSEC.2006.57.
- Garlan, D., Allen, R. and Ockerbloom, J. (2009) 'Architectural Mismatch: Why Reuse Is Still So Hard'. *IEEE Software*, 26 (4; July), pp. 66–69. DOI:10.1109/MS.2009.86.
- GForgeGroup. (2012) GForge Group Collaborative Development Environment. Available at: <http://gforgegroup.com/> (Accessed Jan 2010).
- Ghezzi, C. (2005) 'Service based Computing: Where does it come from? A software engineering perspective' *Keynote address at the International Conference on Service based Computing (ICSOC)*. Amsterdam, Netherlands, 12-14 Dec. 2005 Springer
- Giambiagi, P., Owe, O., Ravn, A. P. and Schneider, G. (2006) 'Language-Based Support for Service based Architectures: Future Directions', *1st International Conference on Software and Data Technologies (ICSOFT 2006)*. Setúbal, Portugal, 11-16 Sept. 2006. Portugal: Springer, pp. 339-344.
- Gilart-Iglesias, V., Macia-Perez, F., Capella-D'alton, A. and Gil-marti'nez-abarca, J. (2006) 'Industrial Machines as a Service: A Model Based on Embedded Devices and Web Services', *Industrial Informatics, 2006 IEEE International Conference on*. Singapore, 16-18 Aug. 2006. Institute of Electrical and Electronics Engineers (IEEE), pp. 630–635. DOI:10.1109/INDIN.2006.275634.

References

- Gill, H. (2005) 'Challenges for Critical Embedded Systems', *Object-Oriented Real-Time Dependable Systems (WORDS) 10th IEEE International Workshop on*. Sedona, USA, 2-4 Feb. 2005. IEEE Conference Publications, pp. 7–12. DOI:10.1109/WORDS.2005.21.
- Gogniat, G., Wolf, T., Burlison, W., Diguët, J.-P., Bossuet, L. and Vaslin, R. (2008) 'Reconfigurable Hardware for High-Security/ High-Performance Embedded Systems: The SAFES Perspective'. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16 (2; February), pp. 144–155. DOI:10.1109/TVLSI.2007.912030.
- Grefen, P., Hoffner, Y., Ludwig, H., and Aberer, K. (2000) 'CrossFlow: Integrating Workflow Management and Electronic Commerce'. *ACM SIGecom Exchanges*, 2 (1), pp. 1-10.
- Gruhn, V. and Thiel, A. (2000) *Komponentenmodellen: DCOM, JavaBeans [sic], EnterpriseJavaBeans, CORBA*. München, Germany and Harlow, UK: Addison-Wesley. ISBN: 3-8273-1724-X / 382731724X
- Ha, W., Sun, H. and Xie, M. (2012) 'Reuse of Embedded Software in Small and Medium Enterprises', *Management of Innovation and Technology (ICMIT), 2012 IEEE International Conference on*. Bali, Indonesia, 11-13 June 2012. IEEE Conference Publications, pp. 394–399. DOI:10.1109/ICMIT.2012.6225838.
- Heckmann, B. (2007) 'Service provisioning in a utility computing environment', *Research Symposium on Security, E-learning, Internet and Networking*. Plymouth, UK, 14-15 June 2007. Plymouth, UK: Information Security and Network Research Group, pp. 185-198.
- Heineman, G. T. and Councill, W. T. (2001) *Component-Based Software Engineering: Putting the Pieces Together*. 1st edition. NHJ, USA: Addison-Wesley. ISBN: 076868207X
- Henry, E. and Faller, B. (1995) 'Large-scale Industrial Reuse to Reduce Cost and Cycle Time'. *IEEE Software*, 12 (5; September), pp. 47–53. DOI:10.1109/52.406756.
- Herzum, P. and Sims, O. (2000) *Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. 1st. edition. New York, USA: John Wiley & Sons. ISBN: 0471327603
- Hesoid and Evelyn-White, H. G. (1914) *The Theogony of Hesoid*. Kindle Edition. Focus Publishing
- Horeis, T. and Sick, B. (2007) 'Collaborative Knowledge Discovery & Data Mining: From Knowledge to Experience', *Computational Intelligence and Data Mining (CIDM) IEEE Symposium on*. Honolulu, Hawaii, USA, 1-5 April 2007. IEEE Conference Publications, pp. 421–428. DOI:10.1109/CIDM.2007.368905.
- Huang, H., Shi, Z.-z., Cheng, Y. and Qiu, L. (2005) 'Service-oriented Knowledge Management on Virtual Organizations', *Computer and Information Technology (CIT), The Fifth International Conference on*. Shanghai, China , 21-23 Sept. 2005. IEEE Conference Publications , pp. 1050–1054. DOI:10.1109/CIT.2005.167.

References

- Human Brain Project. (2011) Human Brain Project - Home. Available at: <http://www.humanbrainproject.eu/> (Accessed Jun. 2011).
- Hunt, A. and Thomas, D. (2004) 'Imaginate software construction'. *Software IEEE*, 21 (5; Sept.-Oct.), pp. 96-97. DOI: 10.1109/MS.2004.1331311
- Isoda, S. (1992) 'Experience Report on Software Reuse Project: Its Structure, Activities, and Statistical Results' *Proceedings of Software engineering (ICSE) the 14th international conference on*. Melbourne, Australia, 11-15 May 1992. New York, USA: ACM, pp. 320-326. DOI:10.1109/ICSE.1992.753509.
- Isoda, S. (1991) 'An experience of software reuse activities', *Computer Software and Applications Conference (COMPSAC), Proceedings of the Fifteenth Annual International*. Tokyo, Japan, 11-13 Sep 1991. IEEE Conference Publications, pp.8-9. DOI:10.1109/COMPSAC.1991.170144.
- Jacobson, I., Booch, G. and Rumbaugh, J. (1999) *The Unified Software Development Process*. Sebastopol. Addison-Wesley. ISBN: 0201571692
- Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. (1992) *Object-Oriented Software Engineering -A Use Case-driven Approach*. 4th edition. Addison-Wesley. ISBN: 0201544350.
- Jacobson, I., Griss, M. and Jonsson, P. (1997) *Software reuse: architecture process and organization for business success*. 1st edition. New York, USA, Harlow, UK and Madrid, Spain: ACM Press and Addison-Wesley Longman. ISBN: 0201924765
- Jammes, F., Mensch, A. and Smit, H. (2007) 'Service-Oriented Device Communications Using the Devices Profile for Web Services', *Advanced Information Networking and Applications Workshops, International Conference On*, Ontario, Canada, 21-23 May 2007. Los Alamitos, USA: IEEE Computer Society, pp. 947-955. DOI:<http://doi.ieeecomputersociety.org/10.1109/AINAW.2007.331>.
- Jansen, S., Brinkkemper, S., Hunink, I. and Demir, C. (2008) 'Pragmatic and Opportunistic Reuse in Innovative Start-up Companies' *IEEE Software*, 25 (6; November), pp. 42-49. DOI:10.1109/MS.2008.155.
- Jennex, M. E. (2009) 'Re-Visiting the Knowledge Pyramid', *System Sciences (HICSS), 42nd Hawaii International Conference on*. Hawaii, USA, pp. 1-7. DOI:10.1109/HICSS.2009.361.
- Jeong, C. and Kim, K. (2012) 'Technology Relationship Analysis Using Problem and Solution Similarities', *Management of Innovation and Technology (ICMIT), 2012 IEEE International Conference on*. Bali, Indonesia, 11-13 June 2012. IEEE Conference Publications, pp. 516-521. DOI:10.1109/ICMIT.2012.6225859.

References

- Jha, M. and O'Brien, L. (2011) 'A Comparison of Software Reuse in Software Development Communities' *Software Engineering (MySEC) 5th Malaysian Conference in Johor Bahru, Malaysia*, 13-14 Dec. 2011. IEEE Conference Publications, pp. 313–318. DOI:10.1109/MySEC.2011.6140690.
- Jiang, M. and Willey, A. (2005) 'Architecting Systems with Components and Services' *Information Reuse and Integration, Conf, 2005. (IRI) IEEE International Conference on*. Las Vegas, USA, 15-17 Aug. 2005. Las Vegas, USA: Institute of Electrical and Electronics Engineers (IEEE), pp. 259-264. DOI: 10.1109/IRI-05.2005.1506483
- Johansson, C., Hall, P. and Coquard, M. (1999) "'Talk to Paula and peter—They Are Experienced" the Experience Engine in a Nutshell'. In: Ruhe, G. and Bomarius, F. eds. *Learning Software Organizations*. Berlin and Heidelberg, Germany: Springer. pp. 171-185. ISBN 978-3-540-41430-8.
- Kaabi, R. S., Souveyet, C. and Rolland, C. (2004) 'Eliciting service composition in a goal driven manner', *2nd international conference on Service oriented computing Proceedings of (ICSOC)*. New York, USA, 15-18 Nov. 2004. ACM, pp. 308-315.
- Karnouskos, S. and Tariq, M. M. J. (2009) 'Using Multi-agent Systems to Simulate Dynamic Infrastructures Populated with Large Numbers of Web Service Enabled Devices', *Autonomous Decentralized Systems, 2009. ISADS '09. International Symposium on*. Athen, Greece, 23-25 Mar. 2009. Institute of Electrical and Electronics Engineers (IEEE), pp. 1–7. DOI:10.1109/ISADS.2009.5207354.
- Kleppe, A. (2003) *MDA Explained: the Model Driven Architecture: Practice and Promise*. 1st edition. Boston, USA: Addison-Wesley. ISBN: 032119442X
- Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnet, M., Erwig, M., Lawrence, J. et al. (2009) 'The State of the Art in End-User Software Engineering' *ACM Computing Surveys (CSUR)*, 43 (3; April), pp. 1-44. <http://citeseerx.ist.psu.edu>. DOI: [10.1145/1922649.1922658](http://dx.doi.org/10.1145/1922649.1922658)
- Kotonya, G., Hutchinson, J. and Bloin, B. (2004) 'A Method for Formulating and Architecting Component and Service oriented Systems' *In Stojanovic, Z. and Dahanayake and, A. Hrsg.) Service oriented Software Engineering: Challenges and Practises, IGP, Hershey, et al.* London, UK: Idea Group Inc., pp. 155-182.
- Kreuzer, K. (2005) Web Service Choreography. Available at: <http://www.st.informatik.tu-darmstadt.de/database/seminars/data/seminar-wscdl.pdf?id=198> (Accessed Oct. 2007).
- Kruchten, P., Capilla, R. and Dueñas, J. C. (2009) 'The Decision View's Role in Software Architecture Practice'. *IEEE Software*, 26 (2; March), pp. 36–42. DOI:10.1109/MS.2009.52.

References

- Krueger, C. W. (1992) 'Software Reuse'. *ACM Computing Surveys*, 24 (2; June 1), pp. 131–183. DOI:10.1145/130844.130856.
- Kumar, A., Neogi, A., Pragallapati, S. and Ram, D. J. (2007) 'Raising Programming Abstraction from Objects to Services', *Web Services (ICWS) IEEE International Conference on*. Salt Lake City, USA, 9-13 July 2007. Lake City, USA: Institute of Electrical and Electronics Engineers (IEEE), pp. 864–872. DOI:10.1109/ICWS.2007.149.
- Lamsweerde, A. van (2000) 'Requirements Engineering in the Year 2000: A Research Perspective', *22nd International Conference on Software Engineering*. Limerick, Ireland: ACM Press, pp. 5-19.
- LaToza, T. D., Venolia, G. and DeLine, R. (2006) 'Maintaining Mental Models', *Software engineering (ICSE) Proceedings of the 28th international conference on*. Shanghai, China, 20-28 May 2006. ACM Press., p. 492. DOI:10.1145/1134285.1134355.
- Lee, R., Harikumar, A., Chiang, C.-C., Yang, H.-S., Kim, H.-K. and Kang, B. (2005) 'A Framework for Dynamically Converting Components to Web Services', *Software Engineering Research, Management and Applications, 2005. Third ACIS International Conference on*. Michigan, USA, 11-13 Aug. 2005. Institute of Electrical and Electronics Engineers (IEEE), pp. 431–437. DOI:10.1109/SERA.2005.8.
- Leite, J. C. S. do P., Yu, Y., Liu, L., Yu, E. S. K. and Mylopoulos, J. (2005) 'Quality-Based Software Reuse', *Advanced Information Systems Engineering (CAiSE) Proceedings of the 17th international conference on*. Porto, Portugal, 13-17 June 2005. Berlin and Heidelberg, Germany: Springer, pp. 535–550. DOI:10.1007/11431855_37
- Lopez, A. Y. and Niu, N. (2011) 'Multiple Criteria Decision Support for Software Reuse: A Case Study', *Information Reuse and Integration (IRI) IEEE International Conference on*. Las Vegas, USA, 3-5 Aug. 2011. IEEE Conference Publications, pp. 200–205. DOI:10.1109/IRI.2011.6009546.
- Ludewig, J. and Lichter, H. (2010) *Software Engineering Grundlagen, Menschen, Prozesse, Techniken*. 2nd edition. Heidelberg, Germany: dpunkt-Verlag. ISBN: 3898646629
- Maamar, Z., Sheng, Q. Z. and Benatallah, B. (2004), 'Towards a conversation driven composition of web services'. *Web Intelligence and Agent Systems*, 2 (2; April 2004): pp. 1263-1570.
- Maunder, C. (2012) CodeProject - For Those Who Code. Available at: <http://www.codeproject.com/>. (Accessed Jan. 2012).
- McCarey, F., Cinnéide, M. Ó. and Kushmerick, N. (2008) 'Knowledge Reuse for Software Reuse'. *Web Intelligence and Agent Systems*, 6 (1; January), pp. 59-81. DOI:10.3233/WIA-2008-0130

References

- McClure, C. L. (2001) *Software reuse: a standards-based guide*. 1st edition. Los Alamitos, USA: IEEE Computer Society. ISBN: 076950874X
- McConnell, S. (1996a) *Rapid development: taming wild software schedules*. 1st edition. Redmond, Wash.: Microsoft Press. ISBN: 1556159005
- McConnell, S. (1996b) 'Who Cares About Software Construction?'. *IEEE Software*, 13 (1; January), pp. 127–128. DOI:10.1109/52.476305.
- McConnell, S. (2006) 'Software Construction, Part 1'. *IEEE Software*, 23 (1; January), pp. 99–99. DOI:10.1109/MS.2006.29.
- Mens, T. and Vangorp, P. (2006) 'A Taxonomy of Model Transformation'. *Electronic Notes in Theoretical Computer Science*, 152 (March 27), pp. 125–142. DOI:10.1016/j.entcs.2005.10.021.
- Meyer, B. (1997) *Object-oriented software construction*. 1st edition. Upper Saddle River, USA: Prentice Hall. ISBN:0136291554
- Microsoft (2010) Inserting, Updating and Deleting Entities in Entity Framework (EOF). Available at: <http://msdn.microsoft.com/en-us/data/ff629457.aspx> (Accessed Aug. 2009).
- Microsoft. (2012a) COM Microsoft. Available at: <http://www.microsoft.com/com/default.aspx> (Accessed Oct. 2008).
- Microsoft. (2012b) Managed Extensibility Framework. Available at: <http://mef.codeplex.com> (Accessed Oct. 2010).
- Microsoft (2012c) Developing Visual Studio Extensions. Available at: <http://msdn.microsoft.com/en-us/library/dd885119.aspx> (Accessed Oct. 2008)
- Microsoft. (2012d) Silverlight - Microsoft Web Platform. Available at: <http://www.microsoft.com/web/page.aspx?templang=de-de&chunkfile=special%5Csilverlight.html> (Accessed Jan. 2011).
- Microsoft. (2012e) Windows Communication Foundation. Available at: <http://msdn.microsoft.com/en-us/netframework/aa663324.aspx> (Accessed Jan. 2011).
- Mili, H., Mili, F. and Mili, A. (1995) 'Reusing Software: Issues and Research Directions'. *IEEE Transactions on Software Engineering*, 21 (6; June), pp. 528–562. DOI:10.1109/32.391379.
- Mishra, D. and Mishra, A. (2009) 'Software Process Improvement in SMEs: A Comparative View'. *Computer Science and Information Systems*, 6 (1), pp. 111–140. DOI:10.2298/CSIS0901111M.

References

- Mohagheghi, P., Conradi, R., Killi, O. M. and Schwarz, H. (2004) 'An Empirical Study of Software Reuse Vs. Defect-Density and Stability', *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, UK, 23-28 May 2004. Washington, DC, USA: IEEE Computer Society, pp. 282-292.
- Morad, S. and Kuflik, T. (2005) 'Conventional and Open Source Software Reuse at Orbotech - An Industrial Experience', *Software - Science, Technology and Engineering, 2005. Proceedings. IEEE International Conference On*, Herzelia, Israel, 22-23 Feb. 2005. Institute of Electrical and Electronics Engineers (IEEE), pp. 110-117. DOI:10.1109/SWSTE.2005.11.
- Moriso, M., Ezran, M. and Tully, C. (2002) 'Success and Failure Factors in Software Reuse'. *IEEE Transactions on Software Engineering*, 28 (4; April), pp. 340-357. DOI:10.1109/TSE.2002.995420.
- Naur, P. and Randell, B. (1968) 'Software Engineering: Report of a Conference Sponsored by the NATO Science Committee', 7-11 Oct. 1968. Garmisch, Germany: Scientific Affairs Division, NATO, p. 231.
- O'Connor, M. J., Nyulas, C., Tu, S., Buckeridge, D. L., Okhmatovskaia, A. and Musen, M. A. (2009) 'Software-engineering Challenges of Building and Deploying Reusable Problem Solvers'. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 23 (04; October), p. 339. DOI:10.1017/S0890060409990047.
- O'Sullivan, A. (2003) 'Dispersed Collaboration in a Multi-firm, Multi-team Product-development Project'. *Journal of Engineering and Technology Management*, 20 (1-2; June), pp. 93-116. DOI:10.1016/S0923-4748(03)00006-7.
- OMG-CORBA. (1999) Object Management Group: CORBA Components. Available at: <http://www.omg.org/corba> (accessed Jul 2007).
- Oracle (2012) Java SDK, Available at: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. (Accessed Jan. 2012)
- Papazoglou, M. P., Traverso, P., Dustdar, S. and Leymann, F. (2007) 'Service-Oriented Computing: State of the Art and Research Challenges'. *Computer*, 40 (11; November), pp. 38-45. DOI:10.1109/MC.2007.400.
- Parreiras, F. S. (2012) *Semantic Web and Model-driven Engineering*. Hoboken, USA: John Wiley & Sons. ISBN:978-1-1180-0417-3
- Paulisch, F. (2008) '40 Jahre Softwareentwicklung: Von Den Anfängen Bis Heute'. *Objektspektrum*, 8 (2008), pp. 10-11.
- Peltz, C. (2003) 'Web Services Orchestration and Choreography'. *Computer*, 36 (10; October), pp. 46-52. DOI:10.1109/MC.2003.1236471.

References

- Penserini, L., Perini, L., Susi, A. and Mylopoulos, J. (2006) 'From Stakeholder Needs to Service Requirements' *Service-Oriented Computing: Consequences for Engineering Requirements*, Minneapolis, USA, 12. Sept. 2006. Institute of Electrical and Electronics Engineers (IEEE)
- Petrasch, R. and Meimberg, O. (2006) *Model Driven Architecture: Eine Praxisorientierte Einführung in die MDA*. 1st. edition. Heidelberg, Germany: dpunkt Verlag. ISBN:3-89864-343-3
- Piccinelli, G., Finkelstein, A. and Williams, S. L. (2003) 'Service-Oriented Workflows: The DySCo Framework', *Euromicro Conference, Proceedings. 29th*. Stockholm, Sweden, 1-6 Sept. 2003. IEEE Conference Publications, pp. 291-297. DOI: 10.1109/EURMIC.2003.1231552
- Picot, A. (2003) *Die Grenzenlose Unternehmung: Information, Organisation Und Management Lehrbuch Zur Unternehmensführung Im Informationszeitalter*. Neuauflage. Wiesbaden, Germany: Gabler. ISBN: 3834921629
- Pinello, C., Carloni, L. P. and Sangiovanni-Vincentelli, A. L. (2008) 'Fault-Tolerant Distributed Deployment of Embedded Control Software'. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27 (5; May), pp. 906-919. DOI:10.1109/TCAD.2008.917971.
- Pino, F. J., García, F. and Piattini, M. (2007) 'Software Process Improvement in Small and Medium Software Enterprises: a Systematic Review'. *Software Quality Journal*, 16 (2; November 21), pp. 237-261. DOI:10.1007/s11219-007-9038-z.
- Poulin, J. S. (1997) *Measuring software reuse: principles, practices, and economic models*. 1st edition.. Reading, USA: Addison-Wesley. ISBN: 0201634139
- Prieto-Diaz, R. (1993) 'Status Report: Software Reusability'. *IEEE Software*, 10 (3; May), pp. 61-66. DOI:10.1109/52.210605.
- Qu, G., Ji, S. and Nsakanda, A. (2012) 'Project Complexity and Knowledge Transfer in Global Software Outsourcing Project Teams: A Transactive Memory Systems Perspective', *System Science (HICSS) 45th Hawaii International Conference*. Hawaii, 4-7 Jan. 2012. Institute of Electrical and Electronics Engineers (IEEE), pp. 3776-3785. DOI:10.1109/HICSS.2012.488.
- Rada, R. (1995) *Software reuse*. Oxford,UK: Intellect. ISBN: 1871516536
- Randolph, J. J. (2009) 'A Guide to Writing the Discussion Literature Review'. *Practical Assessment, Research & Evaluation*, 14 (13), pp. 1-13
- Ravichandran, T. and Rai, A. (2003) 'Structural Analysis of the Impact of Knowledge Creation and Knowledge Embedding on Software Process Capability'. *IEEE Transactions on*

References

- Engineering Management*, 50 (3; August), pp. 270–284.
DOI:10.1109/TEM.2003.817278.
- Rockart, J. F. (1979) 'Chief Executives Define Their Own Data Needs'. *Harvard Business Review*, 57 (2; April), pp. 81–93.
- RosettaNet. (2004) RosettaNet. Available at:
<http://www.rosettanet.org/cms/sites/RosettaNet/> (accessed Oct. 2008)
- Rothenberger, M. A., Dooley, K. J., Kulkarni, U. R. and Nada, N. (2003) 'Strategies for Software Reuse: a Principal Component Analysis of Reuse Practices'. *IEEE Transactions on Software Engineering*, 29 (9; September), pp. 825–837.
DOI:10.1109/TSE.2003.1232287.
- Rowley, J. (2007) 'The Wisdom Hierarchy: Representations of the DIKW Hierarchy'. *Journal of Information Science*, 33 (2; February 15), pp. 163–180.
DOI:10.1177/0165551506070706.
- Rücker, B. and Backschat, M. (2007) Enterprise Java Beans 3.0. 2nd Edition. München, Germany: Elsevier Verlag, 2nd Edition. ISBN: 978-3-8274-1510-3
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorenzen, W. (1991) *Object-Oriented Modelling and Design*. Englewood Cliffs: Prentice Hall, ISBN: 8120310462
- Sandhu, P. S., Aashima, P. K. and Sharma, S. (2010) 'A Survey on Software Reusability', *Mechanical and Electrical Technology (ICMET) 2nd International Conference on*. Singapore, 10-12 Sept. 2010. IEEE Conference Publications, pp. 769–773.
DOI:10.1109/ICMET.2010.5598467.
- Santana de Almeida, E., Alvaro, A., Lucredio, D., Cardoso Garcia, V. and Romero de Lemos Meira, S. (2004) 'RiSe Project: Towards a Robust Framework for Software Reuse', *Information Reuse and Integration (IRI): Proceedings of the 2004 IEEE International Conference on*. Las Vegas, 8-10 Nov. 2004. Las Vegas, USA: IEEE Conference Publications, pp. 48–53 DOI:10.1109/IRI.2004.1431435.
- Schmidt, D. C. (1999) Why Software Reuse Has Failed and How to Make It Work for You. C++ Report magazine (January). Available at: <http://www.cs.wustl.edu/~schmidt/reuse-lessons.html> (Accessed Aug. 2009).
- Schoop, R. (2012), 'Key note presentation', *Emerging Technologies & Factory Automation: 17th IEEE International Conference on*, Krakow, Poland, 17-21 Sept. 2012.
- Schuster, H., Baker, D., Cichocki, A., Georgakopoulos, D. and Rusinkiewicz, M. (2000) 'The collaboration management infrastructure', *Dataengineering 16th International Conference on (ICDE)*. San Diego, USA, 28 Feb.-3 March 2000. IEEE Computer Society, pp. 677-678.

References

- Seedorf, S. (2010) *Ontologie-gestützte Entwicklung komponentenbasierter Anwendungssysteme ein wissensbasiertes Informationssystem zur Unterstützung der Entwicklung und Wartung von Geschäftskomponenten (KompIS)*. Frankfurt am Main, Germany: Peter Lang. ISBN 978-3-631-60535-6.
- Selic, B. (2003) 'The Pragmatics of Model-driven Development'. *IEEE Software*, 20 (5; September), pp. 19–25. DOI:10.1109/MS.2003.1231146.
- Sen, A. (1997) 'The Role of Opportunism in the Software Design Reuse Process'. *IEEE Transactions on Software Engineering*, 23 (7; July), pp. 418–436. DOI:10.1109/32.605760.
- Seriai, A., Bastide, G. and Oussalah, M. (2006) 'How To Generate Distributed Software Components From Centralized Ones?'. *Journal of Computers*, 1 (5; August), pp. 40–52. DOI:10.4304.
- Sessions (1998b) *COM and DCOM: Microsoft's vision for distributed objects*. New York, USA: John Wiley & Sons.
- Sessions (1998a) *Component Oriented Middleware. Component Strategies*. ISBN:047119381X
- Sheng, Q. Z., Benatallah, B., Dumas, M. and Mak, E. O. (2002) 'SELF-SERV: A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment' *Demonstration Session of the 28th International Conference on (VLDB)*. Hong Kong, China, September 2002. VLDB Endowment, pp. 1-4.
- Shiva, S. G. and Shala, L. A. (2007) 'Software Reuse: Research and Practice', *Fourth International Conference on Information Technology (ITNG'07)*. Las Vegas, 2.-4. Apr. 2007. Las Vegas, NV, USA: Institute of Electrical and Electronics Engineers (IEEE), pp. 603–609. DOI:10.1109/ITNG.2007.182.
- Siedersleben, J. (2006) *Moderne Softwarearchitektur: Umsichtig Planen, Robust Bauen Mit Quasar*. 1. Auflage. Heidelberg, Germany: dpunkt-Verlag. ISBN: 3898642925
- Siegel, J. (2000) *CORBA 3: Fundamentals and Programming*. 2nd edition. New York, USA: OMG Press. ISBN: 0471295183
- Singh, M. P. and Huhns, M. N. (2005) *Service-oriented computing semantics, processes, agents*. 1st edition. Chichester, UK: John Wiley & Sons. ISBN: 0470091487
- Skogan, D., Grønmo, R. and Solheim, I. (2004) 'Web service composition in UML', *Enterprise Distributed Object Computing Conference (EDOC) Proceedings. 8th IEEE International*. California, USA, 20-24 Sept. 2004. Institute of Electrical and Electronics Engineers (IEEE), pp. 47-57. DOI:10.1109/EDOC.2004.1342504

References

- Slyngstad, O. P. N., Gupta, A., Conradi, R., Mohagheghi, P., Rønneberg, H. and Landre, E. (2006) 'An Empirical Study of Developers Views on Software Reuse in Statoil ASA', *Empirical software engineering (ISESE): Proceedings of the 2006 ACM/IEEE international symposium on*. Rio de Janeiro, Brazil, 2006. New York, NY, USA: ACM Press., p. 242. DOI:10.1145/1159733.1159770.
- SOA4D (2012) SOA4D Forge: Willkommen. Available at: <https://forge.soa4d.org/> (Accessed June 22).
- Software Engineering Institute .(2004) Enterprise Security Management. Available at: <http://www.sei.cmu.edu/library/abstracts/reports/04tn046.cfm>: (Accessed Dec. 2007)
- Sommerville, I. (2011) *Software engineering*. 9th edition. Boston, USA: Pearson. ISBN:0137053460
- SparxSystem (2012) Enterprise Architect. Available at: <http://www.sparxsystems.com.au/> (Accessed Nov. 2012).
- Stake, R. E. (1995), *The art of case study research*, Thousand Oaks, CA: Sage
- Stojanović, Z. (2005) *A Method for Component-Based and Service-Oriented Software Systems Engineering*. Ph. D.. Technical UniversityDelft.
- SUN. Enterprise JavaBeans. (2008) Oracle8i Enterprise JavaBeans and CORBA Developer's Guide. Available at: <http://java.sun.com/products/ejb/docs.html> (accessed Oct. 2008).
- SUN Microsystems. (2006) JDKTM 5.0 Documentation. Available at: <http://java.sun.com/javaee/5/docs/tutorial/doc/JavaEETutorial.pdf> (accessed Oct. 2007)
- SWEBOK. (2004) Guide to the Software Engineering Body of Knowledge (swebok). Available at: <http://www.swebok.org> (Accessed May 2008).
- Szyperski, C. (2002) *Component Software: Beyond Object-oriented Programming*. 2nd edition. New York, USA and London, UK: ACM Press and Addison-Wesley. ISBN:0-201-74572-0
- Taweel, A., Delaney, B., Arvanitis, T. N. and Zhao, L. (2009) 'Communication, Knowledge and Co-ordination Management in Globally Distributed Software Development: Informed by a Scientific Software Engineering Case Study', *Global Software Engineering, 2009 (ICGSE): Fourth IEEE International Conference*. Limerick, Ireland, 13-16 July 2009. Institute of Electrical and Electronics Engineers (IEEE), pp. 370-375. DOI:10.1109/ICGSE.2009.58.

References

- Thörn, C. (2010) 'Current State and Potential of Variability Management Practices in Software-intensive SMEs: Results from a Regional Industrial Survey'. *Information and Software Technology*, 52 (4; April), pp. 411–421. DOI:10.1016/j.infsof.2009.10.009.
- Tomer, A., Goldin, L., Kuflik, T., Kimchi, E. and Schach, S. R. (2004) 'Evaluating Software Reuse Alternatives: a Model and Its Application to an Industrial Case Study'. *IEEE Transactions on Software Engineering*, 30 (9; September), pp. 601–612. DOI:10.1109/TSE.2004.50.
- Tracz, W. (1994) 'Software Reuse Myths Revisited', *Software Engineering, 1994, Proceedings (ICSE): 16th International Conference on*. Sorrento, Italy, 16-21 May 1994. IEEE Comput. Soc. Press., pp. 271-272. DOI:10.1109/ICSE.1994.296788.
- Rashid, A. and Aksit, M. (2006) 'Transactions on Aspect-oriented Software Development'. Lecture Notes in Computer Science, New York: Springer.
- Tsai, C.-H., Zhu, D.-S., Chien-Ta Ho, B. and Dash Wu, D. (2010) 'The Effect of Reducing Risk and Improving Personal Motivation on the Adoption of Knowledge Repository System'. *Technological Forecasting and Social Change*, 77 (6; July), pp. 840–856. DOI:10.1016/j.techfore.2010.01.011.
- Tsai, W., Malek, M., Chen, Y. and Bastani, F. (2006) 'Perspectives on Service-Oriented Computing and Service-Oriented System Engineering', *Service-Oriented System Engineering, 2006 (SOSE): Second IEEE International Workshop*. Shanghai, China, Oct. 2006. Institute of Electrical and Electronics Engineers (IEEE), pp. 3–10. DOI:10.1109/SOSE.2006.24.
- Turban, E., Kelly, R. and Potter, R. E. (2001) *Introduction to information technology*. 3rd edition. New York, USA: John Wiley. ISBN: 0471347809
- Ven, J. S., Jansen, A. G. J., Nijhuis, J. A. G. and Bosch, J. (2006) 'Design Decisions: The Bridge Between Rationale and Architecture'. In: Dutoit, A. H., McCall, R., Mistrík, I. and Paech, B. eds. *Rationale Management in Software Engineering*. Berlin and Heidelberg, Germany: Springer. pp. 329-348. ISBN: 3540309977.
- Visser, W. (1990) 'More or Less Following a Plan During Design: Opportunistic Deviations in Specification'. *International Journal of Man-Machine Studies*, 33 (3; September), pp. 247–278. DOI:10.1016/S0020-7373(05)80119-1.
- Vlaar, P. W. L., van Fenema, P. C. and Tiwari, V. (2008) 'Cocreating Understanding and Value in Distributed Work: How Members of Onsite and Offshore Vendor Teams Give, Make, Demand, and Break Sense'. *MIS Quarterly*, 32 (2), pp. 227-255.
- Vliet, H. van (2008) *Software engineering: principles and practice*. 3rd edition. Chichester, UK and Hoboken, USA: John Wiley & Sons. ISBN: 0470031468

References

- W3C. (2000) Natural Language Semantics Markup Language: W3C Working Draft. Available at: <http://www.w3.org/TR/nl-spec/> (Accessed Feb. 2009).
- W3C. (2004) RDF - Semantic Web Standards. Available at: <http://www.w3.org/RDF/> (Accessed Feb. 2009).
- W3C. (2007) SOAP Specifications. Available at: <http://www.w3.org/TR/soap/> (Accessed June 2010).
- W3C (2006) Web Services Eventing Available at: <http://www.w3.org/Submission/WS-Eventing/> (Accessed Feb. 2008).
- W3C. (2009) OWL 2 Web Ontology Language Document Overview. Available at: <http://www.w3.org/TR/owl2-overview/> (Accessed Feb. 2009).
- Wang, G. and Fung, C. K. (2004) 'Architecture Paradigms and Their Influences and Impacts on Component-based Software Systems', *System Sciences: Proceedings of The 37th Annual Hawaii International Conference on*. Big Island, Hawaii, 5-8 Jan.2004. Institute of Electrical and Electronics Engineers (IEEE), pp. 272–281. DOI:10.1109/HICSS.2004.1265643.
- Welke, R. J. (1994) 'The Shifting Software Development Paradigm'. *ACM SIGMIS Database*, 25 (4; November): pp. 9-16.
- White, J., Hill, J. H., Gray, J., Tambe, S., Gokhale, A. S. and Schmidt, D. C. (2009) 'Improving Domain-Specific Language Reuse with Software Product Line Techniques'. *IEEE Software*, 26 (4; July), pp. 47–53. DOI:10.1109/MS.2009.95.
- Williams, S. and Kindel, C. (1994) The Component Object Model: A Technical Overview -White Paper. Available at: <http://msdn.microsoft.com/en-us/library/ms877981.aspx> (accessed Oct. 2007).
- Xu, Y., Tang, S., Xu, Y. and Tang, Z. (2007) 'Towards Aspect Oriented Web Service Composition with UML', *Computer and Information Science (ICIS): 6th IEEE/ACIS International Conference on*, Melbourne, Australia, 11-13 July 2007. Institute of Electrical and Electronics Engineers (IEEE), pp. 279–284. DOI:10.1109/ICIS.2007.185.
- Yin, R. K. (2003), *Case study research: Design and methods* (3rd ed.), Thousand Oaks CA: Sage.
- Ye, Y. (2001) 'An Active and Adaptive Reuse Repository System' *System Sciences: Proceedings of the 34th Annual Hawaii International Conference on*. Maui, Hawaii, 3-6 Jan. 2001. Maui, Hawaii: IEEE Computer Society Press, DOI: 10.1.1.95.9799
- Ye, Y. and Fischer, G. (2005) 'Reuse-Conducive Development Environments'. *Automated Software Engineering*, 12 (2; April), pp. 199–235. DOI:10.1007/s10515-005-6206-x.

References

- Zheng, Y. (2007) 'A Knowledge Based Production Line Development Management System', *Wireless Communications, Networking and Mobile Computing, 2007 (WiCom): International Conference on*. Shanghai, China, 21-25 Sept. 2007. Washington, USA: Institute of Electrical and Electronics Engineers (IEEE), pp. 5834–5837. DOI:10.1109/WICOM.2007.1432.
- Zinn M., Fischer-Hellmann, K. P., Phippen A. D. and Schütte, A. (2010) 'Finding Reusable Units of Modelling - an Ontology Approach', *Proceedings of the 8th International Network Conference (INC 2010)*, Heidelberg, Germany, July 2010. Network Research Group, pp. 377-386.
- Zinn, M., Fischer-Hellmann, K. P., Phippen, A. D. and Schütte, A. (2011a) 'Information Demand Model for Software Unit Reuse', *Software Engineering and Data Engineering (SEDE): ISCA 20th International Conference on*. Las Vegas, USA, 20-22 June 2011. International Society for Computers and their Applications (ISCA), pp. 32-39.
- Zinn, M., Fischer-Hellmann, K. P., Phippen, A. D. and Schütte, A. (2011b) 'Reusable Software Units Integration Knowledge in a Distributed Development Environment', *Software Knowledge (SKY): International Workshop on*. Paris, France, 26 Oct. 2011. SciTePress, pp. 24–35. DOI:<http://dx.doi.org/10.5220/0003699000240035>
- Zinn, M., Fischer-Hellmann, K. P. and Schopp, R. (2012a) 'Reuseable Software Unit Knowledge for Device Deployment', *Conception of Complex Automation Systems (Entwurf Komplexer Automatisierungssysteme, EKA)*. Magdeburg, Germany, May 2012. Magdeburg, Germany: IFAK, pp. 99-110.
- Zinn M., Fischer-Hellmann, K. P. and Schoop, R. (2012b) 'Case-based reasoning approach for re-use activities', *Proceedings of the 3th International Workshop on Software Knowledge (SKY 2012)*, Barcelona, Spain, Insticc, pp. 31-42.

Internal conference references

- Zinn, M. (2007) 'Service Based Software Construction Process', *Proceedings of the Third Collaborative*. Plymouth, UK, 14-15 June 2007. Plymouth, UK: Information Security and Network Research Group, pp. 169–184.
- Zinn, M., Turetschek, G. and Phippen, A. D. (2008) 'Definition of Software Construction Artefacts for Software Construction', *Proceedings of the Forth Collaborative Research Symposium on Security, E-Learning, Internet and Networking*. Wrexham, 6-7 Nov. 2008. Plymouth, UK: Centre for Information Security and Network Research, pp. 79-91.

Appendix

A. Content of data medium

The printed version of this thesis contains a Digital Video Disk (DVD). This data medium contains a digital copy of this thesis and published papers (file name includes year, conference name. Additionally, the digital copies of the 11 publications produced during the Ph.D. research are available on the data medium (the file names start with the abbreviation of the conferences and are followed by the year of publication).

B. Methodology of literature review

b.1 Categories

Basically, the literature review of this thesis follows the discussion of writing literature review published by Randolph (2009) in the Journal of Practical Assessment, Research & Evaluation in 2009.

By using the taxonomy for literature review of Cooper (1988) five relevant characteristics exist: focus, goal, perspective, coverage, organisation, and audience. The use of these characteristics in this thesis is shown as follows:

Focus: This characteristic includes four categories: research outcomes; research methods; theories; and practise or application. The literature review in this thesis is used to:

1. explain reuse in general
2. identify and discuss relevant keywords in the field of reuse
3. show the different research and problem areas of reuse (including SCAC related topics)
4. underline the problem of missing knowledge in reuse (including SCAC related topics)

Appendix

5. show the historical view on reuse and the focused problem (including SCAC related topics)
6. critically discuss problems of existing solution approaches (focusing SCAC related topics)
7. discuss the contribution of this thesis to the research area of reuse

For these objectives different perspectives were used. For objectives 1, 2, 3, 4, 5 and 7 the category ‘Theorie’ with focus on state of the art discussion is used. For the objectives 3, 4, and 7 the categories ‘Theorie’ and ‘Practise or applications’ were used mainly. ‘Research Outcomes’ also support the discussion of objectives 3, 4, 5, and 6. The objective 7 were achieved by reviewing literature based on the ‘Research methods’.

Goal: In order to fulfil the mentioned objectives, the goals of this literature review are:

- the generalisation of findings and outcomes of ‘missing knowledge in reuse research’ to
- identify central issues to
- create a line of argument for the innovative solution of a service-oriented provisioning of reuse activities (focus on software construction activities).

Perspective: The literature review is structured by using the mentioned objectives. Literature used for an objective discussion is first discussed in a neutral position of the author of this thesis. The different literature will be related to each other also using this neutral position. In most of the cases (objectives 2, 3, 4, 6, and 9) the discussions have to be related to the research of this thesis or need a critical analysis. Hereby, the perspective of the author is not neutral.

Coverage: Randolph (2009) shows four different coverage scenarios for conducting a review. This ranges from a review of all existing literature to a purposive collection of literature. This literature review focuses on a purposive selection of literature, therefore, only journal papers, conference papers, and specifications of standards (e.g., processes or technologies) were used. These

documents were searched by using digital libraries of IEEE, ACM, Springer, CiteSeerX, and Thinkmind. Also, documents were selected by analysing references cited previously in studies, journals or conference papers.

Note: Following the literature review in Chapter 2, Chapter 3 focuses on the relation between the identified problems and existing industrial environments. Therefore, references are used which are not part of the scientific resources (i.e., internal studies of companies and web pages).

Organisation: Typical forms of organisation of a literature review are historical format, conceptual format, and methodological format. This literature review mainly uses a conceptual format and is structured using the above mentioned objectives. As a result, the review follows the order of these objectives. Inside each objective discussion the conceptual format is also used, but the structure differs for the different. An exception is objective 5. This historical discussion about reuse and the historical view on the missing knowledge problem is organised chronologically by decade.

Audience: The complete thesis including the literature review is used to demonstrate the capability of the author to perform research at Ph.D. level. Therefore, the audience is the review committee for this Ph.D. thesis.

b.2 Stages of a literature review

Regarding the discussed and suggested structure of a literature review by Randolph (2009) the stages may be:

- A rationale for conducting the review
- A research question that guides the review (Problem formulation)
- A plan for collecting data (including selection process) (Data collection)
- A plan for data evaluation and analysis (Data evaluation, analysis, and interpretation)

- A plan for presenting the data (Data presentation)

Randolph (2009) shows based on Cooper (1982) that each stage has four characteristics: Research question asked, primary function in research, procedural differences that create variation in review conclusion, and sources of potential invalidity in review conclusion.

In this review the mentioned objectives use these stages. In the following sections, the stages were discussed related to each objective. As a logical result each objective follows the same structure.

b.3. Problem formulation

For an exact formulation of the problem, it is relevant to define a literature review question (also called secondary research). In the literature review of this thesis each objective uses its own secondary research question:

1. *Objective: 'From the previous literature, what is known about software unit reuse?'*
2. *Objective: 'From the previous literature, what are relevant keywords known regarding software unit reuse and the problem of missing knowledge in reuse?'*
3. *Objective: 'From the previous literature, what are relevant researches or problem areas known in the area of reuse?'*
4. *Objective: 'From the previous literature, what is known about the problem of missing knowledge in the area of reuse?'*
5. *Objective: 'From the previous literature, what is known about the problem of missing knowledge in the area of reuse?'*
6. *Objective: 'What is known about approaches to solve the problem of missing knowledge in the area of reuse?'*

Appendix

7. Objective: *'From the previous literature, what is known about different perspectives on the problem of missing knowledge in the area of reuse and software construction activities?'*

Note: Objective 4 and 5 uses the same question to lead the literature review. But these objectives differ in their goals and data analysis.

Randolph (2009) mentioned that the other relevant part of the 'Problem formulation' is the determination of inclusion or exclusion. This literature reviews two inclusion/exclusion principles:

1. Every existing document is excluded from the review.
2. From the list of excluded documents, studies can be included if they meet all of the following criteria (except optional criterias):
 - a. The study is in English OR in German language (German is only allowed if (1) the publisher is a scientific institution (e.g., University) or (2) written by an relevant author in the field or (3) no English reference can be found of the studies or an relevant statement of it).
 - b. The study is a journal or conference paper or included as scientific work in one of the following digital libraries: IEEE, ACM, Springer, CiteSeerX, journals or conferences.
 - c. (Optional) The study was mentioned as reference for a statement about reuse by other previously selected studies.
 - d. (Optional) The study was mentioned by other researches or relevant people in the area of reuse during personal contact with the author of this thesis.
 - e. The study identifies itself or is identified by other studies as a state of the art paper for the topic of reuse or some subtopic of them.

- f. The title or abstract contains one or more of the following primary keywords (or synonyms): reuse, software unit reuse, reusability, knowledge, software reuse, software product line, software construction, missing knowledge, reuse activities, and software construction process.
- g. The conclusion of the study includes new or existing statements of the focused topic of software reuse.
- h. The study focuses the discussion of the novel contribution of this research to the research area of software unit reuse.

Each objective follows these principles.

b.4. Data collection

Initially the data were searched in the digital libraries (mentioned in inclusion/exclusion principle 2.b) using the primary keywords (mentioned in inclusion/exclusion principle 2.f).

The different objectives, thereby, have different aims:

1. Objective: The goal is to identify a pivotal set of articles together presenting the area of software unit reuse in general.
2. Objective: The goal is to identify relevant term definitions or discussions in software unit reuse.
3. Objective: The goal is to identify a pivotal set of articles together presenting research and problem areas of reuse in general.
4. Objective: The goal is to identify a pivotal set of articles explaining the problem of missing knowledge in the area of reuse.

Appendix

5. Objective: The goal is to identify a pivotal set of articles explaining the problem of missing knowledge in the area of reuse in the history.
6. Objective: The goal is to identify a pivotal set of articles explaining solution approaches for the problem of missing knowledge. Especially the problem of missing knowledge of reuse activities (focusing software construction activities).
7. Objective: The goal is to identify a pivotal set of articles including details to the gap of research for reuse of software construction activities and the related problem of missing knowledge.

The separation between relevant and non-relevant documents is a process including two different steps. Figure 147 shows this process that is described in relation to the figure as follows. After a search for documents in the mentioned libraries the first step (Step 1) is to separate documents based on their content shown in the abstract, conclusion, and the state of the art section (if provided). The papers can now be separated into three different types (Step 2). The first type includes all papers without any useful content (Type 1). The second type of document are documents only providing interesting references which have to be checked for relevance (using this process; Type 2). The third type of documents includes content describing different specific properties/parts of reuse or that make common statements about it. These documents may include interesting references also. These references have to be proofed for relevance (using this process description).

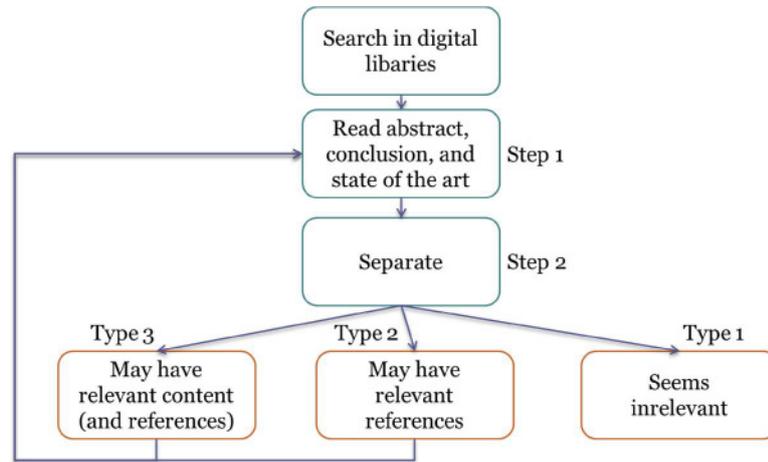


Figure 147 - Sketched process for document separation

In general, each objective uses this data collection procedure. They only differ in the relevance checking part. The author (reader of the documents) has to look into the abstract, conclusion, and the state of the art section and proof which objective goals this document may contribute to.

b.5. Data evaluation

In the data evaluation phase of this stage the following procedure model was used: The selected documents were read completely. Parallel to this task interesting statements regarding the area of software unit reuse were written down on a note paper or dashboard with each document being ascribed a unique number. The same number is written down on the note paper or dashboard. After collecting these notes the note paper or dashboard were scanned / photographed and collected in a ‘cookbook’. Each objective followed this data evaluating process, but they differ in the identification characteristics for the interesting information. Appendix Section C shows and explains an example regarding information collected from a study relating to the different methods

Appendix

described in this section. In the following, the basic rules for identifying interesting information for each objective are shown:

Information is seen as interesting for objective 1 if

- it contains one or more general statements about reuse,
- it contains a specific statement about reuse or different parts of reuse,
- it contains a discussion or opinion statement or
- it contains a statement of other studies (reference).

Information is seen as interesting for objective 2 if

- it contains or creates special words and related them to the topic of reuse or missing knowledge in reuse,
- it explains existing words of the reuse or missing knowledge area or
- it discusses existing words of the reuse or missing knowledge area.

Information is seen as interesting for objective 3 if

- it contains a problem or research area discussion which is related to reuse or
- it discusses practice or theoretical problems.

Information is seen as interesting for objective 4 and 5 if

- it contains a description of the focused problem ,
- it discusses the scenarios leading to this problem ,
- it discusses the impacts of this problem (i.e., for software engineers) or
- it discusses solutions for this problems.

Information is seen as interesting for objective 6 if

- it discusses solutions for the problem of missing knowledge.

Information is seen as interesting for objective 9 if

- it discusses the view on the problem of missing SCAC knowledge in common or critically.

b.6. Data analysis and interpretation

Based on the previous steps the data analysis is processed in three steps focusing on a quantitative research.

1. The first step is identifying the evidence an objective wants to prove positively as an outcome of the literature review.
2. The next step is to create a hypothesized causal link between the collected data and the evidence.
3. The last step is to identify and relate contrary findings and rival interpretations.

The objectives differ in their related evidence. In the following, the objectives goals and the necessary evidences are described. Also, a description of contrary findings or rival interpretations is given. Based on Randolph (2009) this is an relevant information in the description of a qualitative data analyses of a dissertation.

The first objectives want to explain software unit reuse in general. The statement of this objective is that reuse is a wide field. Also, it is relevant to mention the concrete fields of software unit reuse. Both facts are used to demonstrate the authors' knowledge about the field of software unit reuse. Contrary findings and rival interpretations are declarations which see reuse as a small or not relevant field of software engineering. An outcome of objective 1 is a list of relevant fields of reuse. It contributes to the primary research background information and identification of research direction.

Appendix

The second objectives' goal is to identify relevant keywords. Thereby, it is necessary to demonstrate that these keywords exist and have the same meaning. Both facts are used to demonstrate the knowledge of the author about the field of software unit reuse. Contrary findings and rival interpretations that differ to the most common definition are collected for each keyword. An outcome of this objective is a list of keyword definitions which contribute to the primary research by providing a basic definition of relevant terms used.

The third objective proves the field of possible research areas and problem areas. Only studies are used which discuss topics of one of the focused research areas. Contrary findings and rival interpretations are statements that declare a specific topic not to be in a specific research or problem area. The outcome of this objective is a literature based overview of research and problem areas of reuse in software engineering. The contribution to the primary research is the demonstration that the primary research is in a valid research field of reuse and it handles an existing problem identified by other studies.

The fourth objective identifies the problem of missing knowledge in reuse in detail (including SCAc related knowledge). Studies supporting this objective show that this problem exists. Statements are seen as evidence if they demonstrating or discussing the problem, scenarios leading to the problem, and impacts the problem creates. All three statement types are used to identify the problem. Contrary findings and rival interpretations and declarations do not identify the problem as a problem or explicitly disagree with other studies on one of the three statement types. The outcome of this objective is a problem definition based on the three statement types. The resulting contribution to the primary research is a clear definition of the focused problem based on the second research.

Appendix

The fifth objective is a variation of objective 4. Here, the historical organisation is relevant. The evidence of this objective is that the problem still exists in the past and is not a result of activities or technologies of the present. Contrary statements are research results of studies which declare the problem of missing knowledge (especially SCAC related knowledge) as not existing or incorrect. The outcome of this objective is a chronological timetable of studies related to the problem of missing knowledge in the field of reuse. It contributes to the primary research as an evidence for an older not fully solved problem.

The sixth objective discusses existing solution approaches solving the focused problem. The focused evidence is to demonstrate that existing solutions do not solve the problem completely. Thereby, a solution has to be sustainable, correct, and do not include a learning or interpretation process. Contrary findings and rival interpretations differ from that perspective by demonstrating that a specific solution has one or more of the relevant characteristics. The outcome of this is the identification of a missing solution in the research field for the problem of missing knowledge in reuse. The contribution to the primary research is the identification of a gap in the span of existing solution area.

The ninth objective is a variation of objective 8. Here, the novelty of the research is focused. A statement which identifies the research gap, the given solution or parts of it as invalid or wrong or identifies the used research methods as invalid or wrong are contrary findings or rival interpretations. The contribution to the primary research is the evidence of a novel solution in the scope of the used literature. This is also a contribution to the research field.

From the following three steps shown at the beginning of this section, the second step is to generate a link between the evidence and the found statements. Each piece of evidence is represented by an

Appendix

ID. Therefore, the different objectives are classified by their main topic relation. Objective 1, 2, and 3 focuses on reuse (general perspective; class REUSE (R)). Objective 4 and 5 focusses on the topic of problems of reuse (class PROBLEM (P)). Objective 6 and 7 are linked to the topic of solutions (class solution (S)). The final two objectives, 8 and 9, relate to the topic of the contribution of this research (class CONTRIBUTION (C)). Each statement identified as relevant for this literature review is written down in the cookbook (see Appendix Section C) and related to one of the classifications.

To fulfil the last step, the statement has to be identified as supportive or contrary to the evidence used for the related objective. Therefore, it is marked as evidence (E) or contrary (C).

Figure 148 shows an example of a statement analysis and interpretation used in this literature review. Each statement gets an ID (1) and a statement summary. Then it is related to an objective (3). The last step is to decide if this statement is evidence or contrary to the evidence of the objective.

(1)	(2)	(3)	(4)
14	Failure in Reuse: If changes occur only reuse processes are changed and not human factors or repositories	R3, P4	E

Figure 148 - Example of a cookbook form for data analysis and interpretation

Note: A complete example of a protocol sheet is shown in Section C.

b.7. Data presentation

The last relevant information in a literature review methodology is the data presentation stage. As mentioned before a conceptual organisation of the literature is used. The discussion of each objective is structured in the thesis as follows:

Appendix

Objective 1 is discussed first in Section 2.2.1. It is followed by the discussion and definition of relevant keywords in Section 2.2.2. This fulfils objective 2. In Section 2.2.1, based on objective 3, a big picture about the research areas is created.

After this introductory part, objective 4 analyses the problem of missing knowledge in Section 2.2.3. This includes the fulfilment of objective 5 by discussing an historical perspective in Section 2.2.3.1. For objective 7, Section 2.2.3.2 and Section 2.2.3.3 to demonstrate the contribution of the primary research from the perspective of the literature and fulfils the last objective.

Note: To fulfil objective 3, Chapter 3 discusses solution approaches.

C. Document evaluation protocol example

The literature references were summarised in an evaluation protocol based on the methodology of the literature review shown in Section B. Figure 149 shows an example of a protocol sheet and is explained as follows. The sheet has 4 areas. The first area (1) includes the title of the related document (i.e., a journal paper) and the number of the protocol sheet in the protocol book.

13 Software Reuse Myth Revisited		(1)	
UID:	37	Publication type:	Journal
I/E principles:	Default	Reader:	Zinn
Supports Objective Goals:	R3,P1,S1	Found by:	IEEE "Software Reuse"
Relevant paper:	No	Main Author:	Tracz
Types of statement quality:	Low quality	Included in thesis:	NO
Date of publication:	1994		
Statements		(2)	
ID	Statement	Supported Objectives	Evidence / Contrary
1	Software Reuse is a not technical problem	R3, P1, S1	C
2	No special tools needed for software reuse	R3, P1, S1	C
3	Reusing code does not result in huge increases of productivity (with example calculation)	R3	E
4	Software Reuse != Hardware Reuse	R1	E
5	Reused Software != reusable software	R1	E
6	SR is only planned reuse	R3, P1, S1	C
Additional Information		(2)	
References to check:	None	(3)	

Figure 149 - Document evaluation protocol example

The second area (2) includes additional data, as for example, a unique identifier (UID), the inclusion and exclusion rules (c.f. Section B), the listing of supported objectives, the identification whether the paper is relevant for the thesis, type of quality (represents the personal opinion of the author), date of publication, the publication type (e.g., journal paper), the reader, main author, the publisher, and a note to state whether the paper is used inside the thesis.

Appendix

In the third area (3) each piece of evidence is represented by an ID. Therefore, the different objectives are classified by their main topic relation. Objective 1, 2, and 3 focuses on reuse in (general perspective; class REUSE (R)). Objective 4 and 5 focusses on the topic of problems of reuse (class PROBLEM (P)). Objective 6 is linked to the topic of solutions (class solution (S)). The final objective, 7, relates to the topic of the contribution of this research (class CONTRIBUTION (C)). Each statement identified as relevant for this literature review is written down in the protocol by summarising the statement and related to one of the classifications.

To fulfil the last step, the statement has to be identified as supportive or contrary to the evidence used for the related objective. Therefore, it is marked as evidence (E) or contrary (C).

In the last area other possible (relevant) references cited by the focused publication can be listed.

D. Additional research on software unit base technologies

The research uses classes, components, and services as research objects for software construction activities. The following overview of the base technology concepts (object, component, and service orientation) of these software units can be used for the discussion about the research topic. In the following, procedure models and software construction properties of these concepts are shown.

d.1. Object-oriented software construction

The purpose of object-oriented software construction is the development of applications by the means of object instances and their interaction models. To reach this purpose, two different elementary concepts are used: Object orientation and objects as units of modelling.

Nowadays three approaches are used within the object-oriented construction: object-oriented analysis (OOA), object-oriented design (OOD) and object-oriented programming (OOP). In the OOA the requirements are determined and transferred into a document. The result is a technical description with object-oriented drafts (OOA model). In the OOD the requirements are transferred into a specification. Various diagrams are created here, to transfer the requirements into the object-oriented world. Since UML is considered the standard, UML diagrams are usually used. The specification, obtained from this, is the basis for the OOP. The obtained specification is the basis for the OOP. As a result, of OOP a software or a part of software is created.

Through this kind of object-oriented thinking it becomes possible to illustrate the real world viewpoint inside a technology concept.

Here, the units of modelling are objects. Objects have properties and methods. Modelling and programming occurs through the instancing of different objects and the mutual call of methods. Hereby, objects display the attempt to illustrate real world or virtual objects. The interfaces are

Appendix

especially relevant here (Siedersleben, 2006). Figure 150 shows the typical representation of a class with properties (fields) and methods in UML notation. Sommerville (2011) shows that objects can be used in a local and in a distributed scenario.

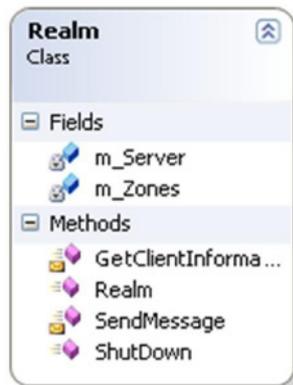


Figure 150 - UML like representation of a class

Modern development environments show a multitude of possibilities for software engineers. Eclipse and Visual studio (.NET) are particularly well-known. These developing environments support the software engineer in object-oriented thinking and programming. This happens, among other things, by the application of syntax highlighting, intellisense, graphic possibilities of modelling, and automatic code production, as for example, generic programming (Czarnecki and Eisenecker, 2000) or snippet technology (Microsoft, 2008). All together it has tried relieve the user of "writing work". In addition, the user is offered the possibility, also to access other technologies (as far as the base architecture admits this). For example, it is possible to integrate Web Services into Visual Studio.NET. The representation corresponds to a class. The methods of the Web Service are methods of this class. The software engineer can handle the Web Service in the usual

Appendix

object-oriented manner (Microsoft, 2012e). To achieve this, the development environment has to be tuned to the programming language and the according underlying architectures.

In addition, development environments, exhibit the possibility of extension by (e.g., addons or plugins). This leads, for example with Eclipse and Visual studio.NET, to the fact that other programming languages are also supported. The aim here is to make the advantages in the user guidance, offered by these IDEs, also available to be used in other programming languages. New technologies or procedure models can be simply integrated as an extension into development environments. Eclipse Foundation (2008), for example, shows a multitude of extensions for the development environment Eclipse.

The general procedures conform to the order of the approaches OOA, OOD, and OOP. According to Siedersleben (2006) most procedure models are based on one of the following models: Workflow model, data flow-/ activity model, roll-/ action model. Typical models of object-oriented development built on the above mentioned models are, (e.g., waterfall model, iterative model, prototyping, extreme Programming, and Scrum).

Note: The mentioned models are assumed as known. For further information see Bunse and Knethen (2008).

With the technical conditions, particularly languages, architecture, and runtime environments are interesting. In the course of time a multitude of object-oriented programming languages were developed.

The languages Java, C ++, VisualBasic, PHP5, and .NET based languages like C# and VisualBasic.NET show a high distribution. A special interest is centred on platform independent programming languages. Its advantage is the possibility to execute binary code, generated by

Appendix

means of these languages, on different operating systems. This however requires a corresponding runtime environment within the operating systems. Figure 151 shows the platform construction of the .NET architecture. Here, it can be seen that .NET programming languages are based on a common language specification (CLS). Furthermore, the diagram shows that .NET offers a common language infrastructure (CLI) for .NET programs, which allows platform independence.

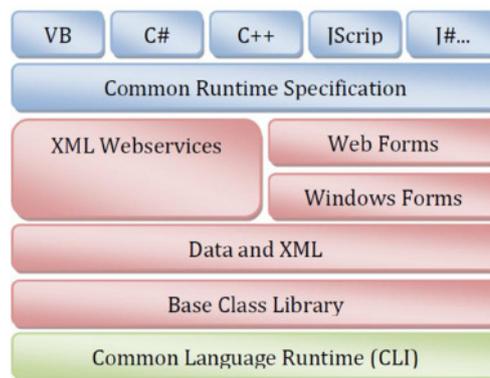


Figure 151 - Technical architecture of the .NET Platform

Other platforms, as for example Java, show a different architecture. The aim, however, as with the .NET platform is to offer a row of possibilities.

The platforms shown offer a runtime environment to the user, meaning a possibility to execute an application which is based on this platform and which also uses its possibilities. Basically, the runtime environment constitutes an interface which translates the commands from the application to the processor. The other way around, the resources of an operating system are offered as a uniform interface within the runtime environment. In contrast to the executed applications, the runtime environment itself is operating system-dependent. Figure 152 shows an example of the

Appendix

runtime communication of .NET for object-oriented languages. The Java runtime communication is similar.

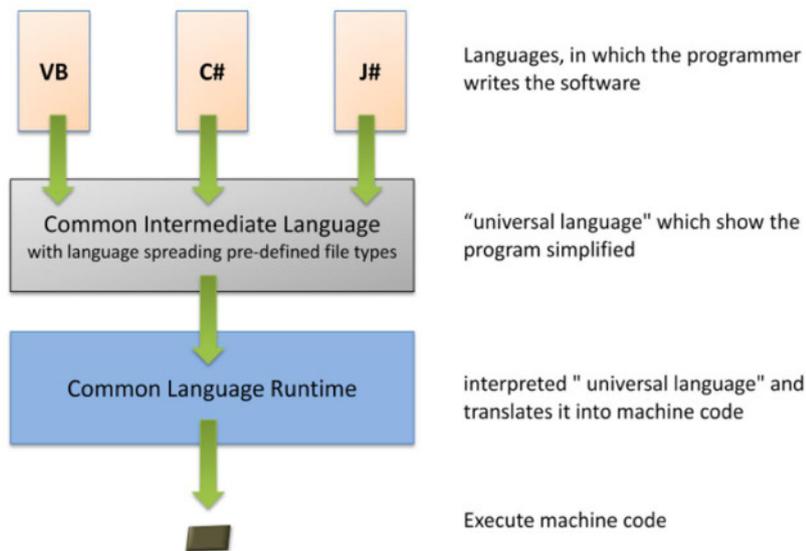


Figure 152 - .NET runtime environment communication (Computerbase 2008, online)

Modern software architectures for object orientation differ from former ones in their varied possibilities and the more efficient memory management. .Net and JavaEE display well-known architectures (Siedersleben, 2006). In both architecture sceneries, attention was paid to the fact that there is a multitude of interfaces to other technologies and systems.

With the investigation of the actual research areas, particularly the area of the retroactive object orientation stands out. Approaches as for example Visual Basic display that it is interesting to enhance systems / approaches that are already in existence with object orientation, to use them in an object-oriented environment or behaviour.

d.2. Component-based software construction

Component-based construction has the purpose to develop software by the composition of ready components. The suitable components were developed by a previous step or were bought.

The key concepts of component-based software construction are component models. These indicate the framework for the software to be developed. During the development, the designer works within this framework. The components which should be used has to fit in such a component model. Accordingly, during the development of components, the interfaces, as well as the adaptation to the runtime environment which is mostly given by a container, have to be looked after. Thereby, it is resorted to an interface description language which is suitable to the component model.

Another key concept is that components usually constitute independent assemblies. These assemblies are processed and integrated with the help of prescribed rules (depending on the component model).

In component-based construction, components are the units of the modelling that means components are the essential units in planning, draft, and construction. If the software engineer is at the level of the construction, only finished components can be joined to form an application.

Welke (1994) and Dahanayake, Sol and Stojanović (2003) indicate that the technology only accounts for one part of the component-based solution. The other parts are the procedure models. For software engineers there are a multitude of models for the component-based software development. Table 62 lists the best known methods and approaches as well as their differentiation signs according to Stojanović (2005). Dahanayake, Sol and Stojanović (2003) indicate with their investigations that the manner in which models handle components depends on the underlying object-oriented methodology. They point out that most units of modelling are built in a

Appendix

conventional manner, for example with object-oriented languages. Thereby, these units contain the peculiarities of the underlying approaches and technologies.

	RUP	Select Perspective	Catalysis	KobrA	UML Comp.	BCF
Availability	Book, web page, consultancy, training	Book, web page, consultancy, training	Book, web page, consultancy, training	Book, papers	Book, papers, consultancy	Book, papers, consultancy
Background	Industry	Industry	Academic & Industry	Academic & Industry	Academic & Industry	Academic & Industry
Maturity	Widely used in practice	Used in Practice	Used in Practice	Not fully applied in practice	Not fully applied in practice	Not fully applied in practice
Method concerns	Development, management	Development, management	Development	Development, management	Development	Development
Use of a method	Regularly used in industry	Regularly used in industry	Catalysis based methods used	Used by KobrA consortium	Potentially used in industry	Potentially used in industry
Elements of a development process	Workflows, guidelines, templates	Phases, guidelines	Rough guidelines, patterns	Phases, activities, guidelines	Workflows, activities	Phases, guidelines, patterns
Method input	Requirements, use cases	Business processes, use cases	Use cases, domain model	Requirements specification	Use cases, domain model	User's requirements, domain model
Method output	Application models and software	Application models and software	Application specification	Application specification	Application specification and models	Application specification and models
Tool support	Rational product family (Rational Rose, etc.)	Select Component Factory	COOL tool family, now Advantage tool family	Enabler Workbench and Repository	No specific tool; UML-based tools used	No specific tool; UML-based tools used
Modelling techniques	UML	Business Process Management Catalyst, UML, ERD	UML	UML-based	UML (with extensions)	UML-based
View on components	Implementation	Design & Implementation	Design & Implementation	Design & implementation	Design & implementation	Design & implementation
Component design	Not applicable	Service package, UML subsystem	Stereotype	Stereotype of the UML class	Stereotype of the UML class	Not specific
Component implementation	Component diagram	Component package,	Package, Software components	Realization components	Not specific	Software components
Defined design patterns	No	Yes	Yes	No	No	Yes
Component repository	No	Yes	No	Yes	No	No
Reusability	Software components	Components, patterns	Components, patterns,	Design-level and software	Design-level and software	Design and software

Appendix

	RUP	Select Perspective	Catalysis frameworks	KobrA components	UML Comp. components	BCF components, patterns
Incremental & Iterative	Yes	Yes	Yes	Yes	Yes	Yes

Table 62 - Comparison of component-based procedural models (Stojanović, 2005)

Table 62 is based on the analysis from Stojanović (2005) the procedure models: Rationally Unified process, Select Perspective, Catalysis, cobra, UML Components, and business Component Factory are briefly displayed and the relations to the component-oriented construction are explained.

The Rationally Unified Process (RUP) constitutes a procedure model, which was developed by the company Rational software in 1995 (cf. Jacobson, Booch and Rumbaugh, 1999). RUP is based on the combination of the rationality approach and the Object Process (cf. Jacobson, Chrsiterson et al., 1992). Concerning the contents, a complete software life cycle can be illustrated with RUP. Single steps, as for example requirement analysis, testing, etc., are defined as workflows. RUP is based on the use of UML. The notation for components, used by the UML, is also used by the RUP approach. Through this, a basic support for components is given. RUP, however, is no component-based procedure model. Only a general modelling framework on basis of object orientation can be used. RUP defines a component as follows:

„A non-trivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. A component conforms to, and provides the physical realization of a set of interfaces.” (Kruchten, Capilla and Dueñas, 2009, p. 284)

Another procedure model is Select Perspectives (Allen and Frost, 1998, pp. 251; Apperly, 2003, pp. 13),, which was created through the combination of the Object Modelling Technique (OMT; Rumbaugh et al., 1991) and the Use Case controlled Objectory method from (Jacobson, Chrsiterson et al., 1992). Like RUP, Selected Perspective was no procedure model for the construction of

Appendix

components at its beginning. (Apperly, 2003) nevertheless, indicate that services can be used as an extension for the handling of components. Basically, there are three notation forms which are used by Selected Perspective (based on Apperly, 2003): Business process modelling with computer science corporations (CSC), Catalyst methodology (CSC, 1995), UML notation for class- and component concepts and data modelling with Entity-relationship diagram (ERD). In Select Perspective, a component is a compiled object which displays a certain service through defined interfaces. Within the Select Perspective approach, components are stored in a component repository.

Catalysis (D'Souza and Wills, 1998) is a component-based approach which is based on the use of UML. In contrast to other approaches (as for example RUP) there are no activities like for example project management, quality management, and configuration management. During the use of Catalysis, so-called process patterns are created. These fit in a certain problem and certain basic condition.

Catalysis indicates according to (Fettke, Intorsureanu and Loos, 2002, p. 8 translated) the following properties:

- *”The model supports exclusively a component-oriented development. To achieve this, a so-called product family approach is selected. The authors understand several software systems which are based on an amount of the same components under the term of product family.”*
- *”The model is based on a very iterative and incremental procedure which leads to very short development times.”*

- *”Even if the approach requires no continuous formal specification of software systems, value is nevertheless placed on a high quality of the specification of a system.”*

Stojanović (2005) shows that Catalysis uses some object-oriented methods and displays a process of development which constructs software by means of high level demands.

Kobra displays a procedure model for software development which is based on the software life cycle definition from the Product Line Strategy. The purpose is to use the component or the concept standing behind it during the whole life cycle. Hereby, an underlying framework will be defined from which components and applications can be built (Framework Engineering). For the conversion, different UML charts are used in the Kobra approach. Kobra constitutes a result of an investigation of the companies Softlab GmbH, Psipenta GmbH, GMD ridge, and Fraunhofer IESE which was ordered by the German government. Kobra is based on preceding methods (Cleanroom-, Fusion- and Catalyst methods) and is compatible with other approaches like the RUP and OPEN Process Framework (OPF). Within the Kobra approach, component repositories are defined. (cf. Atkinson and Muthig, 2002)

Through the use of UML-Components, another possibility to describe components arises (Cheesman and Daniels, 2000). The basis forms the architecture consideration of the desired combination of components and the representation of single components for itself. Figure 8 shows an example of a component chart with UML 2.0 specification.

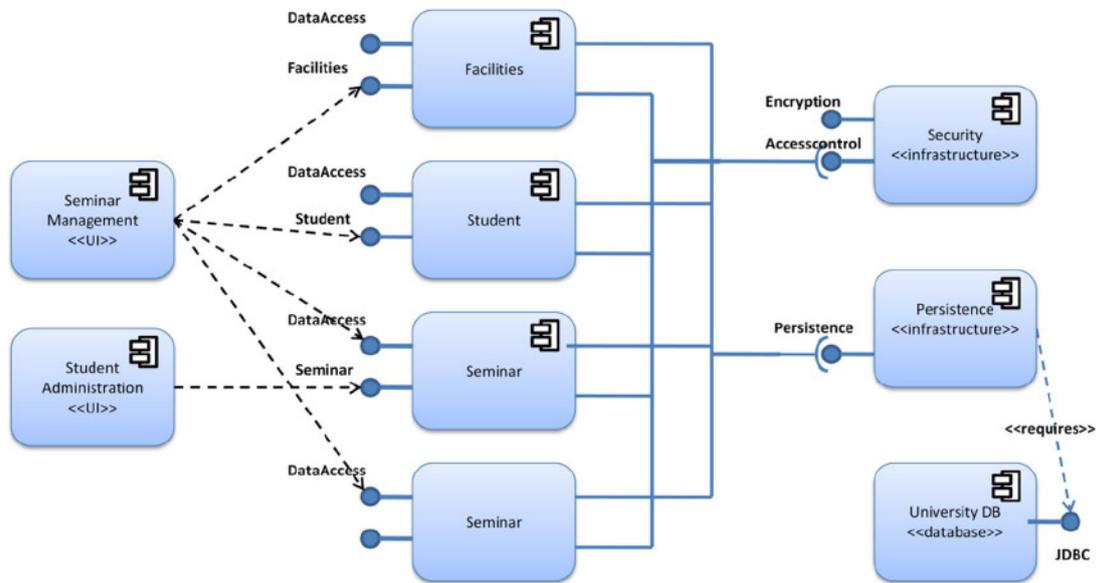


Figure 153 - UML component diagramm example (Ambler, 2008)

Basis of this representation is the possible interfaces of a component and the relations with other components. UML components, in contrast to other component models, do not provide the conversion into a component. Additionally, there are different applications (e.g., Enterprise Architect; SparxSystem, 2012) which allow an automatically conversion of UML Components into components.

The next example is the approach of the Business Component Factory (BCF). Herzum and Sims (2000) show in their approach that applications can be built with components. Thereby, they use an own classification of components (language class, distributed components, business components, business component systems, and system components) and a three-part sequence of their action model:

- A framework that contains the properties of different component-based approaches

Appendix

- A component Factory environment
- Applications which guarantee an automatic generation of components.

During the development with BCF, an application is defined by means of the used frameworks. The component-factory generates the components required. This approach, however, assumes that all information exists in the framework as well as in the generators of the factory.

The technical conditions for component-based software construction depend on the choice of the component model. Heineman and Council (2001) and Szyperski (2002) indicate that a component model defines the following standards:

Component implementation, -naming, -interoperability, -adaptation, -composition, -evolution and -development. Hereby, a component model also determines the elements that are relevant for software engineers: Implementation language, architecture, platform, and runtime environment. A software engineer has to be aware of these properties and elements of the used component model during the complete software life cycle.

In the following, the three component models are examined more exactly concerning the properties relevant for software engineers.

Component Object Model (COM): COM shows a language-independent component standard and was introduced in 1993 by the company Microsoft (see Williams and Kindel, 1994).

The technical properties of a COM component identify themselves as follows: A COM component owns a unique ID (global unique identifier; GUID) through which the component can be identified. This GUID is loaded into one COM component server. Thereby, the component is available and can be used or be administered.

Appendix

Another property is the construction of the interfaces of COM components. The COM technology groups functions in interfaces. Three kinds of interfaces are distinguished: The first interface is the observance of binary standards to enable calls between components (or component parts). This corresponds to the rules for dynamic method calls in Dynamic Link Libraries (DLL). The second interface is also a standard interface for the administration of the COM component through a direct application and the COM-server. The third kind of interface is software engineer-dependent. The methods offered here depend on the developed contents and show the user's content. With the help of these three interfaces, a component can be found, administered, and used. According to Microsoft (2012e), COM components (in addition to the already mentioned properties) are language-independent, platform-independent, object-oriented, and location-independent.

Since COM components are conventional EXE and DLL files with advanced interface, the methods can be called directly. The COM server offers this through a standardised method. Thereby, the language independence is given. It presents itself differently with the platform independence. COM components are loaded and administrated only on a Windows operating system. In 1996 Microsoft showed that the offer can also be extended from local to remote calls. This is a Distributed Component Object Model (DCOM) designated system which allows the use of a COM-Component from another system (Microsoft, 2012a; Session, 1998). The property object-oriented refers to the development, as well as to the use of components. This kind of the components is created in an object-oriented way and owns object-oriented contents. During the usage, a component is instantiated and the client application can use the methods of the component. The location-independence is guaranteed through the registration of the component. The offer is managed by the COM server. For the client applications there is only the COM server and the interfaces of the

Appendix

component. The physical location of the components files is hidden and is not relevant for client applications.

Another type of component models is Enterprise Java Beans (EJB). This form introduced by Sun Microsystems in 1998 is an extension of the client-sided model Java Beans as a part of the Java Enterprise Edition (Java EE, formerly called J2EE).

Basis for this component model is the programming language Java and Java EE as an extension platform. EJBs are developed in Java, as object-oriented and are also used object-oriented. There are two kinds of Beans: session and message driven Beans.

A session Bean constitutes a component which exists only during a client / server connection (cf. Rucker and Backschat, 2007). These contain the logic of workflows and Use Cases. Basically, they are distinguished between stateless and statefull session beans. Stateless means that a session Bean owns no state, regarding the according client application that is represented by it exhibits no status data at this point (e.g., a normal web page). Statefull, on the other hand, means that a client application has a state (e.g., a web shop). A message driven Bean is in contrast to the session Bean permanently available, (e.g., to the persistent saving of data). They serve asynchronous communication. The Bean instances deployed and offered in a special container (EJB container). It fulfils the duties of lifecycle management, state management, security, transaction management, and persistence management. The container, therefore, offers the runtime environment and controls the life cycle of the instances. Special interfaces (Enterprise of service) offer access to, for example, data banks, and message services. Among other things, an EJB server offers the according communication infrastructure for incoming client messages and administers EJB container.

Appendix

For the interaction with the client application, Enterprise Beans possess two kinds of interfaces: the home and the object interface. These interfaces are generated by the container. The container monitors these interfaces and can react to information which runs through the interfaces if required. The home interface offers methods for the lifecycle management of the component. The object interface offers the software engineer-dependent part (user's content).

EJB Containers run in a runtime environment of an EJB server. Sun Microsystems provides a specification for an EJB server which, however, contains only basic information. Subjects like memory and transaction management of the server are left to different manufacturers.

Additionally, a remote interface is made available for the use of Beans. This, however, is development-dependent and programmed with Java Remote Method Invocation (RMI) and Application Programming interface (API). With the usage of Java Interface Definition Language (IDL) and/or Common Object Request Broker (Corba) the Beans can also be used by not Java applications. Sun Microsystems assigns the role 'Bean developer to the software engineer in the developing scenario. (SUN, 2008; Rucker and Backschat, 2007)

The CORBA Component Model (CCM) was developed in 1999 by the OMG Group. Like EJB, it is a server-sided component system (cf. OMG-CORBA, 1999). On account of the decreasing economic interest in regard to Corba, this thesis does not go further into this component model. For further information see Stojanovic (2005), OMG-CORBA (1999), and Siegel (2000).

Regardless of the choice of component models, components constitute a special case if they have to be developed first. In this case other construction forms can be used, as for example a component can be developed with the use of technologies like object-oriented, procedures and tools. This is why software engineers can also resort to suitable development environments. The basic conditions

result from technical contents and technical conditions (e.g., component model). In the example of EJBs the components are also first compiled by the according EJB container and then implemented. Hence, any development environments can be used, as far as they support the interfaces and technologies which are required by components or the component model.

Component-based software construction exhibits some possibilities of connection with other types of construction (see Section D).

d.3. Service-based software construction

The purpose of service-based construction is to build systems from already existing services. The development of services itself is separated here from the development of a system. When considering a service-oriented construction, the fact that it is an interface technology is relevant (i.e., in contrast to other forms of construction), the implementing and its context dependency is (almost) irrelevant. There is also a considerable difference with coupling. Often, services are loosely coupled components. Objects and components, however, have a certain degree of coupling.

The key concepts are, according to Wang and Fung (2004) and Tsai et al. (2006) the services in services themselves, service description / registration / discovery and composition / binding. As already mentioned, the components of modelling within the service-based construction are depicted by services.

Papazoglou et al. (2007) indicate that there are two different areas in the area of development activities: Methods for service-based engineering and design time models.

The first area focuses on service-based software construction using orchestration and choreography. Hereby, business processes are loaded with services and executed. A special aspect

Appendix

of the description within these two approaches is the Behavioural interface. Thereby, time, data flow and data supervision are described within processes (Ghezzi, 2005; Papazoglou et al., 2007).

Orchestration and choreography show two different, but also complementary approaches to the composition of commercial processes. On account of the popularity of these approaches, process orientation is often seen in connection with service-based construction. An orchestration describes the interaction with internal and external Web Services at the message level from the point of view of a compiled commercial process (Peltz, 2003). The W3C consortium defines in its Web Service glossary (W3C, 2004, online):

”An orchestration defines the sequence and conditions in which one Web Service invokes other Web Services in order to realize some useful function. I.e., an orchestration is the pattern of interactions that a Web Service agent must follow in order to achieve its goal.“

Regarding Busi et al. (2005) there are three characteristics in which orchestration and choreography differ: Compiled processes, interaction design and activity state.

Kreuzer (2005, online) shows this differentiation as follows: *”Compiled processes are not specified in choreography, interactions are described from a global view without a central control unit, involved web services administer their activity state decentralised. By contrast, orchestration is based on the existence of a central control unit (orchestrator engine). The orchestrator engine serves the coordination of the interactions and saves the activity state of the involved processes centrally. Orchestration describes compiled processes based on the assumption that the processes are “animated” by engines.“*

Figure 154 shows the cooperation of orchestration and choreography (Peltz, 2003, p. 47).

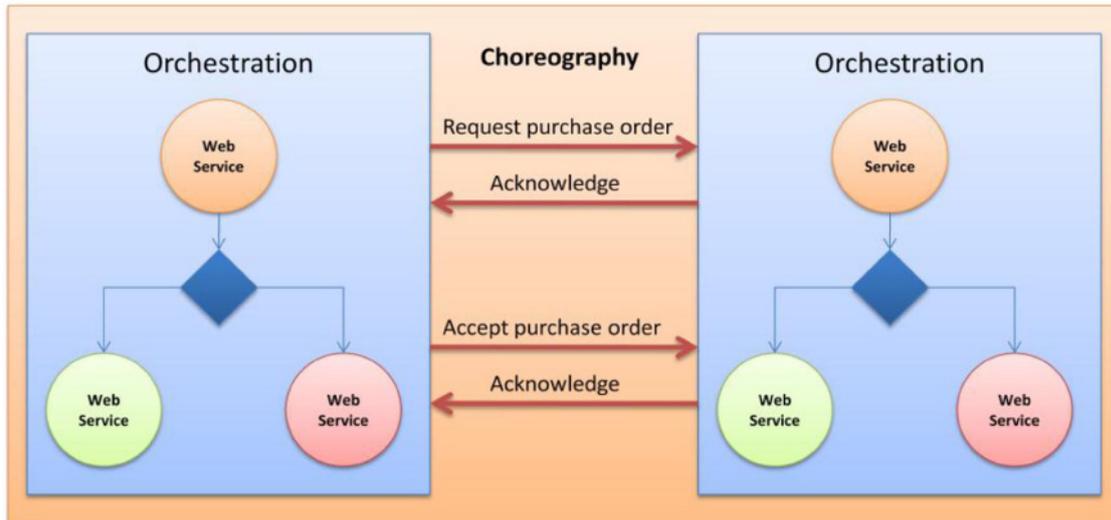


Figure 154 - Web Service Choreographie and Orchestrierung (Peltz, 2003, p. 47)

Kreuzer (2005, online) defines Behavioral interface as follows: "At the level of service composition, behavioral interfaces can be distinguished beside choreography and Orchestration as another point of view [...]. A behavioral interface grasps the behavioral aspects between interactions and complements with it the interface description which is delivered for a web service by a WSDL document. In contrast to choreography the global look at the interactions is cancelled, because the interactions are looked at only from the point of view of a process. Like choreography, behavioral interface describes no internal processes, but only outwardly visible ones. Behavioral interface describes dependence which can appear: in the data flow, with time dependence, with message correlation and with transactions." Kreuzer (2005, online)

Arkin et al. (2007) indicate, that WS-BPEL, (e.g., behavioral interfaces in ebXML, are used as abstract processes like collaboration protocol profile).

Appendix

The second area of the research is directed on methodologies to design or construction time. A methodology is described by Lamsweerde (2000). Hereby, an exact requirement analysis is carried out. At the end, this is conveyed into a chart which consists of operations and/or components. The operation components (which also show the purposes of the stakeholder) are replaced later with Web Services (Kaabi, Souveyet and Rolland, 2004; Penserini et al., 2006). Kumar et al. (2007) points further out that nowadays such an estimate falls in the area of the semantic web. Therefore, Web Services are not necessary for the construction time, but only the demands are relevant for a Web Service, which is dynamically searched and integrated if necessary based on semantic descriptions.

Nevertheless, in literature another area can be identified: This is the use of services within development environments or typical programming languages (i.e., C# or Java). Giambiagi et al. (2006) indicate how programming languages adapt the service-based approach. Hereby, most of the time, the approach is chosen which integrates services in a way that does not change the programme behaviour or the programming language is chosen.

Building up on the state of the art approaches different applications were developed to realise this form of construction. For the development of service implementations, based on orchestration or choreography, different software producers provide suitable applications. International Business Machines Corporation (IBM) Websphere and Microsoft Biztalk Server, for example, serve for the modelling and execution of processes which were augmented with Web Service.

In the area of the adaptation of services through programming languages the widespread development environments, as for example Eclipse and Visual Studio are a good examples. Both development environments deal with object-oriented languages. Web Services can be added and

Appendix

can be used like normal libraries, (i.e., the software engineer uses the Web Service as a class to generate an object instance). Then the actual methods, offered by the Web Service, are displayed as methods of the instantiated object. (Microsoft, 2012e; SUN, 2008)

Some technical conditions are necessary to use services. In the area of service-based construction some description languages were developed in dependency on the approaches. In the area of orchestration, among other things, BPEL4WS and Business Process Modelling Language (BPML) are found. These XML based description languages are used to define (commercial) processes. In addition, both languages command the possibility to integrate Web Services into processes. (Andrews et al., 2003; Peltz, 2003)

The XML based language WS-CDL is a description language in the area of choreography. With the help of this language the interaction between processes can be defined exactly.

Another language is ebXML. The description of electronic commercial processes with the help of XML is the purpose here. Thereby, Costs and expenditure for small business or consortia should be lowered. EbXML defines some standards, for example (based on Clark et al., 2001):

- ebXML architecture (ebXML Technical Architecture Specification) an XML schema for commercial processes (business process Specification schema)
- a Registry Service (Registry Service Specification) with a Registry Information Model
- ebRIM a Message Service (Message Service Specification).

RosettaNet constitutes a competing development to ebXML. The purpose of this project is to automate commercial processes between suppliers. Each process can be described and implemented on the basis of the RosettaNet Implementation Framework (RNIF). The documents and transactions between the process steps are created with the help of the message Guideline

Appendix

document (professional level) and the XML-Message-Guideline-Document (technical level; RosettaNet, 2004).

Beside the description languages of the processes and the operations, the descriptions of services and their interfaces are also relevant. Siedersleben (2006) shows that the exact description of interfaces belongs to the most relevant points. Today, description languages like Web Service Choreography Interface (WSCI) and Web Service description language (WSDL) are used for the description of services (including the interfaces).

As with components the implementation of services is carried out with conventional technologies and methods. (Bieber and Carpenter, 2001)

Another basic module is the architecture in which a service is operational. Beside the Universal Description, Discovery and Integration (UDDI) architecture other platforms exhibit other or rather similar structures. Benatallah et al. (2005) indicates a line of platforms, as for example:

- CMI (Collaboration Management Infrastructure)
- SELF-SERV
- DySCo Framework
- eFlow and CrossFlow (Casati and Shan, 2001; Grefen et al., 2000)

Collaboration management infrastructure is according to Schuster et al. (2000) an approach in which services with certain parameters are defined. In addition, an infrastructure is shown by means of a State-Machine approach. Within this infrastructure services can not only be assembled to applications, but also be exchanged at the runtime. Certain parameters are here assigned to a service. The system selects a suitable service to the runtime.

Appendix

Self-Serv is an approach used for a dynamic service architecture on the basis of a Peer2Peer (P2P) network. Three kinds of architecture units are distinguished here: elementary and composite services and service communities. Elementary services are single services which do not call other services. Composite services are a combination of different services. Service communities are repositories which contain services and distribution logic. Basically, at an inquiry to a community service, the architecture decides by means of business logic, which way the call has to go. This means depending on certain parameters (e.g., availability) it is decided which unit of the community processes this service. With composed services, this is carried out for each service individually. (cf. Sheng et al., 2002)

Piccinelli, Finkelstein and Williams (2003) show a Framework which carries out the orchestration at business level. The underlying technical expenditure is carried out by means of the Dysco-Framework. The aim here is to make the Web Services interchangeable within a workflow without the expenditure of manual updates.

Some literature shows in the area of service orientation which research areas still have to be examined. Four big subject areas can be identified (based on Papazoglou et al., 2007): Service foundation, service composition, service management, and service design / service engineering

Service Foundation (Papazoglou et al., 2007): In the area of service foundation, especially subject dynamics, quality and infrastructure support are asked for. Papazoglou et al. (2007) notes that in order to permit dynamic (re)configurable services architectures at the runtime, an improvement to the service discovery has to be made. This also assumes a research of infrastructure support of data, process and application integration.

Service Composition (Papazoglou et al., 2007): Particularly subjects from the area of business are desirable in the area of service composition. The subjects: comparability, use, and availability constitute the largest research areas. At this point the following services are listed: semantic, typed, and plausible services. Since there is no standard for quality characteristics in this area yet, researches for possibilities of the quality of service are also pending. Accompanying this, the comparability, which serves as a basis for the research area „autonomous composition of services“, is also to be examined.

Service Management (Papazoglou et al., 2007): In the area of service management, particularly the investigation areas are interesting which deal with the independence and the automation of services. This means services should be self-configuring and, thereby, adapt themselves to their environment or to the context (self-configuration services). Additional interest is in services which configure themselves automatically. This means self-analysis and an independent repair of services (self-healing services). Self-optimising services are based on the same idea. Here, an independent analysis and criteria are also necessary (see Heckmann, 2007). Self-protecting services are also interesting, (i.e., an implementation of security aspects within services).

Service Design and Service Engineering (Papazoglou et al., 2007): The suggested research areas in the area of Service engineering, are directed mainly at the dealings with services in software development. Thereby, it is pointed out that there is a lack of design principles for the creation of services. In addition, there is only rudimentary support or methods for the integration of service development in conventional software development. Xu et al. (2007), Skogan, Grønmo and Solheim (2004) and Bauer and Huget (2005) show at the example of the UML, that services can be

displayed with UML. However, there is no notation for services in UML. It is also indicated that there is a lack of analysis possibilities for services.

In addition, the question of service-based control within the development is pending, concerning the use of services within development methods.

Furthermore, some trends show up. Connecting possibilities of service with component technology is a particularly distinctive trend. Breivold and Larsson (2007) point to a comparison framework as a base to the connection between the service-based and component-based approaches. Hereby, two scenarios are focused:

1. Service-based architectures within component-based software development.
2. The possibilities of service-based architectures for system development.

Apperly (2003) and Stojanović (2005) indicate, for example, that context-dependent components can be hidden behind services. The disadvantage of the context dependency is encased. Kotonya, Hutchinson and Bloin (2004) show a hybrid modelling approach for both technologies. This approach extends over the formal description of standardised components and the representation of a procedure model for the development of hybrid systems. Jiang and Willey (2005) show a similar approach in their article. A multi-tier architecture with a component- and a service-based layer is displayed as a system-architecture here. Figure 155 shows this system architecture.

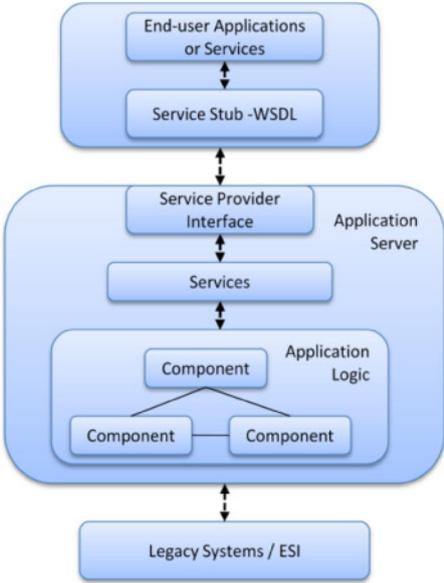


Figure 155 – Multi-tier architecture using services (based on Jiang and Willeam 2005)

Furthermore some trends appear, which try to extend the use of service-based construction or service, as for example Conversation-Driven Composition (Maamar, Sheng and Benatallah, 2004).

E. Schneider Electric internal reuse study and key notes

The thesis refers to two internal studies at Schneider Electric. The first one was conducted in 2009. The aim was to identify the reuse of hardware and software units. Results of this study were presented in a key note presentation at the 17th IEEE International Conference on Emerging Technologies & Factory Automation (ETFAs) in September 2012. Dr. Ronald Schoop representing Schneider Electric was the responsible person for the study and the keynote presentation speaker. Schneider Electric permits the use of figures and tables of this keynote presentation.

The second study was conducted in 2012. The aim was to identify positive and negative impacts of reuse in distributed development projects of different business units. The results of this study were not published at the submission of this thesis (March 2013). Dr. Ronald Schoop representing the company Schneider Electric is the responsible person for the study. Schneider Electric allows the use of figures, but with constraints. Names of persons, products or customers have been removed.

Schneider Electric does not permit the inclusion of the reuse study white papers for use in this thesis. For the review committee of the University of Plymouth, Dr. Ronald Schoop is the contact person for further information about the study.

Contact information:

Dr. Ronald Schoop | Schneider Electric | Industry Business - STS | VP Technology | Edison Group Master Expert (L3)

Phone: +49 (0) 9391/606-2390 | **Mobile:** +49 (0) 172 65 96 423 | **Fax:** +49 (0) 9391/606-2158 |

Email: ronald.schoop@schneider-electric.com | **Site:** www.schneider-electric.com |

Address: Schneider Electric Automation GmbH, Schneider Platz 1, 97828 Marktheidenfeld, GERMANY

F. Case study measurement value results

This section shows the tables including the measured values for the case study. In Chapter 6 the tables are shown including the average values. The tables in this section show the measured values for each user. 12 different SCAs measured.

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	1:28:42	0:39:50	9	35	1	1
User M (2)	1:32:01	0:44:11	10	33	1	1
User M (3)	2:10:32	0:59:38	19	70	1	1
User M (4)	0:49:02	0:28:22	9	41	1	1
User P (1)	0:03:28	0:02:13	2	8	1	1
User P (2)	0:03:01	0:01:37	2	5	1	1
User P (3)	0:03:05	0:02:21	2	7	1	1
User P (4)	0:02:45	0:02:22	2	5	1	1
Expert M	0:13:21	0:03:43	3	25	1	1
Expert P	0:02:34	0:01:54	2	6	1	1

Table 63 - Measured values for the DPWS Java Stack transformation

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	0:18:32	0:06:21	9	52	1	1
User M (2)	0:45:05	0:20:34	7	49	1	1
User M (3)	0:30:45	0:13:56	7	46	1	1
User M (4)	0:18:15	0:09:10	6	56	1	1
User P (1)	0:02:51	0:01:55	2	7	1	1
User P (2)	0:03:14	0:02:34	3	7	1	1
User P (3)	0:03:07	0:02:43	2	6	1	1
User P (4)	0:02:45	0:02:22	1	5	1	1
Expert M	0:07:45	0:02:45	4	23	1	1
Expert P	0:02:46	0:01:56	3	6	1	1

Table 64 - Measured values for the DPWS Java Stack integration

Appendix

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	0:25:36	0:16:23	5	19	1	1
User M (2)	0:40:13	0:29:52	8	22	1	1
User M (3)	0:31:29	0:18:45	8	24	1	1
User M (4)	0:29:06	0:14:51	8	22	1	1
User P (1)	0:03:17	0:02:38	3	8	1	1
User P (2)	0:02:26	0:01:41	3	8	1	1
User P (3)	0:02:49	0:02:00	2	8	1	1
User P (4)	0:03:14	0:02:44	2	5	1	1
Expert M	0:09:08	0:07:21	2	7	1	1
Expert P	0:02:01	0:01:54	2	4	1	1

Table 65 - Measured values for the DPWS C Stack transformation

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	1:13:55	0:44:22	16	22	1	1
User M (2)	1:36:35	0:56:16	12	15	1	1
User M (3)	1:42:04	1:06:10	22	37	1	1
User M (4)	1:10:03	0:37:17	9	53	1	1
User P (1)	0:03:18	0:01:30	2	8	1	1
User P (2)	0:03:04	0:02:10	3	9	1	1
User P (3)	0:03:20	0:02:22	2	7	1	1
User P (4)	0:02:45	0:02:22	2	7	1	1
Expert M	0:37:56	0:04:24	4	20	1	1
Expert P	0:02:23	0:01:16	4	7	1	1

Table 66 - Measured values for the DPWS C Stack integration

Appendix

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	0:19:14	0:09:31	5	12	1	1
User M (2)	0:21:52	0:13:15	8	26	1	1
User M (3)	0:17:45	0:13:55	9	17	1	1
User M (4)	0:29:18	0:16:58	9	36	1	1
User P (1)	0:03:25	0:01:48	2	6	1	1
User P (2)	0:02:54	0:02:19	2	6	1	1
User P (3)	0:03:45	0:03:02	2	9	1	1
User P (4)	0:02:45	0:02:11	2	5	1	1
Expert M	0:03:43	0:02:07	3	7	1	1
Expert P	0:02:45	0:02:03	2	4	1	1

Table 67 - Measured values for the Log4J transformation

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	0:49:34	0:21:03	14	39	1	1
User M (2)	0:45:33	0:21:03	18	21	1	1
User M (3)	0:17:45	0:14:25	9	17	1	1
User M (4)	0:12:34	0:06:44	3	7	1	1
User P (1)	0:03:07	0:02:45	3	4	1	1
User P (2)	0:02:46	0:02:19	2	5	1	1
User P (3)	0:02:32	0:03:02	4	5	1	1
User P (4)	0:02:46	0:02:11	2	4	1	1
Expert M	0:03:56	0:02:15	2	7	1	1
Expert P	0:02:15	0:01:45	2	5	1	1

Table 68 - Measured values for the Log4J integration

Appendix

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	0:22:41	0:19:45	19	31	1	1
User M (2)	0:16:00	0:14:32	22	29	1	1
User M (3)	0:19:06	0:14:23	15	25	1	1
User M (4)	0:26:45	0:21:34	20	42	1	1
User P (1)	0:02:45	0:02:17	2	6	1	1
User P (2)	0:02:54	0:02:31	2	2	1	1
User P (3)	0:03:05	0:02:45	3	8	1	1
User P (4)	0:02:43	0:02:23	2	6	1	1
Expert M	0:05:13	0:01:10	2	9	1	1
Expert P	0:03:04	0:02:45	2	7	1	1

Table 69 - Measured values for the EWS .NET transformation

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	0:28:22	0:05:45	20,27%	5	9	1
User M (2)	0:41:56	0:04:08	9,86%	4	52	1
User M (3)	0:36:10	0:29:01	80,23%	6	10	1
User M (4)	0:26:54	0:20:12	75,09%	5	7	1
User P (1)	0:02:30	0:01:56	77,33%	3	6	1
User P (2)	0:02:54	0:02:31	86,78%	2	5	1
User P (3)	0:02:10	0:01:45	80,77%	3	7	1
User P (4)	0:02:34	0:02:20	90,91%	2	5	1
Expert M	0:09:08	0:05:06	55,84%	4	7	1
Expert P	0:02:01	0:01:54	94,21%	2	4	1

Table 70 - Measured values for the EWS .NET integration

Appendix

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	0:19:58	0:15:03	12	21	1	1
User M (2)	0:18:09	0:15:08	14	24	1	1
User M (3)	0:26:10	0:19:01	19	28	1	1
User M (4)	0:26:12	0:21:11	14	19	1	1
User P (1)	0:03:01	0:02:13	2	7	1	1
User P (2)	0:02:54	0:01:58	2	6	1	1
User P (3)	0:02:10	0:01:50	3	7	1	1
User P (4)	0:02:30	0:02:05	3	7	1	1
Expert M	0:09:08	0:05:06	2	7	1	1
Expert P	0:02:46	0:02:16	3	7	1	1

Table 71 - Measured values for the EWS J transformation

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	0:53:23	0:18:03	9	40	1	1
User M (2)	0:41:56	0:29:05	9	52	1	1
User M (3)	0:36:10	0:29:01	16	37	1	1
User M (4)	0:26:54	0:20:12	23	7	1	1
User P (1)	0:02:30	0:01:56	3	7	1	1
User P (2)	0:02:54	0:02:31	2	6	1	1
User P (3)	0:02:10	0:01:45	3	7	1	1
User P (4)	0:02:30	0:02:20	2	6	1	1
Expert M	0:15:03	0:05:06	4	7	1	1
Expert P	0:02:51	0:01:55	4	5	1	1

Table 72 - Measured values for the EWS J integration

Appendix

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	0:27:50	0:19:05	31	49	1	1
User M (2)	0:28:31	0:21:02	19	47	1	1
User M (3)	0:23:00	0:16:36	21	34	1	1
User M (4)	0:21:02	0:19:02	34	42	1	1
User P (1)	0:02:30	0:01:55	6	7	1	1
User P (2)	0:01:59	0:01:34	4	6	1	1
User P (3)	0:02:22	0:01:46	6	7	1	1
User P (4)	0:03:05	0:02:21	4	8	1	1
Expert M	0:12:35	0:05:56	3	9	1	1
Expert P	0:02:02	0:01:50	3	4	1	1

Table 73 - Measured values for the Log4.NET integration

	Time needed	KR Time	KR used	Task done	Success	Valid
User M (1)	0:18:04	0:17:00	15	24	1	1
User M (2)	0:14:45	0:12:48	17	27	1	1
User M (3)	0:24:09	0:17:23	12	37	1	1
User M (4)	0:19:56	0:15:23	21	39	1	1
User P (1)	0:02:55	0:02:33	2	6	1	1
User P (2)	0:03:05	0:02:46	2	7	1	1
User P (3)	0:02:31	0:02:03	3	7	1	1
User P (4)	0:02:55	0:02:02	3	8	1	1
Expert M	0:06:54	0:05:06	2	7	1	1
Expert P	0:02:29	0:01:56	2	6	1	1

Table 74 - Measured values for the Log4.NET integration

G. Additional identified problems and requirements for future studies

The research of the thesis focuses on different knowledge problems and concludes that it is possible to support an inexperienced user in handling these problems. This is demonstrated in the example of three software construction activities (i.e., special variants of transformation, integration, and deployment). During the research, additional requirements were identified. In different discussions with other researchers (e.g., Ph.D seminars), these requirements were identified for future research, as extension of the main research of this thesis. The results of this section are not published yet.

In the first part of this section the identified problems are summarised. In the second part the relation of the problems to the three SCAC focused by primary research is explained. In the last part of this section further requirements are defined for each problem SCAC relationship.

g.1. Identified problems for software engineers

In the following subsection further problems of software engineers are summarised. These were identified during the Ph. D. research.

g.1.1. Software engineering problems for software engineers

For the software engineer, two kinds of problem arise: specialised problems of the single software construction forms, and problems of the construction forms in comparison. In the following text, the single specialised problems of the previous paragraphs are confronted and explained from the point of view of the engineer.

Problems of the different key concepts: Different construction forms are based on different key concepts. For the developer, in order to be able to make a suitable decision on the application, the

problem is to know the properties and peculiarities of these key concept problems as they arise. This requires suitable knowledge about the key concepts.

Problems of different technologies: Problems for developers are also positioned in the technology area. As previously shown, the construction forms already differ on account of their purpose architectures which show considerable technological differences. Nevertheless, it is necessary to connect single technical problems with each construction form in order to use them in the development.

Problems of the multitude: The problem of the multitude of programming languages, such as object orientation, is not found in the same manner with component-based and service-based construction. Though components and services can be created with conventional methods and technologies, this, however, is second-rate with the use of these ‘interfaces’ based construction forms. Particularly with service-based construction it is rather irrelevant how a service is implemented (Breivold and Larsson 2007). All construction forms have a multitude of concrete technology variants.

Component models and component worlds: The problem of component models and worlds is not found in this form in other construction forms. Though in the object-oriented construction, a strong (economic) relation to the programme paradigms is found, e.g., .NET and JavaEE. Nevertheless, this shows no oligopoly.

In service-based construction there is currently no such dependence. Such information from professional and/or market-political viewpoints, however, can be relevant for the developer (Szyperski 2002b). Service does show a kind of world perspective looking on different protocols.

Often, the most commonly used protocols are SOAP and REST (Singh and Huhns, 2005), and both are not compatible.

Problem of the availability: The problem of availability is of crucial importance for developers. To guarantee the operability, i.e., the frictionless execution and operation of software, all units of software have to be available. If a unit is not available, a programme must be able to react to it. With the handling of objects, in most cases local resources which can be verified, are meant. Though with the handling of components, local resources also exist in most cases, during the construction time, however, only the interfaces are handled. The existence of the resource is not always mandatory. This behaviour is even more pronounced with service-based construction (Breivold and Larsson 2007) and leads to the problem of the availability at runtime (Kumar et al. 2007).

Problems of the re-use and the design: Another problem for the developer arises from the enclosure or granularity of the reuse. While with components the granularity varies from very open (white box) to complete enclosure (black box), there is a pure black box system with services (Breivold and Larsson 2007). Objects, however, are customisable in terms of reusability, like components, but the basic principle of object-orientation is even more open (white box). Nevertheless, it is shown that the reuse in object-orientation is sparse. All three construction forms focus on the reuse, however, in different ways and with different possibilities. The developer has to consider these different modelling behaviours.

Problem of the market: The problem of the market, as described for components (Szyperski 2002b) is also true for the other construction forms. Especially in the example of services

(Fitzgerald et al., 2006). The units of modelling, which a developer selects or develops, are to be arranged in a vertical or horizontal market (Szyperski 2002).

Problems of the completeness: The problem of the completeness does not pose itself in the area of the components and services. The software engineer normally uses interfaces to execute functions of a service or component. But engineers have to know the structure of a component (including external dependencies). On the service side this is not relevant. (Breivold and Larsson 2007)

Problems of the context dependence: Here, similar problems show themselves. Since the implementation of services is encased, only a low dependence exists (Breivold and Larsson 2007). Often services provide all information (in an interface description) so a ‘client’ can create all needed dependencies by themselves.

With objects, the developer can influence the context dependence to a certain degree. Since these dependencies are present at different levels of the development, the developer has to know them intimately.

Problems of the different views: Within one of the three approaches the advantages and disadvantages for the developer stay the same. Breivold and Larsson (2007) discuss using different studies showing that a homogeneous use of the construction forms and the components of the modelling are rather unusual. In this case the different views mix and the developer has the problem in deciding which estimate to follow. Hereby the question arises, which methods and procedure models can be used? In the area of the service-oriented construction, suitable methods, and procedure models are missing (Fitzgerald et al. 2006).

g.1.2. Knowledge Management problems for software engineers

The problems in the area of knowledge management are directly connected to the work of software engineers. In this section the general problems are discussed from the perspective of software engineers.

Problem of knowledge storing: For software developers the problem of knowledge storing occurs if no external process is given (Boden and Avram 2009). The storing of data and information is not a problem in software engineering. Content management tools (e.g., GForgeGroup 2012) support the development process by storing this type of information. However, a commonly used tool or process to force the problem of storing knowledge, especially in the area of reuse, is missing. The result is the knowledge vaporisation effect (see Ven et al. 2006).

Problem of knowledge learning: The need to learn new knowledge is an relevant part of the vocation of a software engineer. This is based on two facts in the area of software development: the changing tasks and fast growing nature of technologies and information (Ajila 2006). Software engineers have different ways to learn such new knowledge. Typically professional or self-training sessions, magazines, or podcast support are common examples. However, the problem is the given time, money, and the learning possibilities of a person.

Problem of knowledge receiving: In addition to the experience about sources of information, a software engineer has to know how to reach or access such sources of information. Each repository system is in place to advance different approaches, e.g., it may be necessary to authenticate in some repository systems (Ajila 2006). Some systems offer standardised approaches such as web portals, while others use advanced specialised applications, in addition, different types or use.

Problem of knowledge search: For software engineers, the problem arises in the functionality of finding information. Search engines such as Google allow you to search in many different systems

for information. Search results of general search engines such as these provide a variety of results that do not match the desired result.

Normally, software engineers are familiar with their own special in-house or free open source repository where they are able to search for information. The number of internal corporate repositories increases with the size of the company. Normally a software engineer is not aware of all existing repositories in their environment (i.e., in a global company). This is particularly true for private repository of other software engineers.

Problem of knowledge using: The problem of the application of knowledge depends on the different ways a software engineer wants to perform, as well as the objects that are necessary for implementation. Thereby, the focused problem is to interpret the given information and knowledge parts correctly for (re)use.

Problem of knowledge distribution: In different projects software engineers have to share their knowledge. Typically this can be done by arranging meetings supported by different presentation media (i.e., audio, video, or pictures) or by using knowledge management tools. Next to the discussed problems of searching and using of knowledge the problem arises to distribute knowledge in a way that it can be understand correctly by others. In contrast to the problem of knowledge use, the software engineer who is the knowledge owner has to look for it (see Taweel et al. 2009; Boden and Avram 2009).

g.1.3. Industrial problems for software engineers

The problems in the area of industrial informatics are more practical and directly connected to the work of software engineers. In this section the problems shown in Section G.1 are discussed from the perspective of such engineers.

Problem of localisation (single software engineer): The industrial example in Section 3.2.1.2 creates a problem of localisation for software engineers (Bosch and Bosch-Sijtsema, 2010). One team member can be located on a different site than others of the same team. To exchange data is not only a problem of different time zone or culture but also a question of communication (see Taweel et al., 2009).

Problem of localisation (multiple teams of software engineers): The problem of localisation also occurs for a multiple teams of software engineers (see Taweel et al., 2009). Different teams are placed on different locations. Teams, as well as, single software engineers, have to communicate with each other.

Problem of missing knowledge exchange: The example in Section 3.2.1.2 shows that knowledge exchange between teams is missing. This can also be found in the analysis of other real development projects (see Boden and Avram, 2009 and software engineers may not be able to work with other solutions than the solutions they already know. In a worst case scenario people are not able to fulfil their work or cooperate with teams using different versions of the same knowledge (based on interpretation issues; see Qu, Ji and Nsakanda, 2012; Choi, Lee and Yoo, 2010).

Problem of reachable knowledge: The problem of missing knowledge exchange is also based on the problem that software engineers use different types of repositories. These types of repositories reach from handmade notes or files on the personal hard disc to a team or department, companywide or community repository system (see Ajila, 2006; Ha, Sun and Xie, 2012). To know how to connect to these repositories is a problem. A software engineer (or a team) has to know where these repositories may found and how to use them. In this case the use of a repository includes the topic of security.

Problem of missing time: Reuse needs time and has to be planned (see Ajila, 2006; Frakes and Isoda, 1994). This is valid from the perspective of the reuse creator (including the role of the supporter). Next to the ‘normal’ fact that people are overloaded with work, reuse estimates time especially in following activities:

- Creation
- Reuse (Selection)
- Reuse (Adaption)
- Reuse (Integration)

Software engineers have to handle the time they have and the time they need to reuse a software unit.

Problem of return of invest: In the example of Schneider Electric the positive effect of reuse was created after the third or fourth reuse iteration of a software unit. The unknown time and the high risk of making fundamental failures for future reuse may be reason why companies does not use ‘reuse’ in their development projects in the past.

Problem of missing support: The reuse of ‘unknown’ software units may speed up with the support of experts. If such experts are not available, the software engineer or a development team is under constraint to investigate the possibilities and limitations of a software unit. In the example of Schneider Electric the support was necessary in the most reuse iterations.

The first idea was to use the requirements to analyse existing approaches and their realisation for the level of handling knowledge of software construction activities. In this section the problems will be summarised. In the second section requirements will be explained briefly using the example of the three example SCAs of the main research. This section can be used for further discussions.

Table 75 summarises the problems.

		Single	Combined	Relation to other problems
Software Engineering	Problems of the different key concepts	X		
	Problems of different technologies	X	X	X
	Problems of the multitude	X	X	X
	Component models and component worlds	X	X	
	Problem of the availability	X	X	X
	Problems of the reuse and the design		X	X
	Problems of the completeness	X	X	X
	Problems of the context dependence	X	X	X
Knowledge	Problems of the different views		X	
	Problem of knowledge storing	X		
	Problem of knowledge learning	X		
	Problem of knowledge receiving	X		
	Problem of knowledge search	X		
	Problem of knowledge using	X		
	Problem of knowledge distribution	X		
Industrial	Problem of localization (Single)	X	X	X
	Problem of localization (Multi)	X	X	X
	Problem of missing knowledge exchange	X	X	X
	Problem of reachable knowledge	X	X	X
	Problem of missing time	X	X	X
	Problem of missing support	X	X	X

Table 75 - Single or combined visualisation

g.1.4. Problem selection for reuse activities

In previous sections different problems of the three perspectives on reuse in software engineering, handling of knowledge, management and industrial context were shown. Section G.1 demonstrated how these problems have common effects on software engineers. In the following, the problems of software engineers will be discussed on the basis of concrete reuse activities of software units. The focus changes from the general view on problems of software engineers to concrete activities

Appendix

which include the possibilities for such problems. Table 76 shows an overview of the discussed problems and the three focused reuse activities. This mapping will be explained in this section.

Note: This mapping shows typical problems in the three reuse activities and is represents the perspective of the author of this thesis based on the previous discussions. Specialised SCACs may be shown differently to this perspective.

		Transformation	Integration	Deployment
Software Engineering	Problems of the different key concepts	X	X	X
	Problems of different technologies	X	X	X
	Problems of the multitude	X	X	X
	Component models and component worlds	X	X	X
	Problem of the availability	X	X	X
	Problems of the reuse and the design			
	Problems of the completeness	X	X	X
	Problems of the context dependence	X	X	X
	Problems of the different views	X	X	X
	Problem of knowledge storing	X	X	X
Knowledge	Problem of knowledge learning	X	X	X
	Problem of knowledge receiving	X	X	X
	Problem of knowledge search	X	X	X
	Problem of knowledge using	X	X	X
	Problem of knowledge distribution	X	X	X
Industrial	Problem of localization (Single)	X	X	X
	Problem of localization (Multi)	X	X	X
	Problem of missing knowledge exchange	X	X	X
	Problem of reachable knowledge	X	X	X
	Problem of missing time	X	X	X
	Problem of missing support	X	X	X

Table 76 - Problems in the focused reuse activities

Basically Table 76 shows that each of the problems is related to each of the different Software Construction Activities. But the table shows no differences between the SCACs. To understand the different impact of the problems to the SCACs it is necessary to describe type of impact to the

SCAs. For that reason a general structure of activities is used to explain the relation between the focused problems and the different activities. Therefore an activity has a preparation phase for information received, an input configuration phase, an execution phase, and an output.

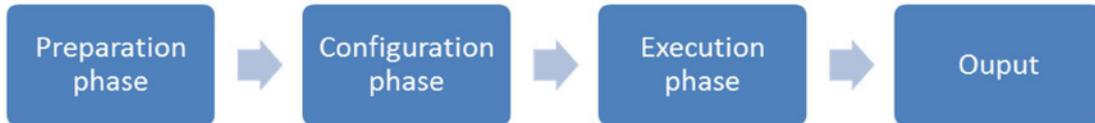


Figure 156 - General structure of activities used for explanation

g.1.4.1. Problems and transformation activities relationship

In this area, expertise is demonstrated by knowing exactly how an appropriate transformation application has to be prepared so that it can be executed. This includes information and parameters needed for the performed transformation to produce the correct result and the distribution of the result necessary knowledge for the result creation.

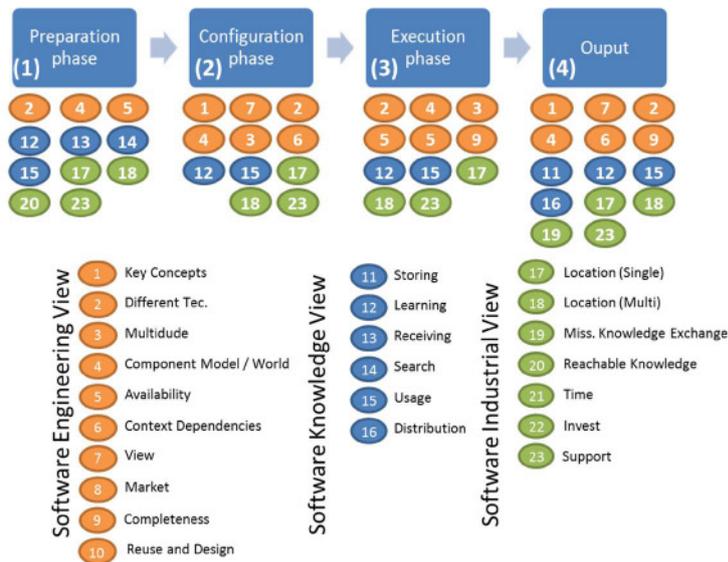


Figure 157 - Relation between a general transformation activity and focused knowledge based problems

Appendix

A based on the structure of a general activity, a general transformation activity can be explained as follows (c.f. Figure 157).

(1) In the preparation phase a user searches and receives all necessary information for the preparation and execution of a transformation activity. (2) The configuration phase includes the installation, configuration, and set of parameters of the transformation tool. (3) The execution phase includes the start of the transformation application. (4) The result of a transformation is another software unit.

Figure 157 shows the relation between the focused problems and the 4 different parts of a general transformation activity. This relationship is now explained in more detail using the transformation activity example (see Section 3.1.1).

The problem of key concepts and the problem of the view mainly occur for software engineers during the configuration phase and for the output. In the configuration a key concept may be expressed by a parameter. The example in Section 3.1.1 does not require such information. As a result, the engineer has to select a transformation tool handling the key concepts of a unit. The output of a transformation also includes different key concepts. In the example of IKVM a user has to decide if a library or an executable has to be the resulting software unit.

The problem of component models/component worlds and the problem of different technologies occur for the software engineer in the same way as the problem of key concepts. It might be necessary to set technology or component model information for the input software unit and the output software unit in the preparation phase. Therefore, the engineer has to receive this information before in a preparation phase. Also, perhaps the problem occurs for the transformation tool itself. For transformation tools different technologies may exist. For example, a

Appendix

transformation of the Java based DPWS libraries into .NET libraries may be performed by using IKVM or an MDD based application. The engineer has to know which transformation technology creates a useable result. The question of the component world and model is also relevant for the transformation result. The engineer's transformation activity might focus a specific technology or component model result (see example in Section 3.1.1).

The multitude problem occurs in the example of transformation predominantly in the configuration phase. There perhaps exists a multitude of single software units or software unit types. Based on the used transformation tool this has to be part of the configuration. Also a multitude of transformation applications may exist which differ in their feature level. An example is the SVCUtils which exists in different versions based on the existing versions of Microsoft .NET.

The problem of reuse and design, as well as the problem of the market, are not relevant for transformation activities.

The problem of context dependencies occurs in the configuration phase. Tools like IKVM or SVCUtils need information about existing dependencies and their locations. The use of the dependencies is necessary to gain a full overview of a software unit. Sometimes the dependencies will be copied into the transformation result. In the example of IKVM and DPWS the transformation results have new dependencies which are mainly IKVM files. A software engineer has to know which dependencies are necessary and how they have to be copied or configured for the transformation tool.

The problem of available occurs from the perspective of this analysis in the preparation and execution phase. The parts of the software unit have to be available for search and download. During

Appendix

the execution of the transformation these parts have to be available for the transformation tool. Additionally the transformation tool has to be available also.

The problem of completeness is relevant in the execution phase and the output of a transformation. In the execution phase information and parts of a software unit have to be completed to perform the transformation. The transformation result has to be complete also, to be shipped to another reuse activity or a repository.

The problem of knowledge distribution and knowledge storing occurs for a software engineer in the output of a transformation activity. If the output is created, an engineer might want to store the transformation activity knowledge for personal reuse or for others. This leads to the problem of distribution.

The problem of knowledge learning occurs in each part of an activity. An engineer has to learn how to find all necessary information, prepare the tool and the software unit, execute it and handle the transformation result.

In a transformation activity the problem of knowledge receiving and searching occurs in the preparation phase. A software engineer has to handle these problems before starting the configuration phase. After receiving the necessary knowledge it can be used. The problem of using knowledge occurs for an engineer in the preparation phase. The knowledge is the combination of information with the transformation application that is using the information to create an output. In the IKVM/DPWS example the user sets all parameters necessary to start an application which transforms Java byte code to .Net byte code.

Appendix

The problem of the use of knowledge mainly occurs in the configuration and execution phase. The user has to know how to configure a transformation tool and how to execute it. The IKVM example shows that a user has sometime to add information or data to the output manually.

The problem of localisation (single/multi) occurs for the software engineer in all parts of a transformation activity. First of all the repositories to search and retrieve information might be localised in different locations.

The same scenario is valid for the problem of reachable knowledge. Also the transformation tool itself may be localised in other locations. As a result, it is difficult to execute the tool. If this scenario occurs, the problem extends itself to the preparation phase. The engineer has to know how to access the environment of the tool to configure it. If an engineer wants to distribute their transformation result, it is necessary to know how to distribute it to other locations. This is part of the problem of knowledge exchange in industrial environments. Additionally the engineer has to know other repositories, their location, and handling to exchange the knowledge of the transformation activity.

If an engineer want to distribute his transformation result it is necessary to know how to distribute it to other locations. This is part of the problem of knowledge exchange in industrial environments. Additionally the engineer have to know other repositories (including their location and handling) to exchange the knowledge of the transformation activity.

The problem of support occurs in each part of a transformation activity. A software engineer may need support during the preparation, configuration, and execution phases, and the handling of the output of a transformation activity.

g.1.4.2. Problems and integration activities relationship

In this area, a software engineer’s expertise is demonstrated by knowing exactly how to integrate a software unit into an IDE. This includes knowledge about different techniques and the configuration of the IDE and the software unit. The result is a development project extended with a new software unit.

A based on the structure of a general activity, a general integration activity can be explained as follows (c.f. Figure 158).

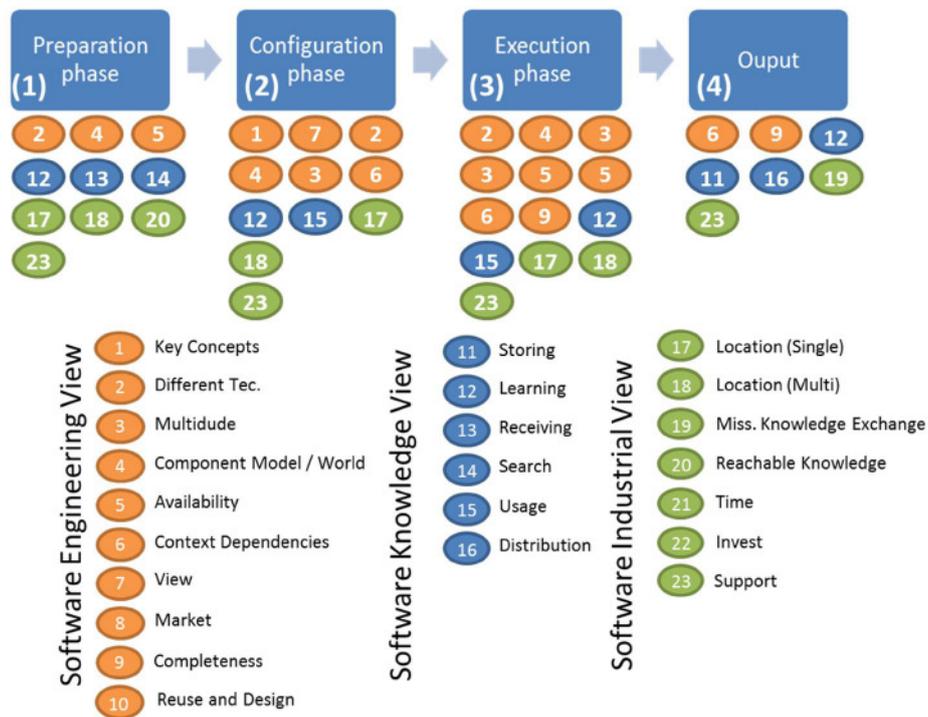


Figure 158 - Relation between a general integration activity and focused knowledge based problems

(1) In the preparation phase, a user searches and receives all necessary information for the configuration and execution of an integration activity. (2) The configuration phase includes the configuration of the IDE and the preparation (including the creation of file and folder structure) of

the software unit. (3) Also in this phase the software engineer has to decide on the type of integration (i.e., referencing, only copy, and so on). The execution phase includes the manual integration of the software unit or parts of it into the IDE by the software engineer. (4) The result of integration is a software development project including the integrated software unit.

Figure 158 shows the relation between the focused problems and the 4 different parts of a general integration activity. This relationship is now explained in more detail using the transformation activity example (see Section 3.1.3).

The problem of key concepts and the problem of the view mainly occur for a software engineers during the preparation and configuration. In the preparation a key concept may be relevant to identify the correct software unit (see Section G.1.1). It is expressed by a search request (see Picot, 2003). The software engineer has to know the key concepts of a unit that is used in a special IDE. The software engineer might use this experience to configure the IDE or to prepare the software unit in the configuration phase. The example in Section 3.1.3 does not require such information. But an example can be identified by analysing Visual Studio and Eclipse. Both IDEs are able to handle libraries, source-code, and service information. For the user it is relevant to know how these IDEs handle these key concepts or views. Web Services in Visual Studio are handled like components. As a result, the user does not notice the difference, but have to know it.

The problem of component models/component worlds and the problem of different technologies occur for the software engineer in the same way as the problem of key concepts. It might be necessary to set technology or component model information in the preparation phase. Also such information is relevant to identify a software unit during the software unit search. However, such information may be used to identify the correct IDE and/or configure the IDE for the software unit.

Appendix

Therefore, the engineer has to receive this information before via the preparation phase. Perhaps the problem occurs also for the IDE itself. It might be that there are different technologies or component worlds for IDEs. Good examples are .Net and Java libraries that are used in Visual Studio (for .Net libraries) and Eclipse (for Java libraries). This shows that the component world problems of software units also exist in the area of IDEs. The engineer has to know which integration technology in an IDE creates a useable result. Also the technology itself can be different. An example is the .NET technology which now exists in 7 releases (.NET 1.0, 1.1, 2.0, 3.0, 3.5, 4.0, and 4.5).

The multitude problem occurs in the example of integration mainly in the configuration and execution phase. There may be a multitude (different versions) of single software units or software unit types. These units can differ in their structure or technology, for example. The software engineer has to know how to handle such multitudes. Also a multitude of IDEs may exist which differ in feature level. An example is Visual Studio. The versions Visual Studio (VS) 2008, VS2010, and VS2012 differ in their integration handling and in the support of technologies that can be supported.

The availability of a software unit is mainly relevant during the execution of integration. Often, this is combined with the problem of completeness. While classes and components have to be complete and available, services can be integrated by using only the interface information (e.g., WSDL file). Also, the availability of the IDE is necessary to fulfil the integration activity. In general, the availability of information and activity knowledge is necessary from the beginning of the preparation phase.

Appendix

Both the problem of reuse/design is not related by transformation activities from the perspective of the author.

The problem of context dependencies occurs mainly in the execution phase and for the output. IDEs, like Visual Studio checks dependencies in some cases, for example for the integration of service information and in cases of early and late binding. If the dependencies are not available, the integration will fail. For the integration of libraries as references, dependencies will not be checked. The error occurs if the software engineer tries to build the development project or at runtime of the software tool or a unit that is created by the development project. To create a valid integration, a software engineer needs the experience for such occurrences. Sometimes the IDE has to be configured in the configuration phase. An example is the path settings for Eclipse for additional paths where libraries (dependencies) can be searched. Another dependency example is shown in the integration example in Section 3.1.3. The software unit requires some other software units integrated in the development projects (IKVM files). Additionally some other files have to be copied in the same directory. These files are and their location next to the main software unit is required by the main software unit (dependencies). Like in a transformation activity engineers has to know such dependencies.

The problem of completeness is relevant in the execution phase and the output of an integration activity. In the execution phase information and parts of a software unit have to be completed to perform the integration. The integration result has to be complete also, to be used during the compilation or the runtime of the development project.

Appendix

The problem of knowledge distribution and knowledge storing occurs from the output of a transformation activity. An engineer might want to store the integration activity knowledge for personal reuse or for others. This leads to the problem of distribution.

The problem of knowledge learning occurs in each part of an activity. An engineer has to learn to find all necessary information, prepare the IDE and the software unit, integrate it and learn about the structure of the integration result.

In an integration activity the problem of knowledge receiving and searching occurs in the preparation phase. A software engineer has to handle these problems before starting the configuration phase. After receiving the necessary knowledge it can be used. The problem of using knowledge occurs for an engineer in the configuration and execution phase. The knowledge is the combination of information with the IDE that is using the information to create an output. Also the user has to know how to use the IDE to create this output. The example in Section 3.1.3 shows that a software unit includes different elements that have to be integrated differently using different techniques of the IDE.

The problem of localisation (single/multi) occurs for the software engineer in three parts of an integration activity. First of all the repositories to search and retrieve information might be located in different areas. The same scenario is valid for the problem of reachable knowledge. Also the IDE itself may be localised on other locations. As a result, it is difficult to execute the integration. If this scenario occurs the problem extends itself to the preparation phase. The engineer has to know how to access the environment of the IDE to configure it. Next to the needed time for performing an integration activity the problem of time and the problem of ROI are not interesting for transformation activities.

The problem of reachable knowledge occurs mainly in the preparation phase. Also the IDE itself may be located in other areas. As a result, it is difficult to execute the tool. If this scenario occurs the problem extends itself to the preparation phase. The engineer has to know how to access the environment of the IDE to configure it.

If an engineer wants to distribute their integration result it is necessary to know how to distribute it to other locations. This is part of the problem of knowledge exchange in industrial environments. Additionally the engineer has to know about other repositories and their behaviour to exchange the knowledge of the integration activity.

Following the time needed for performing an integration activity, the problem of time is not interesting for integration activities.

The problem of support occurs in each part of an integration activity. A software engineer may need support during the preparation, configuration and execution phases, and the handling of the output of an integration activity.

g.1.4.3. Problems and deployment activities relationship

In this area, a software engineer's expertise is demonstrated by knowing exactly how to deploy a software unit into a device. This includes knowledge about different software units, deployment and device techniques, and the configuration of these technologies. The result is a deployed software unit in an embedded device.

A based on the structure of a general activity, a general deployment activity can be explained as follows (cf. Figure 159).

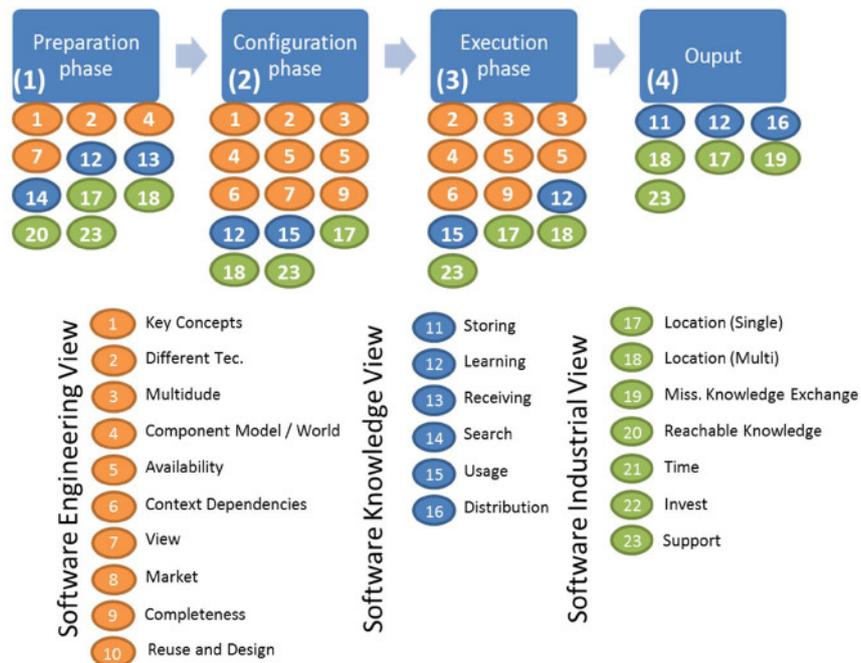


Figure 159 - Relation between a general deployment activity and focused knowledge based problems

(1) In the preparation phase a user searches and receives all necessary information for the preparation and execution of a deployment activity. (2) The configuration phase includes the configuration of the deployment tools and the preparation (including creation of file and folder structure) of the software unit. (3) Also in this phase the software engineer has to decide which deployment technology is compatible to the device and software unit. The execution phase includes the use of different tools preparing the input files and configuration, and finally the upload of the final package into the device. (4) The result of a deployment activity is the prepared software package and a deployed software unit.

Figure 159 shows the relation between the focused problems and the 4 different parts of a general deployment activity. This relationship is now explained in more detail using the transformation activity example (see Section 3.1.5).

The problem of key concepts and the problem of the view mainly occur for a software engineer during preparation and configuration. In the preparation phase a key concept may be relevant to identify the correct software unit. It is expressed by a search request (see Picot, 2003). The software engineer has to know the key concepts of a unit that is used in a special deployment environment. Often, the deployment tools are specialised to a specific key concept or technology. This experience of a software engineer might be useful in configuring the deployment tools or to prepare the software unit in the configuration phase. The three examples in Section 3.1.5 do require such specialised information. Each device or platform has different behaviours in handling software units.

The problem of component models/component worlds and the problem of different technologies occur for the software engineer in the area of deployment in different parts. It might be necessary to set technology or component model information in the preparation phase. Also such information is relevant to identify a software unit during the software unit search. However, such information may be used to identify the correct deployment platform or technology and/or configure the device or the platform for the software unit. Therefore, the engineer has to receive this information before in a preparation phase. Perhaps the problem occurs also for the device or the platform itself. It might be that there are different technologies or component worlds that exist for different devices. Often, the programming language C is used. There are compilers available in different IDEs and different platforms, but mostly the device vendors support specific environments. Also examples exist were

Appendix

vendor specific component models or technologies have to be used. This is shown in the deployment example in Section 3.1.5.

The multitude problem occurs in the example of deployment mainly in the execution phase. There may be a multitude (different versions) of single software units and software unit types. These can differ, for example, in their structure or technology. The software engineer has to know how to handle such multitudes. Also a multitude of devices or deployment platforms may exist which differ in their support and deployment process level. The example discussion of different device types and deployment platforms is made by Zinn et al. (2012).

The availability of a software unit is particularly relevant during the configuration and execution of deployment. All information has to be available. Often, this is combined with the problem of completeness. The configuration phase sometimes required the creation of deployment packages out of existing data. Next to the deployment tools in the execution phase, this requires additional tools (e.g., an IDE).

The problem of reuse and design is seen as not relevant for transformation activities.

The problem of context dependencies occurs in the execution phase and for the output. A device or deployment platform, as shown in the example in Section 3.1.5 often requires the use of external libraries (dependencies). These support the software engineer so that a software unit is able to run in the runtime environment (firmware) of the device. If the dependencies are not available, the deployment might fail or the software unit will not run or react correctly in the runtime environment. Sometimes the device or the deployment platform has to be configured in the configuration phase. An example is shown in the example in Section 3.1.5 where special configuration files have to be set as part of a deployment package.

Appendix

Because of the need of the availability of relevant parts during the configuration and the execution phase it is also necessary to have the software units parts complete.

The problem of knowledge distribution and knowledge storing occurs for a software engineer for the output of a deployment activity. If the output (a final deployment packages for one deployment platform or device) was created, an engineer might want to store the deployment activity knowledge for personal reuse or for others. This leads to the problem of distribution.

The problem of knowledge learning occurs in each part of an activity. An engineer has to learn to find all necessary information, prepare the deployment tools and the software unit, integrate it and handle the deployment result.

In a deployment activity the problem of receiving knowledge and searching occurs in the preparation phase. A software engineer has to handle these problems before starting a configuration phase. The problem increases based on the specialised knowledge that is necessary to create a deployment package for a device. After receiving the necessary knowledge it can be used.

The problem of using knowledge occurs for an engineer in the configuration and the execution phase. The specialised deployment platforms or device types needs specialised configuration. The knowledge is the combination of information with the deployment tools that use the information to create an output, as well as, the tools or procedure necessary to configure the deployment activity. Also the user has to know how to use the different tools to create this output. The example in Section 3.1.5 shows that a software unit includes different elements that have to be configured and transformed to get deployed.

The problem of localisation (single/multi) occurs for the software engineer in all parts of a deployment activity. First of all the repositories to search and retrieve information might be

localised in different locations. The same scenario is valid for the problem of reachable knowledge. Also the deployment tool itself may be localised in other locations. As a result, it is difficult to perform the deployment. If this scenario occurs, the problem extends itself to the preparation phase. The engineer has to know how to access the environment of the tools to configure them. If an engineer wants to distribute the deployment result (i.e., the final build software unit to other teams using the same devices) it is necessary to know how to distribute it to other locations. This is part of the problem of knowledge exchange in industrial environments. Additionally the engineer has to know other repositories to exchange the knowledge of the deployment activity. This includes experience about location and handling of such repositories.

Following the time needed for performing a deployment activity, the problem of time and the problem of ROI are not pertinent for deployment activities.

The problem of support occurs in each part of a deployment activity. A software engineer may need support during the preparation, configuration, and execution phase, and the handling of the output of a deployment activity.

The problem activity relation shown in Table 76 can be described in more detail using the previous problem analyses.

g.2. Requirements definition based on focused Problems

Section G.1 introduces the software reuse activities and shows their relation to the discussed problems of the previous sections. Based on this relationship, a requirement list will be now created. By using this list, both existing solution approaches and the focused solution approach can be analysed and evaluated. Each requirement is represented by a question and subdivided into the three activities and their relation to the problem focused on by the requirement.

Appendix

Table 76 shows the discussed problem in relation to the introduced activities but to reach the aim of creating a measurement table based on the following requirements this perspective must change. Figure 160 shows the information flow used to explain the relationship between the focused problems and the reuse activities. Therefore a reuse of an activity needs information of a software unit (A) and the specific activity (B). This information will be used to create a reuse result (C). Sometimes the result is usable in an additional reuse (of the result or the same activity) or has to be published (D). Figure 160 also shows the relation to the general activity defined in Section G.2.

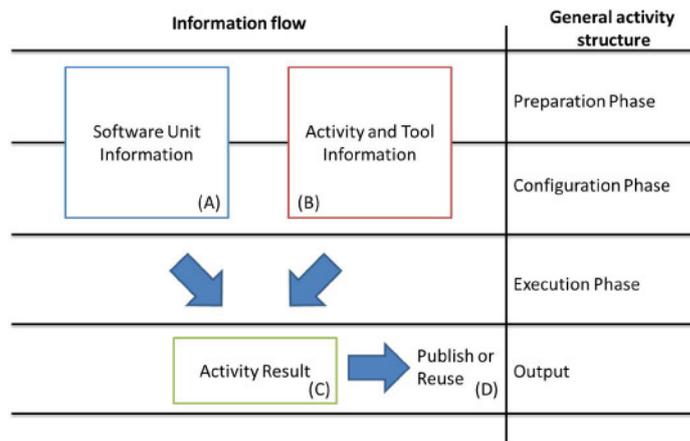


Figure 160 - Information flow of a activity reuse

The following general requirements can be formulated for the section (A), (B), (C), and (D):

Requirement A: The problem of specific knowledge about a software unit can be represented as follows as a question: ‘Is the system under investigation in a position to provide the user the knowledge needed for the reuse of a software unit?’ This question refers to whether the system under investigation to specific properties which allows the user to know or to get information about a software unit for reuse as needed. The question is answered with YES or NO.

Requirement B: The problem of concrete knowledge about reuse activities can be summarised as follows as a question: ‘Is the system under investigation in a position to provide the required knowledge to perform concrete reuse activities to a user?’ This question refers to whether the system under test conditions has specific properties that allow the user to know or get information about reusing activities of a specific software unit to perform this activity. The question is answered with YES or NO.

Requirement C: The problem of automated application of knowledge reuse activities can be summarised as follows: ‘Is the system under investigation able to support a user to perform automatically or partially automatically a software reuse activity?’ This question refers to whether the system under test conditions has concrete properties, which allows the user to reuse activities on a specific software unit which is automated or partially automated by the system. The question is answered with YES or NO.

Requirement D: The problem of the preservation of knowledge can be represented with the following question: ‘Is the system under investigation in a position to support a user in the preservation of knowledge and information?’ This question refers to whether the system under investigation provides specific properties to the user which allows them to obtain information that usually cannot be received because of missing user knowledge. The question is answered with YES or NO.

g.2.1. Requirements based on the software engineering problems

Following, the requirements based on the discussed problems in Section G.2 will be defined and related to a general requirement (A-D). Each sub requirement receives an identifier (ID). This ID consists of the main requirement number (1-23), the first character of the focused activity

Appendix

(Integration, Transformation or Deployment), the first character of the related part in the general activity process (Preparation, Configuration, Execution or Output), and a number (1-N) (if the problem occurs multiple times in a general activity process part). Therefore, the ID is only a summary of the analysis in Section G.2 for each problem.

Requirement 1: The problem of the different key concepts is represented by the following question: ‘Is the system that is to be tested able to simplify the problem of different key concepts for the user?’ This question seeks to determine whether a system has specific features to software engineers in the use of decisions to be made on the basis of key concepts to help. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements.

Based on the analysis in Section G.2 of this problem for the three SCAs the following sub requirements can be defined (see Table 77).

ID	Requirement question ‘Is the system under investigation able to...’	Requirement question answer
1TC1	...supports the user to configure the transformation tool based on the given information about key concepts?	YES/NO
1TO1	...supports the user to handle the key concepts of the transformation result in later reuse?	YES/NO
1IC1	... supports the user to configure the integration tool based on the given information about key concepts?	YES/NO
1DP1	... supports the user to search a deployment tool or software unit based on the given information about key concepts?	YES/NO
1DC1	...supports the user to configure the deployment tool based on the given information about key concepts?	YES/NO

Table 77 - Sub requirement list for the problem of key concepts

Requirement 2: The problem of different technologies is represented by the following question: ‘Is the system under investigation in a position to simplify the problem of different technologies or

Appendix

types of technology user?’ This question seeks to determine whether a system has specific features to support software engineers in the use of different technologies. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. The analysis in Section G.2 for this problem may be expressed by following sub requirement list (See Table 78)

ID	Requirement question ‘Is the system under investigation able to...’	Requirement question answer
2TP1	...supports a user to identify technologies information on a software unit for transformation?	YES/NO
2TC1	...supports a user to configure a transformation tool based on the technical properties of a software unit?	YES/NO
2TE1	...supports a user to execute transformation tool based on different technical properties for a transformation activity?	YES/NO
2TO1	...supports the user to handle the different technology problem of the transformation result in later reuse?	YES/NO
2IP1	...supports a user to identify technologies information on a software unit for integration?	YES/NO
2IC1	...supports a user to configure an integration tool based on the technical properties of a software unit?	YES/NO
2IE1	...supports a user to identify an integration tool based on different technical properties for a integration activity?	YES/NO
2DP1	...supports a user to identify technologies information on a software unit for deployment?	YES/NO
2DC1	...supports a user to configure an deployment tool based on the technical properties of a software unit?	YES/NO
2DE1	...supports a user to identify an deployment tool based on different technical properties for a deployment activity?	YES/NO

Table 78 - Sub requirement list for the problem of different views

Requirement 3: The problem of multitude, (i.e., the large number of different programming languages in the areas, of the identified software construction types) is represented by the following question: ‘Is the system under investigation in a position to solve or simplify the problems arising from the multitude (i.e., large number of programming and specification languages) for the user?’ This question seeks to determine whether a system has specific features to support software

Appendix

engineers, for example, in the use of different programming languages. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements.

Table 79 shows the sub requirement list for this requirement.

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
3TC1	... supports a user by configure a transformation based on different multitudes of a software unit?	YES/NO
3TE1	... supports a user by executing a transformation based on different multitudes of transformation tools?	YES/NO
3IC1	...supports a user to configure an integration tool based on the different multitudes of a software unit?	YES/NO
3IE1	...supports a user to executing an integration tool based on different multitudes of a software unit?	YES/NO
3IE2	...supports a user to executing an integration tool based on different multitudes of transformation tools?	YES/NO
3DC1	...supports a user to configure a deployment tool based on the different multitudes of a software unit?	YES/NO
3DE1	...supports a user to executing a deployment tool based on different multitudes of a software unit?	YES/NO
3DE2	...supports a user to executing a deployment tool based on different multitudes of deployment tools?	YES/NO

Table 79 - Sub requirement list for the problem of multitude

Requirement 4: The problem of different component models and components worlds is represented by the following question: 'Is the system being tested able to solve the problem of software engineers' lower experience in the use of different component models and worlds, or able to simplify it?'

This question seeks to determine whether a system has specific features to support software engineers in the use of different component models and worlds. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. In Table 80 the sub requirement list for this requirement based on the analysis in Section G.2 is shown.

Appendix

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
4TP1	...supports a user to identify component world or model information on a software unit for transformation?	YES/NO
4TC1	...supports a user to configure a transformation tool based on the component world or model properties of a software unit?	YES/NO
4TO1	...supports the user to handle the different technology problem of the transformation result in later reuse?	YES/NO
4IP1	...supports a user to identify component world or model information on a software unit for integration?	YES/NO
4IC1	...supports a user to configure an integration tool based on the component world or model properties of a software unit?	YES/NO
4IE1	...supports a user to identify an integration tool based on different component world or model for a transformation activity?	
4DP1	...supports a user to identify component world or model information on a software unit for deployment?	YES/NO
4DC1	...supports a user to configure a deployment tool based on the component world or model of a software unit?	YES/NO
4DE1	...supports a user to identify a deployment tool based on different component world or model for a deployment activity?	YES/NO

Table 80 - Sub requirement list for the problem of different component models and components worlds

Requirement 5: The availability problem is represented by the following question: 'Is the system under investigation in a position to support a user in identifying all the information a particular activity requires?' This question refers to whether a system has specific features to software engineers with all the necessary information (e.g., files and instructions) to make available. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. In Table 81 the sub requirement list for this requirement based on the analysis in Section G.2 is shown.

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
5TP1	... supports a user to access all parts software unit for a transformation activity?	YES/NO
5TE1	... supports a user having all necessary parts of a software unit available to execute a transformation?	YES/NO
5TE2	... supports a user having the IDE available to execute a transformation?	YES/NO

Appendix

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
5IP1	... supports a user to access all parts software unit for a integration activity?	YES/NO
5IE1	... supports a user having all necessary parts of a software unit available to execute an integration?	YES/NO
5IE2	... supports a user having the IDE available to execute an integration?	YES/NO
5DC1	... supports a user having all necessary parts of a software unit available to configure a deployment activity?	YES/NO
5DC2	... supports a user having the additional tool available to configure a deployment activity?	YES/NO
5DE1	... supports a user having all necessary parts of a software unit available to execute a deployment activity?	YES/NO
5DE2	... supports a user having the deployment tool available to execute a deployment activity?	YES/NO

Table 81 - Sub requirement list for the problem of availability

Requirement 6: The problem of context dependencies is represented by the following question: 'Is the system under investigation in a position to simplify or to solve the problem of context dependency, of a software user, for a user?' This question seeks to determine whether a system has specific features to the user in the use of a software module and the problems that may arise from the different context dependencies to simplify or solve. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. Table 82 shows the sub requirement list for this requirement based on the analysis in Section G.2.

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
6TC1	... supports a user to configure a transformation tool based on the context dependencies of a software unit?	YES/NO
6TO1	... supports the user to handle the (new) context dependencies of the transformation result in later reuse?	YES/NO
6IC1	... supports a user to configure a integration tool based on the context dependencies of a software unit?	YES/NO
6IE1	... supports a user to integrate the context dependencies of a software unit?	YES/NO
6IO1	... supports the user to handle the context dependencies in the integration result in later reuse?	YES/NO

Appendix

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
6DC1	... supports a user to configure a deployment tool based on the context dependencies of a software unit?	YES/NO
6DE1	... supports a user to deploy the context dependencies of a software unit?	YES/NO

Table 82 - Sub requirement list for the problem of context dependencies

Requirement 7: The problem of different perspectives can be expressed as follows: 'Is the system under investigation in a position to solve or simplify the problem of different perspectives (object orientation, component orientation, and service orientation) for users?' This question refers to whether a system has specific features to support software engineers in the use of different perspectives that this study has focused upon. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. The analysis in Section G.2 for this problem may be expressed by following sub requirement list (See Table 83)

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
7TC1	... supports the user to configure the transformation tool based on the given information about view on a given software unit?	YES/NO
7TO1	... supports the user to handle the view on the transformation result in later reuse?	YES/NO
7IC1	... supports the user to configure the integration tool based on the given information about view on a given software unit?	YES/NO
7DP1	... supports the user to search a deployment tool or software unit based on the given information about views?	YES/NO
7DC1	... supports the user to configure the deployment tool based on the given information about view on a given software unit?	YES/NO

Table 83 - Sub requirement list for the problem of different perspectives

Requirement 8: The question of horizontal and vertical markets can be described as follows: 'Is the system capable to solve or simplify the problem of horizontal and vertical markets for the user?' This question refers to whether a system has specific characteristics in order to use a

Appendix

software component in both horizontal as in vertical markets and systems. This focus on the fact that a software unit may be reused in development projects of other vertical or horizontal markets. The question has to be answered with YES or NO. This requirement has no identified sub requirements (see analysis in Section G.2)

Requirement 9: The problem of completeness can be illustrated with the following question: ‘Is the system capable to solve or simplify the problem of completeness of a software unit for the user?’ This question refers to whether a system has specific properties to be able to support a software engineer in a software component to use in full. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. In Table 84 the sub requirement list for this requirement based on the analysis in Section G.2 is shown.

ID	Requirement question ‘Is the system under investigation able to...’	Requirement question answer
9TE1	... supports a user having all necessary parts of a software unit to execute a transformation?	YES/NO
9TO1	... supports the user to handle completeness of the result in later reuse?	YES/NO
9IE1	... supports a user having all necessary parts of a software unit to execute an integration?	YES/NO
9IO1	... supports the user to handle completeness of the result the development project?	YES/NO
9DC1	... supports a user having all necessary parts of a software unit to configure a deployment activity?	YES/NO
9DE1	... supports a user having all necessary parts of a software unit to execute a deployment activity?	YES/NO

Table 84 - Sub requirement list for the problem of completeness

Requirement 10: The problem of reuse type can be represented with the following question: ‘Is the system capable to simplify or solve the problem resulting from the number of different types of reuse, to a user?’ This question refers to whether a system has specific properties that support software reuse engineers in a software unit with the preferred reuse type. The question has to be

answered with YES or NO. This requirement has no identified sub requirements (see analysis in Section G.2)

g.2.2. Requirements based on the Knowledge management problems

Requirement 11: The problem of knowledge storing can be represented with the following question: ‘Is the system able to store knowledge about software units and software reuse activities?’ This question will answer whether the system is able to store the knowledge learned by an expert user about a reuse activity or a related software unit. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. In Table 84 the sub requirement list for this requirement based on the analysis in Section G.2 is shown.

ID	Requirement question ‘Is the system under investigation able to...’	Requirement question answer
11TO1	... supports the user to store the transformation activity knowledge for later reuse?	YES/NO
11IO1	... supports the user to store the integration activity knowledge for later reuse?	YES/NO
11DO1	... supports the user to store the deployment activity knowledge for later reuse?	YES/NO

Table 85 - Sub requirement list for the problem of knowledge storing

Requirement 12: The problem of knowledge learning can be represented with the following question: ‘Is the system able to support a user by learning knowledge about software units and software reuse activities?’ This question aims to uncover whether the system is able to support the learning process of a non-experienced user about a software unit and related reuse activities. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. The analysis in Section G.2 for this problem may be expressed by following sub requirement list (See Table 86)

Appendix

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
12TP1	... supports a user to learn how to search a software unit or a transformation tool for a transformation?	YES/NO
12TC1	... supports the user to learn knowledge about transformation activity configuration?	YES/NO
12TE1	... supports the user to learn knowledge about transformation activity execution?	YES/NO
12TO1	... supports the user to learn knowledge about the transformation result for later reuse?	YES/NO
12IP1	... supports a user to learn how to search a software unit or a integration tool for a transformation?	YES/NO
12IC1	... supports the user to learn knowledge about integration activity configuration?	YES/NO
12IE1	... supports the user to learn knowledge about integration activity execution?	YES/NO
12IO1	... supports the user to learn knowledge about the integration result for later reuse?	YES/NO
12DP1	... supports a user to learn how to search a software unit or a deployment tool for a transformation?	YES/NO
12DC1	... supports the user to learn knowledge about deployment activity configuration?	YES/NO
12DE1	... supports the user to learn knowledge about deployment activity execution?	YES/NO
12DO1	... supports the user to learn knowledge about the deployment result for later reuse?	YES/NO

Table 86 - Sub requirement list for the problem of knowledge learning

Requirement 13: The problem of knowledge receiving can be represented with the following question: 'Is the system able to support a user by receiving knowledge about software units and software reuse activities even if this person is usually not able to receive it?' This question examines whether the system is able to support the receiving of knowledge for a non-experienced user about a software unit and related reuse activities. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. In Table 87 the sub requirement list for this requirement based on the analysis in Section G.2 is shown.

Appendix

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
13TP1	... supports a user to receive knowledge of a transformation activity?	YES/NO
13IP1	... supports a user to receive knowledge of a integration activity?	YES/NO
13DP1	... supports a user to receive knowledge of a deployment activity?	YES/NO

Table 87 - Sub requirement list for the problem of knowledge receiving

Requirement 14: The problem of knowledge search can be represented with the following question: 'Is the system able to support a user by searching knowledge about software units and software reuse activities even if this person is usually not able to search for it?' This question investigates whether the system is able to support the searching of knowledge for a non-experienced user who is usually not able to create a search request. The question has to be answered with YES or NO. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. Table 88 shows the sub requirement list for this requirement based on the analysis in Section G.2.

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
14TP1	... supports a user to search for knowledge about transformation activity?	YES/NO
14IP1	... supports a user to search for knowledge about integration activity?	YES/NO
14DP1	... supports a user to search for knowledge about integration activity?	YES/NO

Table 88 - Sub requirement list for the problem of knowledge search

Requirement 15: The problem of knowledge use can be represented with the following question: 'Is the system able to support a user with (re)use knowledge about software units and software reuse activities even where they do not have the Professional experience to do this?' This question indicates if a system has specific properties to support a non-experienced user to use knowledge about a software unit to a related reuse activity. The question has to be answered with YES,

Appendix

PARTIALLY YES or NO depending on the answers of the sub requirements. Based on the analysis in Section G.2 of this problem for the three SCAs the following sub requirements can be defined (see Table 89).

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
15TC1	... supports the user to use specific knowledge to configure a transformation tool?	YES/NO
15TE1	... supports the user to use specific knowledge to execute a transformation tool?	YES/NO
15TO1	... supports the user to use knowledge to create a final transformation output?	
15IC1	... supports the user to use specific knowledge to configure a integration tool?	YES/NO
15IE1	... supports the user to use specific knowledge to execute a integration tool?	YES/NO
15DC1	... supports the user to use specific knowledge to configure all necessary deployment tools?	YES/NO
15DE1	... supports the user to use specific knowledge to execute a the deployment?	YES/NO

Table 89 - Sub requirement list for the problem of knowledge using

Requirement 16: The problem of knowledge distribution can be represented with the following question: 'Is the system under investigation able to support an expert user by distributing knowledge about software units and software reuse to other users?' This question examines if a system has specific properties to support a user and provide knowledge given by this user to other users. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. In Table 90 the sub requirement list for this requirement based on the analysis in Section G.2 is shown.

Appendix

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
16TO1	... supports the user to distribute the transformation activity knowledge for later reuse to other users?	YES/NO
16IO1	... supports the user to store the integration activity knowledge for later reuse?	YES/NO
16DO1	... supports the user to store the integration activity knowledge for later reuse?	YES/NO

Table 90 - Sub requirement list for the problem of knowledge distribution

g.2.2. Requirements based on the industrial context problems

Requirement 17: The problem of supporting multiple reuse of a software unit within a team in different locations can be expressed with the following question: 'Is the system to be examined in a position to improve multiple reuse of the same software unit within software development teams whereby members are placed in different locations?' This question refers to whether the system under investigation has concrete properties, which improves the multiple reuses within a team which has localisation problems. This time both improvements and simplification in dealing with the software are meant to block. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. Based on the analysis in Section G.2 of this problem for the three SCAs the following sub requirements can be defined (see Table 91).

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
17TP1	... supports a single user to search for a transformation or software unit in a different location?	YES/NO
17TC1	... supports a single user to configure a transformation in a different location?	YES/NO
17TE1	... supports a single user to execute a transformation in a different location?	YES/NO
17TO1	... supports a single user to share the transformation result to different locations?	YES/NO

Appendix

ID	Requirement question ‘Is the system under investigation able to...’	Requirement question answer
17IP1	... supports a single user to search for an integration or software unit in a different location?	YES/NO
17IC1	... supports a single user to configure an integration in a different location?	YES/NO
17IE1	... supports a single user to execute an integration in a different location?	YES/NO
17DP1	... supports a single user to search for a deployment activity or software unit in a different location?	YES/NO
17DC1	... supports a single user to configure a deployment tool in a different location?	YES/NO
17DE1	... supports a single user to execute a deployment activity in a different location?	YES/NO
17DO1	... supports a single user to share / deploy the deployment activity result to different locations?	YES/NO

Table 91 - Sub requirement list for the problem of localisation (single)

Requirement 18: The problem of multiple reuses within different development teams can be represented with the following question: ‘Is the system under investigation in a position to assist different development teams in reusing the same software units even where these teams are in different locations?’ This question refers to whether the system under investigation has a specific property for the reuse of a particular software unit for different development teams in the same way. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. Table 92 shows the sub requirement list for this requirement.

ID	Requirement question ‘Is the system under investigation able to...’	Requirement question answer
18TP1	... supports a team to search for a transformation or software unit in a different location?	YES/NO
18TC1	... supports a team to configure a transformation in a different location?	YES/NO
18TE1	... supports a team to execute a transformation in a different location?	YES/NO
18TO1	... supports a team to share the transformation result to different locations?	YES/NO
18IP1	... supports a team to search for an integration or software unit in a different location?	
18IC1	... supports a team to configure an integration in a different location?	YES/NO
18IE1	... supports a team to execute an integration in a different location?	YES/NO

Appendix

ID	Requirement question ‘Is the system under investigation able to...’	Requirement question answer
18DP1	... supports a team to search for a deployment activity or software unit in a different location?	YES/NO
18DC1	... supports a team to configure a deployment tool in a different location?	YES/NO
18DE1	... supports a team to execute a deployment activity in a different location?	YES/NO
18DO1	... supports a team to share / deploy the deployment activity result to different locations?	YES/NO

Table 92 - Sub requirement list for the problem of localisation (single)

Requirement 19: The problem of missing knowledge exchange can be represented with the following question: ‘Is the system which is under investigation able to support teams or persons in knowledge exchange about software units or reuse activities?’ This question examines if a system has specific properties to support a user providing knowledge to other users. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. Based on the analysis in Section G.2 of this problem for the three SCAs the following sub requirements can be defined (see Table 93).

ID	Requirement question ‘Is the system under investigation able to...’	Requirement question answer
19TO1	... supports a user to share the transformation result knowledge to different others person for later reuse?	YES/NO
19IO1	... supports a user to share the integration result knowledge to different others person for later reuse?	YES/NO
19DO1	... supports a user to share the deployment result knowledge to different others person for later reuse?	YES/NO

Table 93 - Sub requirement list for the problem of missing knowledge exchange

Requirement 20: The problem of the reachable knowledge can be illustrated with the following question: ‘Is the system capable of making knowledge as provided for by a team or a person reachable to others?’ This question refers to whether a system has specific characteristics to make it

Appendix

easier to reach knowledge about reusing software units in the company environment. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. In Table 84 the sub requirement list for this requirement based on the analysis in Section G.2 is shown.

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
20TP1	... supports a team to receive transformation activity knowledge?	YES/NO
20IP1	... supports a team to receive integration activity knowledge?	YES/NO
20DP1	... supports a team to receive deployment activity knowledge?	YES/NO

Table 94 - Sub requirement list for the problem of reachable knowledge

Requirement 21: The problem of the high expenditure of time for reuse can be illustrated with the following question: 'Is the system capable in reducing the time needed for the reuse of a software unit?' This question refers to whether a system has specific characteristics to reduce the reuse of a software unit in time. The question has to be answered with YES or NO. The analysis in Section G.2 shows no sub requirement.

Requirement 23: The problem of excessive support requirements can be illustrated with the following question: 'Is the system under investigation able to reduce the support time effort for a single team reusing a software unit?' The question aims to examine whether the system has a specific property, the support costs for the initial creation of a software unit and/or shorten the reuse of a software unit. The question has to be answered with YES, PARTIALLY YES or NO depending on the answers of the sub requirements. Based on the analysis in Section G.2 of this problem for the three SCAs the following sub requirements can be defined (see Table 89).

Appendix

ID	Requirement question 'Is the system under investigation able to...'	Requirement question answer
23TP1	... reduces support effort for preparing of a transformation activity?	YES/NO
23TC1	... reduces support effort for configuration of a transformation activity?	YES/NO
23TE1	... reduces support effort for execution of a transformation activity?	YES/NO
23TO1	... reduces support effort for creating a final transformation activity result?	YES/NO
23IP1	... reduces support effort time during preparing of an integration activity?	YES/NO
23IC1	... reduces support effort for configuration of an integration activity?	YES/NO
23IE1	... reduces support effort for execution of an integration activity?	YES/NO
23IO1	... reduces support effort for creating a final integration activity result?	YES/NO
23DP1	... reduces support effort for preparing of a deployment activity?	YES/NO
23DC1	... reduces support effort for configuration of a deployment activity?	YES/NO
23DE1	... reduces support effort for execution of a deployment activity?	YES/NO
23DO1	... reduces support effort for creating a final deployment activity result?	YES/NO

Table 95 - Sub requirement list for the problem of excessive support requirements

These relationships are summarised in Table 96 and used in Chapter 6 to compare existing solutions as well as the focused approach of this thesis with these requirements. Therefore, Table 96 shows explicitly how many sub requirements can be identified for each part of the general activity structure for each SCAC (T, I, D). As a result Table 96 shows in detail the difference between the different SCACs related to the 23 requirements.

	Preparation Req A / B	Configuration Req A / B	Execution Req C	Ouput Req D
Req 1	T0, I0, D1	T1, I1, D1	-	T1, I0, D0
Req 2	T1, I1, D1	T1, I1, D1	T1, I1, D1	T1, I0, D0
Req 3	-	T1, I1, D1	T1, I2, D2	-
Req 4	T1, I1, D1	T1, I1, D1	T0, I1, D1	T1, I0, D0
Req 5	T1, I1, D0	T0, I0, D2	T2, I2, D2	-
Req 6	-	T1, I1, D1	T0, I1, D1	T1, I1, D0
Req 7	T0, I0, D1	T1, I1, D1	-	T1, I0, D0

	Preparation	Configuration	Execution	Ouput
	Req A / B	Req A / B	Req C	Req D
Req 8	-	-	-	-
Req 9	-	T0, I0, D1	T1, I1, D1	T1, I1, D0
Req 10	-	-	-	-
Req 11	-	-	-	T1, I1, D1
Req 12	T1, I1, D1	T1, I1, D1	T1, I1, D1	T1, I1, D1
Req 13	T1, I1, D1	-	-	-
Req 14	T1, I1, D1	-	-	-
Req 15	-	T1, I1, D1	T1, I1, D1	T1, I0, D0
Req 16	-	-	-	T1, I1, D1
Req 17	T1, I1, D1	T1, I1, D1	T1, I1, D1	T1, I0, D1
Req 18	T1, I1, D1	T1, I1, D1	T1, I1, D1	T1, I0, D1
Req 19	-	-	-	T1, I1, D1
Req 20	T1, I1, D1	-	-	-
Req 21	-	-	-	-
Req 22	-	-	-	-
Req 23	T1, I1, D1	T1, I1 D1	T1, I1, D1	T1, I1, D1

Table 96 - Requirements relationship

g.2.3. Additional requirements based on different views

In preparation for this thesis no predefined analysis system or approach was identified fitting the different discussed problems. Also no classification for solutions for the perspective on software construction (or reuse) activities and the problem of missing knowledge was identified.

Due to these reasons, a new solution classification for this thesis has to be defined. It uses the requirement analysis shown in the previous section and is based on different views.

Class I “Activity View”: The different sub requirements shown in the last section are divided into three focused SCACs. Therefore, it is relevant to know which SCAC types are supported by a solution. The requirement can be identified by listing the supported SCAC types.

Class II “Software Unit View”: The different requirements discussed in the last section do not identify whether a solution only focusses on one type of software unit. Therefore, it is relevant to know if a solution focuses on multiple technologies, key concepts or views.

H. Information demand model for software unit reuse

To demonstrate the relationship between the discussed problems and knowledge the Software Reuse Information Demand (SRID) model is used in the thesis. This model has been created as part of the research for this thesis and published in 2011 (see Zinn et al., 2011). It extends the problem explanations for software engineers with visualisations of knowledge problems. The model is based on the Information Demand model for company managers created by Picot (2003). This section explains the structure and application of the model and shows how it can be used for further analytic test.

h.1. Description of information demand

Information Demand (ID) is defined as the type, amount and quality of knowledge that a person needs to fulfil a task within a specific time frame. Measuring ID is difficult, because it is dependent upon the task definition, the goals, people involved (Picot, 2003), and the knowledge criteria (Boh, 2008). Two categories of ID can be identified: Objective Information Demand (OID) and Subjective Information Demand (SID). OID describes all information which solves the user’s problem. This information is the amount of existing information that will theoretically solve a specific problem. It can be described as a set of solutions. Similar to OID, SID describes all information that is supposed to solve the problem, from the subjective point of view of a user and because of that; this information may not be able to provide a real solution. Another relevant factor in the area of ID is the information provision (IP). This defines information that is provided by a

Appendix

system and can be utilised by formulating an information query (IQ). IQ is most commonly created by a user who is searching for a solution which is executed by a search system. This query is based on the SID of the user. The useful result is referred to as an Actual Information State (AIS). Within the scope of this thesis this is the intersection of the areas of SID, OID, IP, and IQ. Figure 161 illustrates this relationship.

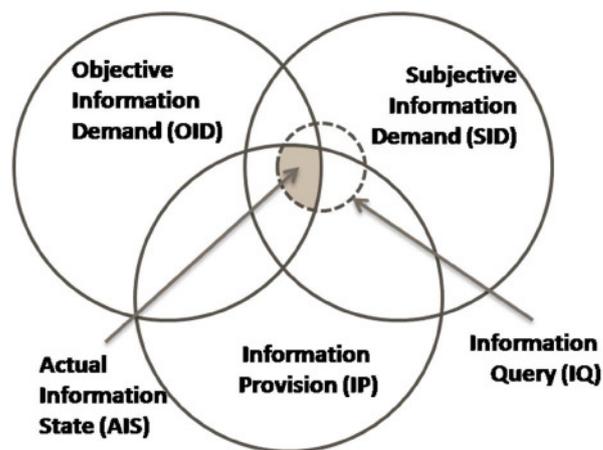


Figure 161 – Original information demand model by Picot (2003, p. 106)

As a result, of this model, AIS is defined as the area of the information model which includes information that

- (1) achieves the task,
- (2) can be understood by the user,
- (3) can be enquired after by the user, and
- (4) is provided.

h.2. Definition of information demand from the software unit reuse perspective

The original definition of an information demand model (Picot, 2003) is not related to the area of software reuse. This relation can be established by adjusting the perspective of software reuse. The OID can be related to as a container describing all software units that may solve a problem. From the software reuse perspective, the relevant tasks of reuse are to find and integrate reusable software units. These units have to fulfil technical, functional, and business requirements (Shiva and Shala, 2007). As a result, OID describes all software units that can solve the problem. SID is related to the user's ability to express technical, functional, and business information about a required software unit. This also includes units that do not solve the problem, contrary to the user's beliefs. Previous studies show that this is an relevant problem of knowledge reuse (Boh, 2008). In the reuse area, IP can be defined as the real availability of reusable software units and their descriptive information. Thus, IP is realised by repository systems responsible for providing software units. Typically, software units are provided by repository systems (Ajila, 2006). The user creates and accomplishes an IQ by using special tools (e.g., software reuse environments; Garcia et al., 2006). The AIS in the area of software unit which includes all reusable software units that:

- (1) theoretically solve the problem,
- (2) are provided by a unit provider or repository system,
- (3) are understood by the user,
- (4) are described by the user request (part of the users' query).

The goal is to increase overlapping areas between the different elements of an information demand model in order to increase the amount of solutions, which is also an relevant aim of the software

unit reuse area. This can be achieved by defining/identifying the critical success factors of software reuse and relating them to the model (Picot, 2003).

Figure 162 shows the relationship between the underlying information demand and the SRID model.

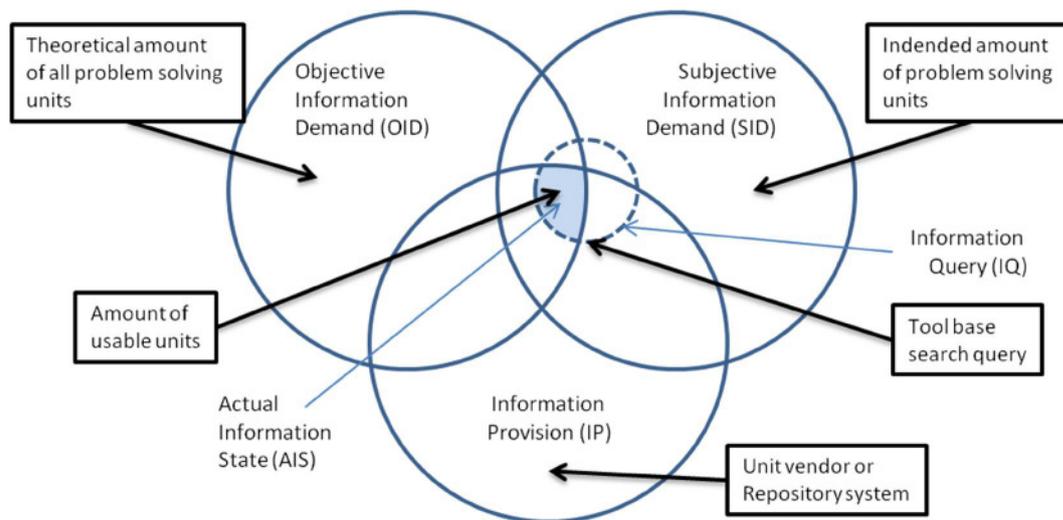


Figure 162 - SRID model related to original information model

h.3. Use of information demand

The underlying information demand model maps used processes to the different parts of the model. By doing so, potential problems of a process field can identified (see Picot, 2003). In the following, this is explained using the SRID model and the Critical Success Factors methodology.

To increase the overlap of OID and SID, the Critical Success Factors methodology (cf. Rockart, 1979) can be used (cf. Picot, 2003). This approach may be used for business success factors, but can also be adapted for other domains (e.g., Enterprise Security Management; Software Engineering Institute, 2004). Using this approach, factors that are required to fulfil a task are

Appendix

identified. In the case of the Critical Success Factors methodology, these factors are: reuse is focused, the unit exists, the unit is available, the unit was found, and the unit is valid.

These factors will be related to specific elements of the information demand model. Based on this relationship, custom analyses can be performed to identify possible risks in a project (see Picot, 2003).

Therefore, an analysis process based on this may be structured as follows:

- (1) Identify relevant success factors for a project.
- (2) Apply the factors to an information demand model.
- (3) Identify problems and risks by establishing which information demand elements are affected by a problem or risk that could affect an applied success factor.
- (4) Prepare training sessions to minimise the risks or problems.

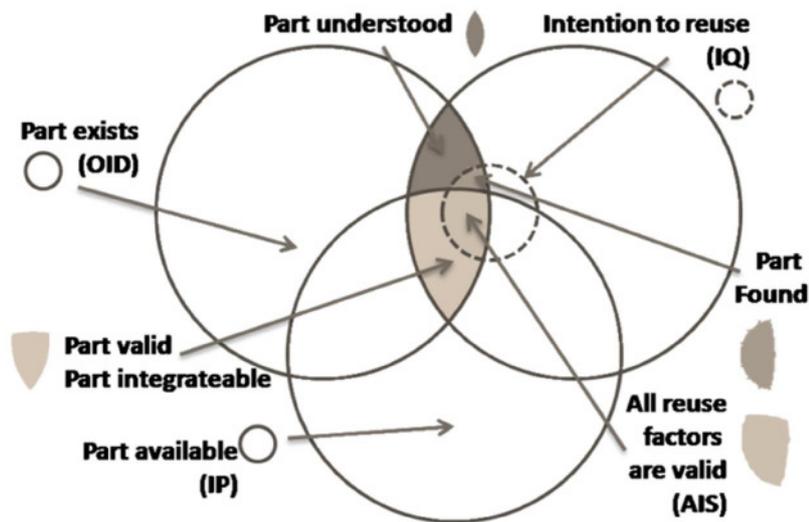


Figure 163 - Critical success factors (of Frakes and Fox, 1996) in the SRID model

Figure 163 shows an example of this procedure. The success factors of Frakes and Fox (1996) are mapped to the SRID model. From the perspective of Frakes and Fox (1996) a software unit is successfully reused if it is focused, a unit exists, the unit is available, the unit was found, the unit is valid, and it is capable to be integrated. This can be combined with the SRID model as follows (cf. Figure 163).

- **Intention for reuse:** This is the user's aim to reuse a software unit. In the scope of the SRID model, this is shown by the IQ definition.
- **Part exists:** A unit exists if it is theoretically possible and is practically able to solve the problem. In the SRID model, this corresponds to the OID.
- **Part available:** A unit is available if it is provided for by a unit vendor or a repository system. This complies with the IP area of the SRID.
- **Part found:** A unit has the state "found" if it is theoretically possible, understandable by the user, requested, and found by the user. This is shown in this model with AIS.
- **Part understood:** A unit is understood if it is theoretically possible and the user is able to understand it. In the SRID model, it corresponds to the overlapping area between OID and SID.
- **Part valid and part able to be integrated:** Both success factors depend on three properties:
 - They have to be part of the theoretical amount of solutions (OID)
 - They have to be provided by a vendor or system (IP)
 - They have to be part of the user's subjective information demand (SID)

Appendix

Based on such a mapping the critical information demand areas can be identified. This may result in specialised training sessions or other supporting activities. To reduce the risk for each success factor (see further discussion, Zinn et al., 2011 or Appendix Section I).

Note: In this thesis, the analysis using the success factors is not used, because people can use different success factor models. This section will only illustrate how the SRID model can be used. In the following sections, it is more relevant to use the SRID model as a visualisation tool for missing knowledge examples.

I. Published papers

In this section relevant papers are included published by the author. All published papers can be found on the data medium (cf. Section A).

Development of a CASE-tool for the Service-Based Software Construction

M.Zinn, K.P.Fischer-Hellmann and A.D.Phippen

Centre for Security, Communications and Network Research,
University of Plymouth, Plymouth United Kingdom
e-mail: mail@marcuszinn.de, K.P.Fischer-Hellmann@digamma.de,
andy.phippen@plymouth.ac.uk

Abstract

In today's software development, applications for computer aided software engineering (CASE) are widespread and necessary. Most procedure models or technologies use CASE-tools. This publication shows the conversion of the fundamental idea of service-based software construction into a software system. The focus is set on the system architecture, the fundamental information model and the transformation model based on it.

A goal is to represent the needed information to build up and use a CASE-tool for service-based software construction.

Keywords

Computer aided software engineering (CASE), service-based software construction process, software construction artefact, unit of modelling, transformation, integration

1. Introduction

Software development processes as well as the actual software development are in many cases supported by applications (CASE-tools). Well-known CASE-tools are e.g. Innovater, Rational Rose, Enterprise Architect und Together. Typical contents are support modes like for example graphic user interface, information preparation, input simplification (Wizards) and automation of process steps. (Deneva, 1999)

Some applications are single technology applications. That means these applications usually support only one type and/or technology of software development processes/software development in form of an executable application. Other CASE-tools offer the possibility of extension.

Typical examples of this kind of applications are development environments. With the consideration of Eclipse, for example, it becomes clear that various technological approaches can be realised. These include component- and service technology as well as software development approach models like model driven development. One of the most important factors is the interoperability between different CASE-tools and the supporting applications and processes (IEEE, 2007) (Garcia-Magarino and Gomez-Sanz, 2008). (Deneva, 1999) and (Deneva and Terzieva, 1996) show that CASE-tools are an important factor of success for approach models or technologies

since this supports the propagation of the approach by a practical application. The important factors are: Simplification in handling, possible integration in available tools or environments and easiness to in understanding.

The service-based software construction process is an add-on for existing procedure models. It supports the developer in reusing software units (classes, services and components) and all important information like documentation and specification which is related to these units. In addition, the reuse is supported by a transformation model, which can transform software units into another form. This reuse is an added value because it offers more information from a single data. Furthermore, this reuse is build upon a new information model. As a novelty this data management and transformation system is provided by a single service. As pointed out in a previous publication, software development is not a question of the location of a developer or the needed data. The question is how the developer can handle this data (Zinn, 2007).

The aim of this publication is to show which models and architecture can be used to create a CASE-tool for the support of the service-based software construction process. This process is the underpinning methodology. Therefore, it is necessary to explain the basic concepts. The next section shows an overview of different models and technologies needed for the service-based software construction. This includes the explanation of artefacts, units of modelling, service-based software construction procedure model, information model, transformation model and integration model and builds a common understanding. Following this, a section demonstrates the structure of the CASE-tool built upon the explained models and technologies.

2. Basic Concepts

2.1 Artefacts and units of modelling

For the explanation of the application shown in this publication, it is necessary to define some terms. In the area of software development, which deals with composing of bigger software units (McConnell, 1996), the term “unit of modelling” is used (Wang and Fung, 2004) and (Zinn, 2008). The service-based software construction uses components, services and classes (objects) as units of modelling (UOM). This shows the scope of this kind of software development: Objects (Object-oriented construction), Components (Component-based construction) and Services (Service-oriented construction). See (Sommerville, 2007), (Szyperski et al., 2002), (Papazoglou et al., 2007) and (W3C, 2004) for the used definitions classes(objects), services and components. The service-based software construction process extends the view of UOMs. As a result, a UOM is not only the implementation of a software unit. A UOM in this area is a container for the implementation and all other information which depends on the implementation and which is important for the reuse. For example, (Pfleeger and Atlee, 2009) show that documentation, specification and test information are also important for reuse. Another very important reason for the storage of all this data is the search for UOMs. Reusing a UOM is easier than to find it again. The “correct” search for data is very complicated and is based on the metadata that can be used for the search. Today semantic search, based on different ontologies, is very popular. This means data and definitions will

be connected in one database by the use of semantics. These semantics can be used to find data entities (see (Stuckenschmidt, 2009) and (Hitzler, 2008)). Important information of a UOM is “Transformation”. A UOM does not only contain describing information, but also supports to carry information which can transform UOM data. A transformation can be, for example, a transformation of one technology into another such as transforming Java byte code into .NET byte code (Frijters, 2008) or a UML diagram into Java source code. Thus the original UOM can be used in another (technology) domain. The service-based software construction process focuses on the transformation of implementation data. Moreover, it can be necessary to transform describing information. The service-based software construction process defines “Transformation” like the definitions of Mode Driven development (MDD/MDSD) (Meimberg et al. 2006), (Stahl, 2007) and Generative Programming (GP) (Czarneck and Eisenecker, 2000). Transformation means in this area that information will be transformed into the same or another domain specific model for later reuse. (Garcia-Magarino, 2008) shows different modelling languages and frameworks in the area of CASE-tool interoperability. (Czarnecki and Helsen, 2003) show a classification of different model transformation attempts. Within the scope of this research MDD and GP are focused. In this publication transformation is done by using existing transformation tools.

Another important term is “Software Construction Artefact (SCA)”. An SCA is a container for UOMs which corresponds to the same problem area. For example: An Artefact for calculating the mathematical GAUSS function carries 3 UOMs. The first is a Java Class, the second a .NET component and the last one a Web service. This example shows that the common scope is important to put UOMs into the same artefact and not the technical view on the UOMs.

2.2 Service-based software construction procedure model

The service-based software construction process is the underpinning methodology for the software described by this paper. It contains a maximum of four possible phases.

1. Development of an outer structure (Precondition phase)
2. Choice of the units of modelling and their transformation rules
3. Creation of missing artefacts / units of the modelling (optionally)
4. Transformation of the units into the external structure

The first phase is to create an outer structure. This means the developer has to create, for example, a class structure which can be extended by adding UOMs. In the second phase the user searches for artefacts which correspond to his search criteria. The list of possibly suitable artefacts is then examined closer to identify certain units of the modelling which can be used. In addition, the units of modelling contain transformation data which can be used to customise the unit. If the research proves that components are missing or no components can be found in the available repositories, the components must be created independently. This means that step three is an optional step. When all components are identified, they are transmitted into the created external structure (see first phase) by means of the transformations selected in phase two. The creation of the external structure is a step preceding the

software construction. The service-based software construction process does not define the creation of the outer structure but uses it as a precondition. Example: A developer wants to create a small program which is able to calculate Pi. To do this he starts to create an empty class structure (outer structure). Afterwards he continues by searching a UOM for calculating Pi. The result of the search is an SCA which provides him four UOMs: A webservice, a java component, a C# Class and a transformation which transforms the C# Class into a VB class. The developer selects the webservice and adds it to the outer structure. The phases 2, 3 and 4 also represent the procedure model. Through the consideration of this procedure model the Use Cases become visible. This, however, only constitutes summarised Use Cases: 1. Select UOM, 2. Transform artefact and 3. Create UOM.

2.3 Basic information, transformation and integration model

The fundamental information model defines the contents of a software construction artefact. A software construction artefact contains different implementations of the three kinds of units of modelling, but all implementations refer to the same problem area. Each unit contains "legible", "illegible" and "reference" information. Legible information serves the user during the identification and transformation phase. This can be documentation information, cost information, specification information and modelling information. "Illegible" data are information which can be ascribed to the automated use by the machine. Thereby, component-specific and component-unspecific information are distinguished. Specific data are implementation (with objects) and binary data (with components). Unspecific data can occur with more than one component kind such as interfaces, UnitTest and transformation data. In figure 1 the illegible data is described as "NRDataType" and legible data as "RDataType".

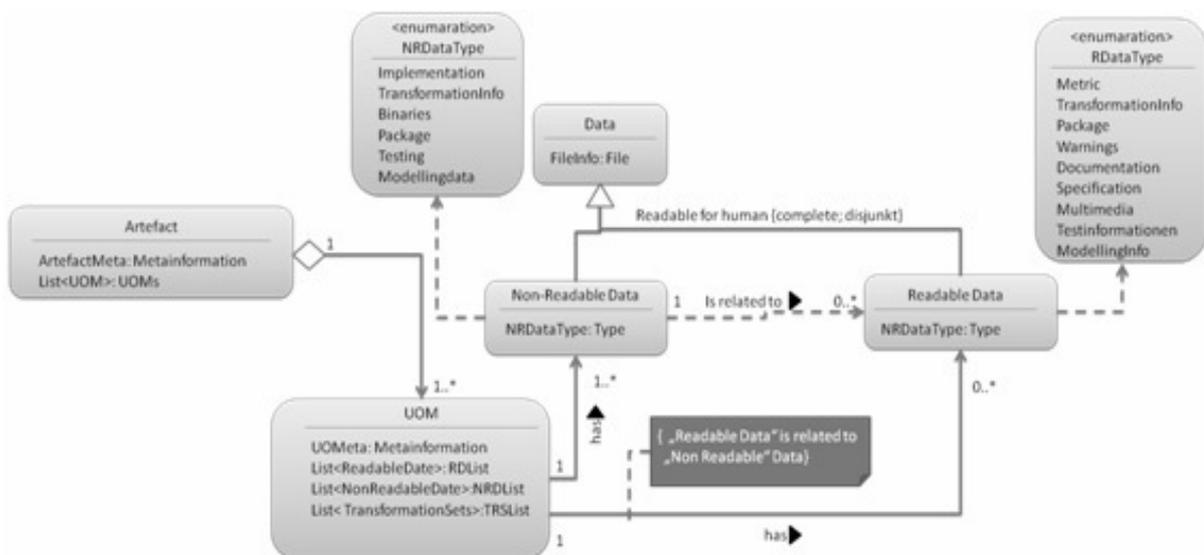


Figure 1: Information model core

This figure also shows a sketch of the information model core. The important statement is the relationship between the Artefact and the UOMs. Also the content of the UOM is important. Beside the basic data model, there is one more transformation and integration model. The transformation model describes the input and output data

as well as the transformation rules of a transformation. The input data are a special subset of data from the information model whose contents depend on the respective unit of modelling. A transformation rule is the description on how to transform the input into the output. Since this area has not deepened further in the actual research yet, an application-supported transformation is anticipated in this case. Therefore, a transformation needs a set of parameters and the respective start-up files which launch a transformation. The output results from the used transformation rule or the used transformation application. This scenario requires an enhancement of the basic information model. UOMs must contain transformation sets and these sets are composed of transformation rules. Figure 2 shows the enhancement of the information model with transformation entities. So a UOM can have transformation sets, which consist of different transformation rules. Input and the output of these rules are file based data.

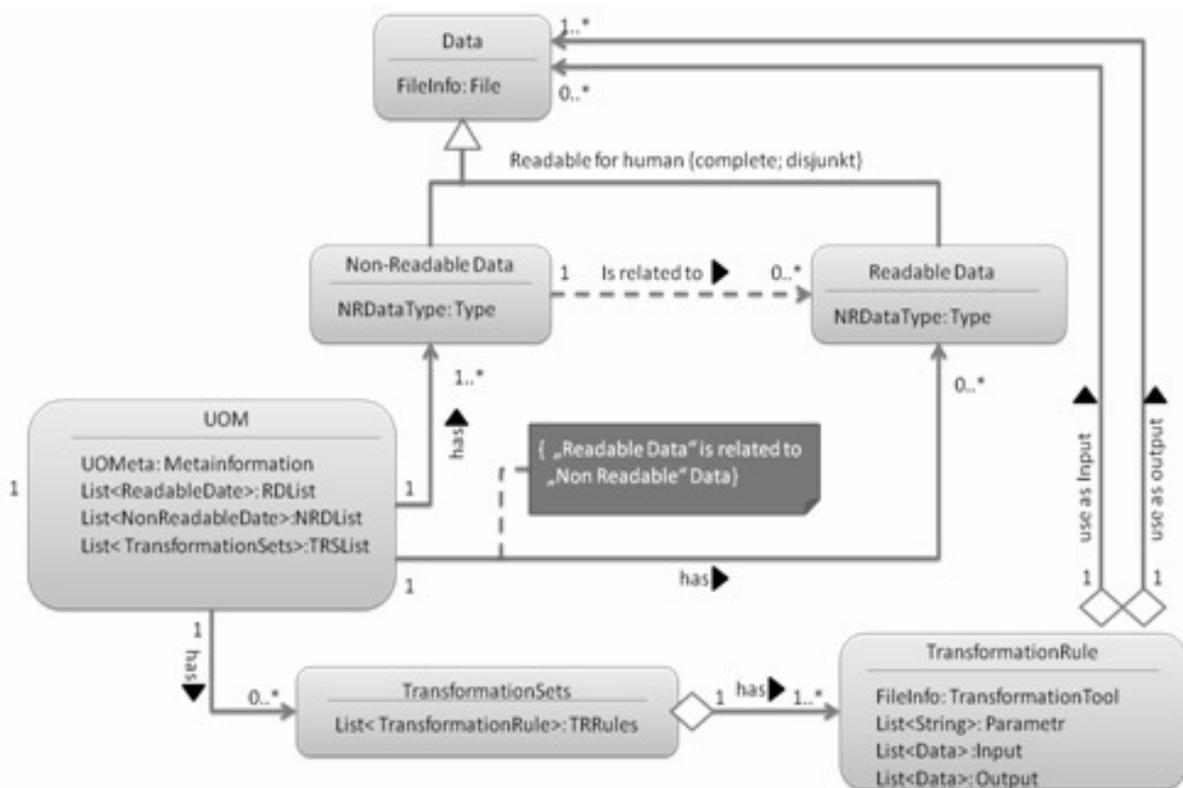


Figure 2: Transformation entities in the information model core

After the transformation of data, these must be integrated into the respective development environment. Therefore, a model is required which recognises the destination structures and mounts the input data. Figure 3 sketches the relationship between the transformation and integration model. The left side shows the transformation concept which is explained above. The right side illustrates the integration into an IDE, whereas the output files of the transformation are the input files of the integration.

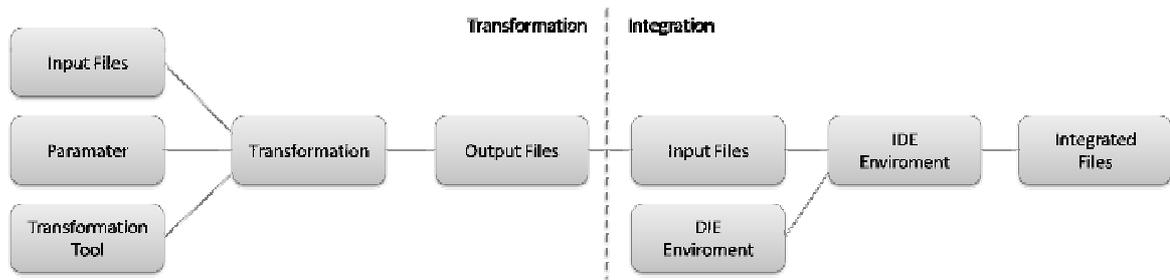


Figure 3: Simple transformation and integration model

3. A service-based software construction CASE-tool

Upon this information, a transformation and an integration model, which are sketched in the last paragraph, a set of CASE-tools were built. This section shows the important points relating to the service-based software construction process.

3.1 System architecture

During the development of the application, the following system architecture was realised (see figure 4):

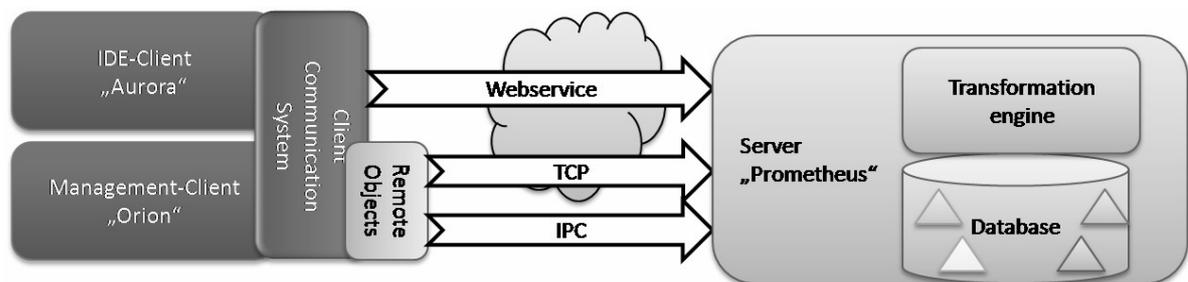


Figure 4: System Architecture

Hereby, the server provides the administration of artefacts and their information, for example, save, delete and search of artefact information. Here, artefact information is saved in a data base. In addition, the server contains the transformation engine. The packaging and provision of this engine, the artefact database and the search functionality through the server as a service belong to the central core of the service-based software construction and represent one novelty of the service-based software construction approach. The management client provides a simple graphic user interface for the server functionality. On one hand, this serves for the management of artefact information and, on the other hand, as an easy test environment of functions (during the development phase). This client is available as a stand-alone application and as an integrated user interface in the development client. The development environment client serves for the use of the server functions within an IDE as a CASE application. This client is able to query for artefact information. The representation of detailed information occurs in an additional web browser as a window within the IDE or in an external browser. In contrast to the management client, the development client does not influence the artefacts such as the deletion of an artefact on the server. The real job beside the search of artefacts and units of

modelling is the transformation and the integration of transformation results in the current development project. Three different forms as a communication-form are possible : Inter process channel (IPC) (Microsoft, 2009), Transmission Control protocol (TCP) (Microsoft, 2005) and a web service. The IPC serves as a remote means of communication between processes within local systems, as for example client and server are executed on the same machine. In contrast, TCP serves for the communication between client and server and, as a result, the programs can be located on different systems. At closer inspection it can be recognised that IPC and TCP constitute the basic communication methods. The web service, shown in figure 4, serves the provision of a uniform interface and uses the basic communication methods again.

3.2 Interfaces

The system exhibits interfaces in two points: 1. with the use of the remote object for service-based construction. 2. With the use of the web service for service-based construction. The first includes features such as create, update and create artefacts or UOMs. In addition, another remote object for the control of the server is available. This is not explained closer since it concerns standard methods for the control of an application (for example restart and stop). Figure 5 shows the interfaces of the remote objects and the web service.

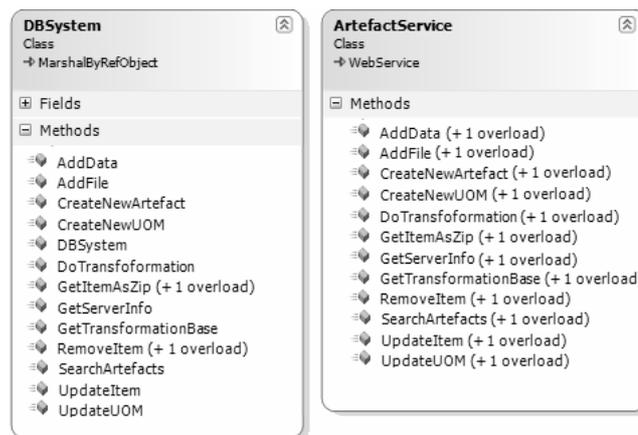


Figure 5 – Remote object and web service interfaces

The basic methods of the interface DBSystem exhibit self created data structures (classes) as parameters which are technology specific (in this case .NET technology). To guarantee the compatibility of the SCA and UOM description with other technologies, all self made (description)classes were made serialisable in the first step. An example for this is that the objects of such classes can be transformed into a form of XML. Since XML is basically a text based form, it can also be converted into other technologies, for example, by the use of web services. Furthermore, attention was paid to the fact that all basic types used in this approach are also serialisable. In the second step, overloaded methods (secondary methods) were developed which contain only serialisable data. This includes, for example, that parameters as well as return values are textual information. Furthermore, serialisation is important to exchange describing data. Figure 6 shows an example of a simplified serialised SCA description which includes a UOM and meta information.

```

<Artefact TypeOfArtefact="Function" ID="1b929d23-ec39-4926-856f-50cc7bf097b4" CreationDate="03.07.2009 08:58:51"
  Path="1b929d23-ec39-4926-856f-50cc7bf097b4">
  <ListOfUOMS>
    <UOM UomType="Component" ID="ec3e9c49-625c-415e-8c49-c400a00f9614" Path="ec3e9c49-625c-415e-8c49-c400a00f9614">
      <TransformationSets />
      <ListOfFileStructure />
      <Metainformation Author="TestInformation" Description="TestInformation" Version="TestInformation" />
    </UOM>
  </ListOfUOMS>
  <Metainformation Categories="MathFunction" Server="Dedalus" WebSite="www.marcuszinn.de"
    ChangeLog="" Description="Gauss function" Version="1.0">
    <Categories>
      <string>Zinn</string>
    </Categories>
  </Metainformation>
</Artefact>

```

Figure 6: Example of a serialised software construction artefact

3.3 Transformation engine

The transformation engine is part of the repository server and can be used by a web service call. It transforms input information into another form by executing transformation rules sets. These rule sets are stored in UOMs. Each rule set consists of transformation rules. A rule in turn contains a program fetch of a transformation tool, a set of parameters which correspond to the transformation tool and input files (see figure 3). These input files can be files of the UOM or output files of a previous rule in the same rule set. Example: To transform WSDL information into a C# client and server stub, the SVCUTIL tool of the Windows SDK can be used. SVCUTIL needs the parameter “language:C#”. So the program fetch is “*svcutil.exe test.wsdl language:C#*”. The output of this call is a C# file (test.cs) which includes different server and client classes and all types are needed. The transformation rule set contains only one transformation rule. This rule is based on a transformation base (svcutil.exe) and two corresponding parameters (language:C#) and the input file (test.wsdl) which is part of the original UOM. The developer now selects this transformation and the repository server executes it. The result will be transferred to the user. If the user uses the IDE client, the result will automatically be added to his project environment.

4. Conclusion and future work

The service-based software construction process is an extension of procedure models and focuses on reuse. It consists of three parts and contains technical innovations. The first part is the idea of a common database for software construction artefacts (SCA) and units of modelling (UOM). This means all necessary information of a software unit (class, component, service), such as implementation, documentation, specification and test information, will be stored in a UOM. Thus a UOM includes all data which was produced during its development. This data can be used by different roles (like software architect or developer) and in different development phases (like requirement or development phase) for reuse. An SCA includes all UOMs which refer to the same problem as a solution. Therefore, the developer can easily manage and search for technical solutions. This stored data focuses on reuse. This means that it is a software unit reuse database / content management system. It

is important for reuse to have a single system to manage entities and their knowledge which can be reused. Users, like software architects or developers, use different data to make decisions. If this data is stored at one place, it is easier to find and use it. The second part includes a transformation engine which can transform UOMs. Transformation means that a UOM can be converted into another form, as for example a Java binary code into a .NET binary code. This also implicates that possible transformation rules will be stored inside the UOM. These rules should be able to transform the implementation of a UOM implementation into another technical or domain specific form and give added value for the reuse of the UOM. Thus the reuse database becomes extended with transformation rules. Together with this transformation engine and the underlying transformation model the database changes from a content management system to a software construction system. At the moment no system is existing which stores all UOM data combined with transformation data and functionality to generate an added value. Therefore, the shown system is an innovation. Part three is a service which offers all the database and transformation functionality to the developer. The innovation of this service is that the complete functionality of the construction system (managing, searching and transformation) is capsulated in one single interface. By using a service as a communication node to such a software construction system, different developers can use one single repository from different locations. As an added value of this, the costs for software developing will be reduced. Lower license fees for different content management systems and transformation tools is only one example for this reduction of costs. The system shown in this publication shows the important parts of the service-based software construction process as a technical implementation. It also shows, that the up to now ascertained models (communication, artefact information and transformation) can be realised in a CASE-tool. The reason is that the ascertained Use Cases were realised. That means construction, search, deletion and fitting of artefacts or units of modelling are basically possible. It was also shown that transformation and use of transformations on artefact information is possible.

Some details of the important data model are not shown in this publication if the output of a transformation rule is saved in a stand-alone unit of modelling, such as the solved consistency problem. But this approach still has open question. One of these questions is the need for a semantic search. By using all possible information, as for example documentation and specification with a full text search, the search results will not be very useful. The system shown in this publication uses full text search. But the developer needs a system which only offers him applicable results for his search question. Therefore, semantic search is an important research area for the service-based software construction process. Another important question is the research for other methods of transformation. The transformation model shown in this publication only supports tool-based transformation. It must be analysed whether there are other transformation methods which must be added to this model.

5. References

Czarneck, K. and Eisenecker, U. (2000), "Generative Programming", Indianapolis, USA Addison Wesley. ISBN: 978-0-201-30977-5

- Czarnecki K. and Helsen S., (2003), "Classification of Model Transformation Approaches", *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, pp. 32-49, <http://www.softmetaware.com/oopsla2003/mda-workshop.html>
- Daneva, M. and Terzieva, R. (1996), "Assessing the Potentials of CASE-Tools in Software Process Improvement A Benchmark Study", *Proceedings of the Fourth International Symposium on Assessment of Software Tools*, pp. 104-108. Toronto, Canada
- Deneva, M. (1999), "A Best Practice Based Approach to CASE-tool Selection", *Proceedings. Fourth IEEE International Symposium and Forum on Software Engineering Standards*, pp. 100-111, Curitiba, Brazil
- Frijters, J. (2008), "IKVM.NET Home Page", <http://www.ikvm.net/>, (accessed 02 2009).
- Garcia-Magarino, I. and Gomez-Sanz, J.J. (2008), "Model-based Methodologies Framework for Defining Model Language Metamodels for CASE Tools", *Proceedings of the 2008 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, pp. 14-23, Budapest, Hungary
- Hitzler, P., Kröttsch, M., Rudolph, S. and Sure, Y. ,(2008), "Sematic Web: Grundlagen", Springer Verlag, Heidelberg, Germany
- IEEE (2007), "IEEE Recommended Practice for CASE Tool Interconnection — Characterization of Interconnections", *Software and Systems Engineering Standards Committee*, <http://ieeexplore.ieee.org/>, ISBN: 0-7381-5233-1
- MConnel, S. (1996), "Who cares about software construction.", *IEEE, Software Vol. 13 Issue 1*, 01, pp. 128-129
- Meimberg, O., Petrasch, R., Thoms, K. and Fieber, F. (2006), "Model Driven Architecture.", Heidelberg: DPunkt, ISBN: 3-89864-343-3
- Microsoft (2009), "Interprocess Communications", [http://msdn.microsoft.com/en-us/library/aa365574\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa365574(VS.85).aspx) (accessed 06 2009).
- Microsoft (2005), ".NET Remoting 2.0", <http://msdn.microsoft.com/de-de/library/bb979586.aspx> (accessed 06 2009).
- Papazoglou, P., Traverso, P., Dustdar, S. and Leymann, F (2007), "Service-Oriented Computing: State of the Art and Research , available Challenges", <http://ieeexplore.ieee.org> (accessed 12 2007).
- Pfleeger, S. L. and Atlee J. M. (2009), "Software Engineering Theory and Practice", 4th Edition, Prentice Hall, USA, ISBN: 978-0-13-606169-4
- Stahl, T., Völter, M., Efftinge, S. and Haase, A. (2007), „Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management“, Dpunkt Verlag, Heidelberg , Germany, ISBN: 978-3-8986-4448-8
- Stuckenschmidt, H. (2009), „Ontologien“, Springer Verlag, Heidelberg, ISBN:978-3-540-79330-4
- Sommerville, I. (2007), „Software Engineering“, Vol. 8th, Addison-Wesley, Munic, Germany, ISBN: 978-3-8273-7257-4

Szyperski, C., Gruntz, D. and Murer, S. (2002), “Component Software Beyond Object-Oriented Programming”, Vol. 2nd Edition, pp.35-48, Addison-Wesley., New York, USA, ISBN 0-201-74572-0

W3C (2004), “W3C Web Service Glossary”, <http://www.w3.org/TR/ws-gloss/>, (accessed 12 2006).

Wang, G. and Fung C. K. (2004), “Architecture Paradigms and their influence and impacts on component-based software system”, pp. 902 72a, *Proceedings of the 37th Annual Hawaii International Conference on System Science*, ISBN: 0-7695-2056-1

Zinn, M. (2007), “Service based software construction process.”, *Proceedings of the 3rd collaborative research symposium on Security, E-learning, Internet and Networking (SEIN '07)*, pp. 169-184, Network Research Group, University of Plymouth, Plymouth, GB. ISBN: 978-1-8410-2173-7

Zinn, M. (2008), “Definition of software construction artefacts for software construction.”, *Proceedings of the 4th collaborative research symposium on Security, E-learning, Internet and Networking (SEIN '08)*, pp 79-91, Network Research Group, Wrexham, GB, ISBN: 978-1-8410-2173-6

Finding Reusable Units of Modelling – an Ontology Approach

¹Marcus Zinn, ²Klaus P. Fischer-Hellmann, ¹Andy D. Phippen, ²Alois Schütte

¹Centre for Security, Communications and Network Research
University of Plymouth, United Kingdom

²h_da – University of Applied Sciences Darmstadt, Germany
mail@marcuszinn.de

Abstract: Today’s software units (classes, components and services) have a huge number of information that is needed or produced during the development and use of these units. In fact, a single piece of information can have different values depending on the point of time in the entire lifecycle. The availability of certain information, as for example documentation, determines among other things the capabilities of a unit. Again, other information is necessary and critical for the success of the entire development process when applying certain procedure models. Retrieval of these units and their contents is important for re-use. There are no suitable models that consider the different units and their contents. Also the current searching behaviour of software developers and architects has not been covered yet. Due to this fact, the benefits in performing reusing software units and the development of software processes are decreasing.

This paper discusses an ontology approach that can be used as a foundation for the search of such units. Moreover, this part of the ontology is focused on the actual searching behaviour of software developers and the finding of units.

1 Introduction

In the object-oriented software development, different units of modelling are used. Every type of unit provides a different amount of information that can be used differently [ZFP09]. Typical units are classes, components and services [WF04]. In the scope of this paper, a component has the meaning of a deployed component. There are two problems: development issues related to a common view of these different units [WF04] and the search for these units [WJS09]. The search for units as a research subject has already been studied for some time. [Pr91] and [MBC91] proposed first approaches. [Ga06] and [LAP04] show a list of the different attempts that have been developed until now. Among other problems, the following problem has been identified:

“Efficient search and retrieval is needed, to assure that the developer is capable of finding previously built reusable assets.” [Ga06]

For this reason, the question arises what an efficient way for a search could be. The current research focuses on the use of semantics in form of ontologies as a foundation of a search (see [TSB09] and [BSW08]). Some of these new attempts focus on the representation of the technical circumstances, as described in [HNK09]. Other studies concentrate on the grammatical structure in such a search [WJS09]. These approaches assume a complicated predefined input behaviour. [He94] showed already in 1994 that there is a significant gap between the description of the problem and that of the solution. Therefore, components are described functionally whereas the searcher actually describes the problem.

In the following paragraphs, the results of the analysis of the present “searching behaviour” of software engineers are presented. Based on these results, an existing ontology is extended. This work is part of a research on a service-based software construction process (SSCP) incorporated the field of Software Reuse Environments. The paper contributes to the research area with the enhancement of an ontology for supporting the search of units of modelling. Aim of this paper is to define the extension of an ontology in order to reflect today’s searching behaviour of software engineers. This can be used in a semantic model to find units of modelling. Therefore, the input behaviour must be determined and modelled. Furthermore, an ontology defined by the authors within the scope of the basic research should be extended. This paper concludes with the fact that the input behaviour does not have to be changed when searching for reusable units in order to achieve exact results.

2 Analysis of the searching behaviour for units of modelling

To get a first impression how software developers tend to search for reusable pieces of software, a questionnaire was given to a group of software development experts. Conducting a representative survey is left as future work in this research. The 15 participants of this questioning were software developers, software architects, and technical project managers who had at least a three-year experience in software development. When the survey was performed, the test persons were active in different software development projects in one of the following areas: CAD, automation, power, or in general software development. 93% of the feedback indicated the use of a general search engine (in most cases www.google.de) in order to search for units. A relationship can be seen in the examples given by the test persons (i.e., “Class C# Device Discover”) and the given search criteria (e.g., manufacturer and technology). Therefore, the analysis shows that there is important and optional information in this relationship (see Layer 1 in Figure 1). The result of the questioning is presented in the following paragraphs as “actual searching behaviour”. Besides, it constitutes a hypothesis of the authors. From the examples of the search enquiries, the analysis of the given search information is displayed in Figure 1. The important information from Layer 1 refers to the functional application object (or content purpose) for that the functionality of the unit (Layer 2) is searched. Layer 2 corresponds to the following structure:

Searched technical contents (application object) + optional describing information (for the technical contents and/or the technical unit).

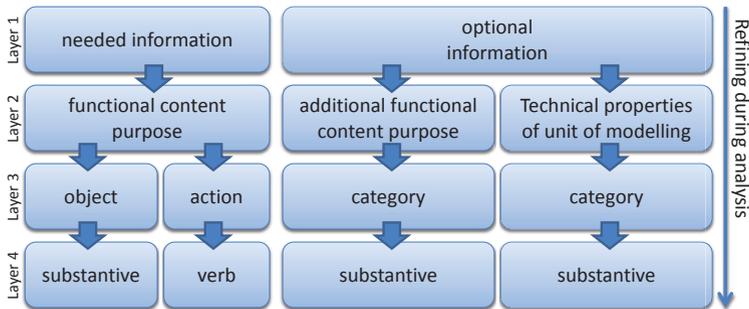


Figure 1: Structure of the search input

In addition, the given examples of the search show that the functionality is described in most cases by an action and an object connected with this action, for instance, “printer” (object) “search” (operation or action) (Layer 3). This example shows the simple substantive-verb relationship in the grammatical area (Layer 4). [WJS09] groups such relationships within the scope of the parsing for search algorithms and refers to it as “Advanced Similarity Word Pair”. Optional information (Layer 1) is divided into two areas. On the one hand, the application object (functional content purpose) is further described. On the other hand, the technical properties of the unit itself can be described (Layer 2). In both cases, categories are used, for example, “WebService C# Device Discover”, and it can be assumed that a web service based on the C# technology is searched. In the grammar of the search, it only concerns a few substantives. From the pattern shown above, the following grammatical construction can be derived:

Functional content purpose (Substantive + Verb) + additional functional content (Substantive) + technical content (* Substantive).*

Beside the grammatical construction and the contents of the search, another important point reveals itself in the analysis: The “problem-solution” relationship. All test persons described the solution in their search (e.g., a class carries out a function for the Gaussian algorithm), but not the problem. Therefore, this factor is interesting because a component may solve different problems (perhaps also in a different way). Furthermore, a problem can refer to several solutions. When questioning the participants why they do not search for the other position (in this case the problem), the answers were quite different. Two types of responses were mentioned remarkably frequently:

- 1) During the search for the problem, solutions can be hardly found.
- 2) During the search for the problem, the problem must be described precisely in order to find a precise solution.

This stands in contrast to the statement from [He94] that the searching person describes the problem, not the solution.

3 Problems of finding units of modelling

Nevertheless, the approach described in Chapter 2 contains some problems:

Full-text search: The use of the search engines making text comparisons can lead to false or not usable results [TSB09].

Substantive verb description: A simple substantive-verb structure in a relationship-based search engine faces following problems: On the one hand, the substantive can be selected unspecific ally (i.e., device), although a printer can be the searched object. This entails the generation of unsuitable hits during the search. On the other hand, verbs and substantives can have synonyms (e.g., graphics and display graphics) or they can be wrongly associated [WJS09]. The last example also shows that in some cases it may be a matter of interpretation. From the point of view of the automation, “machine is computable” seems illogical because a machine does not change. However, this statement makes sense from the CAD point of view because a machine must be recalculated by the change of knowledge-based properties [CI06]. This includes the reconsideration of engine space due to the update of cubic capacity size. This instance can lead to a change of the whole vehicle. In addition, a problem arises concerning the existence or non-existence of a word in another language. Thus, a search launched with the German expression “Gerät suchen” will not be able to find a component described as “Device discover”.

Consistency of the statement: The shown example “web service C# Device Discover” does not state to which “web services” and “C#” they really refer. Hence, a search formulated such can lead to false results:

- 1) A web service written in C # is searched that performs “Device Discovery”.
- 2) A component is searched that contains a web service or rather uses one and performs “Device Discovery”. This component should have been developed in C#.

For an exact allocation of the given information, other details are missing. In this example, the information is clearly allocated to “C#”. The problem in this case is that such an input does not specify whether the information is optional or mandatory.

Problem-solution relation: A search for the solution as described above presents all the solutions that fit to given keywords and their relations. By means of ratings (evaluations, frequency of the choice, etc.), statistical probabilities can be determined for the best result [Ga06]. It is, however, an open question whether information about the problem is missing or has already been considered satisfactorily in the solution description. In spite of these significant problems and open questions involved, the survey shows that this searching behaviour nevertheless is actually used. Therefore, an attempt was made to cover the existing searching behaviour in an ontology. On this basis, it can be investigated how to improve the search result while using the same input behaviour.

4 A search ontology for reusable units of modelling

4.1 Structure

In context of the current research, an ontology to the subject “service-based software construction process” was developed by the authors in order to counteract the problems explained in Chapter 3. This ontology serves only the search of units of modelling. Here, certain modelled properties were incorporated by other ontologies (e.g., technical component properties from [Ga05]) since these have already been edited. This also includes the description of technological facts (components, services, etc.). Figure 2 shows the distribution within this ontology.

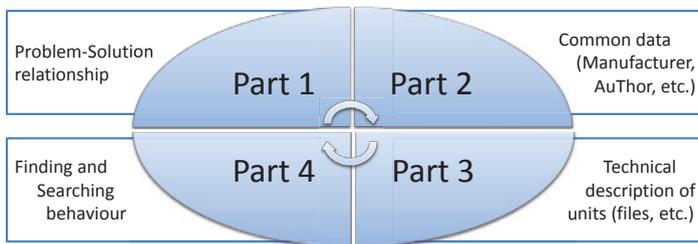


Figure 2: Structure of the reuse ontology

Part 1 shows the access to the ontology: “the problem-solution approach”. This is still the untreated part of the whole research. Part 2 contains general “business information” about the solution as, for example, manufacturer, name, and author. In Part 3, the solution is described as a technical unit; that is, type of unit, technology, file format, files, etc. In the fourth part, the technical contents are described. Possible descriptions are made, for example, in form of a substantive-verb combination and they also contain some optional information. This part of the ontology will be described in this publication.

If an instance of the ontology is generated (e.g., by the registration of a newly developed unit), the user must specify various information that is stored in the suitable areas of the ontology. Furthermore, the data can be entered automatically into Part 3 of the ontology, for instance. This is possible because the technical data is automatically detectable such as file size, file type, file name, and technology. Nevertheless, the data from the other sections of the ontology is not automatically detectable.

In the following, the modelling of the searching behaviour displayed in Chapter 2 will be described in more detail. This corresponds to Part 4 of the ontology. Moreover, it is focused on the problems indicated in Chapter 3.

4.2 Description of the technical and professional contents

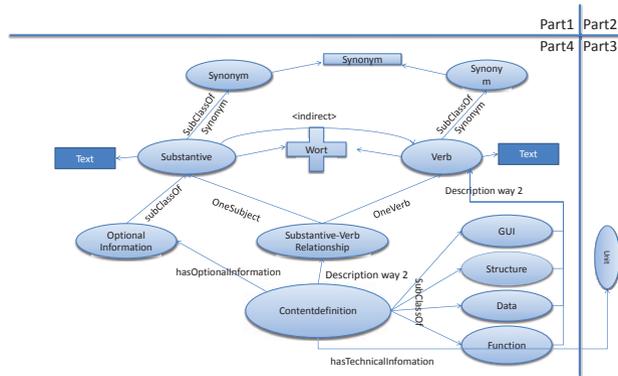


Figure 3: Model of the finding based upon substantive-verb relation

A unit has a so-called “content definition” describing the technical contents. It also uses two different ways of description. These ways are related to each other. The modelling of the optional information for the technical properties is made in Part 3 of the ontology and will not be described in this publication.

4.2.1 Way of description 1

The first way of description is the substantive verb combination explained in Chapter 2 and represents a technical and domain specific description of the contents. In detail, each element (substantive and verb node) of this tuple has a text field and each substantive and verb can have a translation. Within this ontology, this corresponds to a text. At this point, however, an ontology shortcut to a language ontology allowing translations is planned. For this reason, Figure 3 is simplified and as a result the element “word” (+ icon) is displayed as a shortcut to a word ontology. Based on it, cross-language searching and finding elements are possible. Therefore, “device discover” corresponds to “device search”. Similarly, the translations will also proceed with synonyms. A “printer” is a special “device”. Thus, the search for a device may deliver “printer” if appropriate. At this point, an ontology can also be used (“- icon” synonym).

The optional information for the application object is also displayed and modelled as substantives. In order to create the description of the technical properties, Part 3 of the ontology is related to the content definitions object.

4.2.2 Way of description 2

The second part of the description of the solution defines the technical contents from the point of view of its intended purpose. This definition is based on the fact that a unit of modelling may be seen from three different perspectives: Functional contents, technical properties, and technical contents. As previously presented [ZTP08], a component carries only one certain technical content type. Therefore, a component offers either func-

tions such as simple data, user interfaces or it provides structure information to the solution of a problem.

4.2.3 Search variations on the basis of the ways of description

Because of the mentioned features, an ontology-based search can be simply expressed: for instance, “function device discovery”. In this example, “function” represents optional information. Hence, an attempt was made to generate an indirect relationship between the technical and the professional contents. Two variations have evolved that will be explained in the following.

Variant 1: Verbs as synonyms for technical content types

In this case, four contents types (Data, Function, UI and Structure) are associated with certain verbs. These verbs fit to the content type (e.g., a function is a content type of something that executes something; UI is a content type of something that illustrates). Thus, a function can be “executed” or graphics can be “illustrated”, for instance. Table 1 shows some examples of a possible assignment.

Content type	assigned verbs
Function	calculate, execute, accomplish, bear, manage
Data	offer, suggest
UI	show, present, demonstrate
Structure	structure, align, regulate, arrange, classify

Table 1: Technical content type-verb relation

For each instance of the ontology, this allocation would be firmly “wired”. Moreover, only a few verbs are associated to the content types. With the help of this assignment, the search could be “execute device discover”. In addition, synonyms and translations are available for this search.

Variant 2: Direct links of the substantive verb tuple with the content types.

In contrast to variant-1, the verbs from the substantive-verb tuple are now connected directly with the content types. Although the allocation from Table 1 can be maintained, every entered verb, however, must receive an allocation. A result would be that the search enquiry “device discover” will search for units offering a function that searches for devices. In comparison to the first variant, only two words are required instead of three.

4.2.4 Realisation of a search

To launch the search, a search query must occur at first such as “device discover C#”. Part 4 of the ontology can be used for the identification of the substantive-verb tuples. All the other terms (in this case “C#”) are understood as optional terms and are searched

for within the remaining parts of the ontology. C# is a technology whose relationship with the component is modelled in Part 3 (technical information). From the perspective of the ontology, the search is called “unit has content definition with tuple (device-discover) and has a relationship with C#”. Moreover, with the perspective of the search in Variant 2, it is obvious that the user searches for a function and not for simple data, a user interface, or structure information.

4.3 Problem solution

In Sections 4.1 and 4.2, the ontology does not solve every problem mentioned in Chapter 3. The use of ontologies in order to avoid the problems of text-based search is not new [TSB09]. It is already known that because of their logical structure, ontologies are suited to perform inheritance hierarchies. An example that could be expressed is “device is a printer” [St09]. The use of simple substantive-verb tuples describing the technical contents with fixed verb-content mapping presents a novelty. In comparison to a 100% semantic search with input methods considered complicated [WJS09]; this approach can lead to more “wrong” search results. However, a “substantive-verb content type” triple can arise from a search enquiry. This is the result of the semantic assignment of substantive-verb-tuple to a unit as well as the allocation of the verb-content type “substantive verb”. As a result of such a search, only the units owning this triple are performed. In contrast to a text-based search that seeks words in all texts of a data record, the input words are analysed in their relationship and are only searched if they are related to that relationship. The optional information is used to improve the search result. Variants 1 and 2 from Chapter 4.2.3 indicate that there are different possibilities to model the relations between professional and technical contents. Variant 2 is identical to the searched input behaviour but provides more exact results. Table 2 shows the search results of the given input “device discover C#”:

Dataset	Text-based search	Variant 1	Variant 2
Some component with the description “device Microsoft discover c#”	hit	no hit	no hit
Some component with a description triple “device-discover-function” and optional description “Microsoft c#”	hit	no hit	hit
Some component with a description triple “device-discover-data” and optional description “Microsoft c#”	hit	no hit	no hit

Table 2: Example of search results

The text-based search in Table 2 provides a hit for each data record because the searched data is available. Variant 1 delivers no hit because the search enquiry does not display the substantive-verb-content type triple. Variant 2, however, delivers exactly one hit. Although only the tuple “device-discover” was entered, the triple “device discover function” was implicitly also searched. This reduces the number of possible hits in contrast to the entered tuple.

5 Conclusion and future work

The ontology approach shown in this paper contains a semantic modelling of the following (search) input pattern for the search of reusable units:

Functional content purpose (Substantive + Verb) + additional functional content (Substantive) + technical content (* Substantive).*

This allows the searching behaviour that appears to be broadly applied nowadays to text-based search engines also to be applied to semantic search engines. Thereby, it is possible to make use of the usual advantages of an ontology as, for example, using a shortcut to other ontologies and the advantages of a semantic search (see [St09]). This leads to a better result in contrast to a text-based search (see [TSB09]) because a text-based search only compares the searched words with the dataset. In order to have an exact result, the searched words must be in a certain semantic relationship (*Substantive + Verb*) and must be implicitly combined with the technical content type (Structure, UI, Data or Function) of the searched unit. This simple approach combined with the typical information about reusable units of modelling (i.e., manufacturer and technical information) represents an innovation to the area. This paper shows in an example that this approach works. The search pattern can be grasped completely in an ontology without changing the effort or the input for the user. In addition, this publication shows that the input behaviour of software engineers identified as typical does not have to be changed and a better search result can be achieved in comparison to a text-based search.

However, not every problem is solved by the solution presented in this publication. On the one hand, it is not finally clarified whether it is better to describe the solution only with a search or whether the problem should be described as well. On the other hand, the possibility to recognise whether the optional information describes the application object, the technical unit, or the technical contents is missing. Within the scope of the further research of “service-based software construction”, these two open problem formulations, in particular, will be analysed in more detail. However, the searching behaviour still is supposed not to change.

References

- [BSW08] Bast, H.; Suchanek, F.; Weber, I.: Semantic Full-Text Search with ESTER: Scalable, Easy, Fast. In: International Conference on Data Mining Workshops (ICDMW '08)-Proceedings, Pisa Italy, 2008; pp. 959-962
- [Cl06] Claassen, E.: Protection of Intellectual Property in the Product Development Process. In: Proceedings of the 11th Seminário Internacional de Alta Tecnologia, Universidade Metodista de Piracicaba, Brasil, 2006
- [Ga06] Garcia, V. C.; De Almeida, E. S.; Lisboa, L.B.; Martins, C. A.; Meira, A. R. L.; Lucredio, D.; De M. Fortes, R. P.: Toward a Code Search Engine Based on the State-of-Art and Practice. In: Asia Pacific Software Engineering Conference-Proceedings, Bangalore, India, 2006; pp. 61-70

- [Ga05] Gangemi, A.; Grimm, S.; Mika, P.; Oberle, D.; Lamparter, S.; Sabou, M.; Staab, S.; Vrandečić, D.: Core Software Ontology - Core Ontology of Software Components - Core Ontology of Services, 2005, online available at <http://cos.ontoware.org>, (Accessed 14.1.2009)
- [He94] Henninger, S.: Using Iterative Refinement to Find Reusable Software. In: IEEE Software Journal, Vol. 11(5), 1994; IEEE; pp. 48–59
- [HNK09] Hewett, R.; Nguyen, B.; Kijsanayothin, P.; Efficient Optimized Composition of Semantic Web Service. In: IEEE International Conference on Systems, Man, and Cybernetics (SMC 2009) Proceedings. San Antonio, USA, 2009; IEEE; pp. 4065 - 4066
- [LAP04] Lucrédio, D.; Almeida, E. S.; Prado, A. F.: A Survey on Software Components Search and Retrieval. In: Proceedings of the 30th EUROMICRO Conference, Rennes, France, 2004; IEEE/CS Press; pp. 152–159
- [MBC91] Maarek, Y. S.; Berry, D. M.; Kaiser, G. E.: An Information Retrieval Approach for Automatically Constructing Software Libraries. In: IEEE Transactions on Software Engineering Journal, Vol. 17(8), 1991; IEEE; pp. 800-813
- [Pr91] Prieto-Díaz, R.; Implementing faceted classification for software reuse. In: Communications of the ACM, Vol. 34(5), 1991; ACM; pp. 88–97
- [St09] Stuckenschmidt, H.: Ontologien – Konzepte, Technologien und Anwendungen, Springer Verlag, Heidelberg Germany, 2009, ISBN 978-3-540-79330-4
- [TSB09] Tümer, D.; Shah, M. A.; Bitirim, Y.: An Empirical Evaluation on Semantic Search Performance of Keyword-Based and Semantic Search Engines: Google, Yahoo, Msn and Hakia. In: 4th International Conference on Internet Monitoring and Protection, Venice/Mestre, Italy, 2009; IEEE, pp. 51-55
- [WF04] Wang, G., Fung, C. K.: Architecture Paradigms and Their Influences and Impacts on Component-Based Software Systems. In: Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS'04), Big Island, Hawaii, 2004; IEEE ; pp. 1-10
- [WJS09] Wang, H., Jing, L., Shao, H.: Research on Method of Sentence Similarity Based on Ontology. In: Proceedings of the First WRI Global Congress on Intelligent Systems (GCIS 2009), Xiamen, China, 2009; IEEE; pp. 465-469
- [ZFP09] Zinn, M.; Fischer-Hellmann, K. P.; Phippen, A. D.: Development of a CASE tool for the service based software construction. In: Proceedings of the 5th Collaborative Research Symposium on Security, E-learning, Internet, and Networking (SEIN'2009), Darmstadt, Germany, 2009; Centre for Security, Communications and Network Research; pp. 134-144
- [ZTP08] Zinn, M.; Turetschek, G.; Phippen, A. D.: Definition of software construction artefacts for software construction. In: Proceedings of the 4th Collaborative Research Symposium on Security, E-learning, Internet, and Networking (SEIN'2008), Wrexham, UK, 2009; Centre for Security, Communications and Network Research; pp. 79-91

Device services as reusable units of modelling in a service-oriented environment - An analysis case study

M. Zinn¹, A. Bepperling¹, R. Schoop¹, A. D. Phippen², K. P. Fischer-Hellmann²

¹Schneider Electric Automation GmbH, Steinheimer Str. 117, D-63500 Seligenstadt, Germany,
{marcus.zinn, axel.bepperling, ronald.schoop}@de.schneider-electric.com

²Centre for Information Security and Network Research, University of Plymouth, Plymouth United Kingdom,
andy.phippen@plymouth.ac.uk, K.P.Fischer-Hellmann@digamma.de

Reusing software components is an important but not standardised task in software engineering. This will become a problem if requirements change in the future. By the use of web services in service-oriented architecture, as a communication interface for devices, the automation area has to define how to handle these reusable units. At the moment, no standardised way is defined. This publication analyses the use of web services in a service-oriented environment as a communication interface of physical devices in the area of automation. An analysis matrix will be built by the use of defined research challenges of web services and known reuse aspects in the area of software engineering. The results of a Device Profile for Web service (DPWS) case study will be compared to this matrix in order to define research steps and important factors for device services for the future.

Keywords: Device Profile for Web service (DPWS), web services, reusability, software engineering, service-oriented architecture (SOA)

I. INTRODUCTION

Service-oriented architecture (SOA) is a conceptual approach to define a scope of services and an environment which will be used to communicate with and between these services [1]. [2] describes three kinds of services: Services of the first and second generation, as well as future services. Services of the first generation were distinguished by the fact that they were independent and not integrated services, as for example the query of data. The demands on services management, quality, operational safety, interoperability, security, and trust were very low. With the growing propagation and rising requirements, the second generation of services was created. This generation characterises itself through the possibility of a bundle of services which are dependent on each other and which can be offered. Furthermore, these services are components of bigger systems which display business processes among other things. Through this integration the requirements of the first generation were expanded by the following points: Lifecycle, Quality of Service (QoS), and service level agreement. This second generation constitutes the "State of the Art" of services. In addition, it is indicated that the next generation must become better at the point of "Easy to use". This refers particularly to the high dependencies of today's services on context. Services, however, offer further topics, which are being examined at the moment. These are, among other

things: service level agreements (SLA), quality of service, and security (see [2] and [3]).

According to [4], the characteristics of a today's services are defined as follows: Services are

- equipped with standard interfaces and flexible collaboration contracts
- able to communicate ("All to all-mode", at any time)
- platform-independent

In addition, services can be searched, browsed, and joined dynamically to form new services or applications. A typical representative of these services is the web service technology. In this case, the already displayed properties of services are described by the web services description language (WSDL) [5].

Beside the use of services in software engineering, different use cases for services exist in other domains, such as telecommunication and trade markets.

Another example is the automation area. In this area, it is important to increase the flexibility, modularity, and reconfigurability of automation systems by using new information and communication technologies [6]. In relation to agent-based systems and SOA, different approaches were developed in this area [7] and [8]. The usage is demonstrated by the fact that SOA supports typical automation requirements like collaborative automation in sense of autonomous, reusable, and loosely-coupled distributed components. [9] and [10] constitute web service as a useful technology approach of SOA in this domain.

One technology approach is the Device Profile for Web Service (DPWS) standard [11]. It extends the basic web services definition with the information required by electronically devices (like footprint, performances, security, and event driven messaging) [12]. Other technologies still exists like Jini, OSGi and UPnp. [13] shows an comparison and a requirements analysis of these four technology.

The aim of this paper is to extend the research map for DPWS given by [14]. This extension is a list of open research tasks that appear if DPWS based web services will be used as reusable units for software engineers in the area of software engineering. The special focus is set on research challenges and reusability of web services in this area. To do this it is necessary to define different criteria for an analysis. The analysis will be done at the example of a DPWS case study. DPWS was selected because of it is the new standard and it is

interesting to see if this standard can be used for all important areas of services.

This paper is structured as follows: After this introduction, chapter 2 outlines the basics of web services as units of modelling in software engineering and relationship to the Device Profile for Web Service standard. Chapter 3 forms an analysis matrix that will be used in Chapter 4 to analyse a DPWS case study with the requirements of reuse of modelling units and the research challenges of web services and SOA. The paper terminates in chapter 5 with the conclusion and a roadmap extension for future research challenges for the DPWS standard.

II. WEB SERVICES AND DPWS: "STATE OF THE ART" IN THE AREA OF SOFTWARE ENGINEERING

A. WEB SERVICES

Ref. [15] shows a comparison of different service technologies to CORBA Trader, JavaBeans Context, Jini, OSGi, and web services. Services, however, are only components with a special interface. A SOA constitutes an architecture to implement and execute these components. Programmes that are based on an architecture like this use the following criteria of the SOA [4]:

- Loose interconnection/coupling
- Services can have a or no state
- Services constitute the building blocks for modelling and development
- The core of the architecture constitutes the service definitions, descriptions, search- and access-protocols and quality information.
- Services are self-describing
- Search and find-functions

The actual combination of services to a service-based application is divided into two different approaches: Orchestration and choreography. Ref. [3] describes how orchestration services interoperate. This includes business logic and the order of execution of single services which are controlled by a single endpoint. This form of construction is very popular in order to copy model processes. A typical process-based modelling language is BPEL, for instance [16]. This approach finds its use, for example, in Enterprise Service Bus (ESB) or Enterprise Services Hub (ESH) that constitute a middleware for communication [3].

In contrast to the orchestration which serves the execution of a business process choreography constitutes a semantic approach [3]. It is defined on how different endpoints can communicate with each other.

Today's service-based construction is based on services at runtime (cf. orchestration). Nevertheless, some researches indicate that in the scope of the software development cycle, services can also be inserted at other points in time, such as in design and compilation time [17]. Therefore, the difference to the component-based construction arises through the fact that the units of modelling are services, i.e. loosely linked components. This form of construction only works with interfaces, instead of typical components which are context dependent. The implementation and dependencies of the services are secondary or of no interest. For this reason,

service-based construction approaches do not only support the development of software, but also the development of systems [18]. [3] and [2] put up additional research directions for service-oriented software engineering (SOSE).

In the commercial area, the approaches Windows Communication foundation WCF [19] and SCA (Service Component Architecture [20]) have proved to be particularly useful for developing with services. The companies of products for software development have already upgraded. The company Intel even goes one step further and uses services not only to access software components but also hardware components. This technology of the service-oriented Infrastructure (SOI) project is based on the approach "*Hardware as Services (HAS)*" [21]. A similar approach is found in the European FP6 Project SOCRADES [22]. In the scope of this project, the DPWS standard was used to build web service interfaces for electronic devices. Another new approach with the aim to generate a SOA and web service based middleware for devices, is the Hydra Project of the European Information society [23].

B. Device Profile for Web Services (DPWS)

DPWS was first used as a unit of modelling to develop a service infrastructure in the ITEA SIRENA (Service Infrastructure for Real-time Embedded Network Application) project.

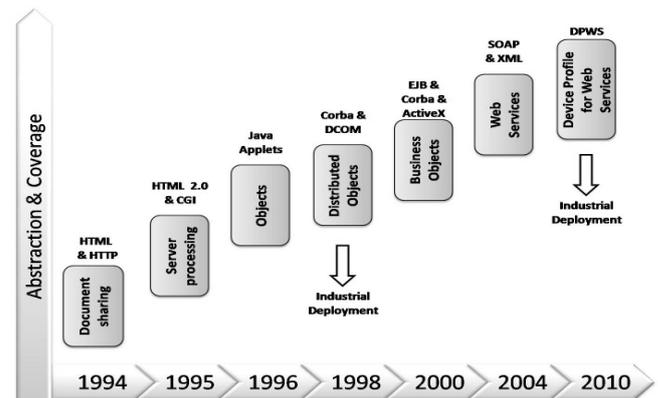


Fig. 1. Interfaces with higher levels of abstraction and more distribution of processing

The focus of this infrastructure was the area of embedded network application [24] Figure 1 pointed out DPWS in the history interfaces abstraction.

Now DPWS is an OASIS standard. As a web service extension, DPWS can use all web service features (cf. section A) and it focuses communication between web services securing messages, dynamic discovering, describing a DPWS web service and subscribing for web service events [1]. This builds upon other web service protocols (cf. figure 2) (for more details see [12]).

, By the use of these protocols, DPWS shows its own architecture which is important to know for the use inside software engineering. By using DPWS, a device has two different types of services: hosting and hosted services.

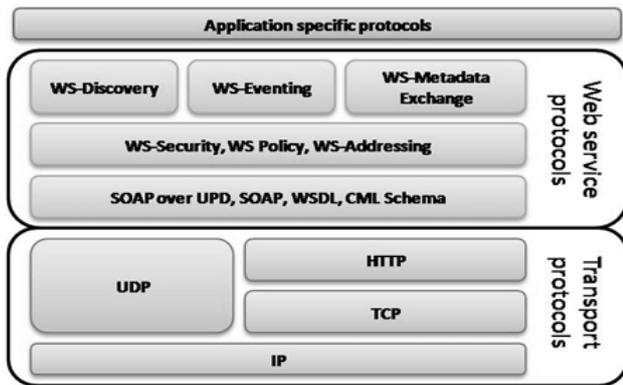


Fig. 2. Basis protocol stacks used by DPWS

A hosting service is a service that represents the device itself. It is used to configure the device, get default data (like name, ID, etc) from it, and publish the hosting services. Hosted services represent functionality that is provided by the device, such as the printing functionality of a printer. Figure 3 shows this relationship.

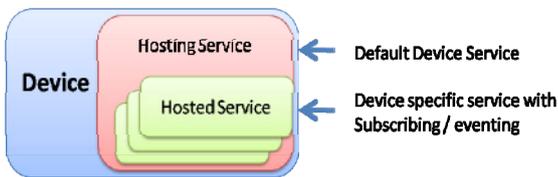


Fig. 3. Hosted and hosting Service in a DPWS Device [Basis 38]

Ref. [24] shows the advantages and disadvantages of using DPWS,

TABLE 1
ADVANTAGES AND DISADVANTAGES OF DPWS

High acceptance	DPWS has a high acceptance in the area of software development and in the automation market. Ref. [25] This is based on the fact that DPWS was created by a consortium of Microsoft and some printer manufacturer. DPWS is a default part of the operating system Windows Vista and is supported by Microsoft .NET. Additionally, there are API implementations for the .NET and the JAVA Framework.
SOAP limitations	The possibilities of communications (traffic) are limited by the used transport protocol. DPWS uses SOAP for messaging at the application level. On one hand, it is good to have a simple system on limited hardware devices. On the other hand, the complexity of the used API will increase.
Functional simplicity	The function provided by a DPWS stack is very simple because embedded devices have limited resources. Moreover, the DPWS is only an infrastructure for hosting services. Thereby, hosting services (including their special functionality) must be developed by protocols the device manufacturer.
Web service compatibility	DPWS is compatible to the normal web service standard. So it has the same advantages as units of modelling in a service-oriented software development (SOSE) environment.

III. CRITERIA OF AN ANALYSIS MATRIX FOR (DEVICE) WEB SERVICES

To make a statement about the reusability properties and the future research challenges of DPWS based web services,

it is important to define the analysis criteria. Thereby, the analysis matrix consists of two parts: reusability criteria and future research challenges criteria of web service. Since a web service is a unit of modelling inside the software engineering, existing analysis of this area will be used to create a criteria list. Thus, the analysis matrix focuses on the reuse of web service and the focus on the future and it will use a simple illustration of the displayed results: ‘+’ (good support, existing) and ‘-’ (bad support, not existing).

A. Reusability criteria for web service

The reusability was considered to be a big advantage of object orientation. However, the reuse is still a considerable problem in the (object-oriented) world [26].

“Component-based software engineering is an approach based on reuse for the definition, implementation, and composition of loosely coupled, independent components to systems.” [27]

Components are the units of modelling of component based software engineering. Because of the fact that web services are very similar to components [27], the important properties of components can be used as reusable factors for web services: Context dependence, component models and component worlds, vertical and horizontal markets, Reuse at design level and Quality factors.

Context dependence [27]: Context dependence can be defined as a dependence on an element which lies beyond the sphere of influence of a component. An example is the libraries of a runtime environment which are assumed by a component. Similar to object orientation, the purpose of a component is to be reused for a certain target area. With the rising number of context dependencies which should increase the reuse in a suitable context, the possibility of use, however, decreases. Normally, web services are known as components without direct context dependencies. But the system, which is hosting the web service, has these dependencies. As a result, the user has an indirect connection to the problems of context dependence.

Component models and component worlds [27] and [28]: Special context dependence is the fact that a component belongs to a component model. Beside the exact form and the properties of the components which correspond to the model, a component model also specifies how components can *speak* with each other (interaction standard) and connect to each other (composition standard). Moreover, a component model can own implementations of different manufacturers (component worlds). Similar to the multitude of object-oriented languages, there are also different approaches of component models which are mostly incompatible to each other. Web services normally do not have this kind of problem.

Reuse at design level: [27] shows four kinds of reuse of components which are differentiated by visibility and convertibility of the component implementation. These constitute a classification. With the help of this classification, the access possibility of information of the components is divided. The problem which originates hereby is that components must account to which reuse class they belong. In

addition to the problem "How information is carried", the question "What is carried?" is still present. The literature shows the following granularity programming- and script languages, libraries, interfaces, messages and protocols, patterns, framework and system architecture [27]. Therefore, [27] indicates that components are not able to carry several pieces of design level information at the same time. Web services have the same problems currently, but they focus only on one kind of reuse classification. A web service is a black box, which means the user only knows the interface and nothing more.

Quality factors: Ref. [27] shows that the question of quality is very crucial for components. It is, however, not explained more exactly how quality is defined with components. Ref. [29] defines the quality of components with the help of formal correctness and completeness of interfaces. Measuring quality factors is very difficult. In this paper the only question on quality is if the used technology supports quality factors. In the resulting matrix this can be used for a common decision about the use of the service. If a service will be used, the user has to look to the own quality requirements and the given quality factors of the service. This can not be provided by this paper.

Additional reuse information: Besides the shown factors other information is also useful for reuse. Ref. [30], [31] and [32] show that the use of the following information increases the reuse rate if they exist and they will be used, for example, for Documentation, Specification and unit test data. Additionally, this kind of data can also be reused.

Repository & discovery: Another important property of reuse is the discovery of reusable data. The "correct" search for data is very complicated and is based on the metadata that can be used for the search. Today semantic search, based on different ontologies, is very popular. This means data and definitions will be connected in one database by the use of semantics. These semantics can be used to find data entities [33] and [34]. The analysis matrix will be extended by the question if a kind of content management system (for example repository) or a subscription system (like UDDI) can be used. But UDDI is only a brokering system. The information about a web service (e.g. documentation, web service location, etc.) must be saved in a single repository system in which a discovery system can search for the information [33].

B. Research challenges of (web) services

As a base for future research, the classification of research topics for (web) services of [3] will be used: Service Foundation, Service Composition, Service Management, Service Monitoring, Service Design and Service Engineering.

Service Foundation [3]: Service Foundation provides an adaptable middleware infrastructure. In this area, especially subject dynamics, quality and infrastructure support are demanded. Ref. [3] notes that in order to permit dynamic (re)configurable service architectures at the runtime, an improvement to the service discovery has to be made. This also assumes a research of infrastructure support of data,

process and application integration. The analysis matrix will use the following criteria based on [3] of this topic.

TABLE 2
SERVICE FOUNDATION CRITERIA

Dynamically reconfigurable runtime	The service runtime infrastructure must automatically adapt (distributed) service components and resources to create an optimal architectural configuration. This configuration has to be optimal for application and user requirements.
End-to-end security solutions	Services, service-based applications and their infrastructure should provide end-to-end security solutions on transport and application level.
Infrastructure support for data and process integration	There is a need for consistent data access, irrespective of the data type (including format, source, or location). Additionally, it should be possible to integrate processes into existing processes. This includes inserting SOA-Applications or web service into existing processes.
Semantically enhanced service discovery	Service discovery have to use semantic expressions. The discovery itself should work with minimal user interaction. So requester and provider of a service need a semantic language that can be used. This language must include semantic annotations combined with QoS characteristics and service description (based on WSDL).

Service Composition [3]: In particular, subjects from the area of business are desirable in the area of service composition. The subject's comparability, use and availability constitute the largest research areas. At this point the following services are listed: for example semantic, typed and plausible services. Since there has not been any standard for quality characteristics in this area yet, researches for possibilities of the "Quality of service" are also pending. Accompanying this, the comparability, which serves as a basis for the research area „autonomous composition of services“, is also to be examined. For the analysis, the following points based upon [3] are interesting:

TABLE 3
SERVICE COMPOSITION CRITERIA

Composability analysis for replaceability, compatibility and	<i>"Service conformance ensures a composite service's integrity by matching its operations with those of its constituent component services. It imposes semantic constraints on the component services and guarantees that constraints on data that component services exchange are satisfied. Service conformance comprises both behavioural conformance as well as semantic conformance. The former guarantees that composite operations do not lead to spurious results, while the latter ensures that they preserve their meaning when composed and can be formally validated."</i> [3]
Dynamic and adaptive processes	It is necessary that services and processes use adaptive service capabilities. This assumes a technology-based extension for services and processes/application using it. With this extension services and processes can "continually morph themselves to respond to environmental demands and changes without compromising operational and financial efficiencies." [3]. CF. self-configuring, -optimising, -healing and -adapting services.

QoS-aware service compositions	Service compositions have to understand QoS information such as policies, performance levels, security requirements, service-level agreement (SLA) stipulations from other services or applications. This assumes that services can interchange these information and automatically adapt to them.
Business-driven automated	Services and service-oriented applications should provide themselves as usable business units. This assumes an abstraction of the technological, application and current business level. This abstraction should enable the composition of distributed business processes and transactions.

Service Management & Monitoring [3]: In the area of service management are interesting, particularly the investigation areas, which deal with the independence and the automation of services. This means services should be self-configuring and thereby adapt themselves to their environment or to the context ("self-configuration services"). Additional interest is in services which "cure" themselves automatically. This includes self-analysis and an independent repair of services ("self-healing services"). Self-optimising services are based on the same idea. Here an independent analysis and criteria are also necessary [35]. Self-protecting services are also interesting, as for example an implementation of security aspects within the services. For the analysis the following points based upon [3] are of interest:

TABLE 4
SERVICE MANAGEMENT & MONITORING CRITERIA

Self-configuring management services	Service has adapted and optimised automatically to a given environment. So they can be easily installed and used.
Self-adapting management services	Service must react on changing requirements of the environment (including market changes). These requirements can include deploying new instances, changing instances or changing runtime characteristics.
Self-healing management services	Service has to detect malfunctions. This calls for special diagnosis, discovery and reaction functionality. These must discover, diagnose, and react to disruptions. The runtime environment should not be affected.
Self-optimising management services	Service has to monitor service components and resources and adapt them to the current end-to-end, process, or business needs. Self-optimising management services improve overall utilization or ensure the timely completion of particular business transactions.
Self-protecting management services	Services have to anticipate, detect, identify and protect it and their resources. This includes detection of hostile activities, such as unauthorized access and use, virus infection and proliferation and denial-of-service attacks. If a threat occurs, the service has to start counteractive measures.

Service Design and Service Engineering [3]: The suggested research areas in the area of service engineering are directed mainly at the dealings with services in software development. Thereby, it is pointed out that there is a lack of

design principles for the creation of services. In addition, there is only rudimentary support or methods for the integration of service development in conventional software development. Ref. [36] and [37] show at the example of the UML that those services can be displayed with UML. However, there is no notation for services in UML. It is also indicated that there is a lack of analysis possibilities for services.

TABLE 5
SERVICE DESIGN & SERVICE ENGINEERING CRITERIA

Engineering of service applications	A service-oriented engineering methodology and an adapted development environment are needed. These have to enable modelling, development, deployment, test and configuration of services and an SOA-based application.
Flexible gap-analysis techniques	"Gap analysis purposes a business process and services a realization strategy by incrementally adding more implementation details to an abstract service/process interface. Such a strategy considers several service-realization possibilities such as green field development, top-down and bottom-up development, meet-in-the-middle development, and development based on reference models." [3]
Service versioning and adaptivity	Developers and Development environments should introduce techniques to discover and select suitable external services, detect problems in service interactions, search for alternative solutions, monitor service-execution sequences step by step, and make appropriate upgrade and version services.
Service governance	Services must meet the functional and QoS objectives within the context of the business unit and the enterprises within which they operate. This is important because a service is part of many views and domains which are cross-organizational. So service-oriented application will consist of service fragments that different organizations must maintain separately.

IV. ANALYSIS OF A "DEVICE PROFILE FOR A WEB SERVICE"-SCENARIO

A. THE DEVICE PROFILE FOR A WEB SERVICE SCENARIO

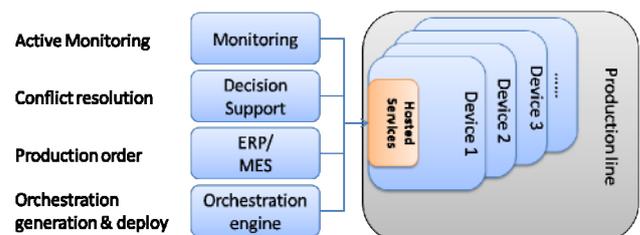


Fig. 4. System architecture of the DPWS scenario

In the scope of the SOCRADES project an experimental DPWS scenario was built. The authors were directly and indirectly participants of this scenario. Figure 4 shows the system architecture. The basic configuration of the architecture is shown in figure 4

The web services of each workstation of the production line (see figure 5) will be "programmed" by the orchestration engine. Thereby, the devices provide the following hosting services:

- Managing of the device (lifter or conveyor)
- Subscription and Messaging via WS-Events

In addition to the orchestration engine, three other tools are using the web services:

- Decision Support System: This system tries to solve conflicts which occur if the panel can use different production line ways (see figure 5 current panel position).
- ERP / MES System: The production orders will be sent by the ERP/MES system. The production line will execute the orders.
- Monitoring System: The monitoring system records the single events coming from different devices and shows them in a graphical way.

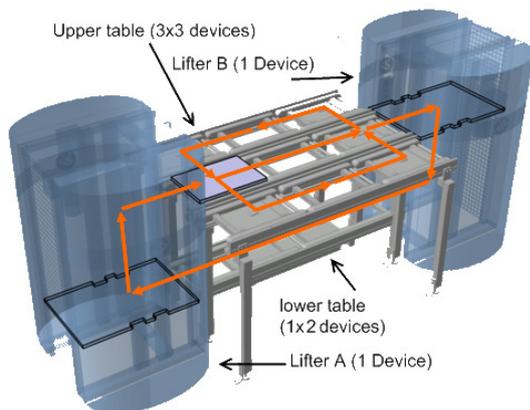


Fig. 5. Configuration of a production line based on [22]

Basically the scenario is based on the following process: With the help of the orchestration engine and a special graphical tool the production process will be created and uploaded into the devices. The Monitoring and the decision system register for the device events. If the ERP system sends an order, the production line tries to execute this order. If a conflict occurs, the decision system tries to help the specific device by sending special orders. The web services are developed with a C-based DPWS API.

B. REUSABILITY ANALYSIS IN A SERVICE-ORIENTED ENVIRONMENT/ARCHITECTURE

For the analysis of the shown scenario there are two different points in time important: development- and runtime.

Reusability comparison

Six criteria of the scope of reusability are used.. Context dependence is the first criteria. To use a DPWS web service a special API is necessary. Therefore, DPWS has context dependencies. In addition, the scenario shows during development time that the different APIs (C and Java) react differently to the events. This shows that the context dependencies are complex. DPWS has an own component model and has only one dependency on the web service model. But there is no special relationship between the typical component worlds like Java or NET. Because the DPWS service is a web service it is also a black box for a developer. Only the interface is known. But normally it is better to know some of the internal processes or structures. Especially the web service is managing an electronic device and not a piece

of software. Normally a DPWS service is offering some simple data like name and version which can be used by a software engineer. But to set this information is not a fix rule. Thus the point “Reuse at design level” decreases the reusability in the area of software engineering. DPWS does not support any extended quality factors for software engineering yet. Also in the scenario, no quality measurement was used. In the topic “Additional reuse information”, different views are important. During the development time, documentation and specification were developed. These artefacts can be “reused”. DPWS only provides small text fields for information during the runtime. Special device description approaches exist [38] which are used during the runtime of the scenario to acquire additional information. So DPWS has approaches to get additional information for reuse. This is also important for the last point of the reusability analysis. In order to find and manage this information, special content management systems are necessary. DPWS itself does not provide any support for this. During the runtime and the development time of the scenario, no system was used. Nevertheless, this is one of the important points for reusability. But DPWS service can be very easily discovered. Thus and the self describing possibilities makes DPWS service extremely useful compared to normal web services

Research challenges comparison

In the area of service foundation, four different factors are defined. During the developing time the hosted and hosting services were built and dynamically loaded into the devices. This shows that DPWS Services can easily adapt to dynamically reconfigurable runtime architectures. For example: if a device is “online”, its services can be easily reconfigured. The second point “End-to-end security solutions” is handled by the DPWS stack. Since DPWS uses WS-Protocols it can also use security protocols. In the experimental scenario a special binding object was built. This object inherits the basic functionality of a DPWS binding of the C-Stack. This also includes normal HTTP Binding Security options. A DPWS based service can also be programmed in a way that it only accepts security based binding. Infrastructure support for data and process integration does exist. In the scenario the work packages are simulated by the ERP System. This system receives feedback from the production line and the ERP System or from a user. It has to acknowledge some states of the production line. By using events and subscribing, it is very easy to bring a system or application into a process. Semantically enhanced service discovery is not supported by the DPWS. It only supports information fields, as for example name, friendly name and device scope of a lifter device.

For Service Composition there are four points in the focus of this research. The point “Composability analysis for replace ability, compatibility, and process conformance” is not supported by the DPWS system. DPWS does not have behavioural or semantic conformance. The only composition of service was part of the orchestration engine, which did such conformance by itself. The built services in this scenario do not support dynamic and adaptive processes. It is possible

to build services which adapt themselves, but this has not been part of the DPWS API yet. Due to the fact that DPWS uses web service protocols, QoS-aware service compositions are possible (see dynamically reconfigurable runtime architectures), but it is not as highly detailed as it will be required in the future. Business-driven automated compositions were made by the orchestration engine. With a Petrinet based implementation it reacts on events. The same scenario was made with the Decision system. In the scope of Service Management & Monitoring, DPWS does not support any of the criteria. In the scope of Service Design und Service Engineering, four different criteria exist. The first criterion "Engineering of service applications" is handled by the scenario. In the development time services are built with the help of an integrated development environment (IDE). In this case the IDE was Visual Studio 2005. During the runtime a service-based application was built by the orchestration of the different device services in a specific process model. So the orchestration engine "built" service-oriented software. Flexible gap-analysis techniques were not used in the scenario. However, the fact that the hosting service can be changed during their runtime is a basic element for future research in this criterium. Service versioning and adaptability do not really exist. The scenario shows that it is possible to receive events from the hosting services, but this depends on implementation. The DPWS Standard does not define interfaces or methods to detect problems in service interactions, to search for alternative solutions or to monitor service-execution sequences step by step. Some simple debugger exist at the moment. Inside the shown scenario the service governance was not shown. The missing functionality in QoS and the orchestration of an external system makes it clear that DPWS is not able to meet business QoS objectives.

V. CONCLUSIONS AND FUTURE WORK

This result is a software engineering extension to the research roadmap of [14] and shows that DPWS based web services can be used for modelling and as interfaces for electronic devices. The outcome of this is that SOA can be used in the area of software engineering and the experience of the case study shows that DPWS based web services are easy to create and used by software engineers. However, the comparison between the important properties of reuse and research challenges of web service (see. Table 5) shows a gap for the use of DPWS in the future. In the scope of reusability, DPWS shows, for example, that topics like context dependencies, Quality factors and Repository & discovery have to get more focus in order to increase reuse for DPWS web services. In the area of research challenges DPWS shows according to this analysis that it has answers to some of the famous research questions such as in Service Design & Engineering and Service Foundation (cf. table 6).

TABLE 5
RESULTS OF THE ANALYSIS OF THE EXPERIMENTAL DPWS SCENARIO

Reusability	Context dependence	-
	Component models and component worlds	++
	Reuse at design level	-

		Quality factors	-
		Additional reuse information	+
		Repository & discovery	-
Research Challenges Criteria	Service Foundation	Dynamically reconfigurable runtime architectures	+
		End-to-end security solutions.	+
		Infrastructure support for data and process integration	++
		Semantically enhanced service discovery	-
	Service Composition	Composability analysis for replaceability, compatibility and process conformance	-
		Dynamic and adaptive processes	-
		QoS-aware service compositions	-
		Business-driven automated compositions	+
	Service Management & Monitoring	Self-configuring management services	-
		Self-adapting management services	-
		Self-healing management services	-
		Self-optimizing management services	-
		Self-protecting management services	-
	Service Design & Engineering	Engineering of service applications	++
		Flexible gap-analysis techniques	+
		Service versioning and adaptivity	-
Service governance		-	

Some criteria fields, however, are not so good covered by DPWS from the view of software engineering, as for example Service Composition and Service Management & Monitoring. But this does not mean that DPWS is the wrong approach for device web service. It shows only the open points of research for the future. By following such guidelines DPWS can be used more easily in the area of software engineering to become more accepted as it is.

REFERENCES

- [1] E. Zeeb, A. Bobek, H. Bohn and F. Golasowski, "Lessons learned from implementing the Devices Profile for Web Services", Proceedings of the IEEE International Conference on Digital Ecosystems and Technologies, 2007.
- [2] B. Fitzgerald et al., "The Software and Services Challenge." Contribution to the preparation of the Technology Pillar on "Software, Grids, Security and Dependability" of the 7th Framework Programme. ftp://ftp.cordis.europa.eu/pub/ist/docs/directorate_d/st-ds/fp7-report_en.pdf, pp. 1-19, 2006.
- [3] M.P. Papazoglou, P. Traverso, S. Dustdar, und F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," Computer, vol. 40, 2007, pp. 38-45.
- [4] G. Wang und C. Fung, "Architecture paradigms and their influences and impacts on component-based software systems," 37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the, Big Island, Hawaii: , pp. 272-281.
- [5] W3C, "W3C Web Service Glossary", 08 2004. <http://www.w3.org/TR/ws-gloss/> (accessed 12 2008).
- [6] M. G. Mehrabi, A. G. Ulsoy and Y. Koren, "Reconfigurable Manufacturing Systems and their Enabling Technologies", International Journal of Manufacturing Technology and Management, Vol. 1, N. 1, pp. 113-130, 2000.

- [7] P. Leitao und F. Restivo, "ADACOR: A holonic architecture for agile and adaptive manufacturing control," *Computers in Industry*, vol. 57, 2006, pp. 121-130.
- [8] A. Colombo and F. Jammes, "Collaborative automation and service-oriented architectures in the industry", Tutorial at the IEEE IECON'06, Paris, France, 2006.
- [9] A. Bepperling, J. Mendes, A. Colombo, R. Schoop, und A. Aspragathos, "A Framework for Development and Implementation of Web service-Based Intelligent Autonomous Mechatronics Components," 2006 IEEE International Conference on Industrial Informatics, Singapore: 2006, pp. 341-347.
- [10] J. Lastra and A. Colombo, "SoA middleware for Manufacturing. Section: Applications in manufacturing and industrial monitoring and industrial surveillance", European Union Info-day IST Program, Brussels, Belgium, 2007.
- [11] Oasis, The Devices Profile for Web Service specification. See <http://www.oasis-open.org> (accessed 12.04.2009)
- [12] F. Jammes und H. Smit, "Service-Oriented Paradigms in Industrial Automation," *IEEE Transactions on Industrial Informatics*, vol. 1, 2005, pp. 62-70.
- [13] D. Barisic, M. Krogmann, G. Stromberg, and P. Schramm, "Making Embedded Software Development More Efficient with SOA," *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW)*, IEEE Computer Society, 2007, pp. 941 - 946.
- [14] G. Cândido, J. Barata, A.W. Colombo, und F. Jammes, "SOA in reconfigurable supply chains: A research roadmap," *Engineering Applications of Artificial Intelligence*, vol. 22, 2009, pp. 939-949.
- [15] H. Cervantes, and R. Hall. "Technical Concepts of Service Orientation." In *Service based Software System engineering Changes and Practises*, by C. Stojanović and A. Dahanayake, 1-26. London, GB: Idee Group Publishing, 2004.
- [16] T. Andrews, et al. "Business Process Execution Language for Web Service v1.1." *IBM developerWorks*. 05 2003. <http://www.ibm.com/developerworks/library/specification/ws-bpel/> (accessed 2006).
- [17] H.P. Breivold und M. Larsson, "Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles," *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*, Lubeck, Germany: 2007, pp. 13-20.
- [18] I. Sommerville, *Software-Engineering*, Munchen, Germany, Pearson Studium, 2007.
- [19] Microsoft. *Windows Communication Foundation*. <http://msdn.microsoft.us/netframework/aa663324.aspx> (accessed 2008).
- [20] OASA. *Service Component Architectur*. 2007. www.osoa.org/display/Main/Service+Component+Architecture+Home (accessed 2008).
- [21] M. Chang, J. He, und E. Castro-Leon, "Service-Oriented in the Computing Infrastructure," *2006 Second IEEE International Symposium on Service-Oriented System Engineering (SOSE'06)*, Shanghai, China: 2006, pp. 27-33.
- [22] Information Society Technologies. *SOCRADES*. 2008. <http://www.socrates.eu/Home/default.html> (accessed 2008).
- [23] Hydra Project, 2010, <http://www.hydramiddleware.eu> (accessed 2010).
- [24] J.M. Mendes, A. Rodrigues, P. Leitão, A.W. Colombo, und F. Restivo, "Distributed Control Patterns using Device Profile for Web Services," *2008 12th Enterprise Distributed Object Computing Conference Workshops*, Munich, Germany: 2008, pp. 353-360.
- [25] H. Bohn, A. Bobek, und F. Golatowski, "SIRENA - Service Infrastructure for Real-time Embedded Networked Devices: A service oriented framework for different domains," *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICNICONSMCL'06)*, Morne, Mauritius: , pp. 43-43.
- [26] J. Ludewig, and H. Lichter. *Softwareengineering - Grundlagen, Menschen, Prozesse, Techniken*. Heidelberg, Germany: dpunkt Verlag, 2007.
- [27] C. Szyperski, *Component software : beyond object-oriented programming*, New York ;London ;Boston: ACM Press ;;Addison-Wesley, 2002.
- [28] V. Gruhn, and A. Thiel. *Komponentenmodelle , DCOM, Javabeans, Enterprise Java Beans, CORBA*. Addison-Wesley, 2000.
- [29] J. Siedersleben, *Moderne Softwarearchitektur : umsichtig planen, robust bauen mit Quasar*, Heidelberg: dpunkt-Verl., 2006.
- [30] M. Zinn, "Definition of software construction artefacts for software construction.", *Proceedings of the 4th collaborative research symposium on Security, E-learning, Internet and Networking (SEIN '08)*, pp 79-91, Network Research Group, Wrexham, GB, 2008
- [31] M. Zinn, "Development of a CASE-tool for the service-based software construction .", *Proceedings of the 5th collaborative research symposium on Security, E-learning, Internet and Networking (SEIN '09)*, Network Research Group, Darmstadt, GB, 2009
- [32] S. Pfleeger, *Software engineering : theory and practice*, Upper Saddle River [N.J.]: Prentice Hall, 2010.
- [33] H. Stuckenschmidt, (2009), „Ontologien“, Springer Verlag, Heidelberg, Germany
- [34] P. Hitzler, et. Al (2008), "Sematic Web: Grundlagen", Springer Verlag, Heidelberg, Germany.
- [35] B. Heckmann, „Service provisioning in a utility computing enviroment.“ *Research Symposium on Security, E-learning, Internet and Networking (SEIN 2007)*. Plymouth, England: Network Research Group, University of Plymouth, 2007., pp. 185-198.
- [36] Y. Xu, S. Tang, Y. Xu, und Z. Tang, "Towards Aspect Oriented Web Service Composition with UML," *6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007)*, Melbourne, Australia: 2007, pp. 279-284.
- [37] D. Skogan, , R. Grønmo, and I. Solheim. "Web service composition in UML." *8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*. California, USA: IEEE, 2004. 47-57.
- [38] Schneider Electric, "Device Description XML", www.schneider-electric.com, (accessed 2009).

Information demand model for software unit reuse

M. Zinn
Centre for Information
Security and Network Research
University of Plymouth
Plymouth, UK

A. Schuette
Centre for Information
Security and Network Research
University of Applied Science Darmstadt
Darmstadt, Germany

K. P. Fischer-Hellmann
Centre for Information
Security and Network Research
University of Applied Science Darmstadt
Darmstadt, Germany

A. D. Phippen
Centre for Information
Security and Network Research
University of Plymouth
Plymouth, UK

Abstract

Typically, reusable software units (like classes, components and services) in object oriented development environments have to provide a certain amount of information. This information is needed during the development and runtime phase. Typical information types are for example documentation, specification, metrics, technical properties, etc. In different usage levels, information may have different values for software engineers. The availability of information determines the capabilities of a unit, due to different users employing different information to decide about the reuse of a unit. Again, other information is critical for the success of the entire development process when applying certain procedure models. Thus, retrieval of these units and their contents is important for reuse. However, this is still a problem since a lot of expert knowledge is needed to find, adapt, and integrate reusable software units. To solve this problem, it is necessary to understand information demand in the area of software unit reuse. This paper discusses an existing information demand model of the knowledge science area and applies it to the field of software unit reuse to support analysis of software reuse problems. To conclude, a model will be defined to visualise the coherence between information demand and software unit reuse knowledge.

1 Introduction

The area of object oriented software development creates many different units of modelling. Frequently used units are Classes (e.g., Java Class), Components (e.g., .NET Libraries), and Services (e.g., WebSer-

vices) [18]. In the scope of this paper, a unit is classified as a deployed and function-complete component that can be reused. Each of these types of units provides different information (like machine or human-readable information) that can be used at different steps within development processes [20]. Two important problems are identified by using these units of modeling: development issues related to a common view of these different units [18] and the decision to reuse a unit upon the available information [16]. Typically, Software Reuse Environments (SRE) support software developers by handling these problems. The main concept is to combine three important usages in one environment: reuse repositories, automatic integration of software units, and searching of these units. Current Integrated Development Environments (IDE) may be seen as SRE systems [8], but no approach completely corresponds to all three general ideas about SRE [8]. It is the authors opinion that the idea of SRE systems is to support software engineers even if knowledge about “finding” and “integrating” is missing. It is especially this missing knowledge about reusable units that avoids or limits the reuse [17]. This poses the question whether this missing knowledge has an impact on the reuse of software units.

First, the term “missing knowledge” has to be defined. The authors research is focused on three typical reuse steps that can be defined as finding, adapting, and integrating. These can be found in the most common or custom created reuse processes. A software engineer needs knowledge to find, adapt, and integrate a unit. If this knowledge is available, the reuse is successful (from the knowledge perspective). However, if the knowledge is not adequate for only one step, the reuse is not successful. This means if a user does not

know how to search a unit he/she may not find it. The result will be a unsuccessful reuse of this software unit. Also if a user found a unit and he is not able to transform or adapt it for his focused technology, the unit will not be reused. In the case of a found and adapted software unit, the user must be able to integrate it into his specific development environment. If the developer is not able to do this, the unit will not be (re) used. Therefore, missing knowledge is defined as non-adequate information for the reuse (focusing: finding, adapting, and integrating) of a unit.

The authors hypothesised that the problem of missing knowledge reduces the reuse of software units and can be visualised on the base of an information demand model.

The goal of this paper is to visualise the interrelation between information demand and software unit reuse. The result may be used to support or analyse software unit reuse. In the following paragraphs, an existing information demand model will be discussed. Based on these results, the information demand model is applied to the area of software unit reuse. An existing critical success factor model for software unit reuse will be mapped to the new model to show that it can be used in the field of software unit reuse. Furthermore, the new model will be used to visualise the problem of missing knowledge. This means to show in a graphical way which kind of knowledge is needed for software reuse. The same visualisation can be used to analyse critical success factors. The paper concludes by discussing whether information demand in the area of software unit reuse can be visualised.

This work is part of the research on a Service-based Software Construction Process (SSCP) incorporating the field of Software Reuse Environments. The goal of this research is to find a semantic model (about finding, adapting, integrating, and deploying of software units) combined with service technology that supports software engineers by performing software reuse (finding, adapting, integration, and deploying) without having all needed information. The paper contributes to the research area by enhancing a basic model to visualise the coherence of missing knowledge and software reuse knowledge. This new model can be used in the future to analyse reuse problems based on missing knowledge.

2 Information demand

2.1 Description of information demand

Information demand (ID) is defined as the type, amount, and quality of knowledge that a person needs

to fulfil a task within a specific amount of time. Measuring ID is problematic because it depends on the task definition, the goals, the people who deal with it [14], and the knowledge criteria [1]. ID can be split into two sub definitions: objective information demand (OID) and subjective information demand (SID). OID describes all information which solves the users problem. This information is the amount of existing information that will theoretically solve a specific problem. It can be described as a set of solutions. Similar to OID, SID describes all information that is supposed to solve the problem, from the subjective point of view of a user. Because of that, this information may not be able to provide a real solution. Another important factor in the area of ID is the information provision (IP). This defines information that is actually provided by a system and can be utilised by formulating an information query (IQ). IQ is a query which is normally created by a user who is searching for a solution which is executed by a search system. This query is based on the SID of the user. The useful result referred to as actual information state (AIS) in this paper is the intersection of the areas of SID, OID, IP and IQ. Figure 1 shows the relation:

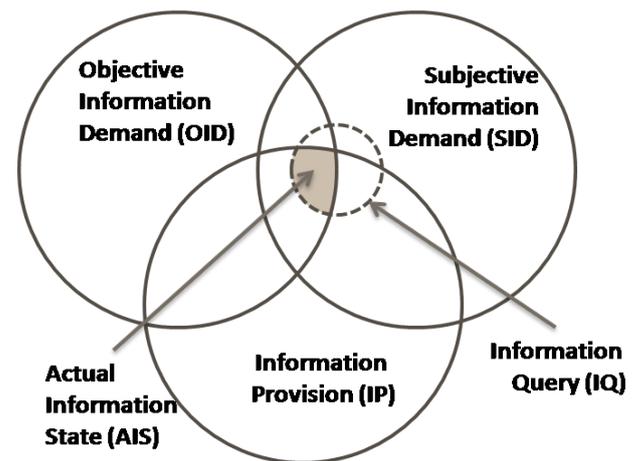


Figure 1: Information Demand [14]

As a result of this model, AIS is defined as the area of the information model which includes information that

- achieves the task
- can be understood by the user
- can be enquired after by the user
- and is provided.

2.2 Use of information demand

To increase the overlapping area of OID and SID, the Critical Success Factors methodology [16] can be used [14]. This approach may be used for business success factors, but can also be adapted to other domains (e.g., Enterprise Security Management [2]) Thereby, factors are identified that are required to fulfil a task. These factors will be related to specific elements of the information demand model. Based on this relationship, custom analysis can be performed to identify possible risks in a project [14]. Therefore, an analysing process based on this may be structured as follows:

1. Identify important success factors for a project.
2. Apply the factors to an information demand model.
3. Identify problems and risks by analysing which information demand element is affected by a problem or risk that could affect a applied success factor.
4. Prepare training sessions to minimize the risks or problems.

The following sections will focus on point 2 and 3 of this process and connect the general idea of information demand with the area of software unit reuse.

3 Information demand in the area of software unit reuse

3.1 Definition of information demand from the software unit reuse perspective

The presented definitions in Section 2 are not related to the area of software reuse. This relation can be created by redefining the terms of Section 1 from the perspective of software reuse. This represents the novel contribution of this publication. The OID can be related as a container describing all software units that may solve a problem. From the software reuse perspective, the important tasks of reuse is to find and integrate a reusable software units. This unit fulfils technical, functional, and business requirements [17]. As a result, OID describes all software units that can be used as solutions . SID is related to the users ability to express technical, functional and business information about a needed software unit. Again, this also includes units that do not solve the problem, contrary to the users beliefs. Previous studies show this as an important problem of knowledge reuse [1]. In the

reuse area, IP can be defined as the real availability of reusable software units and their descriptive information. Thus, IP is realised by repository systems responsible for providing software units. Typically software units are provided by repository systems [1]. The user creates and accomplishes an IQ by using special tools (SRE-Systems)[8]. The AIS in the area of software unit includes all reusable software units that:

- theoretically solve the problem,
- are provided by a component provider or repository system,
- are understood by the user
- are described by the user request (part of the users query).

Figure 2 shows the relationship between the normal information demand and information demand based on software unit reuse. The model is referred to as Software Reuse Information Demand (SRID) model by the authors of this publication.

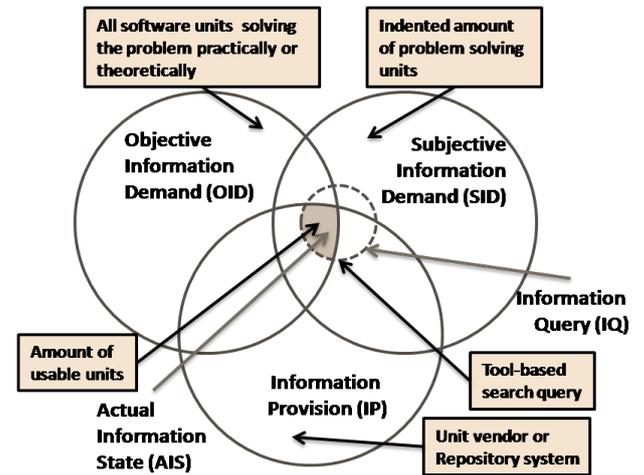


Figure 2: Software Reuse Information Demand (SRID) Model

The goal is to increase overlapping areas between the different elements of an information demand model, in order to increase the amount of solutions, which is also an important aim of the software unit reuse area. This can be achieved by defining/identifying the critical success factors of software reuse and relating them to the model [14]. (See Section 2.2)

3.2 Analysis of critical success factors in software unit reuse

As mentioned in Section 2.2, an Information Demand Model can be used to identify risks by referring

critical success factors to the model. Usually, this is done in the business area [16][14]. The idea is to map a project specific procedure model to an information demand model. The information demand model identifies the information demand fields and the procedure model the typical process steps or aims in a project. By doing the mapping information demand will be mapped to project steps or aims. Project leader can now identify which project step or aim is at risk, because the related information demand. Before this step a project leader must be identify which information demand field is at risk in his project or team. In the scope of this paper, this methodology will be adapted to the area of software reuse. First, critical success factors will be identified. This is necessary because these success factors will be analysed and assessed in the new model. In the area of software reuse, critical success factors can be identified by analysing the typical reusability metric methodologies [17]. The following metric methodologies are taken from the reusability metric analysis of [17] and described briefly.

- Cost productivity model (mathematically based model)
- Return of invest model (mathematically based model)
- Maturity model (mathematical model / process based model))
- Failure modes models (process based model)
- Reusability model (property based model)

The Cost Productivity Model is an approach that aims to calculate the advantages of software development based on cost [17][6]. From a development point of view, the development of a reusable software unit is more expensive than the development of software units without reusable properties [17]. However, from the commercial point of view, software reuse is cheaper than development of a new unit [17]. The Return On Invest (ROI) model analyses the ROI of software development from the perspective of software reuse This methodology is related to the Cost Productivity Model *Cost Productivity Model* [17][13]. The Maturity Assessment Model deals with the measurement of improvements in a software unit reuse process. Therefore, different models exist (see [17],[11], and [4]). The success factors of the Failure Modes Model describe a successful reuse process. This is possible if reusability is focused, a unit exists, the unit is available, the unit was found and the unit is valid. Validity means it is able to fullfil the users requirements, and is capable of integration. A reuse process is not successful if one of these success factors is not applicable [17][7].

The last measurement of success factors to be discussed is the *Reusability Assessment*. The core methodology is to identify reusability success factors of a given reusable software unit, for example fewer parameters [17][6]. In the scope of this analysis, the *Failure Modes Model* is a usable example of success factors that can be used in the SRID model. This is based on the fact that each success factor of this model represents a reuse process step. The result of all steps represent a reusable software unit for a user. The direct link to software reuse makes it easier to map these steps to the SRID Model. Other models based on component properties (like *Reusability Assessment*) or mathematical functions (*Return On Invest model*) are also feasible, but are based on higher abstraction to map them to software reuse demands. In the scope of this paper, it is important to provide a simple example. The success factors of the Failure Modes Model [7] may be mapped to the SRID model in the following way (see also Figure 3):

- **Intention for reuse:** This is the users aim to reuse a software unit. In the scope of the SRID model, this is shown by the IQ definition.
- **Part exists:** A unit exists if it is theoretically possible and able to solve the problem. In the SRID model, this corresponds to the OID.
- **Part available:** A unit is available if it is provided by a unit vendor or a repository system. This complies with the IP area of the SRID.
- **Part found:** Part found: A unit has the state “found” if it is theoretically possible, understandable by the user, requested, and found by the user. This is shown in this model with AIS.
- **Part understood:** Part understood: A unit is understood if it is theoretically possible and the user is able to understand it. In the SRID model, it corresponds to the overlapping area between OID and SID.
- **Part valid Part integrateable:** Both success factors depend on three properties:
 - They must be part of the theoretical amount of solutions (OID)
 - They must be provided by a vendor or system (IP)
 - They must be part of the users subjective information demand (SID)

When using the *Failure Modes Model*, a reuse is successful if all success factors are successfully fullfilled. This is demonstrated by the AIS in the SRID Model

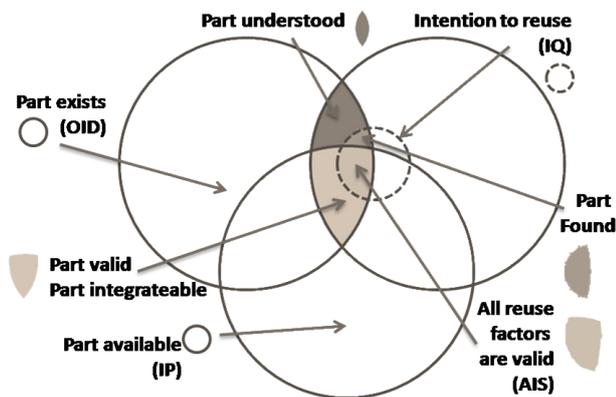


Figure 3: Software Reuse Success factors in the SRID Model

(see Figure 3). Generally, the critical success factors within the Failure Modes Model can be mapped to a model based on information demand in a graphical way. This poses the question how this visualised mapping supports the software engineer.

As mentioned in Section 2.2, the reference between success factors and an information demand model helps to identify risks in projects. For example, the success factor “Part understood” is the overlapping area between SID and OID. If a problem reduces the area of SID, this overlapping area may also decrease. As a result, the specific success factor, which must be fulfilled for successful reuse is vulnerable. This poses a certain risk to the project.

4 Visualising Information Demand Examples in the Software Unit Reuse Area

In Section 3, the SRID model was defined and explained by visualising an example of a reuse metric model. This is now used to explain the specific missing knowledge problems that are discussed in this document. The hypothesis of the authors is that the problem of missing knowledge reduces reuse of software units and can be visualised on the base of an information demand model. As mentioned before, the problem of missing knowledge is applied to the three common reuse steps: Finding, adapting, and integrating. Table 1 shows typical based on missing knowledge in the context of these three reuse steps. In the field of “Finding” users have to know where a repository can be found. This is a problem especially in global companies with repository on different locations. Also the usage knowledge of these systems is important to know. In the field of “Adaption” it is important to

know how to adapt software units to fit special requirements. Typical examples are source code adaption, configuration management, cross compiling, technology transfer (i.e. Java to .Net), and domain transformation. Doing such kind of adaption without knowing the procedure needs a lot of time or is not possible. In the last field (“Integration”) the integration of software units into development environments is focused. Software reuse may at risk if a user does not know how to integrate software units in his development environment. This includes the problem of missing configuration knowledge.

This section aims to visualise two examples of the problems listed in Table 1. The first example describes cases in which users do not know how to formulate a query [17][19] (Refers to problem No. 2). Another example is that users with less advanced knowledge are frequent users of software unit reuse [17][5] (Refers to all problems of Table 1). These examples can be used to demonstrate the process steps mention in Section 3.2:

Identifying important success factors for a project:

For the verification, the Failure Modes Model is used. (see Section 3)

Referring the factors to an information demand model:

For this step, the mapping between the SRID model and the success factors of the Failure Modes Model is used. (see Figure 3)

Identify problems and risk by analysing which information demand element is affected by the problem or risk and which success is vulnerable:

In the first example named “QueryProblem” (QP), the user is not able to formulate a search query. This may have several reasons: missing knowledge about the searched unit, missing knowledge about the tool that is used to formulate the query or the query language. This problem is classified as a “Finding Problem” (see Table 1). Consequently, the size of IQ in the SRID model is affected. According to the mapped Failure Modes Model, the usable area of the “Part found” success factor decreases. This is a logical result because of “Part found” is the overlapping area of OID, SID, ID, and IQ. If one of this areas decrease the overlapping area may also decrease. Furthermore, the AIS will be affected. Because of a query is in itself a kind of validation (from the user’s perspective), it can also affect the Part valid success factor. Figure 4 visualises this.

The other example named “YoungUserProblem” (YUP) is due to the fact that users with less advanced knowledge for example young professionals are frequent users of software unit reuse. This affects the SID, which is defined as the amount of conceivable software units providing a solution. People with less advanced knowledge also have less knowledge about reusable

Reuse Step	Problem No. and Name	Problem Description	Reference
Finding	1 Find repository	User has to know where to find a repository	[10]
	2 Access repository	User has to know how to access a repository	[17]
	3 Use repository	User has to know how to use a repository	[17][19]
Adaption	4 Unit adaption	User has to know how to adept a software unit	[1]
	5 Transformation (Tec.)	User has to know how to transform a software unit into another technology	(e.g. [15])
	6 Transformation (Domain)	User has to know how to transform a software unit into another domain	(e.g. [15])
Integration	7 IDE Integration	User has to know how to integrate a unit into his IDE	
	8 Unit Setup	User has to know how to setup a unit for his specific project	

Table 1: Problem of missing knowledge

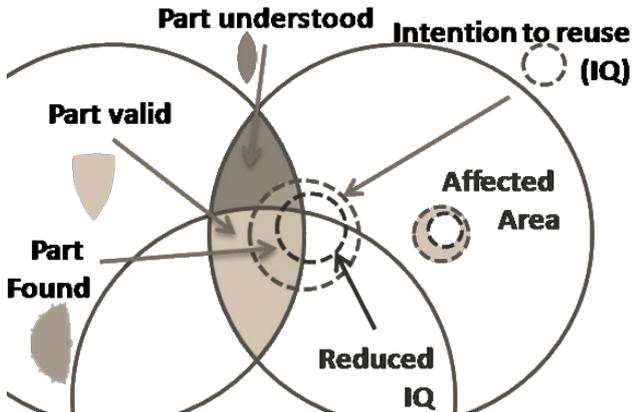


Figure 4: Impact of missing formulation knowledge

software units which means finding, adaption, and integration steps are affected. For example, a software engineer searching a software unit within the area of a specific technology (e.g., SOAP-based web services) could have problems reusing a software unit if he is not an expert for the technology. This example refers to all reuse steps of Table 1. As a result, the success factors “Part understood”, “Part valid”, “Part integratable”, and “Part found” are affected directly. Figure 5 illustrates this.

The result of this analysing step is a list of affected success factors. Figure 6 shows this on the base of the SRID model. *Prepare trainings and sessions to minimise the risks or problems:* Based on the previous analysis, special tasks may be performed to reduce the risk for the success factor. In the QP example, a query tool training for the project team may be a potential solution to limit the risk. The trained users are able to create an execute queries more correctly. As an result the ID area will increase. This may also include an increased overlapping area between SID, OID, and ID. More correct solutions can be found. This is shown in 7 in the affected area of IQ. In the YUP example, individuals can be trained on special software units or

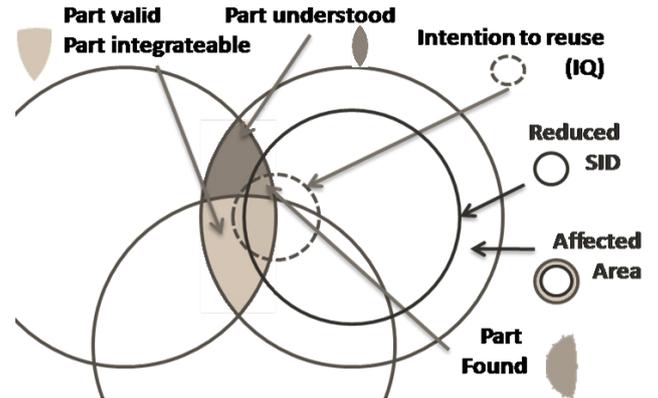


Figure 5: Impact of missing user knowledge

specific technology to increase their knowledge. The result will be an increased area of SID. This includes also and increased overlapping area between OID and SID. As a logical result the risk factors will decrease. This may be seen as a positive effect for the project. Figure 7 visualises a decreased SID. If this happens all overlapping areas of SID also decrease. This is shown in the affected area of SID.

IQ is affected, which also applies to the success factor “Intention for reuse”. As a consequence, the project is directly vulnerable. In the YUP example, the SID, not defined as a success factor, is affected. However, the analysis shows that success factors are also affected (indirectly). Both examples can be visualised in the SRID model. The authors hold the hypothesis that the problem of missing knowledge reduces reuse of software units and can be visualised on the base of an information demand model, as verified in the scope of the given examples. These two examples demonstrate how to identify the impact of problems on critical success factors of software unit reuse by using a model based on information demand. An important difference between the two examples is the directly affected element of the SRID model. In

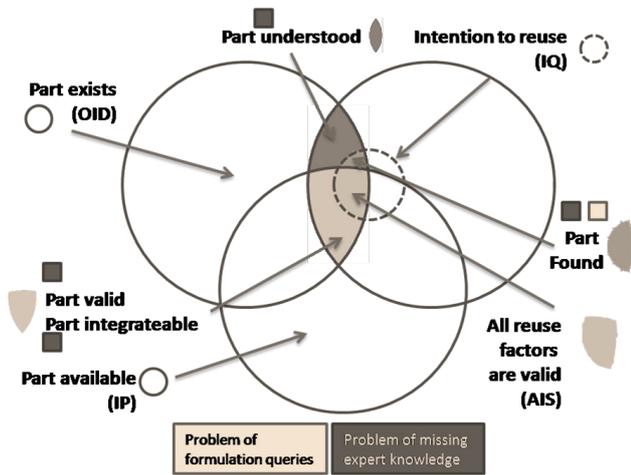


Figure 6: Affected success factors in the SRID model

the QP example, the area of IQ is affected, which also applies to the success factor “Intention for reuse”. As a consequence, the project is directly vulnerable. In the YUP example, the SID, not defined as a success factor, is affected. However, the analysis shows that success factors are also affected (indirectly). Both examples can be visualised in the SRID model. The authors beheld the hypothesis that the problem of missing knowledge reduces reuse of software units and can be visualised on the base of an information demand model, as verified in the scope of the given examples.

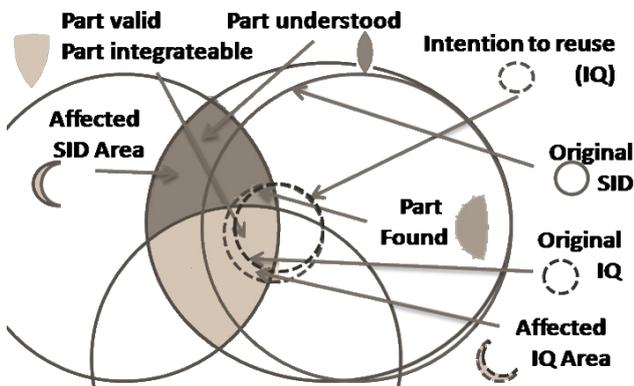


Figure 7: Trained SID (technology knowledge) and IQ (Search tool knowledge) decrease negative effects

5 Conclusion and Future work

The paper demonstrates the relationship between a common information demand model and the software reuse area focused on missing knowledge of finding, adapting, and integrating of a reusable software

unit. As demonstrated, it is possible to adapt the common information model to specific definitions of software unit reuse. As a result, a new information model has been created that is based on software reuse. It is called Software Reuse Information Demand (SRID) model. As a second step of this demonstration, the SRID model was used in a critical success factor analysis to visualise critical success factors as knowledge areas. This analysis shows that the SRID model can demonstrate critical success factors of software reuse. In the last step, two reuse problems based on missing knowledge are taken from a list of identified knowledge based problems as examples. These two examples of software unit reuse problems given by scientific research were mapped to and visualised by the SRID model. This combination (SRID model, project relevant success factors, and project relevant reuse problems based on missing knowledge) may be used as an analysing method of software unit reuse inside software development projects.

The paper demonstrates which success factors in the SRID model may be affected by problems based on missing knowledge. The demonstrated examples also verified that information demand may help to explain the impact of problems in software unit reuse that are based on missing knowledge. In future, the new model may be used to analyse the problems of software unit reuse in more detail. This may help to find solutions to support software developers, software architects, and individuals managing reuse. An important research question for this future analysis method may be how to reduce the impact of missing knowledge to software unit reuse. This question will be addressed in the authors future research. Mapping the SRID model to different reuse metrics or problems can simplify the analysis of software reuse problems from an information demand perspective. The employment of the SRID model as a comparison platform is also to be explored.

Acknowledgements

The authors would like to thank Schneider Electric and the partners of the ICT FP7 project “Cooperating Objects Network of Excellence” (CONET) for their support.

References

- [1] W. F. Boh, Reuse of knowledge assets from repositories: A mixed methods study, *Information and Management*, Vol. 45, No. 6, pp. 365-375, 2008.
- [2] R. A. Caralli, J. F. Stevens, B. J. Willke, W. R. Wilson, The Critical Success Factor Method:

- Establishing a Foundation for Enterprise Security Management, *Software Engineering Institute*, available at <http://www.sei.cmu.edu>, last access 04.2011, 2004.
- [3] C. M. Christensen, *The innovator's dilemma: when new technologies cause great firms to fail*, Harvard Business School Press, 1997.
- [4] T. Davis, The reuse capability model: a basis for improving an organization's reuse capability, *In proceedings Advances in Software Reuse*, pp. 126-133, 1993.
- [5] K. C. Desouza, Y. Awazu and A. Tiwana, Four dynamics for bringing use back into software reuse, *Communications of the ACM*, Vol. 49, No. 1, pp. 96-100, 2006.
- [6] W. Frakes and C. Terry, Software reuse: metrics and models, *ACM Computing Surveys*, Vol. 28, pp. 415-435, 1996.
- [7] W. B. Frakes and C. J. Fox, Quality improvement using a software reuse failure modes model, *IEEE Transactions on Software Engineering*, Vol. 22, pp. 274-279, 1996.
- [8] V. C. Garcia, E. S. de Almeida, L B. Lisboa, A. C. Martins, S. R. L. Meira, D. Lucredio, R. P. de M. Fortes, Toward a Code Search Engine Based on the State-of-Art and Practice. *In proceeding of 2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pp. 61-70, 2006.
- [9] S. Jansen, S. Brinkkemper, I. Hunink and C. Demir, Pragmatic and Opportunistic Reuse in Innovative Start-up Companies, *IEEE Software*, Vol. 25, No. 6, pp. 42-49, 2008.
- [10] A.J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, J. Lawrance, H. Lieberman, B. Myers, M.B. Rosson, G. Rothermel, C. Scaffidi, M. Shaw and S. Wiedenbeck, The State of the Art in End-User Software Engineering, *ACM Computing Surveys*, available at <http://citeseerx.ist.psu.edu>, 2009.
- [11] W. Lam and M. Loomes, Re-engineering for reuse: a paradigm for evolving complex reuse artefacts, *In proceedings of the 20 Annual International Computer Software and Applications Conference (Compsac'98)*, pp. 507-512, 1998.
- [12] A. C. Martins, V. C. Garcia, E. S. de Almeida and S. R de Lemos Meira, Suggesting Software Components for Reuse in Search Engines Using Discovered Knowledge Techniques, *In proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'09)*, pp. 412-419, 2009.
- [13] , J. Poulin, An Agenda for Software Reuse Economics, *Presentation on the International Conference on Software Reuse*, 2002.
- [14] A. Picot, *Die grenzenlose Unternehmung: Information, Organisation und Management Lehrbuch zur Unternehmensfuehrung im Informationszeitalter*, Gabler, 2003.
- [15] A. Punder, S. Haeberling and R. Todtenhoefer, An MDA Approach to Byte Code Level Cross-Compilation, *In proceedings of Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'08)*, pp. 251-256, 2008.
- [16] J. F. Rockart, Chief executives define their own data needs, *Harvard Business Review*, Vol. 2, pp. 81-93, 1979.
- [17] S. G. Shiva and L. A. Shala, Software Reuse: Research and Practice, *In Proceedings of Fourth International Conference on Information Technology (ITNG'07)*, pp. 603-609, 2007.
- [18] G. Wang and C.K. Fung, Architecture paradigms and their influences and impacts on component-based software systems, *In proceedings of the 37th Annual Hawaii International Conference on System Sciences*, Centre for Information Security and Network Research, pp. 272-281, 2004.
- [19] Y. Ye, An Active and Adaptive Reuse Repository System, *In proceedings of the 34th Annual Hawaii International Conference on System Sciences*, pp. 10-19, 2001.
- [20] M. Zinn, G. Turetschek and A. D. Phippen, Definition of software construction artefacts for software construction, *In proceedings of the Forth Collaborative Research Symposium on Security, E-Learning, Internet and Networking*, pp. 79-91, 2008.

Reusable Software Units Integration Knowledge in a Distributed Development Environment

M. Zinn¹, K. P. Fischer-Hellmann², A. Schuette² and A. D. Phippen¹

¹University of Plymouth, Plymouth, U.K.

²University of Applied Science Darmstadt, Darmstadt, Germany*

Abstract. Today's software units (classes, components and services) require large amounts of information during their development and use that can be documented for future reference, like documentation, multimedia files, specification, and models. The availability of certain information, for example documentation, is one of the factors that determines the capabilities of a unit, especially by reusing it. Additional information is necessary and essential for the success of the entire development process when applying certain procedure models, like Rational Unified Process (RUP). Acquiring these units and their content is important for reuse. However, this causes a problem in the area of global cooperation. Currently, approaches are missing that deal with software reuse in distributed software reuse scenarios. Especially the problem of missing knowledge about integration of reusable software units in these scenarios has not yet been addressed. This knowledge is also an important factor for reuse and reuse decisions. As a result software development teams locate at different locations may have problem to integrate exchanged reusable software units. This paper discusses the challenges of integration in distributed reuse scenarios by focusing on an industrial example and create a model extension for a existing reuse system. As an result integration of reusable software units can be done remotely without the necessary integration knowledge.

1 Introduction

In object-oriented software development, various units of modeling are used. Typical units are classes, components, and services [11]. Every unit type provides a certain amount of information that is be used based on their underlying technologies, like service description, documentation, or models [14]. In the scope of this paper, a component is a deployed component. There are two important problems: development issues related to a general view of these different units [11] and the decision to reuse a component based on the available information [3]. Software Reuse Environments (SRE) support software developers by addressing these problems. The general idea is to have three important functions in one combined environment: reuse repositories, automatic integration of software units, and the searching for these units. Current Integrated Development Environments (IDEs) are SRE systems. However, none of these approaches completely fulfill the requirement of SREs to include all functions. [2]. Most of these

* The authors would like to thank the French company Schneider Electric for providing information about distributed existing software engineering scenarios.

SRE systems support the integration of information in a specified environment by using extensions that can directly communicate to a SRE system (e.g. Eclipse and Visual Studio). These functions can be used in distributed and non-distributed scenarios. Typically, in such a scenario, the decision maker, the person who decides to reuse a specific software unit, is the same as the integrator, implementing the reuse. However, there are scenarios in which the decision maker and the integrator are not the same person. In the scope of this paper this is called a distributed scenario because the individuals can be located in different locations and differ in their domain of expertise. Typically, software architects are this kind of decision makers in software development [4]. Figure 1 shows these focused scenarios:

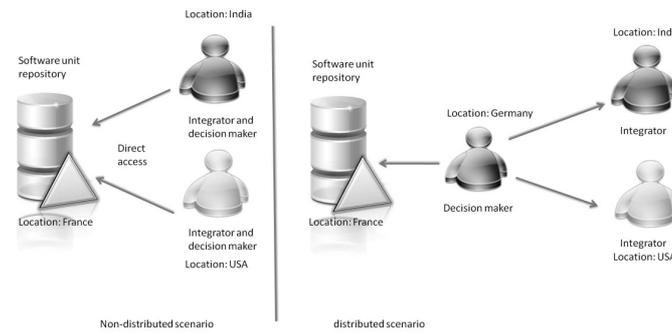


Fig. 1. Distributed and Non-Distributed Scenario.

The authors of this paper hypothesize that the reuse of software units in a distributed scenario has a negative influence on the reuse. These negative impacts can be mitigated by providing an integration model and a service based communication architecture to achieve the integration. Challenges of the distributed software reuse scenario will be discussed in this paper. The aim of this paper is to provide a solution for distributed software reuse scenarios that can be used to support software development. This is achieved by extending an existing software reuse architecture to include an integration model. This solution is the result of research into Service-based Software Construction Process (SSCP) incorporated into the field of SRE and the software unit reuse with limited knowledge. This paper's heuristic value lies within the enhancements to existing SOA-based (Service Oriented Architecture) architectures (SSCP System) by supporting the handling of units of modeling, like classes, components, and services, for use by decision makers with integration tasks. The paper concludes with the fact that supporting distributed scenarios can be done with an integration extension of the SSCP system.

2 Two Problems in a Distributed Reuse Scenario

2.1 Problem Identification

Distributed software development scenarios cause special problems in Software Architecture, Engineering Processes, and R&D Organisation [1]. Especially the sharing of reusable software units between teams have a deep impact on costs:

“A problem observed [...] is that when decoupling between shared software assets is insufficiently achieved is excessive coordination cost between teams. One might expect that alignment is needed at the road mapping level and to a certain extent at the planning level. When teams need to closely cooperate during iteration planning and have a need to exchange intermediate developer releases between teams during iterations in order to guarantee interoperability, the coordination cost of shared asset teams is starting to significantly affect efficiency.” [1]

To get an impression of the problems that may exist in a distributed reuse scenario, it is helpful to observe an real life industrial scenario. For this, data from the company Schneider Electric is used [8]. Information about used technology and methodologies is provided by project leaders of Schneider Electric): Schneider Electric is a French company that focuses on the automation and energy industries. Employing approximately 130,000 people, divided into over 100 organisations, Schneider Electric is divided into in 5 different domains: Building, Industry, Power, IT, and Energy. Each domain has locations all over the world in many different countries. In each of these domains, software development is an important part of the work and the provided software solutions. Typical software development areas are server-, desktop-, web-, and embedded device applications. Various locations work together to fulfil a task and provide a software solution. Thereby, typical units of modelling (like classes, components, and services), implemented in different technologies, are used (like .NET or Java). Schneider Electric uses the typical component worlds (see [10]). Each location uses its own repository for these units. However, the repository types and their usage differs. The authors analysed 6 software development projects of Schneider Electric from 2006 to 2010 that uses a distributed scenario (limited to two locations) evaluating four different aspects: (1) Which partner is developing the Software?, (2) Which partner is making architectural decisions?, (3) Which partner is selecting reusable software units?, and Which partner is integrating the selected reusable software units?

The result of this analysis can be describes as follows: (Answer Q1 and Q2) The characteristics of the analysed scenario include that the partner who selects the reusable components is not the partner who is doing software development. Most time Indian software development teams were responsible and team from other countries are the software designer making architecture decisions.(Answer Q3) Also selecting of reusable software units, like corporate identity or login components are selected by the non developer teams. (Answer Q4) In each analysed project the integrating task was done by the software development team. The previous analysis shows a distribution pattern. The development task and the development decisions are done by separated teams located in different countries. Such patterns include different problems. In the following section the problem of accessibility and integration will be discussed and analysed.

2.2 Accessibility Problem

The accessibility problem for software unit reuse can be explained using the Software Reuse Information Demand Model (SRID) [13], that is based the Information Demand Model [6]. From the SRID point of view of information depends on five factors :

- Objective information demand: This is the entire (theoretical) amount of information that can solve a problem.
- Subjective information demand: This is all information that the user believes can solve the problem.
- Information provision: This is all information that is accessible.
- Information request: This is a search request the user formulates to find information relevant for solving the problem.
- Real level of information: This is all information that is correct, is available to the user, is accessible, and is requested by the user.

All these 5 factors are part of the Software Reuse Information Demand model (see Figure 2). The general problem of information demand arises because the real level of information is a subset that is limited by the user's ability to formulate the request (Real level of information). Figure 2 shows the relationship between the five factors.

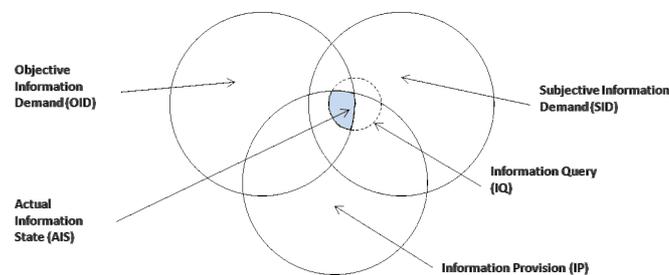


Fig. 2. SRID model [13] (based on [6]).

In the area of accessibility this problem can be identified when a user has to search for a reusable software unit in an external environment. Missing knowledge about a providing system (repository) limits the information request [9]. The user is less able to formulate a request. The severity of this problem comes from most users being junior, inexperienced software developers [9]. Another accessibility problem occurs when the user has no access to the software reuse environment of another location. This is an information demand problem based on infrastructure requirements. In the case of Schneider Electric, both accessibility problems occur.

2.3 Integration Problem

A user who wishes to integrate a reusable software unit has to know about the dependencies, structure, configuration and technology of the unit. Especially configuration has been a problem for some years [7]. This limits the overlapping area between subjective and objective information demand (See Figure 2). The result is a strong limitation of the real level of information. This is demonstrated in the following simple example: In the initial situation the reusable component library for discovery device profile based web service is a .Net library called 'Discovery.dll'. It uses another reference called 'DPWS.dll' that is an unmanaged .NET library and includes some specific libraries that are used by the Discovery.dll file. A configuration file is required ('Config.xml') that has

to be placed in the same directory as the *Discovery.dll* file. By using Visual Studio, the user has to perform following integration steps to create this setup:

1. add the *Discovery.dll* as a reference, because of the user will require functions, structure, user interfaces or data from this library.
2. write a script to copy *DPWS.dll* in the release directory after compiling, because of the unmanaged libraries can not be easily managed by the IDE.
3. add the *Config.xml* to this project and set the copy attribute to “*Copy if newer*”, because of this file contains settings that will be used during runtime.

This example illustrates the complexity of software unit integration for a specific environment. If the user is unaware of these integration steps, the process is likely to be time consuming. Therefore, the problem of integration is to know these additional setup steps. Each reusable unit needs further steps for integration. In a distributed scenario the individual who is aware of these steps cannot be in a different location. In this case the integration problem is also a problem of information demand.

3 Solution Approach

Approaching the problems involves two different models. The first is an integration model and the second an architecture extension to support distributed scenarios. The model, the architecture, and the combination to support distributed software unit reuse, constitutes the scientific contribution of this paper.

3.1 An Integration Model as Reuse Model Extension

In the context of the underlying research, the authors developed an ontology to the subject 'Service-based Software Construction Process' in order to counteract the problems experienced in software unit reuse [14]. Also an environment was build using this ontology and enabling users to do software unit reuse (focusing search, adaption, and IDE integration of units) without the complete necessary knowledge. The used ontology serves only the unified description of units of modelling (classes, components and services). This includes the description of technological facts (components, services, etc.).

The ontology consists of 4 parts. Part 1 shows the access to the ontology: The problem-solution approach' Part 2 relates to 'general business information' about the solution (e.g., manufacturer, name, and author). Part 3 describes the solution as a technical unit (e.g., a type of unit, a technology, a file format, or files). In Part 4 the technical contents are described thereby explaining a semantic search approach that is discussed in a previous publication (See [15]). If an instance of the ontology is generated (e.g., by the registration of a newly developed unit), the user has to specify information that is stored in the appropriate area of the ontology. The data may also be entered automatically into Part 3 of the ontology. This is possible as the technical data is generally detectable (such as file size, file type, file name, and technology). Nevertheless, the data from other sections of the ontology is not automatically detectable. The ontology

A unit has been represented as a common description of classes, components, and services and will now be extended with integration information. The 'Real File' node in the base semantic model may have instances of integration descriptions. Each node in the semantic model contains a unique identifier (ID) and a friendly name property. This is not sketched in Figure 5. Three questions have to be answered in relation to unit integration (Questions are defined from the result of the component analysis of [5]): (1) What is the target platform? (2) How should the unit be integrated? (3) What is the scope of the unit?

The target platform has to be validated before integration can proceed. This is critical, as the current research of the extension model includes parts that are platform dependent. A platform is marked by the 'IDE' node that has a relation to the 'environment' mode of the base model. The meaning of this relation is that the 'IDE' shows which kind of environment can be used to host, build, and execute a software unit. The 'Real File' node has an indirect relation (given by the base model) to an 'Environment' node. This relation describes the appropriate environment to use with this unit. From the semantic point of view, the 'Environment' node can be used to validate the compatibility between a software unit and the platform of the IDE. For example, a class file that requires the .NET framework is generally not compatible with a Java-based environment (such as Eclipse). The process of integration can be illustrated by detailing the various integration patterns of the Visual Studio and Eclipse APIs. The following concepts are necessary from the view of the authors and are provided in both environments Visual Studio.NET and Eclipse (handling in the two environments differs):

- OnlyCopy: This copies a file without referencing it in the solution tree of the project. This is necessary for second level dependencies that are not controlled by the IDE environment.
- WebReference: This marks a file as a web reference. Different IDEs utilize different methods to manage this information. For example, Visual Studio can use a WSDL file to create a reference to a web service that is based on the corresponding WSDL description.
- Reference: This copies a file and includes it in the solution tree of the project. This is a traditional reference that can be included or imported. This is necessary for managing the dependencies of a unit.
- DoNotCopy: This prevents a file from being transferred into a project's environment. All files a unit includes are not necessarily required by the IDE (e.g., documentation).
- InsertAsText: This flags the content of a file to be treated as text when loaded into the IDE. This is useful for code references (using or import) and code snippets.
- CopyAsResource: This flags a file to be used as a resource and includes it in the project (e.g., configuration files).

The scope of the unit is used to create integration packages. For instance, a library refers to another library as a dependency, so both libraries have to be delivered. This relation can be modelled by referring an "Integration package" node to the global "unit" node. A unit is now part of an integration package. Each of these packages includes files with integration descriptions that are related to an instance of the integration package.

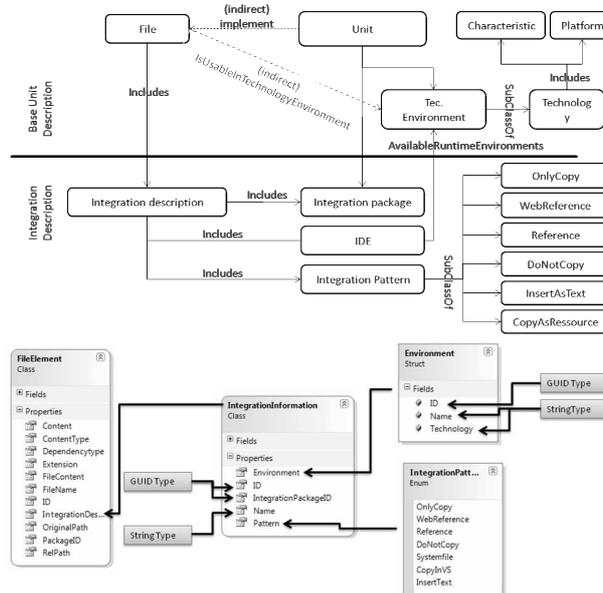


Fig. 5. Integration model extension of Figure 4 (top) and data model variation.

Figure 5 shows a model integration extension in relation with the normal unit description.

3.2 Architecture Extension

In a previous publication, [12] an architecture of a service-based software construction CASE-tool was sketched. Figure 6 shows an overview of this sketched architecture:

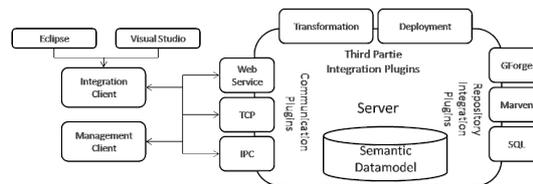


Fig. 6. Communication Architecture extended version of [12].

The architecture is used to implement a distributed system that deals with the underlying topic of missing knowledge in software reuse (see Section 1). It is capable of integrating existing software unit repositories and handles them within the semantic model. The server side of this architecture provides different functions like search, management, transformation, and deployment of software units. On the client side a management client and an integration client are sketched. In contrast to the management client, the development client does not influence the artefacts (groups of software units with the same business context), such as the deletion of an artefact or software

unit on the server. Besides searching for artefacts or units of modelling, the development client is responsible for the transformation and the integration of transformation results into the current development project. In this described scenario the integration client communicates directly to the server. This belongs to a non-distributed scenario. The user searches, selects, and integrates software units into the project, using the integration client which is hosted in the IDE (or the host system). This architecture will be extended by adding an integration plugin next to the *Deployment* and *Transformation*. Analysing the scenarios of Figure 7 two distributed scenarios are identified by the authors using the infrastructure given by the architecture of Figure 6: **Light-weight scenario:** The integration client receives metadata from the management client about the unit(s) that are to be integrated. The integration client is able to do a specific search on the server with a single unit as a search result. **Heavy-weight scenario:** The management client sends the integration information directly to the integration client; there is no need for the integration client to communicate with the server.

The two scenarios differ in the amount of data which has to be exchanged between the management client and the integration client. In the light-weight scenario, the management client sends only metadata to the integration client. Therefore, the integration client can perform the search. In the other scenario, all data required for integration is sent. Figure 7 illustrates both scenarios:

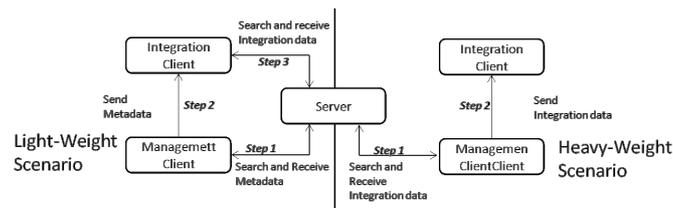


Fig. 7. Light- and Heavy-weight scenario.

Based on the integration extension for the semantic model (see Section 3.1) a data model can be created for communications. [12] shows an XML description of data entities that is used in SOAP based communication between clients and the server. Figure 5 shows the data model that is used for integration.

Based on the light- and heavy-weight scenarios, a service for the integration client can easily be defined. The light-weight scenario involves the integration client requiring the meta data and the ID of the integration package of the unit (see Figure 5). A unit includes all references to file elements, allowing the client to request specific information about the unit from the server. The heavy-weight scenario requires the integration client to know a set of integration information. Therefore, all files and an ID for an integration package is required which describes the integration of the files (see Figure 5). An web service interface supporting both scenarios (based on the data model of Figure 4) may be described as into two operation: *GetIntegrationDataLightWeight(Guid serverID, Guid artefactID, UOM unit, Guid integrationPackageID)* and *GetIntegrationDataHeavyWeight(FileElement[] setOfIntegrationFiles, Guid integrationPackageID)*.

4 Example Scenario Discussion

As discussed in Section 3, the paper focuses on the problems of accessibility and integration. Section 4 now addresses a definition of a problem approach. The relationship between both discussions can be demonstrated with a simple comprehensive example. Given the scenario of Schneider Electric (see Section 2.1), two teams situated in different locations (French and India) are working together on a software development project. The French team is defining the architecture and preselecting existing software units that are developed by the same team. The Indian team is responsible for the real implementation and integration (see Figure 1). **Integration Problem:** The team in India has no information about the structure and the dependencies of the reusable software units. Learning to integrate these units would take a considerable amount of time. By using the focused architecture and integration model of Section 3, the team can use the integration description for automatic or manual integration. As a result, the integration team needs less knowledge about the integration of a specific reusable unit. However, this is only possible if integration descriptions are available. So the French team have to insert the information in the SSCP environment. But only one time. **Accessibility Problem:** The architecture extension discussed in Section 3 allows the French team to send information to the team in India. They may send only unit meta-information (light-weight scenario) or they may send the complete unit description including all information for integration (heavy-weight scenario). In the first case, the Indian team has information about the unit, but they have to connect and use the repository tool of the French team. This only solves one part of the accessibility problem, because this team has to know how to access the repository system. They are however, able to formulate a query for this system. In the second case, the Indian team can directly integrate the unit without accessing the repository tool (see Figure 1). This result is very important. The Indian team does not need to access this repository. The accessibility problem described in Section 3 can be described by the questions 'Where is the repository?', 'How to access it?', and 'How to use it'. At this point the Indian team does not need to handle the repository because of the other team is doing this. As an result the different question does not occur.

Another interesting result is reuse of this integration knowledge. After the French team added the knowledge to the SSCP system, it is reusable at any time. Different teams located around the world can be supported.

The example discussion shows that both scenarios (light and heavy-weight) may be resolved by the solution described in Section 3. However, this depends on the availability of an integration model and the distributed scenario in use.

5 Conclusions

This paper demonstrates the problems of accessibility and integration when using a distributed industrial scenario. This scenario deals with projects that reuse software units and is implemented by two teams in different locations. Accessibility is a problem if one team requires access to the repository system of another team without having knowledge of the tool. Accessibility is also a problem, if there is no access to a repository

system. Integration becomes a problem if the integration team has no knowledge about the structure and dependencies of the reusable software unit. All problems are based on missing information. The result of these problems is a negative influence on software unit reuse (as it may increase integration time, etc.). This illustrates the importance of information in software unit reuse. A described problem approach uses an extended semantic model that describes different software units (classes, components, and services) in a unified way. This extension describes data that is needed to integrate Studio and Eclipse. Based on this, a distributed architecture of a software reuse environment was extended to solve the discussed problems (accessibility and integration). The accessibility problem is solved by using the architecture to get the integration information without the need of connecting to a repository system. The integration problem is solved by providing the integration information as part of the description of the reusable software unit. The model combined with the architecture is the described novelty of this paper. This paper arrives at the conclusion, that the discussed accessibility and integration problems can be solved by providing the correct meta-information and technical infrastructure to deliver the information. Integration of reusable software units should not need expert knowledge. However, this paper only discuss a solution. The created model and architecture extension should be tested in a additional case study by addressing the advantages for software developers in more complex distributed scenarios.

References

1. Jan Bosch and Petra Bosch-Sijtsema. From integration to composition: On the impact of software product lines, global development and ecosystems. *Journal of Systems and Software*, 83(1):67–76, 2010.
2. Vinicius C. Garcia, Eduardo S. de Almeida, Liana B. Lisboa, Alexandre C. Martins, Silvio R. L. Meira, Daniel Lucredio, and Renata P. de M. Fortes. Toward a code search engine based on the State-of-Art and practice. In 2006 13th Asia Pacific Software Engineering Conference (APSEC'06), pages 61–70, Bangalore, India, 2006.
3. Slinger Jansen, Sjaak Brinkkemper, Ivo Hunink, and Cetin Demir. Pragmatic and opportunistic reuse in innovative start-up companies. *IEEE Software*, 25(6):42–49, 2008.
4. Philippe Kruchten, Rafael Capilla, and Juan Carlos Dueas. The decision view's role in software architecture practice. *IEEE Software*, 26(2):36–42, 2009.
5. Jingyue Li, Reidar Conradi, Christian Bunse, Marco Torchiano, Odd Petter N. Slyngstad, and Maurizio Morisio. Development with Off-the-Shelf components: 10 facts. *IEEE Software*, 26(2):80–87, 2009.
6. Arnold Picot. *Die grenzenlose Unternehmung: Information, Organisation und Management Lehrbuch zur Unternehmensfuehrung im Informationszeitalter*. Gabler, Wiesbaden, neuaufl. edition, 2003.
7. Marcello Rosa, Wil M. P. Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Questionnaire-based variability modeling for system configuration. *Software & Systems Modeling*, 8(2):251–274, 2008.
8. Schneider-Electric. Schneider-Electric website. <http://www.schneider-electric.com>, September 2010.
9. Sajjan G. Shiva and Lubna Abou Shala. Software reuse: Research and practice. In Fourth International Conference on Information Technology (ITNG'07), pages 603–609, Las Vegas, NV, USA, 2007.

10. Clemens Szyperski. Component software: beyond object-oriented programming. ACM Press Addison-Wesley, New York, London, Boston, 2nd ed., 2002.
11. G. Wang and C. K. Fung. Architecture paradigms and their influences and impacts on component-based software systems. In 37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the, pages 272–281, Big Island, Hawaii, 2004.
12. M. Zinn, K. P. Fischer-Hellmann, and A. D. Phippen. Development of a CASE tool for the service based software construction. pages 134–144, Plymouth, 2009. Centre for Information Security and Network Research.
13. M. Zinn, A. Schuette K. P. Fischer-Hellmann, and A. D. Phippen. Information demand model for software unit reuse. In The Proceedings of the 20th International Conference on Software Engineering and Data Engineering, pages 32–39, Las Vegas, June 2011.
14. M. Zinn, G. Turetschek, and A. D. Phippen. Definition of software construction artefacts for software construction. pages 79–91, Plymouth, 2008. Centre for Information Security and Network Research.
15. Marcus Zinn, K. P. Fischer-Hellmann, and Alois Schuette. Finding reusable units of modelling - an ontology approach. In Proceedings of the 8th International Network Conference (INC'2010), pages 377–386, Heidelberg, July 2010.



ifak

Entwurf komplexer
Automatisierungssysteme

EKA 2012

DENKFABRIK

**Beschreibungsmittel, Methoden,
Werkzeuge und Anwendungen**

12. Fachtagung
mit Tutorium und Toolausstellung

09. bis 10. Mai 2012
in Magdeburg
Denkfabrik im Wissenschaftshafen

ifak

Institut für Automation und
Kommunikation e.V. Magdeburg



Otto-von-Guericke-Universität Magdeburg
Institut für Automatisierungstechnik

Herausgeber:

Prof. Dr. Ulrich Jumar
Institut für Automation und Kommunikation e.V. Magdeburg
an der Otto-von-Guericke-Universität Magdeburg
Werner-Heisenberg-Str. 1, 39106 Magdeburg
Tel.: +49 391 9901410; Fax: +49 391 9901590; E-Mail: ulrich.jumar@ifak.eu

Prof. Dr.-Ing. Dr. h. c. Eckehard Schnieder
Technische Universität Braunschweig
Institut für Verkehrssicherheit und Automatisierungstechnik
Langer Kamp 8, 38106 Magdeburg
Tel.: +49 531 391-3317; Fax: +49 531 391-5197; E-Mail: e.schnieder@tu-bs.de

Prof. Dr. Christian Diedrich
Otto-von-Guericke-Universität Magdeburg
Institut für Automatisierungstechnik
Universitätsplatz 2, 39106 Magdeburg
Tel.: +49 391 6718499; Fax: +49 391 6711186; E-Mail: christian.diedrich@ovgu.de

Veranstalter:

ifak – Institut für Automation und Kommunikation e.V. Magdeburg
in Kooperation mit dem Institut für Automatisierungstechnik der Fakultät für Elektrotechnik und
Informationstechnik der Otto-von-Guericke-Universität Magdeburg

Veranstaltungsort:

Denkfabrik im Wissenschaftshafen Magdeburg
Werner-Heisenberg-Straße 1
39106 Magdeburg

Programmkomitee:

Prof. Dr. U. Jumar (Magdeburg)	Prof. Dr. R. Findeisen (Magdeburg)
Prof. Dr. C. Diedrich (Magdeburg)	Prof. Dr. G. Frey (Saarbrücken)
Prof. Dr. E. Schnieder (Braunschweig)	Prof. Dr. P. Göhner (Stuttgart)
Prof. Dr. D. Abel (Aachen)	Prof. Dr. H.-M. Hanisch (Halle-Wittenberg)
Prof. Dr. U. Berger (Cottbus)	Prof. Dr. S. Kowalewski (Aachen)
Dr. J. Birk (Ludwigshafen)	Dr. B.-M. Pfeiffer (Karlsruhe)
Prof. Dr. S. Engell (Dortmund)	Dr. G. Rauprich (Leverkusen)
Prof. Dr. U. Epple (Aachen)	Dr. R. Schoop (Seligenstadt)
Prof. Dr. A. Fay (Hamburg)	Prof. Dr. B. Vogel-Heuser (Kassel)
Prof. Dr. M. Felleisen (Pforzheim)	Prof. Dr. P. Winzer (Wuppertal)
Prof. Dr. W. Fengler (Ilmenau)	

© 2012 ifak Institut für Automation und Kommunikation e.V. Magdeburg, 39106 Magdeburg

Alle Rechte, auch das des auszugsweisen Nachdrucks, der auszugsweisen oder vollständigen Wiedergabe (Fotokopie, Mikroskopie), der Speicherung in Datenverarbeitungsanlagen und das der Übersetzung, vorbehalten.

ISBN: 978-3-940961-72-3

Vorwort

Die Tagung EKA – Entwurf komplexer Automatisierungssysteme hat über ihre nunmehr bereits 20-jährige Geschichte nichts an Aktualität verloren. Von Prof. Dr.-Ing. Dr. h.c. Eckehard Schnieder an der TU Braunschweig ins Leben gerufen, wird die EKA seit 2008 im Zweijahresrhythmus gemeinsam vom Institut für Automation und Kommunikation (ifak) und dem Institut für Automatisierungstechnik der Otto-von-Guericke-Universität in Magdeburg durchgeführt.

Schwerpunkte der Fachtagung sind Beschreibungsmittel, Methoden und Werkzeuge für den Entwurf komplexer Automatisierungssysteme. Ein gelingender Brückenschlag zwischen theoretischen Erkenntnissen und deren praktischer Nutzung ist ein wichtiges Anliegen der Tagung. Als deutsch-sprachige Tagung mit wissenschaftlichem Anspruch möchte die EKA den Entwurf komplexer Automatisierungssysteme in der großen Breite der eingesetzten Methoden und der Vielfalt der Anwendungsgebiete beleuchten.

Neben dem Brückenschlag zwischen Theorie und Anwendung dokumentiert das wissenschaftliche Programm der EKA zugleich die Einheit von Regelungs- und Automatisierungstechnik. Der Tatsache, dass zu der gemeinsamen Fachdisziplin sowohl mathematisch inspirierte systemtheoretische Arbeiten der Regelungs- und Steuerungstechnik als auch informatikgetriebene Methoden der Automatisierungstechnik zählen, wird man sich nicht verschließen können. Überdies ist eine solche Differenzierung zwar innerhalb der Fachdisziplin in gewisser Weise verständlich, historisch gewachsen und akzeptiert.

Für Vertreter aus Wissenschaft und Anwendung anderer Disziplinen ist eine Trennung in Regelungs- und Steuerungstechnik auf der einen Seite und Automatisierungstechnik auf der anderen Seite dagegen wenig plausibel. Eine ganzheitliche Perspektive auf die verschiedenen Facetten der Automation ist mit Blick auf die effiziente Lösung automatisierungstechnischer Aufgaben nicht nur hilfreich, sondern vielfach sogar zwingend. In einigen großen Tagungen und Kongressen wird eine solche Gesamtsicht zwar in einem umfassenden Programm geboten, durch einen hohen Grad der Parallelität von Sitzungen gehen wünschenswerte Synergien aber wieder verloren. Bewusst ist die EKA deshalb einsträngig und gut überschaubar gehalten.

Die Komplexität als langjähriger Gegenstand der Fachtagung EKA liefert im Jahr 2012 auch das Motto des deutschen Automationskongresses. Nicht nur technische Systeme, auch die verschiedenen Bereiche unseres Alltags scheinen durch eine ständig wachsende Komplexität gekennzeichnet. Der Kongress AUTOMATION 2012 greift mit dem gewählten Motto „Komplexität beherrschen – Zukunft sichern“ die hiermit verbundenen Herausforderungen auf.

Ob in der Fertigungs- oder Prozessindustrie, der Energiewirtschaft, im Verkehr oder der Medizintechnik – überall kommt der Automation eine Schlüsselfunktion beim Beherrschen komplexer Systeme zu. Getreu ihrem Anspruch steht bei der EKA die Beherrschung des Entwurfs komplexer Automatisierungssysteme durch Beschreibungsmittel, Methoden und Werkzeuge im Vordergrund. Damit fokussiert die EKA insbesondere wissenschaftliche und methodische Aspekte, die bei geeigneter Abstraktion eine Klammer über vielfältige mögliche Anwendungen bilden.

Das Programmkomitee hat nach sorgfältiger Begutachtung aus den eingereichten Beiträgen eine Auswahl für die Tagung EKA 2012 zusammengestellt. Im vorliegenden Tagungsband sind die Endfassungen der von den Autoren eingereichten ausführlichen Manuskripte zu Vorträgen und Postern zusammengestellt, wobei sich die Reihenfolge an der zeitlichen Abfolge der Sitzungen orientiert:

- Beschreibungsmittel
- Posterpräsentation
- Modellierung und Entwurf
- Zuverlässigkeit, Konsistenz
- Werkzeuge
- Anwendungen

Vorgeschaltet ist der Fachtagung EKA 2012 wieder ein Tutorium, dessen Beiträge ebenfalls in den vorliegenden Tagungsband aufgenommen wurden. Das Systemengineering in der Automation wird durch die Interaktionen zwischen technischen Systemen, deren Komponenten, den Bearbeitern verschiedener Professionen und vielfältigen Softwarewerkzeugen bestimmt. Diese Interaktionen gehen über einen rein technischen Datenaustausch hinaus. Erforderlich ist ein eindeutiges Verständnis der Sinnhaftigkeit, d. h. der Bedeutung hinter den Daten. Unter der Überschrift „Semantik in der Automation“ widmet sich das Tutorium der EKA 2012 deshalb dem Aufgabengebiet der semantischen Beschreibung und daraus ableitbaren Assistenzfunktionen.

Im Namen der drei Tagungsleiter wünsche ich Ihnen eine interessante Lektüre des Tagungsbandes der 12. EKA



Prof. Dr.-Ing. Ulrich Jumar
im Namen der Herausgeber

Prof. Dr.-Ing. Eckehard Schnieder
Institut für Verkehrssicherheit und
Automatisierungstechnik
Technische Universität Braunschweig

Prof. Dr.-Ing. Christian Diedrich
Institut für Automatisierungstechnik
Otto-von-Guericke-Universität
Magdeburg

Prof. Dr.-Ing. Ulrich Jumar
ifak – Institut f. Automation
und Kommunikation e.V.
Magdeburg

Inhaltsverzeichnis:

<i>Modellierungsvorschlag zur grafischen Beschreibung alternativer und paralleler Prozessabläufe auf Basis eines Vergleichs bestehender Beschreibungsmittel – Diskussion zur Erweiterung der VDI/VDE-Richtlinie 3682 „Formalisierte Prozessbeschreibung“</i> Lars Christiansen, Tobias Jäger, Frank Schumacher, Alexander Fay (HSU Hamburg)	1
<i>SFC-based Process Description for Complex Automation Functionalities</i> Liyong Yu, Gustavo Quirós, Sten Grüner, Ulrich Epple (RWTH Aachen)	13
<i>Kombinierung constraintbasierter Methoden und Petrinetztechniken zur Modellverifizierung offener, diskreter und reaktiver Systeme</i> Jan Krause, Stephan Magnus (ifak Magdeburg)	21
<i>Semantische Optimierung im Requirements Engineering durch Terminologiemanagement</i> Susanne Arndt (TU Braunschweig)	35
<i>Beschreibungsmittel für Abhängigkeiten zwischen physikalischen und funktionalen Strukturen</i> Markus Göring (Vattenfall), Alexander Fay (HSU Hamburg).....	47
<i>Analyse der praktischen Relevanz verschiedener Beschreibungsmittel im Entwurfsprozess von Produktionssystemen</i> Matthias Foehr (Siemens AG), Arndt Lüder, Alexej Steblau, Matthias Lüder (Universität Magdeburg) ..	61
<i>Entwurf eines rigorosen Modells für die Echtzeitsimulation auf Automatisierungssystemen</i> Kai Krüning, Caspar Kielwein, Ulrich Epple (RWTH Aachen)	73
<i>Konzept und Erfahrungen beim Abgleich mehrerer Domain-Ontologien</i> Thomas Hadlich, Christoph Engel, Christian Diedrich (Universität Magdeburg), Mathias Mühlhause (Siemens AG)	81
<i>Zwei-Ebenen-Modellierung eines automatisierten Straßenverkehrs mit Petrinetzen</i> Matthias Hübner, Eckehard Schnieder (TU Braunschweig)	91
<i>Reuseable Software Unit Knowledge for Device Deployment</i> Marcus Zinn (University of Plymouth/Schneider Electric), Klaus Peter Fischer-Hellmann (Universität Darmstadt), Ronald Schoop (Schneider Electric)	99
<i>Untersuchung zur FPGA-Implementierung von Mess- und Regelungsalgorithmen</i> Irina Gushchina, Bernd Däne, Alexey Moskalev, Wolfgang Fengler (TU Ilmenau).....	111
<i>Funktionsorientierte Auslegung eines Linearantriebs</i> Florian Riekhof, Petra Winzer (Bergische Universität Wuppertal), Linus Wörner, Stefan Kulig TU Dortmund)	121
<i>Ressourceneinsatzplanung für „Robot Farming“ Konzepte in der Montage</i> Volker Zipter, Michael Zürn (Daimler AG), Ulrich Berger (BTU Cottbus).....	139
<i>Modellierung und Simulation von Cyber-Physical Systems</i> Liu Liu, Felix Felgner, Georg Frey (Universität des Saarlandes)	149
<i>Erweiterung des V-Modells® für den Entwurf von verteilten Automatisierungssystemen</i> Timo Frank, Birgit Vogel-Heuser (TU München), Thomas Hadlich, Christian Diedrich (Universität Magdeburg), Karin Eckert (HSU Hamburg)	159

“Reuseable Software Unit Knowledge for Device Deployment”

M. Zinn^{1,3}, K. P. Fischer-Hellmann², R. Schoop³

¹University of Plymouth, CSCAN, Drake Circus Devon PL48AA Plymouth, UK, ²University of Applied Science Darmstadt, Haardtring 100 64295 Darmstadt, Germany, ³Schneider Electric Automation GmbH, Steinheimer Str. 117, 63500 Seligenstadt, Germany
E-Mail: marcus.zinn@plymouth.ac.uk, k.p.fischer-hellmann@digamma.de,
ronald.schoop@schneider-electric.com

Abstract: Deployment of software units into embedded devices requires dedicated knowledge. Usually, this is specialist knowledge, covering technology (hardware platform, software technology) functionality (interfaces, interaction state machines), and processes (deployment procedures, rules). Gaining and applying this knowledge requires time. This paper presents the results of a case study identifying relevant knowledge for device deployment. The study analyses three different embedded device engines supporting service based device deployment. The identified knowledge, including how to deploy software units into the analysed device platforms, is used to construct a new model and to extend an existing semantic service based software unit reuse model. As a result, a usage environment employing this model enables an inexperienced user to repeat the stored deployment procedures without having all the required knowledge. In other words, these users will reuse the stored software unit knowledge. This paper addresses the topic of device deployment and software reuse knowledge.

Keywords: Software Unit Reuse, Deployment, Information, Knowledge

1 Introduction

Deployment of embedded devices is seen as the physical set up of devices in a specific environment (e.g. medical devices [BuDoVi2009]). From a software development perspective, deployment may be seen as the installation of software on a system [BuDoVi2009]. From here on, the term 'deployment' will refer to the latter definition. The conception of embedded devices has changed in the past. Originally, such devices were perceived as [Gill2005]: (1) Specialised on a specific task by limited functionality. (2) Built for an unchanging environment. (3) Limited by resources. (4). Nowadays, they are perceived as embedded systems, which are characterised as being [Gill2005]: (1) (Self-)adaptive, open and more efficient. (2) Capable of dynamically handling multiple tasks. (3) 'Plug and Play'-able for integration. The reason for this change of perception can be seen “as a consequence of the integration of IT” [Gill2005] into the field of embedded systems. The authors perceive this change to be a result of advancements in hardware and software of embedded systems. Over time, hardware became more capable of handling increasingly complex software instructions, more advanced software technologies, and platforms

[GIMaCa2006], [Gill2005]. This increased flexibility enables the implementation of special software features, namely: Fault Tolerance [PiCaSa2008], Security [GoWoBu2008], and Dynamic Infrastructure [KarTa2009]. The high number of available embedded devices poses a problem for software (re)use. This problem is especially apparent in the area of automation where a lot of different types of devices exist. Usually, available devices are distinguished by hardware technology, software technology, form factor, performance classes and safety features. This results in the fragmentation of both software platforms and libraries for embedded devices. Therefore, the task of (re)using such software units for embedded devices is becoming increasingly more complicated and requires special knowledge for adaption, integration, transformation, and deployment. This kind of knowledge is not universally available and might be difficult to acquire, especially for younger professionals [ShiSha2007]. The solution to these problems can be identified as the reduction and simplification of required knowledge that enables the user to deploy (in other words to reuse deployment knowledge of) software units without the complete knowledge previously required. This paper aims to achieve this by extending an existing software unit reuse model with its own deployment description model. This work is part of the research on the Service-based Software Construction Process (SSCP) [ZiFHP2010] incorporating the field of Software Reuse Environments. Its goal is to find a semantic model (about search, adaption, integration, and deployment of software units) combined with service technology that aids software engineers to perform software reuse (search, adaption, integration, and deployment) without having all the previously required knowledge.

2 State of the art examples and related work

One way of handling embedded device deployment is to use deployment engines for communication. Typical examples from the automation area are: Sonata Engine, Dynamic Deployment with DPWS, and OSGi Deployment. The **Sonata** Engine was developed by the companies Inico and Schneider Electric [Sonata2011]. It is a deployment engine for automation devices with a built-in compiler and deployment system, which can be used on different device platforms. An important feature is the built in development environment. Code can be directly entered on a web page running on the device. The compilation and deployment process is performed by the Sonata Engine on the device itself. This feature makes the engine flexible in changing the device functionality. Instead of using this “integrated” method, tools can be developed to control the deployment process externally. **Dynamic Deployment** is based on Web Service Management (WS-Man) and the Device Profile for Web Service (DPWS). WS-Management is a network protocol for XML based web services and is used to exchange SOAP based messages between systems containing management information. WS-Man is commonly employed to manage system resources. It is a standard protocol used by the Desktop Management Task Force (DMTF) and can be combined with different resource description models like DPWS. DPWS is a profile for embedded devices [NiReDri2009]. It defines web service profiles like discovery and can also be used for runtime deployment [Gill2005]. In the SOA4D project WS-Man was combined with DPWS [SOA4D2011] to develop a dynamic

Service Oriented Architecture (SOA) infrastructure for devices. To deploy services, the Dynamic Deployment Engine needs hardware configuration files, WS-Man Resource files (including DPWS Service information), and specific binary files. In contrast to the Sonata Engine, the files for the deployment process will not be created on the device, but externally. **OSGi** (Open Service Gateway Initiative) is a common way of managing software units for devices, such as automotive devices (telematics), smart home devices (home appliances, security systems, energy management systems), and mobile devices (cellular phones, PDAs) [ACE2011]. In the last few years it was also utilized to handle reusable (Java) components on system level. Software for OSGi based devices is distributed in form of packages called 'OSGi bundles'. A bundle contains all interfaces, classes, resource files, and a manifest file in a single JAR file. OSGi can also be viewed as a configuration system [OSGi2009]. Normally, OSGi uses special plugins for the Eclipse IDE. These plugins automatically manage the deployment process automatically for the user. The communication between plugin and device is overseen by a device agent. For this communication the BaseOMA DM protocol over HTTP or HTTPS is used. The exact communication specification is available as Open Source. [OSGi2009]. All three engines simplify the deployment process by providing a standardised way to deploy software on different devices. This helps to reduce the number of different 'deployment ways'. However, the problem persists because of two facts. The first fact is that not every device is able to run a device engine. This is normally limited by the resource requirements of the device engines. The second fact is that also a limited number of 'deployment ways' still require the user to know each of the deployment models. The next section describes an experimental setup which focuses on the second fact.

In the area of deployment of software units to devices different approaches exist, like Rubus, COMDES-II, and ProCom Systems [CaFeMT2010]. Rubus [HaHTNo2008] is a component based model supporting dependency analysis of embedded system that contains multiple embedded devices. With extension Rubus is able to synchronise and update devices. This is done by a Rubus specific model and extensions for the devices and the Rubus system for the necessary communication. Two other approaches, the component based software framework COMDES-II [XuSieAn] and the ProCom System [CaFeMT2010] are very similar. In both approaches an embedded system which contains multiple devices can be modelled by using a 'Virtual Node' concept. Each device in the real system is represented by a virtual node in the used model. For each node deployment content and the deployment process can be modelled. In the way of modelling these approaches differ. COMDES-II uses a combination of a special XML based description language and application extension. ProCom is able to use a combination of existing description languages like SysML and a Model Driven Development approach. All three approaches are made especially for the embedded device area. Each approach handles complete embedded systems. This differs to the approach of this paper. In this paper an existing system will be extended to handle device deployment. This system is prepared for reuse of software units and their specific knowledge. Also the aim of this approach is different. In this paper the

aim is to support software developers to deploy software units to devices without the specific knowledge.

3 Device Deployment Case Study

The case study's aim is to identify and reuse necessary knowledge for embedded device deployment. The study is divided into five steps: (1) Design the experimental environment. This includes the description of the different devices, the expected results, and the procedure model for the experiment. Furthermore, the software units that should be (re)used for the deployment of the different devices must be specified. (2) Analyse the required knowledge for deployment to different devices. The analysis scope focuses on applying knowledge of the reused software unit, the deployment setup, and the communication setup. (3) Define a model based on the collected data. This model should describe the knowledge required for deployment of software units to specific device types. (4) Extend an existing software unit reuse model and system for the possibility of creating and executing deployment rules for different devices. (5) Repeat the deployment process of Step 1 in an existing software unit reuse environment, using the newly extended model of Step 4.

3.1 Experimental Setup

Embedded Device type	File No.	Description	File extension (Type)
Sonata (DPWS)	1	File with structured text. Including Interface Description, Event Handler Description, Hardware configuration, Hardware - Variable Mapping	DAT (XML)
DynDeployment	1	Hardware configuration	INI (Text)
DynDeployment	2	WS Management Files, including Device description, service class, service interface description.	XML (Text)
DynDeployment	3	Binary Files needed as additional assemblies of the base Celio Engine	BIN (Binary)
DynDeployment	4	Compiled File (including all others)	UPL (Binary)
OSGi	1	Interface description file	XML (Text)
OSGi	2	Ressource description file	BIN (Binary)
OSGi	3	Manifest File (Project)	XML (Text)
OSGi	4	Classes for the service	Java Class (Java Code)
OSGi	5	Compiled File (including all others)	JAR (Binary)

Table 1: Object Description for deployment case study

The study's test subjects consisted of the following three embedded devices engines: Advantys STB (distributed I/O device) using the Sonata engine, Advantys STB using the Dynamic Deployment engine, and GX300 Gateway (Ethernet gateway with I/O) using OSGI Deployment. They have some common properties: (1). They are embedded devices with I/O in the automation area. (2) They provide a TCP/IP interface and a web service or remote service interface. (3) Their deployment process may be seen as service based. All chosen devices were prepared for the case study by being connected to a power supply and, via Ethernet, to a TCP/IP network. Also, a computer running Microsoft Windows XP operating system was also connected to the network. This computer contained all required software tools (explained in the following sections) and the deployment objects. A number of different software units were chosen as deployment objects

(required files). The fact that these objects differ in technology and structure is not important for the study. In all three cases a web service was created. Even though the functional content of these web services differs, they must be compatible to the software reuse environment mentioned in Step 5 (see Section 3.5). The results of the study have been proven by an expert of device deployment regarding the focused device engines. The same expert has been observed in Section 4.2. Table 1 describes the deployment objects and shows their related devices. The case study requires the following three statements to be true: (1) For the three different deployment processes a description model can be defined describing all relevant data. (2) The description model can be used to extend the SSCP model. (3) The deployment processes can be reproduced by using the SSCP model extension.

3.2 Device Deployment Analysis

In order to collect all necessary information, an expert has been monitored during the process of deploying the given software units to the different devices. The first investigated test setup was the Sonata [Sonata2011] engine on an Advantys STB. A previously configured file ('Project.dat') was uploaded to the device. Before the upload process, the expert verified the device was running and sent a 'Stop' signal to the device. To be able to restore the device, the Project.dat file currently in use was stored on the computer by sending a 'Save' signal to the device. The expert uploaded the new Project.dat file by sending a 'Load' command. The building and deployment process was initialised by a 'Build' command. In order to start the newly configured device, the Sonata engine requires an additional 'Run' command. The communication knowledge required for the Sonata engine is shown in Fig. 1..

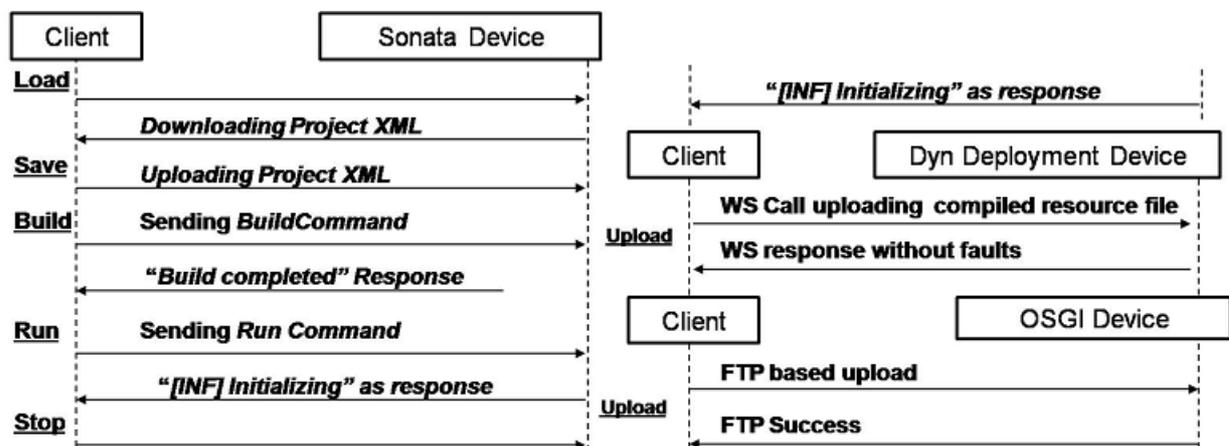


Figure 1: Communication sequence for a the focused deployment procedures

The second test setup was an Advantys STB with Dynamic Deployment. Similar to the first test, the expert verified the power status of the device. As preparation, the hardware configuration file, the WS-Man resource files and the Zelio Engine binary file were compiled into a new binary file ('Project.upl'). This file was then uploaded using a web service call. After the upload, the device operated fully automated. There was no need for manual power switching or additional commands. The expert was not allowed to shut down the device during deployment because this

could damage the electronic device. Fig. 1 summarizes the important communication message. An OSGi based system supports only Java libraries (JAR files). To create these files, a complete Java project must be created using the Eclipse IDE. Such a project requires specific OSGi engine libraries to be included. For this to proceed, the required source code must be developed and a resource file must be created, describing the physical device. This process is similar to the other test setups. The upload was performed automatically by an Eclipse plug-in, created by the OSGi community. With the help of the tool WireShark, the messages between the plug-in and the device were measured. Fig. 1 shows the important messages. The expert had to manually restart the device (power off and on). This is necessary because the device does not support automatic restart by the OSGi engine.

3.3 Defining a model

Based on the results of Section 3.1, the deployment model shown in Fig. 2 can be created.

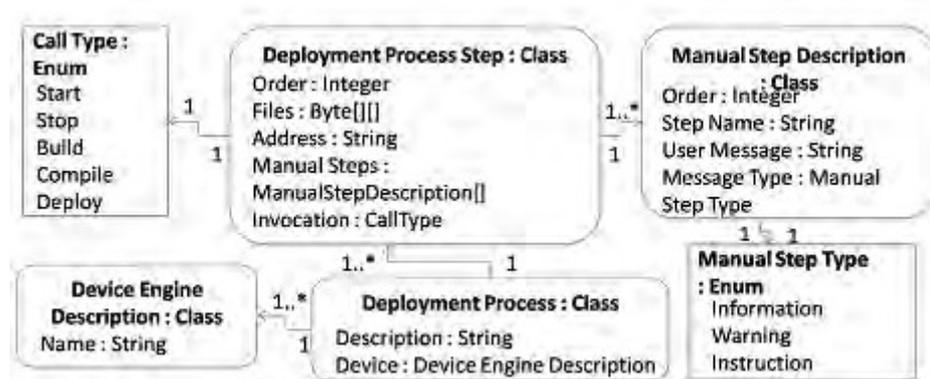


Figure 2: Deployment Model (Case Study Result)

An instance of this model constitutes a complete **'Deployment Process' (DP)**. A DP includes all information and describes all steps that are necessary to successfully complete the deployment. As a result, a DP consists of one or more deployment process steps 'DPS'. A DPS must describe the input files ('Files'), the communication information ('Address'), the description for manual steps ('Manual Step Description'), and the type of deployment call ('Invocation'). Additionally, a DP has a 'Device Engine Description', that represents the device engine used for deployment. The 'Files' are generic files (byte type). The term 'Address' refers to an address text (string type). The 'Manual Step Description' is a more complex type, describing a set of 'User Messages' that can include warnings, information or instructions. A message includes text (string type) and a 'Message type' which defines the type of message within the enumeration of the following entities: (1) 'Warning' - Critical information that must be read by the user. (2) 'Instruction' - Order that must be executed by the user to continue. (3) 'Information' - Information that is informative but not necessary. The last information of a DPS is the 'Invocation' that can be one of the following entities enumerated below. (1) 'Start' - Command to start a device. (2) 'Stop' - Command to stop a device. (3) 'Build' - Command to start the compilation process on a device. (4) 'Upload' - Command to upload one or more files to a device. (5) 'Save' - Command to

download one or more files from a device. This gives rise to the question of how this model relates to the information gathered in Section 3.2. Table 2 shows the instantiated model (without the three DP instances).

TC					DPS			
	OD.	File	AD.		Manual Step			Inc.
	OD.	File	AD.	OD.	Name	User-Message	M-Type	
1	1	None	*	1	Stopping	Active configuration will be stopped	Information	Stop
	2	None	*	2	Saving	Storing current configuration	Information	Stop
	3	Project.xml	*	3	Uploading	Uploading current configuration	Information	Upload
	4	None	*	4	Building	Building new configuration	Information	Build
	5	None	*	5	Running	Start new device configuration	Information	Start
2	1	Project.upl	*	1	Building	Deploying - Device will restart	Information	Build
	2	None	*	2	Building	Do not shutdown device	Warning	None
3	1	Project.jar	*	1	Building	Deploying...	Information	Build
	2	None	*	2	Building	Do not shutdown the device	Warning	None
	3	None	*	3	Restarting	Please restart the device manually	Command	None

Table 2: Instantiated model (Case Study Input) (OD = Order, Ad = Address, Inv = Invocation)

3.4 Extending an existing Software Reuse Model

This case study extends an existing model that describes reusable software units and information for their usage. The model to be extended is the Service based Software Construction Process Model (SSCP Model), which is used in the Service based Software Construction Process (SSCP) approach [ZiFHP2010]. The SSCP Model is a semantic data model that aims to aid software developers, engaged in software unit reuse by enabling them to use one single service to search, adapt, and integrate software units without possessing the otherwise required knowledge. This is made possible by the service offering the necessary knowledge for software unit reuse. The SSCP was chosen because of the following specific attributes required for this study. The SSCP already includes a semantic description model for software units and additional information. This model describes units in a generic way, so a reusable software unit can be anything that can be used for reuse of software units. Based on this description different models about usage (e.g., integration or adaption) are defined. The complete model is extensible, so new models describing other reuse usage activities can be added. The final important fact is that an SSCP application environment continues to exist. This can be used for the study and extended by adapters (plugins) to work with the knowledge about software units described in the SSCP model. This approach classifies the necessary knowledge into two different types: *'Shared-Knowledge'* that can be described in a unified way. This knowledge will be referenced by different software unit reuse activities (e.g., Search, Adaption, Integration and Deployment of the service. An example of Shared-Knowledge is a unified description of classes, components, and services as a software unit. *'Specific-Knowledge'* which is highly particular, not abstractly expressible and has only one specific purpose. Usually, this knowledge is represented by an adapter (plugin) of the SSCP approach handling other applications or systems for software unit reuse activities. An example of Specific-Knowledge is an adapter which integrates software units into a development environment like Eclipse. Therefore, the adapter includes all specific knowledge to handle the tool. In the integration process, the adapter integrates a software unit described by the Shared-Knowledge

into the Eclipse environment. Unlike Shared Knowledge, this information is not available in a unified data model (Shared-Knowledge) and is integrated differently into the SSCP environment (see Section 3.5). Adapters may be used for different software units and their stored knowledge. For example, an adapter integrating .NET software units into the .NET IDE Visual Studio can be used for different .NET software units. General knowledge is described by the SSCP Model and includes different model layers. These layers are classified in three sections: (1) Unit View - Description of units for modelling: Classes, Components, and Services. (2) System View - Description of units for modelling: Classes, Components, and Services. (3) Business View - Description of extended information related to user interaction.

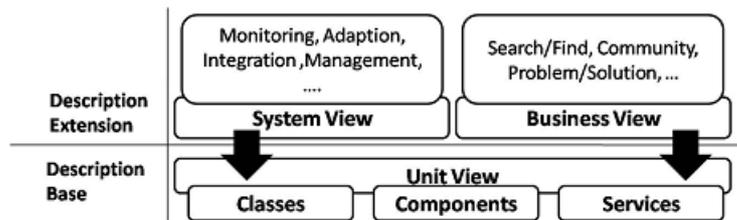


Figure 3: Description layers of the Service based Software Construction Model

These three sections encompass different levels of data (see Fig. 3). The first level 'Unit View' describes software units (classes, components, and services) in a unified way. The 'System View' extends this layer, offering a specific description of different applications of software units from the application or system perspective. It includes information that is required by applications or the system itself to perform or support software unit reuse, for example the Integration Description Model (IDM). By using this model, it is possible to describe the process of integrating a software unit into an integrated development environment (IDE). The IDM is a unified description of integration processes and their required data. The 'Business View' extends the 'Unit View' by providing information that is important for the user (e.g. search or problem definition). An example would be the semantic description of a simpler search function (see [ZiFHPh2010] for more information). This paper aims to extend the 'System View' with a new data level called 'Deployment'. The extension undertaken in this paper is a new model describing extended deployment data for modelling units. In order to extend the model, it is important to understand the structure of the existing SSCP model and to find similarities linking both description models. The SSCP Model is divided into four sections. Each section describes one part of a software unit. Part 1 describes extended data such as it's author and developer. Part 2 describes the software unit as the solution within the range of the related problem. Part 3 describes a unit as a technical component. The last area describes semantic search information.). This document emphasises the importance of Part 3, as all data levels of the 'System View' (see Fig. 3 will be linked to this section, representing technical solutions. Example: Component is developed by .Net Technology that uses the .NET Platform and the VisualBasic .NET programming language. In order to extend the model, A link between the original data model and the new deployment model must be found for extension. This link can be created by defining a 'Deployment Extension' Entity. For this to be obtained, this entity is required to separate the deployment extension from other existing extensions (like transformation). The new model is

then attached to the SSCP model at this entity and receives its own entity called 'Device Deployment'. This entity relates all deployment model extension instances to a specific 'Unit' entity. For this purpose, the model created in Section 3.3 is used, employing the typical basic types string, integer, and enumeration (subclasses). Only the 'File' entity is not created because of a pre-existing file description already exists in the original model. All input files used in the test setups were 'Machine Readable Content' and will therefore only be used by systems and not by humans (like documentation). Therefore, a direct link was created to the 'Machine Readable Content' entity.

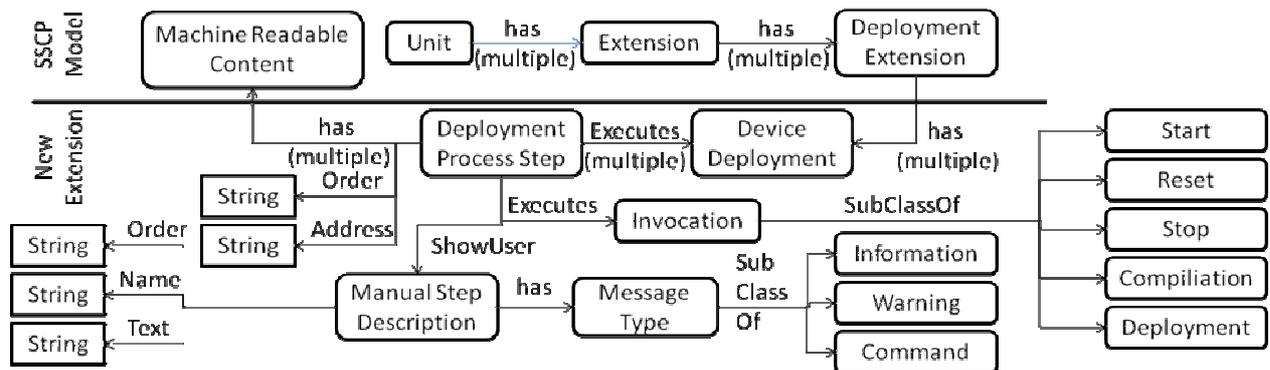


Figure 4: SSCP Model Extension - Device Deployment

3.5 Using Reusable Software Unit Knowledge for Device Deployment

In the case study, the 'Prometheus 2011' Tool which is within an SSCP Environment was extended to support the deployment extension shown in Section 3.3. Three adapters were created to support communication between the SSCP Environment and devices using the three platforms (see Section 3.1). The adapters know how to deploy data into the device (see Section 3.2) and how to start the deployment process (see Section 3.2). Furthermore, the adapters can be reused for future needs by means of the data shown in this study or other data which is compatible with the device. In Test Setup 1 and 3, the adapters themselves implement the protocols. In Test Setup 2, the upload tool was wrapped by the adapter. All adapters use a default interface for device deployment. This interface includes only one method for implementation: *SetDeploymentProcess (DeploymentProcess deploymentProcessObject)*. The classes used in the interface are based on the model description of the SSCP model (see Fig. 4). This constitutes an important part of the SSCP approach. Adapters include specific data for handling more general data that is loaded by the interface call and is used by the adapters to perform the deployment process. Therefore, general data (software units and their usage information) are separated by specific data. The user uploading the software units and thereby creating the deployment activities combines the general data with a specific adapter. Currently, each adapter needs a DP instance for each device or deployment procedure. The data shown in Table 1 was saved in the SSCP environment in a SQL database following the SSCP data model as shown in Table 2. This means a 'Unit' description was created for each software unit. Additionally a 'Deployment Process' description was created and added to each 'Unit'. At the end, each 'Deployment Process' was linked to an adapter. The user interface of the Prometheus 2011 Tool was extended to allow for selection and execution of

deployment processes. This also includes an input method for addresses that are requested by the adapters during the deployment process. Table 2 does not show these addresses due to limited presentation space. Following addresses are used in the three test cases: 192.168.178.26:8089, .30:8088, and.31:8090. After these changes, the system was able to perform device deployment of the stored reusable software units. The new device deployment procedure was tested by two individuals. The first person was the expert of Section 3.2. This was done to test if all device deployment processes were valid and performed correctly. The second person was not specialised in device deployment (a non-expert). Both individuals were able to perform the device deployment successfully. This means that the same necessary knowledge for deployment of software units to specific devices was reused by different people.

3.6 Result Discussion

The first result of the case study was the three different deployment processes of Section 3.1 could be replicated within the SSCP environment (see Section 3.1). In all cases, this resulted in working devices with the same software units. The second result is that the same knowledge was used. The difference between the two experiments is that originally the user (the expert) was required to already possess all necessary knowledge, whereas, the second scenario, the knowledge was part of a specific knowledge system (Prometheus 2011). The amount of deployment knowledge is the same in both scenarios. However, in the second scenario a user without pre-existing knowledge of device deployment (a non-expert) was able to successfully perform it in all three cases. This is the most important result of the case study. After the knowledge was integrated into the environment, it became accessible to users with less pre-existing knowledge. From this stage onwards, the knowledge can be (re)used by untrained users. The study shows that device deployment requires pre-existing knowledge, in this case the address of the different devices. Furthermore, the user has to follow instructions coming from the deployment process that may require him to have an electrical engineering background. The three adapters build for the different deployment procedures are separated from the input knowledge. Therefore, they are accessible for other software units that should be deployed to the same device or device engine. This is not demonstrated in this paper. However, the case study also shows other effects of a user's pre-existing knowledge because knowledge about the SSCP environment is now part of the requirement for device deployment. Test subjects were trained to use the SSCP environment before starting the experiment. Whether acquiring knowledge about the SSCP environment or three unique deployment processes is more difficult for users is to be evaluated. This, however, was not part of this study's investigation.

4 Conclusion and Future Work

During the case study, a description model was defined describing all relevant data for three different deployment processes. This model was used to extend the Software Reuse Information Demand (SRID) Model with deployment activity descriptions. The study includes the analysis of an expert who was deploying different software units to three different device platforms that

require different levels of usage knowledge from a user. The measured knowledge in this analysis was saved into a Service Based Software Construction Process (SSCP) environment that is used to store and execute software reuse knowledge. This stored knowledge was reused by an inexperienced user who was using the extended SSCP environment. The non-expert user was able to create the same results than the expert user, but without the same expertise. The paper's case study demonstrates that it is possible to integrate the usage knowledge for different device deployment platforms in the SSCP environment. This stored knowledge can be (re)used by users that were previously unable to perform device deployment on the different platforms. From the authors' perspective, the following aims, suggested in the beginning, were fulfilled: (1) A description model was defined describing all relevant data for the three different deployment processes. (2) This description model was used to extend the SSCP model. (3) The deployment processes was reproduced by using the SSCP model. As demonstrated in the study, some knowledge is very specific and cannot be generically reused. In future research, this knowledge may be analysed to find a way to decrease the amount of specialised (not generalisable) knowledge. This may make the approach shown in this publication more effective. Section 3.5 indicates that the result of this study is a model enabling inexperienced users to deploy software units to devices. This gives rise to the question whether there is any reduction of time and effort. The number of different devices or device platforms, expert users, and inexperienced users are included in this publication to get a comprehensive impression of the model this paper presents. In future research, this stands to be confirmed by extending the number of participants and devices of this study. Special focus must be laid on devices as this duplication focuses on device engines. Another result demonstrated by the study are that adapters for the SSCP environment can be reused for other software units and should be 'reused' for the same device or device engine. This constitutes a further aspect future research might focus on.

References

- [ACE2011] Apache ACE Project, available at <http://incubator.apache.org/ace/a-brief-introduction.html>, last visted 13 July 2011.
- [BuDoVi2009] Burg, S.; Dolstra, E.; and Visser, E.; "Software deployment in a dynamic cloud: From device to service orientation in a hospital environment", Proceeding of First Workshop on Software Engineering Challenges in Cloud Computing (ICSE 2009), IEEE Computer Society, 2009.
- [CaFeMT2010] Carlson, J; Feljan, J; Maki-Turja, J; and Sjodin, M; , "Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems", 36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2010.
- [EiPfaMu2010] Eichhorn, M.; Pfannenstein, M.; Muhra, D.; and Steinbach, E.; , "A SOA-based middleware concept for in-vehicle service discovery and device integration," Intelligent Vehicles Symposium (IV), 2010 IEEE , pp. 663-669, 2010.
- [GIMaCa2006] Gilart-Iglesias, V.; Macia-Perez, F.; Capella-D'alton, A.; Gil-Martinez-Abarca, J.A.; , "Industrial Machines as a Service: A Model Based on Embedded Devices

and Web Services” , International Conference on Industrial Informatics, IEEE , pp. 630-635, 2006.

- [Gill2005] Gill, H.; , “Challenges for critical embedded systems”, Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005), pp. 7- 9, 2005.
- [GoWoBu2008] Gogniat, G.; Wolf, T.; Burlison, W.; Diguët, J.-P.; Bossuet, L.; and Vaslin, R.; , “Reconfigurable Hardware for High-Security and High-Performance Embedded Systems: The SAFES Perspective”, IEEE Transactions on Very Large Scale Integration (VLSI) System , vol.16, no.2, pp. 144-155, 2008.
- [HaHTNo2008] Hanninen, K.H; Haki-Turja, J.; Nolin, M.; Lindberg, M.; Lundback, J.;, and Lundback, K.-L.; , “The Rubus Component Model for Resource Constrained Real-Time Systems”, In 3rd Int. Symposium on Industrial Embedded Systems, 2008.
- [KarTa2009] Karnouskos, S.; and Tariq, M.M.J.; , “Using multi-agent systems to simulate dynamic infrastructures populated with large numbers of web service enabled devices”, International Symposium on Autonomous Decentralized Systems (ISADS '09), pp. 1-7, 2009.
- [NiReDri2009] Nixon, T.; Regnier, A.; Driscoll, D.; and Mensch A.; “Devices Profile for Web Services (DPWS) Specification”, July 2009, available at <http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.pdf>, last visited July 2011.
- [OSGi2009] OSGi Alliance, “OSGi Service Platform Core Specification and Service Compendium - Release 4, Version 4.2,”, 2009, available at <http://www.osgi.org/Specifications/HomePage>, last visited 13 July 2011.
- [PiCaSa2008] Pinello, C.; Carloni, L. P.; and Sangiovanni-Vincentelli A. L.; , “Fault-Tolerant Distributed Deployment of Embedded Control Software” Transactions on Computer-Aided Design of Integrated Circuits and Systems, IEEE, vol.27, no.5, pp. 906-919, 2008.
- [ShiSha2007] Shiva, S. G.; and Shala, L. A.; , “Software Reuse: Research and Practice”, In Fourth International Conference on Information Technology (ITNG'07), pp. 603–609, Las Vegas, NV, USA, 2007.
- [SOA4D2011] SOA4D Group, “Service-oriented Architectures for Devices”, available at <http://www.soa4d.org>, last visited 13 July 2011.
- [Sonata2011] Sonata Engine, Available at http://www.inicotech.com/products_oem.html, last visited 13 July 2011.
- [ZiFHPh2010] Zinn, M.; Fischer-Hellmann, K. P.; and Phippen, A. D.; Schuette, A.; , “Finding Reusable Units of Modelling - an Ontology Approach”, Proceedings of the Eighth International Network Conference (INC 2010), Heidelberg, Germany, 8-10 July, pp. 377-386, 2010.
- [XuSieAn] Xu Ke; Sierszecki, K.; Angelov, C.; , “COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems”, 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)

Autorenverzeichnis:

Adamczyk, Heiko.....	297	Nke, Yannick.....	189
Arndt, Susanne.....	35	Obst, Michael.....	179
Berger, Ulrich.....	139	Pacholik, Alexander.....	287
Christiansen, Lars.....	1, 239	Rau, Florian.....	171
Däne, Bernd.....	111, 287	Rauscher, Michael.....	199
Diedrich, Christian.....	81, 159, 227, 319	Rehkopf, Andreas.....	171, 267
Ding, Yongjian.....	217	Riekhof, Florian.....	121
Doherr, Falk.....	179	Runde, Stefan.....	253
Drumm, Oliver.....	253	Quirós, Gustavo.....	13
Eckert, Karin.....	159	Schnieder, Eckehard.....	91
Engel, Christoph.....	81	Schoop, Ronald.....	99
Epple, Ulrich.....	13, 73, 277	Schumacher, Frank.....	1
Evertz, Lars.....	277	Sokolov, Sergiy.....	227
Fay, Alexander.....	1, 47, 239, 319	Steb lau, Alexej.....	61
Feldmann, Stefan.....	307	Stein, Christian.....	321
Felgner, Felix.....	149	Strube, Martin.....	239
Fengler, Wolfgang.....	111, 287	Urbas, Leon.....	179
Fischer-Hellmann, Klaus Peter.....	99	Vogel-Heuser, Birgit.....	159, 307
Foehr, Matthias.....	61	Winzer, Petra.....	121
Frank, Timo.....	159	Wörner, Linus.....	121
Frey, Georg.....	149	Yu, Liyong.....	13
Fuchs, Julia.....	307	Zinn, Marcus.....	99
Glimm, Birte.....	319	Zipter, Volker.....	139
Göhner, Peter.....	199	Zürn, Michael.....	139
Gö ring, Markus.....	47		
Grüner, Sten.....	13		
Gu, Chunlei.....	217		
Gushchina, Irina.....	111		
Hadlich, Thomas.....	81, 159		
Hauptmanns, Ulrich.....	217		
Hintze, Elke.....	297		
Hübner, Matthias.....	91		
Jäger, Tobias.....	1, 239		
Kielwein, Caspar.....	73		
Krause, Jan.....	21		
Krätzig, Marko.....	297		
Krüning, Kai.....	73		
Kulig, Stefan.....	121		
Liu, Liu.....	149		
Liu, Zheng.....	227		
Lüder, Arndt.....	61		
Lüder, Matthias.....	61		
Lunze, Jan.....	189		
Maga, Camelia R.....	209		
Magnus, Stephan.....	21		
Moskalev, Alexey.....	111		
Mühlhause, Mathias.....	81		
Müller, Christian.....	171, 267		

ISBN 978-3-940961-72-3

Economic Efficiency Control on Data Centre Resources in Heterogeneous Cost Scenarios

Benjamin Heckmann, Marcus Zinn,
Ronald C. Moore, and Christoph Wentzel
University of Applied Sciences Darmstadt
Haardtring 100, 64295, Darmstadt, Germany
benjamin.heckmann@gmx.de

Andrew D. Phippen
University of Plymouth
Drake Circus, PL4 8AA, Plymouth, U.K.

Abstract—Optimisation of resource selection in hybrid cloud data centres depends on the control of resource usage. The primary criterion for this resource selection is economic efficiency. The presented approach considers operational efficiency aspects in service providing and therefore focuses on technical criteria, such as resource load, as well as economic criteria, such as the costs of resource usage. When services are offered at different service levels the approach enables revenue optimisation in cases of excessive load. The concept is prepared to handle heterogeneous IaaS scenarios.

Keywords—Business, Cloud, Efficiency, Services-oriented Architecture, Utility Computing

I. INTRODUCTION

The following concept characterises an approach to optimise the resource selection in data centres. Both runtime and deployment time are considered as point of decision about the usage of resources. Primary criterion for this decision is economic efficiency.

The project was conducted as an applied research in the field of business informatics in close cooperation with a business partner [1]. The developed approach for efficient control on data centre resources in heterogeneous cost scenarios was also implemented as a proof-of-concept [2]. The concept is restricted to the following technical solutions for IT resource offers specified by our business partner.

IaaS: Offering hardware resources located in data centres (e.g., servers, storage, network) based on virtualisation technologies (e.g., VMware, Xen) is defined as Infrastructure as a Service (IaaS) in this concept. Virtualisation enables the separation of hardware resources into smaller fractions, whereby each fraction offers the same virtual hardware interfaces as an actual hardware. In this context the IaaS focus is on server virtualisation. These server fractions are called virtual machines (VM). Hardware resources can be allocated to VMs as demanded, depending on the features of the virtualisation technology used. IaaS thereby describes the basic management layer for data centre operations.

SaaS: Software applications can be deployed based on an IaaS layer. In this context deploying business software in one or several VMs to ease deployment and operation of multiple parallel instances of this software is called Software as a Service (SaaS). Thereby, SaaS describes the basic layer for the consumer interaction.

Hybrid Cloud: In this paper the provisioning of resources or IT services based on the paradigm of IaaS or SaaS is also called cloud-based provision, conforming to the *cloud* definition of the National Institute of Standards and Technology (NIST) [3]. In this paper hybrid clouds are compositions of clouds offering the same type of service while their operation technology may vary. The services analysed in this project are operated as a hybrid cloud hosted in several data centres across the world. A data centre may expose its resources as a single cloud, but more often as the sum of multiple clouds, each representing an individual technical solution grown over time.

II. BACKGROUND

A. Cost Domains

Here it is assumed that in most cases the technical boundary of a cloud also reflects an individual cost domain. This is true when clouds reside in different data centres, even more obvious in different countries. Clouds can also differ in the applied technology for their operations. Distinguishable cost domains can also originate out of significantly different hardware performance, as in scenarios where older and newer hardware are operated simultaneously within the same data centre.

B. Utility Computing Service Life Cycle

Our concept takes major aspects of Heckmann et al. and extends them significantly. The works of Heckmann et al. reflect the characteristics of a service life cycle (business planning, development and operations) in the context of Utility Computing (UC). The business model of UC offers scalable IT-based services metered by usage.

The main contributions aggregated from these results are:

- Technology-independent Provision Model [4]

The developed component architecture describes the minimum necessary functionalities and dependencies in an operations environment for a UC service. This architecture is used when services should be operated as part of a service-oriented architecture (SOA) and hosted on a cloud platform and are incorporated in an UC business plan. Those scenarios (SOA and cloud and UC) are called UC scenarios herein.

- Technology-abstracted Resource and Cost Simulation [5] When services are orchestrated [6] using other services and used in UC scenarios, complex service cascades are formed. These cascades can be complex both architecturally and economically. Both challenges can be addressed with a simulation framework to analyse the interaction between resource allocation and costs, service orchestration, service purchasing costs, and service pricing. A proof-of-concept implementation of such a simulation framework for multi-tier operations environments was implemented.
- Specification Paradigm for Service Quality [7] Within the introduced results a new approach on agreeing on service level for services in UC scenarios is described. This approach offers a specification paradigm for service quality description straight from a usage perspective. In this case service levels are no more defined by technical conditions. They are called *business service level* (BSL) and are specified by describing the quantity and quality of the consumers' behaviour in using a service.

C. Research Objectives

The research objectives are examined from the perspective of a service provider.

We assume a service provider with multiple data centres spread worldwide. The data centres are operated as a hybrid cloud with multiple clouds per data centre hosting SaaS offers for a multiplicity of varying customers. Each service is offered with more than one service level. Each cloud is considered to be its own cost domain. The research objective is to make potential savings accessible between different cost domains. We provide an approach for a technical solution, including a proof-of-concept implementation. This optimisation should be performed at the initial resource allocation during deployment of a service as well as continuously during its operations.

III. RELATED WORK

This research offers an approach to technically converge the quality-related ontologies of service, experience, and business as introduced by Moorsel [8] or Dobson and Sanchez-Macian [9]. In the literature, three focuses on data centre control related approaches can be found: effective data distribution, quality of service (QoS) in networks [10] and reduction of power consumption. The focus on effective data distribution resides in the field of grid computing. Here large amounts of data have to be distributed over several nodes so that parallel calculations on the data slices accelerate the overall processing of the data. In most grid architectures there is an architectural component called *broker* [11]. This broker controls the distribution, processing and result aggregation, sometimes supplemented by billing or marketplace features, like auctions and bidding. Different approaches are known to accelerate processing, for example using resource reservation or considering the problem as a queueing system [12].

In networks, QoS approaches mainly are focused on the network layer. MDCSim [13] instead offers an approach for

a multi-tier data centre simulation, but focuses their outcomes onto a comparison of Infiniband and 10 Gigabit Ethernet network technologies.

The focus on reduction of power consumption centers on server consolidation. Approaches for load prediction for servers in a single data centre are shown by Speitkamp [14] using historical data analysis, Bi [15] using a non-linear optimisation model or based on a limited lookahead control framework by Kusic [16]. Wang introduces an approach to combine server consolidation and dynamic voltage and frequency scaling [17]. An approach for service level management in distributed infrastructures, including QoS translation and support for self-adaptation, is shown by Freitas [18].

Load balancing on the level of data centres within and between client devices is addressed by Peoples [19].

None of these approaches sufficiently covers the relation between resources, services and consumers introduced in Section 5 of this paper.

IV. RESEARCH APPROACH

The following steps were taken to obtain the research objectives of making potential savings accessible between different cost domains for SaaS providers:

- 1) Analysis of the customer-service-resource relation in SaaS provision scenarios in the context of our business partner (see section V).
- 2) Design of a generalised concept to efficiently control data centre resources in heterogeneous cost scenarios based on the previous analysis (see section VI).
- 3) Implementation of the design as a proof-of-concept (see section VII).

V. ANALYSIS OF THE BUSINESS PARTNER CONTEXT

A. Model of the Customer-Service-Resource Relations

The relationships between a SaaS provider and its customers are modelled with a data structure. This data structure is subsequently used as the basis for the optimisation, and must be modified only when the relationships between the provider and the customers change. The provider and the customers are represented by the nodes in a graph; the edges in the graph represent the services provided.

Customers can have one or more contracts with the provider. A contract applies to one or more consumer groups (e.g., branches) within the organisation of the customer. Each consumer group relates to one or more services of the provider. This relation incorporates the link to two service levels and one usage pattern. A Usage Pattern is a quantitative and qualitative description of the service usage behaviour of a consumer group [7].

The price (per unit) and the contract penalty (per unit) are stored attached to the link between the first service level, a service and a consumer group. Accordingly, the price (per unit) is also stored attached to the link affecting the second service level. A penalty for this relation is not necessary, as it reflects the service usage over and above the contracted usage pattern.

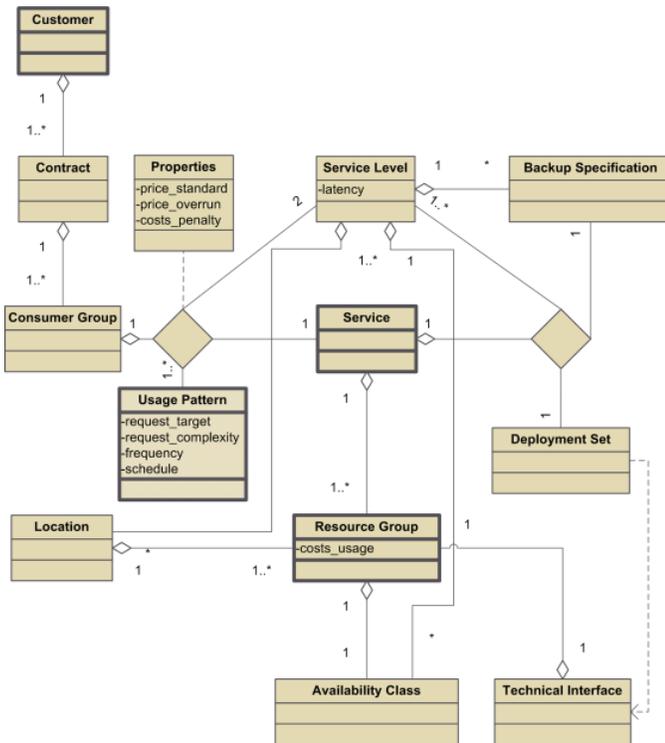


Figure 1. Relations Between Resources, Services and Consumers

Services, one or several, act as connector between provider and customer, more precisely between resource groups on the provider side and consumer groups on the customer side. Also a service relates to a backup specification and a technical deployment set. Such a set contains necessary files and configuration properties for deployment.

Resources are managed in groups. The primary grouping criterion is technical, for example the virtualisation software used. The secondary criterion is the geographical location, for example the hosting data centre. The cost of the resource usage (per unit) is an attribute of a resource. Linked to a resource is the according technical interface for its administration and monitoring (e.g., virtualisation management API). Additionally, an availability class is linked to a resource group. The availability classification enables an abstract categorisation to distinguish between different level of technical availability assurance.

Service level serve as abstract categorisation to differentiate between varying level of service quality. Beside their previously described relations, a service level links to one or more locations, one availability class and one backup specification.

The customer-service-resource relation is elaborated in the data model in Fig. 1.

B. Mediation Conditions

In the research context resource groups are only considered during resource selection when they conform to the required quality properties. Resource selection should respect the technical load of resource groups and customer constraints such

as processing location. Only incoming service requests (e.g., from the consumer towards the service) should be considered.

VI. SOLUTION DESIGN

The required functionality for an efficient control on data centre resources in the analysed context is distributed among two architectural components, named *Service Broker Manager* and *Service Broker Gateway*.

The Service Broker Manager implements the elaborated data model described above and offers interfaces for interaction (e.g., graphical user interface (GUI), application programming interface (API)). Beside the storage of the data model the broker offers a method to match a service request from a certain customer with a suitable resource. The broker continuously analyses the monitoring data from all resource groups and redirects service requests, including service relocation, accordingly.

The matching between a customer's service request and a suitable resource is done in six steps. Preconditions are a given service request and at least two resource groups:

- 1) Service type, service consumer and the service level corresponding to the service request are determined.
Postcondition 1: identifiers for service type, service consumer, and service level are known.
Precondition 2: service request and service type are known.
- 2) Resource demand for the service request is estimated.
Postcondition 2: service request's resource demand is known.
Precondition 3: service request's resource demand, service type, and service level are known.
- 3) Pools of resource groups are selected by available resources and matching service level.
Postcondition 3: two pools of resource groups are known, where each resource group offers enough resources for request processing and one pool complies with the demanded service level and the other does not.
Precondition 4: service request's resource demand, service type, and service consumer are known.
- 4) The estimated revenue per pooled resource group for request processing is calculated.
Postcondition 4: per given resource group the estimated revenue is known.
Precondition 5: service request's resource demand, service type, service consumer, and service level are known.
- 5) Estimated costs for service level violation (latency exception and request failure) are calculated.
Postcondition 5: estimated costs for latency exception and request failure are known.
Precondition 6: two pools of resource groups with sufficient processing resources distinguished by service level compliance, estimated revenue per pooled resource group and estimated costs for latency exception and request failure are known.
- 6) The most efficient opportunity out of the following actions is selected:

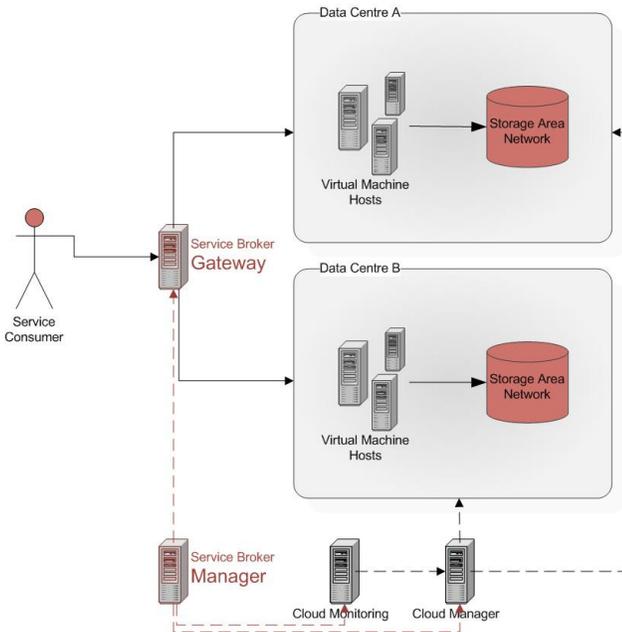


Figure 2. Service Broker Component Architecture

- Request is processed by a service level conforming resource group.
- Request is processed by a non-conforming resource group.
- Request is not processed.

Postcondition 6: action for further request processing determined.

The Service Broker Gateway acts as a load balancer on the network layer. It reroutes service requests to appropriate service instances, including the capability of dynamically shaping the traffic up to the blocking of certain requests. This is especially useful in cases of excessive load. Here requests can be forwarded (or blocked) based on economic efficiency.

This *Service Broker* concept enables resource selection and control on load distribution based on the elaborated relation. An overview on the component architecture is given in Fig. 2.

VII. INITIAL RESULTS

As proof-of-concept the Service Broker Manager including a GUI, API and the request-resource matching method has been implemented. As a scenario for the evaluation of the request-resource matching method a database with four customers, each with two contracts affecting two consumer groups is defined. Five services are available, whereby each consumer group uses two services. As hosting environment two resource groups are provided, hosted in two data centres as varying cost domains. The self-service cloud portal of the business partner uses the Service Broker API to retrieve a suitable resource address during service deployment.

First tests using the self-service portal show the broker's ability to pick the most cost effective resource with enough load reserve. This leads to a significant overall change in

load (and service instance) distribution among the two cost domains. The load distribution shifts in favour of the more cost-effective data centre. Without the broker-enriched self-service portal, the deployment of new service instances took about three weeks for the whole business process to terminate, due to internal measurements of the business partner. Using the broker-enriched portal the deployment time was *reduced to approximately 30 minutes*.

These first outcomes demonstrate the proof-of-concept's ability to efficiently control data centre resources in heterogeneous cost scenarios.

VIII. FURTHER WORK

Feasibility of Business Processes: Our concept creates an opportunity to also associate business process steps with our data model. A similar approach was introduced by Heckmann, but not elaborated to work based on resource load information.

Simulation-based Load Prediction: The Service Broker can be extended based on the simulation framework for Utility Computing elaborated by Heckmann [5]. Instead of retrieving the current load through a service for resource monitoring (referring to step three in Section 6) the broker can use load forecasts.

Utilisation of External Services: From a provider's perspective, at the current stage, the concept only addresses incoming service requests. In addition, the concept could also be extended to represent outgoing service requests to external service providers. This could expand the efficiency of the service provision one step further.

IX. CONCLUSIONS

This paper introduces and evaluates the *Service Broker* concept.

The Service Broker is an approach to optimise the resource selection in data centres. The concept enables the control of resource usage both at runtime and deployment time. In this research context, the primary criterion for resource selection and subsequent request forwarding is economic efficiency. The broker was evolved and evaluated in close cooperation with a business partner. The evaluation of the concept was done through a proof-of-concept implementation presented on CeBIT 2011 as an applied research in the field of business informatics.

The elaborated concept considers technical criteria, such as resource load, as well as economic criteria, such as the costs of resource usage. When services are offered at different service levels the broker enables revenue optimisation in cases of excessive load. Additionally, the concept is independent of the technical solution for resource management (e.g., virtualisation framework) and is prepared to also handle heterogeneous technical scenarios.

REFERENCES

- [1] P. Opper, "T-Systems International GmbH," 2011.
- [2] M. Zinn, "Cebit 2011," 2011.
- [3] P. Mell and T. Grance, "The NIST definition of cloud computing," Jul. 2010, (Access Date: 04/05/2011). [Online]. Available: <http://csrc.nist.gov>

- [4] B. Heckmann, A. D. Phippen, R. C. Moore, and C. Wentzel, "Agreeing on and controlling business service levels in Service-Oriented architectures," *International Transactions on Systems Science and Applications*, vol. Vol. 7, no. No. 3/4, pp. 173–178, Dec. 2011. [Online]. Available: <http://siwn.org.uk/press/sai/itssa0007.htm>
- [5] B. Heckmann, I. Stengel, A. Phippen, and G. Turetschek, "Utility computing simulation," in *ESM'2009 The 2009 European Simulation and Modelling Conference*. Leicester, United Kingdom: EUROSIS-ETI, Oct. 2009, pp. 175–180. [Online]. Available: <http://www.eurosis.org>
- [6] T. Erl, *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, 2007.
- [7] B. Heckmann and A. Phippen, "Quantitative and qualitative description of the consumer to provider relation in the context of utility computing," in *Proceedings of the Eighth International Network Conference (INC 2010)*, Heidelberg, Germany, Jul. 2010, pp. 335–344.
- [8] A. Van Moorsel, "Metrics for the internet age: Quality of experience and quality of business," *5TH PERFORMABILITY WORKSHOP*, 2001. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.3810>
- [9] G. Dobson and A. Sanchez-Macian, "Towards unified QoS/SLA ontologies," in *IEEE Services Computing Workshops, 2006. SCW '06*. IEEE, Sep. 2006, pp. 169–174.
- [10] R. Braden, D. Clark, and S. Shenker, "RFC 1633 - integrated services in the internet architecture: an overview," <http://www.apps.ietf.org/rfc/rfc1633.html>, Jun. 1994. [Online]. Available: <http://www.apps.ietf.org/rfc/rfc1633.html>
- [11] S. Venugopal, R. Buyya, and L. Winton, "A grid service broker for scheduling distributed data-oriented applications on global grids," in *Proceedings of the 2nd workshop on Middleware for grid computing*, ser. MGC '04, 2004, pp. 75–80, ACM ID: 1028506.
- [12] A. Afzal, A. S. McGough, and J. Darlington, "Capacity planning and scheduling in grid computing environments," *Future Generation Computer Systems*, vol. 24, p. 404414, May 2008, ACM ID: 1350010.
- [13] S. Lim, B. Sharma, G. Nam, E. K. Kim, and C. Das, "MDCSim: a multi-tier data center simulation, platform," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 2009, pp. 1–9.
- [14] B. Speitkamp and M. Bichler, "A mathematical programming approach for server consolidation problems in virtualized data centers," *Services Computing, IEEE Transactions on*, vol. 3, no. 4, pp. 266–278, 2010.
- [15] J. Bi, Z. Zhu, R. Tian, and Q. Wang, "Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, 2010, pp. 370–377.
- [16] D. Kusic, J. Kephart, J. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," in *Autonomic Computing, 2008. ICAC '08. International Conference on*, 2008, pp. 3–12.
- [17] Y. Wang and X. Wang, "Power optimization with performance assurance for multi-tier applications in virtualized data centers," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, 2010, pp. 512–519.
- [18] A. L. Freitas, N. Parlavantzas, and J. Pazat, "A QoS assurance framework for distributed infrastructures," in *Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond*, ser. MONA '10, 2010, p. 18, ACM ID: 1929567.
- [19] C. Peoples, G. Parr, and S. McClean, "Energy-aware data centre management," in *Communications (NCC), 2011 National Conference on*, 2011, pp. 1–5.

Automated Reuse of Software Reuse Activities in an Industrial Environment – Case Study Results

Marcus Zinn
University of Plymouth
Plymouth, UK
marcus.zinn@plymouth.ac.uk

Klaus-Peter Fischer-Hellmann
University of Applied Science
Darmstadt, Darmstadt, Germany
k.p.fischer-hellmann@digamma.de

Ronald Schoop
Schneider Electric Automation
Seligenstadt, Germany
ronald.schoop@schneider-electric.com

Abstract - The reuse of prefabricated software units, such as classes, components and services is one of the central topics of software engineering and requires lot of knowledge and experience. Instead of focusing on the knowledge management processes and a resulting lifelong learning process of individuals, this paper shows an experimental study based on an approach of automation of knowledge based reuse activities. This is done by employing a unified view of software construction activities and software units used by these activities in an industrial environment. It concludes that software engineers of different industrial business units and knowledge levels can be supported by performing different software construction activities with only one approach, the result of which avoids a long learning process for software engineers.

Keywords-Automated software unit reuse; software reuse activities; industrial environment; case study.

I. INTRODUCTION

The reuse of software units (like classes, components, or services) requires professional knowledge or expertise. A software unit is a technical unit, and can, therefore, be defined like a software component in the context of this paper:

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition of third parties”. [1]

Typically, software engineers have to acquire this knowledge. In industrial environments, the knowledge depends not only on the technical properties of a software unit but also on the technical environment, technical topic (e.g., embedded devices) and the business topics (e.g., Automation, Datacenters, Mines & Minerals). Today knowledge about software units in a reuse context is a broad field. As adequate description of knowledge in the context of this paper following definition is used:

“... the capability of a man (or an intelligent machine) to use information for problem-solving” [2]

Starting from this point of view a software engineer has to have different kinds of information to perform software reuse, as for example: (1) Information about technical properties such as programming language, necessary

technical environment, and dependencies. A software engineer has to know this information. [3]

(2) Information about interfaces and business context. A software unit solves at least one problem. Typically, the interfaces and provided data types are related to this fact. By handling such a software unit a software engineer have to be aware about this information. [3] (3) Information about the reusable artefact. Today a reusable software unit is more than a single binary file. Related information like test cases, documentation, and versioning are also reusable and sometimes implied. A software engineer has to deal with this related information. [4] (4) Information about related reuse concepts and processes. Software unit reuse is not undertaken if a software engineer decides to perform reuse. Many activities such as search, validation, integration, transformation, and testing are part of a reuse process. A software engineer must be aware of the existence of different reuse processes and technologies.

As a result of these perspectives, reusing a software unit may define as the use of different information about a software unit and a given environment to perform a number of reuse activities. The result is a reused software unit in a software development project.

Based on the high number of different technologies, business context, reuse artefact information and possible reuse concepts or technologies, the amount of necessary knowledge is high. This results in a problem for software engineers. Each time they wish to reuse a software unit they have to know about the relevant activities, and the related knowledge and information. If this knowledge is missing the reuse cannot be carried out successfully.

A solution may be the automation of reuse activities. As shown in the automation industry, this requires the development of supporting systems that are able to perform activities for a user. By automating software reuse activities, software engineers are able to perform these activities without having acquired the complete knowledge. Such an approach would reduce the problem of missing knowledge and was discussed in the past [5] and [6] under the name of “Service based Software Construction Process (SSCP)”. However, the experimental proof of this concept is still missing.

This paper describes the setup and the results of the first

phase of an experiment validating the concept of SSCP, which is described by the following hypothesis:

“Automated Software reuse activities will reduce the problem of missing knowledge in software unit reuse”

This work forms part of the research on a Service-based Software Construction Process (SSCP) incorporating the field of Software Unit Reuse. The goal of this research is to identify a semantic model (about finding, adapting, integrating, and deploying of software units) combined with service technology that supports software engineers by performing software reuse (finding, adapting, integrating, and deploying) without having all needed information. The paper contributes to the research area by demonstrating the positive effect of automated software reuse activities, based on software reuse knowledge on the problem of missing knowledge in software unit reuse, in a real world experiment.

After the problem statement in the next section, the Section 3 shows the focused solution of this paper. This is used in Section 4 to describe the experiment setup and execution. Section 5 discusses the experiment results followed by the conclusion section (Section 6)

II. THE PROBLEM OF REUSE IN MULTIPLE INDUSTRIAL SOFTWARE DEVELOPMENT TEAMS

Typical aims of software reuse are to reduce costs and time in development projects [6]. These are two reasons why reuse of software units is an important part of software development in industrial areas [5]. However, the use of reuse in industrial projects does not guarantee a successful project, a fact, which has been demonstrated by several project studies in the past [6]. Typical problems are [6], e.g. : Misconceptions (reuse == repository, reuse == OO), No non-reuse specific processes modified, No reuse specific processes installed, No training/awareness actions, Reusable assets produced but then not used, Multi contractor / Multi company project, and No production of assets.

The last problem ‘No production of assets’ differs from the others. This problem deals with the fact that a software unit must be developed in order to be reusable [7]. If this is not the case, the amount of required resources is decreased by reuse [6][7]. Based on this statement, the effort to reuse increases after the creation of a software unit and should remain at the same value continuously for each reuse.

An internal study conducted by Schneider Electric [8] indicates a complex but interesting picture. A set of around 50 software units (so-called ‘bricks’ in industry area) has been created and widely reused. The average reuse number is between 9 and 10. The distribution of reuse for different bricks is shown in Figure 1. It starts with a minimum of 3 reuses (the point where typically a cost breakeven would start compared to a non reuse approach) and spans up to 36 reuses.

Relating to the above mentioned fact ‘No production of

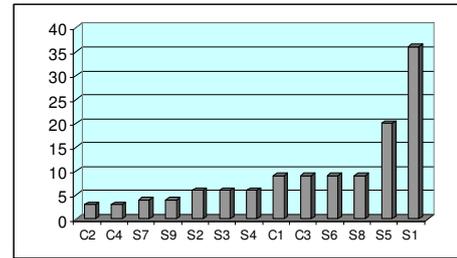


Figure 1. Distribution of reusable bricks [8]

assets’ the study of Schneider Electric shows a dilemma of reuse in industrial environments. A reusable software unit creates additional reuse effort during the creation phase and in reuse phases of each development team which reuses this unit.

Creation Phase Dilemma (CPD): The creation of reusable software includes different phases, which focus the reusability. Typical examples are given by Software Product Line approaches [7]: (1) Generalisation – The interfaces and functions of a software unit must be generalised to increase the reuse probability. (2) Integration – The software unit must be built in a way that it can be integrate in the development projects of other teams. (3) Support – The software unit must be ‘equipped’ with additional reuse artefacts, which support the reuse, e.g., reuses documentation. Additionally, such a unit have to be installed in a system, which provides access to it.

All of these steps require knowledge from an expert user.

Reuse Phase Dilemma (RPD): Each development team has now different challenges for reusing such a software unit. Typically, each team has to find and download the software unit [8]. In the next steps, they have to understand and integrate the unit into their development projects [7]. Sometimes software units must be adapted (transformed) for that specific application [9]. Figure 2 shows also the typical support and maintenance effort, which is created during these steps. This effort is the results from the problem that the development teams have not enough knowledge to perform the described reuse steps.

CPD and RPD are typical theoretical examples discus-

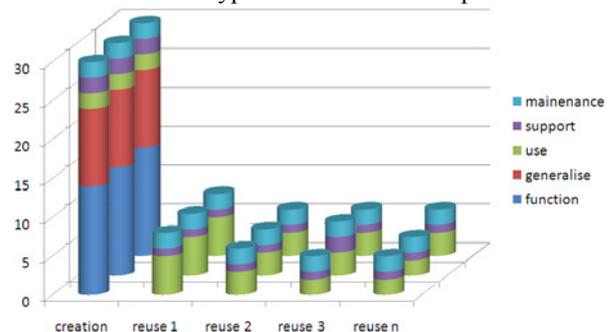


Figure 2. Support and maintenance effort [8]

sions of problems. The reality creates two additional dilemmas in the context of CPD and RPD. (1) **Creator dilemma (CD)**: The creation team is not available for support at the time of reuse (people are loaded with other projects or change team or organization) (2) **Reuser dilemma (RD)**: The reuse teams are different for each development projects, and therefore the exchange of a 'learning curve' between the teams is not possible.

Figure 2 shows that each development team has nearly the same problems and need nearly the same amount of resources. The challenge of reuse based software development in industrial areas is to reduce the sketched dilemmas. The purpose of this study is to show that reuse of a single software unit in multiple teams does not need this amount of resource on both sites: creator and reuser.

III. CONTEMPORARY SOLUTIONS

Nowadays, there are different approaches for the above-mentioned problems. The first approach is so called information systems, which, in general, enable the storage of information. This enables a user to search for information. However, such systems are not designed specifically to address the issue of transformation, but treat the subject of information generally [10]. Generally, such systems can be used to save information about an area of knowledge in textual form, but without the context of knowledge (see [10]). Each software construction activity may be described in this form and may be stored in an information system. The user is now faced with the problem of obtaining this information and interpreting it correctly in order to perform a successful transformation. Usually, information systems are not intended to apply their stored information automatically. But they can be extended for this task [10].

Despite this lack of functionality, information systems comprise a part of this article's advocated solution. Extensions of information systems are so-called Knowledge Base System (KBS) [10]. Such systems are defined as:

"... a method that simplifies the process of sharing, distributing, creating, capturing, and understanding a company's knowledge." [11]

Knowledge systems are not fundamentally designed for the subject of software construction activities. Furthermore, the authors of this article believe knowledge systems are missing a fundamental property: the automated application of stored knowledge for specific tasks. However, there is a lack of systems that have asserted themselves and are not focused on the typical software construction activities of software units. The latter property 'application of knowledge', is also a part of the solution discussed in this article. Basically, the knowledge that is necessary for perform an reuse activity can be stored in knowledge

systems.

The area of software development has currently seen a number of interesting approaches dealing with specific subjects of a software reuse activity. Most of them are specific for one reuse activity type. For example there are two existing approaches for the activity of software unit transformation which are of interest: Model transformation [12] and generative programming [13]. Both approaches have existed for some time and form the basis for approaches that are being used today. Both support software engineers in generating reusable transformation models or rules. However, additional knowledge is necessary to make use of both approaches. This can be found in other activity areas like deployment [14] and Integration [15]. For the integration of software units into Integrated Development Environments (IDE) very specialised solutions exists e.g., Packaging for Eclipse or Packaging for Visual Studio. But these products are too specialised and require different kinds of specialised knowledge from the user.

The above mentioned solutions have one common problem. They assume a high learning curve. But learning how to implement every existing technology or solution for knowledge based problems cost too much time. It is necessary to identify a solution, which is able to support software engineers by performing software reuse activities without a lifelong learning process.

IV. FOCUSED SCENARIO

The basic idea of the targeted solution is that an expert applies knowledge (knowledge extraction) about the software reuse activity of a specific software unit to a system, which is able to perform the activity automatically with a minimum of human interaction based on knowledge. Users who do not have the necessary knowledge are now able to perform this activity (knowledge injection). A learning process for this specific activity and the specific software unit is not necessary. Figure 3 shows this scenario.

The idea was presented in previous [5][6] where its advantage was demonstrated for two reuse activity examples: Integration of software units into integrated development environments (IDEs) [15], and deployment of software units into embedded devices [14].

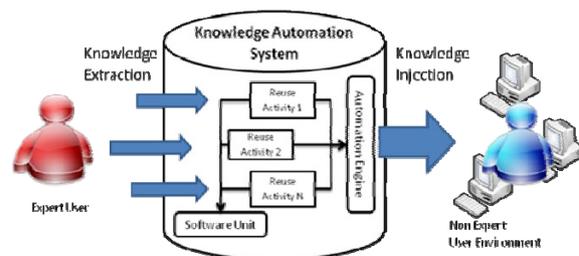


Figure 3. Concept of the focused solution

For the experiment demonstrated in this publication the software construction activities ‘Integration’ and ‘Transformation’ were chosen.

V. THE EXPERIMENTAL SETUP

A. Technical structure and infrastructure

The following section utilises this theoretical description to create the basis for the activities of integration, transformation and deployment of real models. The experiment is performed by software engineers using these models. These engineers try to perform different transformation and integration software construction activities with and without the support of the proposed solution. The second step comprises a description of the design and implementation of the experiment. These descriptions are intended for the replication of the experiment, and to ensure the sustainability of the experiment for the study’s results. The setup of the experiment is divided into three distinct areas:

- (1) Description of the environment,
- (2) Description of the technical structure of the experiment, the necessary elements, and
- (3) Description of the measurement process.

Description of the environment: The experiment was conducted at a German location of the company Schneider Electric (Address Steinheimer Strasse 116, 63500 Seligenstadt, Germany). The company has participated by means of employees at this site and from other international locations using the company intranet. The experiment itself was conducted in normal offices, which provide a connection to this intranet source.

Description of the technical structure of the experiment, the necessary elements: The technical design of the experiment is mainly a hardware and software infrastructure. Figure 4 shows this structure in the environment of the Schneider Electric intranet. Six important elements are involved. The first element is the intranet (1), which is used to connect the various other elements of the technical structure. The second elements (2) are the connected databases, including the software units and complete information about the re-use activities. Four databases are important for the experiment:

- 1) SOA4D: This is an open source repository software unit with further information about device profiles, including four web services. This repository is based on the Forge technology and offers a web interface.
- 2) Prometheus SQL: this is a specially developed Repository. It belongs to the approach and uses a

Microsoft SQL database and Microsoft SQL database interface.

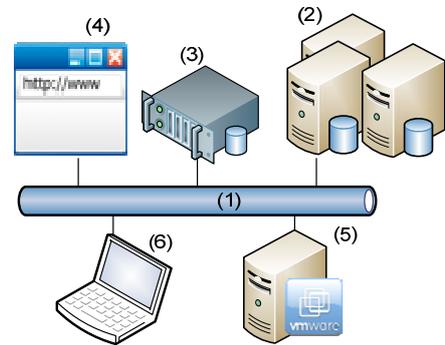


Figure 4. Experimental environment and setup

- 3) DDXML repos: This is a Schneider Electric internal repository that contains XML elements describing embedded devices. Communication with this repository will be achieved via a Web service.
- 4) Brick Catalogue: This is, Schneider Electric internal repository used by all Schneider Electric business units containing software unit.

The third element (3) in the experiment’s design is the Prometheus Server. This comprises the core of the technical structure. The server maintains information about software units and software construction activities in the connected databases and makes this information available to the user. Finally, the Prometheus Server performs requested activities and presents the available results to users. The fourth element (4) is a website through, which the user can communicate with the Prometheus Server. The website runs on a further server and contains a web application giving the user the ability to query information from the server or to perform reuse activities on the server. This web application is named ‘Ecostruxure repository’ and for this experiment the 4.1 version was used. The basic technology of the Web application is Microsoft Silverlight version 4.0. The website used the endpoint ‘/RepositorySearch.html’ and was available within the company’s intranet. The fifth element (5) of the structure is a VM-Ware server. This server is used to fulfil the experiment’s required operating system environment and runs as a virtual machine (VM) to make this available. For the connection to the server VM-Ware Workstation software with version 8.0 was installed on a laptop (6). These elements are common office laptops used within the company Schneider Electric. The laptops were used with the VM-Ware Workstation software with version 8.0. In addition to the computer network environment, there is the possibility to use telephone, internet, voice, conversation, or literature. This is also reflected in the working environment within the company’s sites.

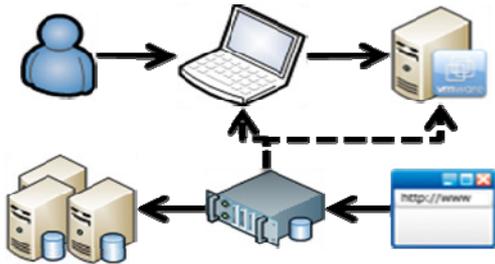


Figure 5. - Basic experiment scenario

Figure 4 shows the scenario based on the experimental setup. Users are able to view the test environment (operating system, the virtual machine) from element (5) (VM-Ware server) by using element (6) (office laptop). Within this test environment, all necessary software applications are found by means searching for information on the Internet, or performing activities on the intranet, as well as various means of communication usually employed by Schneider Electric (FTP, Skype, TELNET). Furthermore, users can now click element (4) (the website) to access and use the Web application, which allows communication with element (3) (Prometheus server). The Prometheus server communicates with the databases that are marked as element (2). Also the Prometheus server interacts with the elements (5) (VM-Ware server) by using element (6) (office laptop) (see Section III). Figure 5 shows this interaction scenario.

Figure 6 shows the different measurement variants in the experimental setup. This can be accomplished by three different (technical) variants. The first is the purely visual recognition of the user's actions and does not require any technical measure (called 'Observer'). The second is to record the user's interactions with the virtual machine as video recording (called 'Recording'). For this, the installed VM Ware Workstation software with version 8.0 is used, which already includes the feature of video recording. The third variant is to log the information (called 'Logging'). This is done in three elements of the experiment's design:

- Create the user data in virtual machines. These data can be analysed after the experiment.
- The Prometheus Server attracts all incoming server requests and performed activities. This information can

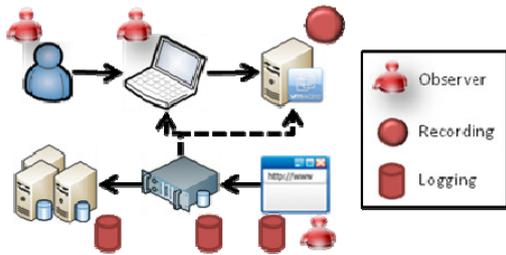


Figure 6. Overview measurement utilities

also be queried after the end of the experiment and used for analysis.

- The data and information are generated and stored in the databases through the interaction of the user.

Description of the technical setup for the measurement and the measurement process itself:

(1) Experimental groups and scenarios: There are a total of three experimental groups: the first group (1) consists of experts for one particular software unit. These individuals receive expert status either because they have created this software unit or are well acquainted with its use. The selection of experts is performed via the Internet from public data of Schneider Electric software units. These data also contain the contact person responsible for this software unit. These people are also asked directly whether they have created the software unit and / or have used it frequently. Altogether the study requires 5 experts. The second experimental group (2) consists of 10 software engineers with the following characteristics: first, the people should actively participate in the software development of a project at the time the experiment takes place. On the other hand, it is important that these people do not have the same expert status as the previously selected 5. The last criterion is that these people are neither expert in the software unit nor in the technology standard development platform for this unit.

The third group (3) is similar to the second experimental group and consist of 10 participants. Therefore, the same rules used for selection of the second experimental group apply.

Note: In this the next phase of the experiment, the total number of participants will be increased up to 30 per group.

Procedure: In principle, there are 3 different experimental groups required to perform seven scenarios. Table 1 shows the different scenarios related to the different groups.

TABLE I. SCENARIOS OF THE EXPERIMENT

Scenario	Description / (GroupID)
(1) Observation of experts	The experts from experimental group (1) performs transformation and / or integration activities (manually). / (1)
(2) Collection of software units and activities	Collection of software units and activities: In this scenario, each of the selected experts from experimental group (1) insert the knowledge about the unit and the specific transformation and, or integration activity into the Prometheus Server. / (1)
(3) Prometheus Validation	The experts perform the same activities as in scenario (1) but now with Prometheus Server support. The expert validates the results. / (1)
(4) Reuse activities with Prometheus	Participants from the group (2) are asked to take over one transformation and integration task. They have to use the Prometheus Server for this purpose. / (2)
(5) Reuse activities without Prometheus	In this scenario, the people placed in the experimental group (3) are asked to take over a transformation or integration task. Activities are repeated so they correspond to those of the experts from scenario (1), The Prometheus Server is not

	used / (3)
(6)/(7) Validation of the results	Validation of the results: This scenario will test the results of the experimental group (2) and (3) by the experts for the respective software unit from experimental group (1) and (2). / (1)

(2) Measurement

In the following section, the methodology of measurement of the experiment will be explained. This includes the definition of the measurable variables and the process of measuring.

Definition of variables: The results of the measurement procedures are stored in the form of variables. In addition, each variable is assigned a unique name within the experiment. In this section, all variables are named and briefly presented. Table 2 shows the different measurable variables in the different scenarios.

TABLE II. OVERVIEW OF VARIABLES

Sc. ID / ID	Name: Description
(1,3,4,5)/(A)	ActivityDuration: How long does it take an expert/user to perform an activity? This variable contains a value that expresses how long the expert takes for the preservation of the task.
(1,3,4,5)/(B)	TaskAnalysisActivityDuration: How long did it take the expert/user to analyse the task initials? This variable describes the time between being presented with the task and the start of work on the computer.
(1,3,4,5)/(C)	TaskActivityDuration: How much time does expert/user spend working on the computer in order to perform the activity? This variable describes the time between the start and completion of work on the computer activity.
(1,3,4,5)/(D)	ActivityCarriedOutSuccessfully: Has the expert/user completed the activity successfully? This variable represents whether an activity was successful or not.
(1,3,4,5)/(E)	UseKnowledgeSources: What kind of knowledge sources did the expert/user use to perform the activity? This variable describes the sources consulted to perform the activity such as the Google phone or contacting another expert for information.
(1,3,4,5)/(F)	MadeSubTasks: What sub tasks did the expert undertake in order to perform an activity?
(2)/(G)	EnterUnitDuration: How long does it take the user to enter all necessary information about a software unit into the Prometheus system? This variable contains a value of the expert testimony of how much time was needed from commencing work on the computer to enter the information of its software unit.
(2)/(H)	EnterActivityDuration: How long does the expert take to enter an activity for a software unit in the Prometheus system? This variable contains a value of the experts' statement of how long since commencing work on the computer it took to input the specific activity of entering the activities information.
(2)/(I)	TotalInputDuration: How long does it take the expert to enter all the information into the Prometheus system? This variable contains a value of expert testimony on how long the whole process of entering all their data took.
(2)/(J)	SuccessfulEntry: Could the expert enter all the important information? This variable tells us whether an expert could enter all the information about a software module and complete activities in the system.
(2)/(K)	MadeSubTasks: What sub tasks did the expert undertake in order to perform an activity?

(3,6,7)/(L)	ResultsValid: Is the result of an activity conducted by Prometheus or without equivalent to the result of the same activity conducted by an expert? This variable indicates whether the expert considers the result of activities performed by Prometheus or without it as good as the result, which was achieved through manual execution of the same activity.
-------------	---

Measurement Execution Process: In Figure 6, three variants of measurement used to measure the variables were introduced. The following section shows, which of these techniques are used for the different variables.

In Scenarios (1), (3), (4), and (5), seven measurements are raised per cycle: (A) The variable 'ActivityDuration' is measured by the observer (measurement variant 1). Here, the observer measures from the time, which he assigns the task to the expert/user up to the time the expert says the task was completed. The time is recorded in whole minutes. (B) The variable 'TaskAnalysisActivityDuration' is determined by the interaction of measurement variant (1) and (2). Here, the observer notes the time at which the task is assigned to the expert/user (see variable 'ActivityDuration'). The end of this phase can be measured at the time when the expert commences an activity on the virtual machine. The time is recorded in whole minutes. (C) The variable activity of 'TaskActivityDuration' determines the interaction of the measurement variants (2) and (1). The point in time at which the activity is started on the virtual machine is measured. The endpoint is the time the expert/user tells the observer that the task was completed. The time is recorded in whole minutes. (E) The variable 'UseKnowledgeSources' is determined by the measurement variants (1) and (2). The observer notes all information coming from the expert's behaviour that cannot be measured by measurement variant (2). The type of measurement (2) also used to analyse, which sources of information accessed through the use of the virtual machine. Typically such sources can be classified by using source names and the type of resource, e.g., (1) co-worker, telephone, and (2) website, Google (Web browser). (D) The variable 'ActivityCarriedOutSuccessfully' is measured by measurement variant (1). The expert/user is asked after the completion of the activity if he has done this successfully. The variable can only be set to yes or no. (F) The variable 'MadeSubTasks' is determined by the measurement variants (1) and (2). Here, the observer notes the progress of the entire task. This can be done based on the recording of the activities in the virtual machine itself, which is operated by the observer both on the external (outside the virtual machine) and internal (within the virtual machine) view. The observer here notes, which activities were measurable, including their start and end time, e.g., starts 10:41 expert uses web browser.

In scenario (2), five measurements are made: (G) The variable input 'EnterUnitDuration' determines the measurement variants (1) and (3). The website (see Figure

4) logs every activity of the user. Accordingly, the entry of the website is the start time and represents the initial value used for the measurement. To avoid error, the observer compares measured time with the automatically measured time. The end time is determined by the expert's signal indicating that he/she has to finish the task. The observer notes down this time. Time is measured in whole minutes. (H) The variable 'EnterActivityduration' is measured by the measurement variant (3) on the Prometheus Server (see Figure 4) and the website (see Figure 4). The server and the website recognize the time of a user's request. Each measurement contains the time and the names of tasks, e.g., 10:00:00 user creates a new software unit. (I) The variable 'EnterActivityDuration' is measured by the measurement variant (1). The observer records the start time point at which he/she hands over the task to the experts. The end time is determined by the expert's signal that he/she has finished the task. The observers take note of this point in time. Time is measured in whole minutes. (J) The variable 'SuccessfulEntry' is measured with the measured variants (1) and (3). Firstly, the expert must inform the observer that he/she was able to enter all information into the system. Secondly, the Prometheus server writes all values into the database. The variable can only be set to yes or no. (K) The variable 'MadeSubTasks' is measured in the same way than in Scenario (1,3,4,5)/(F).

In scenarios (4), (6), and (7) one measurement is made: (L) The variable 'ResultsValid' is captured by the measurement variant (1). The expert examined the results of the performed activity from the scenarios (3), (4), and (5) with the same activity carried out in scenario (1). It tells the observer whether the result has the same value and is usable. The variable can only be set to 'yes' or 'no'.

Definition of Software units and reuse activities: The different scenarios 1-7 are performed in this experiment with the software units shown in Table 3.

TABLE III. USED SOFTWARE UNITS

Name / ID	Description	Tec/ Unit Type / Repository	Integration effort / Transformation effort
DPWS / SU1	Enable devices for WS* profiles	Java / Component / SOA4D	Advanced into Eclipse/Advanced using IKVM
DPWS / SU2	Enable devices for WS* profiles	C++ / Component / SOA4D	Advanced into Visual Studio / None
CWS / SU3	Webservice for data exchange of business units	Soap-C# / Webservice / Prometheus	Normal into Visual Studio / Advanced using SVCUtil
CWS / SU4	Webservice for data exchange of BUs	Java-Android / Class / Prometheus	Advanced into Eclipse / Advanced using Java2SOAP
Code Signing / SU5	Webservice for Code signing	Soap-C# / Webservice / Brick Repos.	Normal into Visual Studio / Normal using SVCUtil

Table 3 shows that five integration and four transformation activities are connected with the five software units. The integration activities typically focus integration of software units on the most common IDEs (Visual Studio and Eclipse). The transformation activities include the transformation of software units on three different transformation tools (IKVM [16], SVCUtil [17] and WSDL2Soap [18])

VI. EXPERIMENT RESULT DISCUSSION

A. Experiment Results

The experiment's results were collected in the way described in the previous section. The next step is to discuss these results. First of all, the result of one software unit with a transformation activity will be discussed in more detail. After this analysis, the results of all software units will be summarised and compared. For this purpose, two perspectives were used for analysing the summarised results: Comparing different groups from the perspectives of (1) activity execution and (2) use of knowledge.

1) Detailed result example

One of the measured software unit is the 'Device Profile for WebServices' Java stack, which enables Java based embedded devices to handle mutable WS* Protocols like Webservice discovery. The transformation task for this software unit was to use IKVM transformation tool to transform the complete DPWS Java Stack into a C# Stack. This task requires knowledge about the DPWS Java Stack (especially the references of the 20 different JAR Files), the .NET Platform and experience in using IKVM. This scenario was taken from a real development scenario of Schneider Electric in the European research project for industrial automation SOCRADES [19].

Expert scenarios (1-3): Scenario 1: In the first scenario, the Expert was measured by performing this task manually. The main result is that the experts needs 14:23 min.. In Scenario 2 it was measured how long the expert needs to insert the software unit and the transformation activity. The initial creation of the software unit into Prometheus needs 12:06 min. and the transformation needs 38:03 min.. In Scenario 3, the expert was observed by using the Prometheus Server to perform this task. He needs 2:04 min. to perform the task and received a 2:56 min. training into the system (this training will only be necessary once per expert). The expert validated the result as a correct transformation.

Non-expert scenarios (4-5): In Scenario 4, five non-experienced software engineers of the industrial areas of Building, Power and Industry (Automation) did the task without support of the Prometheus Server.

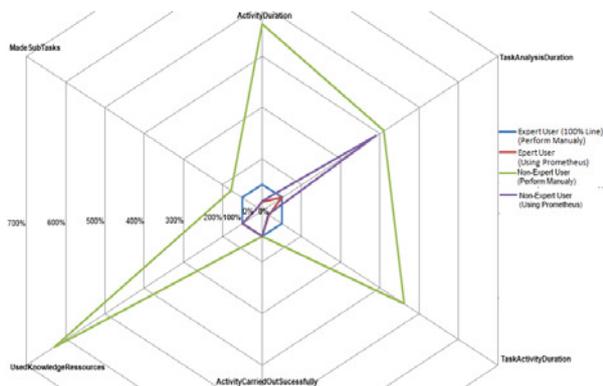


Figure 7. Results of the different groups for DPWS transformation activities

The different participants need 42 min., 90 min., 77 min., 69 min., and 104 min. (rounded off). Thus, the average time was 76 min. (rounded off). The expert validates all final results as valid. In Scenario 5 the participants of group (3) use Prometheus to perform the task. The measured introduction task performing times (in minutes) were (3:03/2:23), (2:56/2:10), (2:33/1:59), (2:45/2:22), and (2:43/2:23). The average time was (2:48/2:18). The expert validates the results as correct results. Figure 7 summaries the results. The validation in Scenario 6 and 7 are not shown in Figure 7 because of all results were valid. Additionally to the measured time the kind of used knowledge resources were measured. Only online websites, downloaded documentation, and the expert were used as knowledge resource. The expert in scenario 1 uses only one knowledge resource (an older development project) 4 times. By adding the necessary information into the Prometheus system of Scenario 2 the expert only uses one knowledge resource (the introduction). In Scenario 3, the experts need only the introduction to perform the activity. The non-expert group (2) of scenario 4 needs multiple resources multiple times. Figure 8 shows the used number of knowledge resources in each scenario (average values).

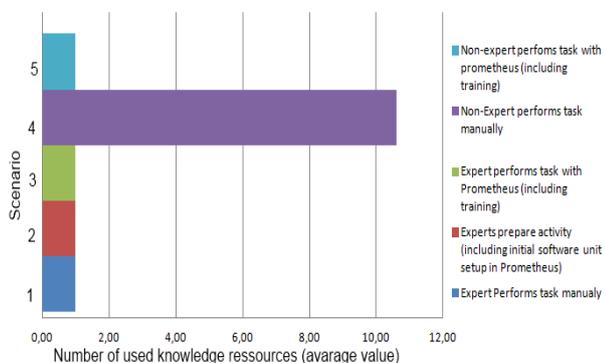


Figure 8. Overview of number of used knowledge resources

The non-expert group (3) of scenario 5 needs only one knowledge resource (the introduction).

2) Comparing of different groups from the perspective of activity execution

Figure 9 and Figure 10 show the results of the three groups in transformation and integration activities measured in the Scenarios 1, 3, 4, and 5. The different results of the software units are summarised by using this type of view. In the context of transformation, Figure 9 demonstrates a clear separation of the different groups. Starting with the Expert Users without Prometheus support (Expert, Scenario 1) as the 100% comparison line, the

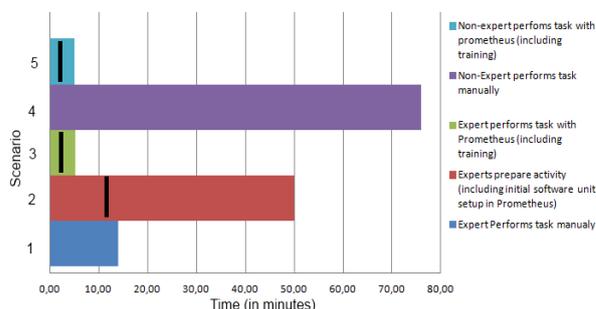


Figure 9. Results of the different groups for transformation activities (5 software units)

measured values of the second group (User with Prometheus support – User (P)) are significantly decreased. This fact is mentioned especially in the variable ‘ActivityDuration’ (1). On the other hand, the Variable ‘TaskAnalysis-ActivityDuration’ (2) is much closer to the comparison line. As a result, Prometheus Users are able to perform a specific activity much faster than an expert user or a Non-Expert user. In comparing the two variables of the comparison line with user (without Prometheus support User) Figure 9 shows a further significant difference. Both variables of the user are decreased. The normal user needed much more time to fulfill the given tasks. But this difference changes by analyzing the results of users (with Prometheus support). Compared to the expert with Prometheus support this group has no significant differences, but compared to the expert group without Prometheus support the measured values decrease significantly. In Figure 9, the two lines of Prometheus supported users are more or less congruent.

As a result of this consideration, it is clear that the Prometheus approach creates a positive effect for Non-Expert User and even for expert users.

Figure 10 shows the measured values for the integration activity. The first interesting point is the general comparison to the results shown in Figure 9. Both pictures show nearly the same result, but the positive characteristics are not so distinct. Only the users (without Prometheus support) performing the integration activity need less time (compared to the 100% comparison line) then the same group was

performing the transformation activity. That both results a nearly the same indicates that the used approach supports software engineers by performing these kinds of activities.

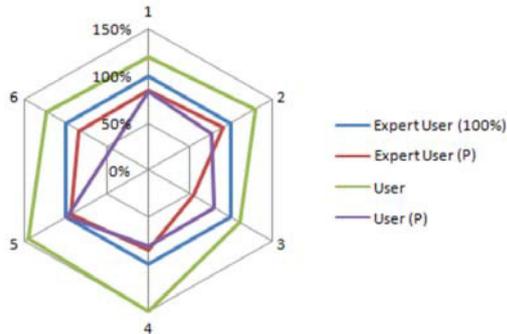


Figure 10. Results of the different groups for integration activities

All users (experts and non-expert user) were able to perform the given activities correctly and needed less time than the expert user (without Prometheus support).

3) Comparing of different groups from the perspective of the use of knowledge

In Figure 9 and Figure 10, it is also mention that most of the expert users (80%) (without Prometheus support) did not use a measurable knowledge base. The other 20% used exactly one knowledge base. All experts or users (with Prometheus support) only used the knowledge base that was the documentation of the Prometheus system. The users (without Prometheus support) performing both the transformation and the integration activity used much more knowledge bases. The most used knowledge base was the internet.

B. Impacts on industrial reuse

In applying the aforementioned approach to industrial environments faced with both creator and reuse phase dilemmas, and therefore no knowledge transfer, leads to the following effect, shown in Figure 11: The effort for the creation team increases by adding the software unit information into the Prometheus system. The theoretical very useful but missing support effort is mostly replaced by the effort for this ‘knowledge injection’.

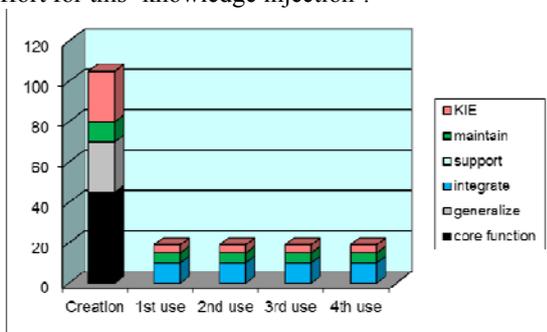


Figure 11. Effects on MTwKIE

The major effect is visible at the reuse site. Even without or just less support, the effort for reuse for single users or team is significantly reduced. In the case of this experiment the reduction of the measured variable are $\sim 38,5\%$ in the transformation activity case compared to the expert user (perform manually) (see Figure 9), $\sim 73,21\%$ in the transformation activity case compared to the non-expert user (perform manually) (see Figure 9), $\sim 38,5\%$ in the integration activity case compared to the expert user (perform manually) (see Figure 10), and $\sim 73,21\%$ in the integration activity case compared to the non-expert user (perform manually) (see Figure 10). This is mainly based on the fact, that expert and non-expert Prometheus users do not spend much time in searching a software unit and preparation/execute a specific reuse task. The same positive effect is expected in the reuse of a software unit multiple teams of different business units. The approach detailed in this paper has two positive effects. First of all, the solution is sustainable for all teams as it is available to all once it has been stored in the system. This is shown by using different participants from different business units. As consequents, all teams will obtain the same result and the same effects described in Figure 9 and 10. Therefore, the way of reuse planned in the creation phase is more sufficient. The second positive effect is the adaptation towards knowledge created in the “reuse” steps. If a team recognizes an alternative way to perform the reuse activities it is able to store this knowledge in the system. This requires training for the use of the Prometheus system, but other teams are now able to decide, which kind of transformation rule they want to use. (Reuser is Creator) Figure 11 shows both positive effects.

VII. CONCLUSION AND FUTURE WORK

The reuse of a software unit consists of different reuse activities. To perform such activities knowledge is required. Especially in an industrial environment this constitutes problem for a single team and in different teams of different business units. This paper shows the structure and result of an experiment aiming to demonstrate that it is possible to automate chosen reuse activities so that less experienced users are able to perform the activities. By comparing a group of software unit experts, a group of less experienced users within a normal development environment, and a group of less experienced users with the support of the focused automation approach following results are obtained: (1) It is possible to automate reuse activities. Expert users store their knowledge into a system, which is then able to perform the activity (knowledge extraction). (2) Less experienced users who are normally unable to perform such activities are now able to do this. (knowledge injection) (3) Analysing of the results demonstrated that this approach has positive effects for reuse of software units in industrial

environments. (4) With automated support, a single team can decrease their reuse costs from the first time of reuse and thereby make it sustainable. Users utilizing the new approach are able to perform an activity faster than the software unit expert because the system provides the complete environment for the activity based on the expert users' knowledge. (5) By reusing the expert's knowledge, the variations are minimized. All teams use the same activity based on the same knowledge. (6) New automated activities are sustainable because the activity will be changed or a new one is stored in the system, therefore it can be used in each new reuse step of each team. Next to the positive effects, this paper's experiment is limited to two software reuse activities: Transformation and Integration. These activities were chosen because they require different amount of knowledge about tools, environment, and software units. But there also other reuse activities like test, validation, and deployment. Especially for deployment, for example on embedded devices, knowledge is required, but not all activities may be automated completely. The next step is the phase two of the experiment. The number of software units is raised to 10 and the number of inexperienced software engineers in the groups 2 and 3 is increased up.

Next to the fact that the results have to be confirmed by repeating the experiment with new software units and other software engineer the process has to be proofed by other companies. For that purpose the process of the experiment has to be formulated in a formal way. Additionally the following aspects are interesting for the future.

Horizontal extension of the research field: The concept presented in this work was demonstrated by using the example of integration and transformation. But, much more than the activities made use of in this experiment still exist in the area of software unit reuse. First, standard activities exist such as testing and validation of interfaces. These activities usually have a high degree of automation. However, these approaches are lacking in one approach, which is used to represent knowledge uniformly and then re-applied to the different existing automation systems. The scientific task is thus to consider whether the approach presented in this work can also be used for other horizontal activities. On the other hand, technological progress can ensure new activities in the area of reuse. The scientific problem in this case is to check whether the approach presented in this work is can also be used for new activities.

VIII. REFERENCES

- [1] I. Sommerville, *Software engineering*, Pearson, 2011.
- [2] F. Bobillo, M. Delgado, and J. Gómez-Romero, "Representation of context-dependant knowledge in ontologies: A model and an application," *Expert Systems with Applications*, vol. 35, no. 4, pp. 1899–1908, 2008.
- [3] N. Juristo and A. M. Moreno, "Reliable knowledge for software development," *IEEE Software*, vol. 19, no. 5, pp. 98–99, 2002.
- [4] R. Oliveto, G. Antoniol, A. Marcus, and J. Hayes, "Software Artefact Traceability: the Never-Ending Challenge," pp. 485–488, 2007.
- [5] M. Zinn, "Service based software construction process," in *Proceedings of the Third Collaborative*, Plymouth, UK, pp. 169–184, 2007.
- [6] M. Zinn, G. Turetschek, and A. D. Phippen, "Definition of software construction artefacts for software construction," in *In proceedings of the*, pp. 79–91, 2008.
- [7] J. Bosch and P. Bosch-Sijtsema, "From integration to composition: On the impact of software product lines, global development and ecosystems," *Journal of Systems and Software*, vol. 83, no. 1, pp. 67–76, 2010.
- [8] V. C. Garcia, E. S. de Almeida, L. B. Lisboa, A. C. Martins, S. R. L. Meira, D. Lucredio, and R. P. de M. Fortes, "Toward a Code Search Engine Based on the State-of-Art and Practice," 13th Asia Pacific Software Engineering Conference (APSEC'06), Bangalore, India, pp. 61–70, 2006.
- [9] T. Mens and P. Vangorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006.
- [10] R. Stair and G. Raynolds, *Principles of information systems*, 10th ed. Boston Mass.: Course Technology Cengage Learning, 2011.
- [11] T. Davenport, *Working knowledge: how organizations manage what they know*, Harvard Business School Press, 2000.
- [12] A. Kleppe, *MDA explained: the model driven architecture: practice and promise.*, Addison-Wesley, 2003.
- [13] K. Czarnecki, *Generative programming: methods, tools, and applications.*, Addison Wesley, 2000.
- [14] M. Zinn, K. P. Fischer-Hellmann, and R. Schoop, "Reuseable Software Unit Knowledge for Device Deployment," presented at the *Entwurf komplexer Automatisierungssysteme (EKA 2012)*, 2012.
- [15] M. Zinn, K. P. Fischer-Hellmann. "Reusable Software Units Integration Knowledge in a Distributed Development Environment," *International Workshop on Software Knowledge (SKY'11)*, pp. 24–35, 2011.
- [16] J. Frijters, "IKVM," *IKVM.NET Home Page*, [Online], <http://www.ikvm.net/>. [retrieved: 09,2012].
- [17] Microsoft, "ServiceModel Metadata Utility-Tool," [Online], <http://msdn.microsoft.com>, [retrieved: 09,2012].
- [18] Apache, "WebServices - Axis." [Online]. <http://ws.apache.org/axis/java/user-guide.html>, [retrieved: 09,2012].
- [19] Socrates, "Socrates Website", [Online], <http://www.socrates.org> [retrieved: 09,2012].

Case based Reasoning Approach for Re-use Activities

M. Zinn^{1,3}, K. P. Fischer-Hellmann² and Ronald Schoop³

¹University of Plymouth, Drake Circus, Devon, PL4 8AA, Plymouth, U.K.

²University of Applied Science Darmstadt, Haardtring 100, D-64295 Darmstadt, Germany

³Schneider Electric Automation GmbH, Steinheimer Str. 112, D-63500 Seligenstadt, Germany
marcus.zinn@plymouth.ac.uk, k.p.fischer-hellmann@digamma.de
{ronald.schoop, marcus.zinn}@schneider-electric.com

Abstract. The development of software applications is partly or entirely based on the re-use of software units. For software engineers, this leads to the problem that it is not possible to know all processes, technologies and supporting applications and the alternatives needed for the re-use of a software unit. As a result software engineers are not able to employ the most optimal solution known. Based on case based reasoning this paper outlines a way to use the stored knowledge of a specific re-use activity in order to give software engineers assistance if they want to perform similar activities. This solution consists of a proposal system for a re-use activity information system. The publication concludes with the result that it is possible to re-use, within a given environment, specific knowledge for other integration activities.

1 Introduction

The re-use of software units is one of the major topics of software engineering. At the same time this topic is also a wide area of scientific research. One of the central questions of this research field is to find a consistent description of software units. The answer to this question is e.g. associated with the following objectives [1]: (1) saving of resources (time and effort), (2) reduction of know-how and greater flexibility when re-using software units.

Due to this question, in the past decades, many different methods and technologies have been developed for the re-use of software units. As an example of current approaches that promote re-use, object orientation, component-orientation and service orientation are mentioned [2].

One of the problems of re-use is to define what a re-usable software unit is [3]. From the conventional view that only the part of a software unit that is actually used again (e.g. binary or source code) is such a re-usable software unit, the trend was created that also additional information (such as documentation, specification, test information, etc.) can be used again. Because of this diversity of information the terms ‘assets’ or ‘artifacts’ are used [4]. As a result a re-usable software unit thus includes many different needs for information within a re-use process.

This diversity poses a problem in answering the scientific question. The complexity of the problem increases because for each re-use technology additional methods

and applications supporting the re-use technology were developed. This strong expansion of data or information is called information explosion [5]. Software engineers have to find their way in this confusing environment.

Potential solutions can be found in the area of knowledge management (KM). KM and information systems (IS) are able to organise knowledge and information to structure and deliver it consistently [6]. Technologies such as semantic models allow the connection of different elements, creating knowledge-based statements about the knowledge of this relation. A typical example of such knowledge relation is found today in social networks and advertising. Social networks are capable of grasping knowledge entered by the user and generating adverts that might interest the user, based on this knowledge. The selection of advertisements is based on the knowledge entered by previous users. For the social networks the operator of knowledge generation is created using an added value. This process is known as ‘Knowledge Harvesting (KH)’ [7].

In principle, the method of KH may also be used in the re-use of software units. This means that knowledge about an existing software unit or a related re-use activity can be used to generate statements for other software units or activities. An existing IS or KM system that is capable of generating software units and their knowledge of software re-use activities and save it to reproduce (to perform it automatically), will be extended. This extension allows for generating predictions about alternative methods and technologies or any other specific application systems that can be used in a software re-use process. The prediction execution is focused in this publication.

2 Problem Identification

Since the scientific question has been not answered and the objectives are not implemented there is the problem that software engineers may have a comprehensive knowledge of all existing re-use technologies and the associated methodology and supporting software applications. In the following the problem of missing knowledge on the methodology and supporting software applications in the re-use of software units is focused. Usually this knowledge is specialised with certain (re-use) technologies or development models. Software engineers typically obtain this knowledge through a learning process. The experience of a software engineer supports him/her in making decisions about the re-use of software units. However, a person acquires this knowledge only when he/she works with such methodologies or applications, or is informed or somebody shows them it. This process is referred to as learning process. If a software engineer wants to solve a sub problem of software re-use (partly) automated, he/she can only do this by knowing about the corresponding application that solves the problem. Applications that a person does not know about are in this case not part of the solution approach designed by the person or the amount of knowledge of the person. This problem can be shown based on the information demand model for the re-use of software units [8].

Fig. 1 shows the structure of the Information Demand Model for the re-use of software units. It demonstrates the problem that a person lacks knowledge about a specific step of the re-use of a software unit. The subjective information demand

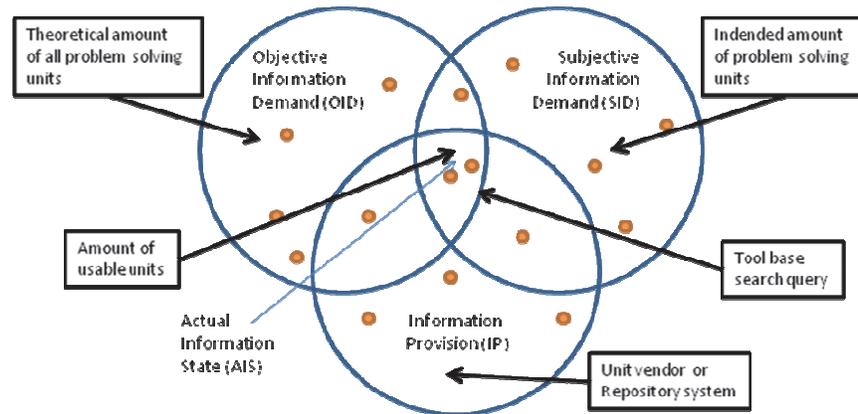


Fig. 1. Information Demand Model [8].

(SID) shown in Fig. 1 contains all solutions a person can imagine. The overlap of all three areas is the solution sets, which recognises a person who is theoretically correct (OID) and are being offered or for the person actually reached (IP). This solution set will be formulated by the inability of an individual to search even more restricted (IQ) [9]. It can be assumed that the amount of actual usable solutions (intersection of all three areas) is much lower than the approaches located in the overlapping area between the objective need for information and the offered solution sets. The reasons for this can be explained as follows:

Younger software engineers do not have much experience and knowledge in software re-use. Interestingly, these people are very interested in the re-use of software units [10]. I.e. The problem of missing knowledge shown in Fig. 1 actually exists for them. Another reason for this problem is that software engineers often have tasks that deal with new technologies or new approaches. Knowledge on the new information is usually limited to those persons concerned [10].

It is assumed that the knowledge of various software engineers is different. For example, it may be that a person is an expert in a service-oriented technology. Another person, however, is rather an expert in the use of object-oriented technologies. Both persons are experts in their field and have worked in this field with the usual methods and applications that support them in the re-use of software units. When swapping these two individuals to each other's technologies it is expected that a certain learning process is necessary to achieve the same knowledge level of a real expert in that particular technology area.

The fundamental problem can be formulated as follows: Due to the fact that many technologies, i.e. methods, processes and software applications for the re-use of software units and related activities, exist (information explosion), the problem arises that a software engineer does not have the complete knowledge to (re)use all this technology. It raises the question how the necessary knowledge, to fill in for the missing part of the activities of re-using software units, can still be made available to a person.

3 Perspective Information and Knowledge

This publication deals with the fundamental question of how to transform data into information and information into knowledge. In addition this knowledge should be available to other persons. This refers to the range of knowledge elements of the re-use of software units. The scientific background of such an investigation can be illustrated using the data–information–knowledge–wisdom hierarchy (DIKW) [11]. Based on DIKW hierarchy the elements ‘Data’, ‘Information’, ‘Knowledge’ and ‘Wisdom’ are defined as follows:

*“Data are defined as symbols that represent properties of objects, events and their environment. They are the products of observation. But are of no use until they are in a useable (i.e. relevant) form. The difference between data and information is functional, not structural. **Information** is contained in descriptions, answers to questions that begin with such words as who, what, when and how many. Information systems generate, store, retrieve and process data. Information is inferred from data. Knowledge is know-how, and is what makes possible the transformation of information in to instructions. **Knowledge** can be obtained either by transmission from another who has it, by instruction, or by extracting it from experience. [...] Wisdom is the ability to increase effectiveness. **Wisdom** adds value, which requires the mental function that we call judgement. The ethical and aesthetic values that this implies are inherent to the actor and are unique and personal.”* [11]

Software units within this publication represent ‘Data’. The DIKW hierarchy uses data to generate information, if a relation of the individual data elements (e.g. parts of the software units) is made to each other. The range of information on a software unit includes all possible information about this unit, such as the description of the technical contents, unit structure, technological information, and information about authors or producers etc. Information turns into knowledge if information is so far connected to each other that it can be used to perform an activity. As part of the re-use of software units, this means that information about a software unit for a user is brought into relation to the extent that these users transform the unit, for example, or integrate it into a development environment (can carry out its activity trap). This last step in the scope of software unit re-use represents a scientific problem [12] and is focused on in this paper.

The area of ‘Wisdom’ is the next step in the processing of knowledge. It is about clearance from the perspective of knowledge to do the right thing. But this step is not part of this publication and is not discussed further. But it is a long term problem in the re-use of software units and should be discussed and resolved.

4 Solution Approach Definition

In the following, an approach to solving the problem described above is outlined. It deals with re-use activities of integration, transformation and deployment of the re-use of software units. I.e. The outlined approach is able to store information about software units and bring it into relationships. This constitutes knowledge and can be

used to describe the above-mentioned activities and perform them with technical support.

An information system that allows saving information and knowledge about software units and re-use activities ([13], [14], and [12]) is used as a base system in this study. Through modeling of re-use activities (e.g., transformation of a software unit or integration of a software unit in a development environment) within this system, it is possible to store such activities of a particular software unit within the information system [12]. By defining and using a service-oriented environment the information system capable is of automatically performing these activities. This means that a user who has knowledge of the software unit and specific activities may deposit this knowledge in the information system. A user, who does not have this knowledge, can use the information system to access this knowledge and use it, even without learning the knowledge. The usual scenario using this information system is explained in more detail in the following example:

Example: A user who is an expert in web technologies has stored a web service software unit in the information system that is able to sign files. As information about the web service, he/she defines, among other things, that the software unit consists of a web service description file (WSDL) and a text document that serves as documentation on the software unit. In addition, the system needs some meta information (e.g. author and functional description) for this software unit, which serves, among other things, to carry out a semantic search for this software unit. After the input of the actual software unit, the expert user defines a transformation activity. He/She indicates that a particular software application (svcutil.exe) from Microsoft is able to transform the software unit (WSDL) file into a source code file containing an implemented web service client. For this transformation, he/she must also specify which information is needed for the transformation. In this case, there are various parameters and the WSDL file of the software unit. The expert also needs to define, that the result of the transformation is a new software unit. After entering this transformation and configuration into the information system, the system is able to offer another user information about a software unit (e.g. download of the software unit and its documentation) and the execution of the related activity (in this case, the transformation of the WSDL file into web service client as source code. Another user searches for a web service that is able to sign files and gets the information from the system including the loaded software unit present. The user can now view the stored metadata and the software unit. In addition, he/she is able to download the data from the software unit. The user can also view information about the stored transformation. Here it is shown that the transformation yields results. The transformation can now be performed by 'pressing on a button' within the information system and the result is delivered to the user to download.

The proposed information system is able to store re-use knowledge about specific software units and perform it. Basically it can be said that the above described problem has been solved. Users without knowledge can perform activities (i.e., unknown applications and methods) with the required knowledge. This statement is only correct if it is assumed that a person wanted to use an application in order to achieve a particular result. However, this person has no knowledge of the application used to get into the activity or methodology. The person now knows that such an activity can be performed. While this is an important factor of the fundamental information for re-

use, such a user is not able to define the same activity for a similar software unit in the information system or carry it out.

Considering Fig. 1, it can be stated: A person is only able to enter re-use knowledge in the information system, if he/she knows (has learned) these activities (knowledge and knowledge application). Conversely, it can be said that a person who has no knowledge of this cannot lack the scenario depicted a way that knowledge on other similar tasks or activities can be transferred.

4.1 Extension of the Existing Solution Approach

During an experiment [12], of which the goal was to underpin the approach of the information system shown in Example 1, the authors recognised that the knowledge of the activities entered by the participants can be used in another context. This knowledge can be used in a predictive system to support people who create new activities. This approach is hereafter called the ‘Predictive Software Re-use Activities (PreSRA)’ and follows a simple principle. The entered information about an activity, which is the input, the output and the characteristics, are stored as patterns within the information system. A user can choose three different ways to work with it. **Search for Activities:** The user can explicitly search for an activity within the information system for a software unit. For this purpose, it determines the type of activity as well as the familiar input and output information. The information system then analyses previously entered activities on this model and can give the user a recommendation, which is already a recorded activity to fit its defined input and output. **Automated Proposal System:** When creating an activity for a software unit, the user has to define an application or select an application known to the system that performs automated activity, e.g. transformation from the application example 1. In addition, the user must define the input and the output of an activity. With this definition, the information system can automatically detect the pattern for this activity based on the user input and compare them with existing patterns in the system. The result is a list of alternative applications which are able to process this pattern or alternative configurations for the already selected application. **Free Use:** Based on the second point ‘Automatic proposal system’ can be defined using another variant. Users can use the information system for activities by entering: (1) Input parameters for an activity and / or (2) the desired output and the desired result of the activity and/or (3) specific information or browse the properties of an activity. The result is an outlined list of possible activities that a user can use. Unlike the first two variants here it is not the goal to find an activity in the information system to re-use a software unit, but to preserve the knowledge of how an activity can be simulated. Such knowledge can be passed to the user i.e. in textual form.

This PreSRA approach also supports user input activity knowledge or users who generally want to identify an application that is capable of performing a certain activity at a certain given pattern. The problem described above will now be solved with the approach. It is noted that not only the automated execution of an activity without knowledge is possible, but also the knowledge that is necessary for the performance users will be provided as a suggestion system. Here are different ways to use this special knowledge. There are three interesting ‘proposal variants’: Proposal system

for the integration of software units in development environments, proposed transformation system for applications, and proposed system for device-based deployment. This division into three systems proposal does not represent the full amount of any possible systems for the re-use of software units, but represents the focus of this publication. The basic study aimed at finding [15], integration [14], and transformation and a special case of the deployment of so-called embedded devices [16]. The topic of integration is focused in this publication.

4.2 Technical Structure of the Approach

The PreSRA approach will be explained using the example of integration of software units into development environments and the existing previously outlined information system. This system will be now explained. This is necessary to understand the context of the data used by the proposed systems.

4.2.1 Information System Architecture

The central core of a basic information system is a data model that uses semantic relationships. This data model is able to store information about software units. Different components of software (plugins) have access this data model to perform different tasks. E.g. repository plugins allow the loading of units from different software repositories, which have different data models. This creates a unified view on different data sets within the information system. Plugins are able to perform re-use activities e.g. transformation and integration. An extension of the basic semantic data model is necessary. Communication plugins allow sending knowledge or information to clients / plugins for further processing (e.g. implementation of activities). It is therefore possible to view, download or use information about software units on other computer systems. At the same time it is possible to perform activities using the information system and send the result to (remote) clients. The information system provides its functionality by using the communication plug-ins in the form of various communication technologies (e.g. SOAP or REST based web service). The basic scientific investigation, however, focused on web service technology.

4.2.2 Used Data Model

The basic data model consists of four areas [15]. The first section describes metadata about the software unit, such as authors, support information, creation date, etc. The second section deals with the representation of the software unit as a solution or problem. The basic focus of the investigation is not on this scientific problem, but this area was reserved for further research in the data model. The third section describes the technical part of the software units. Takes place simultaneously in this area, a semantic model that the search of a software unit using a noun-verb combination allows [15]. The fourth section describes a software unit from a technical perspective. I.e. the contents of a software unit are defined by its physical data. Figure 2 shows this part of the data model.

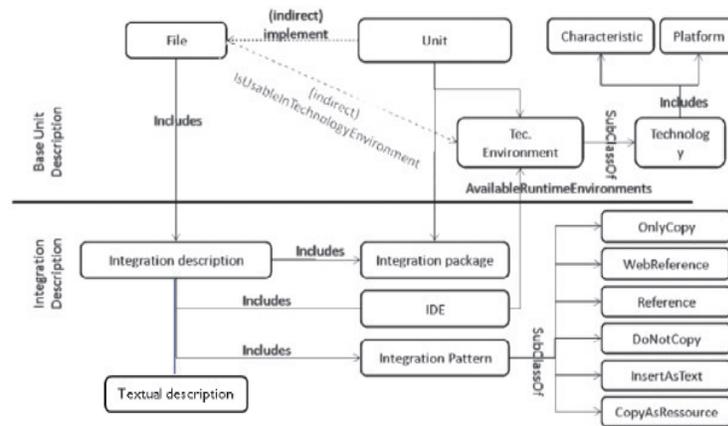


Fig. 3. Integration activity extension [14].

5 Information System as Integration Activity Proposal System

In the following the data model extension for integration of software units stated in the previous section is used to explain the concept of a proposal system. This paper introduces the basic data model and the expansion of the existing data model of information systems used for the storage of integrating knowledge. This includes the execution of integration on the basis of this knowledge [12]. For this purpose the data model shown in Figure 3 was used. The basic principle of the information system is to generalize knowledge for a particular activity and furthermore, generalize knowledge about the use of standardized interfaces specifically in software units. Accordingly, the data model shown in Figure 3 is a generalization of various models that are required for execution of integration knowledge. The most important information for the integration of a software unit is (1) Which files of a software unit can be used and (2) How these are integrated into a development environment. This not only directly affects the files of a software unit but also influences their possible dependencies. Figure 3 (1) describes the presented information and specifies the files to be integrated. These are files belonging to the software unit whose dependencies have to operate in the development environment. These dependencies can be part of the information contained in the Information system (for example, additionally stored files) or files or environment variables that must be part of the development or runtime environment of the system. In this area, (2) the information shown in Figure 2 comprises the information of the development environment. Among other things, this describes which type of development and runtime environment and which associated configurations are needed for the unit. Furthermore, this demonstrates a classification set that specifies the fundamentals of the way in which a software unit is integrated. The following content samples can be derived from this information (see Figure 3):

Table 1. Information of integration activities objects.

Typ	Description	Symbol
File(s)	All files participating in the integration process. This includes all kinds of information available about each file, eg Technology, type, name, size, etc.	File Set<File>
Integration type(s)	The integration of each file type participating in the integration process. This includes all information shown Figure 3.	Type Set<Type>
IDE(s)	Any development environment described by means of the data model. This includes any information, e.g. Name, supported technology platform, environmental variables, system files, operating system	IDE Set<IDE>
Dependency(ies)	Each dependency of a file, e.g. Technical environment, platform, environmental variables, system files, folders and file structures, relation to other software units, etc.	Dep Set<Dep>

Due to the pattern shown in Table 1, the following content model relationships can be derived from the data model shown in Figure 3¹:

Table 2. Input and output patterns table for integration activities.

		Input pattern			
		Type, Set<Type>	File, Set<File>	IDE, Set<IDE>	Dep, Set<Dep>
Output pattern	Type, Set<Type>	(✓)	✓	✓	✓
	File, Set<File>	✓	(✓)	✓	✓
	IDE, Set<IDE>	✓	✓	(✓)	✓
	Dep, Set<Dep>	✓	✓	✓	(✓)

By entering one of the input patterns defined in Table 2 can one can specify a corresponding output pattern. This will be illustrated by the following example:

While entering the files a person wants to integrate, i.e. Information, Technology, type, name, size, etc., the information system is able to compare this with stored knowledge of previous integrations. This process discovers which of the stored files contain identical or similar information. An output pattern can be used from the resulting quantities of suitable integrations as this creates information integration type, used IDE and further necessary dependencies. This includes the information content of each sample, e.g. IDE platform. This process is called case-based reasoning [17].

This type of search can be used for each of the input pattern's elements defined in Table 2. In addition, elements of the input pattern can be logically linked to obtain a more accurate result. This is illustrated by the following example. The information system can be asked with which IDE it is possible to integrate necessary files and how to define the integration type of each file specifying how these should be integrated (see Table 3). All saved integrations will be compared to see whether a similar

¹Due to the semantic relationships of the entire data model other input and output patterns can be identified. The results presented in this work are patterns and therefore represent examples that serve for direct use. This also applies to the patterns shown for other usages.

process to arrive at the content of these files can be provided to others with the same type of integration. The resulting set of integrations indicates that such integration is possible and specifies which IDEs can be performed.

Table 3. Search patterns.

		Input pattern			
		Type, Set<Type>	File, Set<File>	IDE, Set<IDE>	Dep, Set<Dep>
Input (& relation)	Type, Set<Type>	not useful	not useful	✓	maybe not useful
	File, Set<File>	not useful	not useful	✓	maybe not useful
	Result			✓ (& relation)	

6 Discussion of the Focused Approach

As part of the solution to the problem of ‘making knowledge available to people’, this publication presents two problem approaches. The first solution is the basic information system that was considered and discussed earlier from the perspectives [13] and [12]. This system provides access to information and the execution of activities through the use of stored knowledge, which is also available through remote communication systems. With this approach, the following objectives in relation to the basic problem are to be met: (1) Users need no knowledge of the software repository, including the repository’s location as well as the means to access and operate it. (2) Users require no knowledge to perform an activity. The information system and the expansion of automation plug-ins enable the integration, transformation and deployment of software units. As discussed in the first section, knowledge can be used with this solution, even if a person is not aware of this. The idea presented in this publication extends the information system with a proposal system (PreSRA) which is able to work with the accumulated knowledge about integration. It is also to offer capability of such users to further knowledge based added value (i.e. transformation or deployment). There are three considered scenarios: (1) Automatic creation of activities: Due to the fact that the proposed system is able to compare input patterns with existing patterns, the system is also capable of generating a re-use activity from a given input pattern. Thus, for example by entry of file information, a comparison with other transformations can be performed. If transformations are found, the system can process these files and is able to create an automatic transformation from it. This transformation can then be verified by an end user. The system can then use these transformations in its knowledge base. (2) Search by activities: The examples showed in the previous section show that the PreSRA system can be used i.e. to search. This applies to any activity that a particular input pattern expenditures by testing the knowledge base to a greater or lesser amount of expenditure patterns. (3) Transfer of knowledge: Looking at the data model extensions for the activities of the integration,

transformation and deployment are each composed of individual steps. Figure 3 shows a manual step description. A suggestion system may be adapted so that it not only stores automatically running activities in the information system, but can also serve as a step by step description. A user is then able to perform every single step of an activity manually. This helps the user learn the knowledge that is necessary for a particular activity. Besides the search for knowledge and execution of activities (i.e. integration), users are now able to define knowledge activities (i.e. integration) without having the appropriate knowledge. Additionally, they are also capable of using this knowledge to instruct other users. This solves the problem discussed above, that people without knowledge are unable to instruct others. This applies only within the scope of this paper and under the use of its proposed integration activity. In addition, this system allows the user to generate templates to enable other users to learn knowledge for re-use activities (i.e. integration).

7 Conclusions

This publication focuses on the problem of users inability to perform re-use integration activities of software units due to a lack of knowledge. Additionally, these individuals were not able to use existing knowledge to solve similar problems or to support other people. At the same time, this publication outlines a solution to these problems. An existing information system can (automatically) perform such re-use activities based on expert knowledge it received as input. This information system has been extended in this paper to analyse the input of knowledge and non-expert users can use it as a suggestion system. This enables users to ask the system for information, e.g. Software units in form of activities and/or the system can create such activities from existing activities or even execute them. In addition, it was shown that the system's knowledge of the activities can be made available to the user by using a case-based reasoning approach, which enables them to repeat these activities manually and thus acquire the knowledge themselves. This represents a solution for people with no knowledge, defined in the context of re-use activities like integration. This approach can be used for future research, including other activities, such as automated. Likewise, the problem may be the definition of Wisdom 'of knowledge from the perspective of re-focusing of software units'. It is also necessary to consider whether the method described before is applicable to other domains.

References

1. E. Henry and B. Faller, "Large-scale industrial reuse to reduce cost and cycle time", IEEE Software, Bd. 12, Nr. 5, S. 47–53, Sep. 1995.
2. G. Wang and C. K. Fung, "Architecture paradigms and their influences and impacts on component-based software systems", in 37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the, Big Island, Hawaii, pp.. 272–281, 2004.
3. I. Sommerville, Software engineering. Boston: Pearson, 2011.

4. R. Oliveto, G. Antoniol, A. Marcus, and J. Hayes, "Software Artefact Traceability: the Never-Ending Challenge", pp. 485–488, 2007.
5. C. Alvarado, J. Teevan, M. S. Ackerman, and D. Karger, "Surviving the Information Explosion: How People Find Their Electronic Information", Massachusetts institute of technology - artificial intelligence laboratory, Apr. 2003.
6. F. O. Bjørnson and T. Dingsøy, "Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used", *Information and Software Technology*, Vol. 50, No. 11, pp. 1055–1068, Oct. 2008.
7. N. Nakashole, M. Theobald, and G. Weikum, "Scalable knowledge harvesting with high precision and high recall", pp. 227, 2011.
8. M. Zinn, K. P. Fischer-Hellmann, A. D. Phippen, and A. Schütte, "Information Demand Model for Software Unit Reuse", presented at the ISCA 20th International Conference on Software Engineering and Data Engineering (SEDE-2011), Las Vegas, Nevada USA, S. 32–39, 2011.
9. A. Picot, *Die grenzenlose Unternehmung: Information, Organisation und Management Lehrbuch zur Unternehmensführung im Informationszeitalter*, Wiesbaden: Gabler, 2003.
10. S. G. Shiva and L. A. Shala, "Software Reuse: Research and Practice", in *Fourth International Conference on Information Technology (ITNG'07)*, Las Vegas, NV, USA, pp. 603–609, 2007.
11. J. Rowley, „The wisdom hierarchy: representations of the DIKW hierarchy“, *Journal of Information Science*, Bd. 33, Nr. 2, S. 163–180, Feb. 2007.
12. M. Zinn, K. P. Fischer-Hellmann, and R. Schoop, „Automated Reuse of Software Reuse Knowledge in an industrial environment – Case Study Results“, presented at the 17th IEEE International Conference on Emerging Technologies & Factory Automation, Krakow, 2012.
13. M. Zinn, K. P. Fischer-Hellmann, and A. D. Phippen, „Development of a CASE tool for the service based software construction“, presented at the 5th Collaborative Research Symposium on Security, E-learning, Internet, and Networking (SEIN'2009), Plymouth, 2009, pp. 134–144.
14. M. Zinn, K. P. Fischer-Hellmann, and R. Schoop "Reusable Software Units Integration Knowledge in a Distributed Development Environment", *Second International Workshop on Software Knowledge (SKY2011)*, pp. 24–35, 2011.
15. M. Zinn, K. P. Fischer-Hellmann, and A. Schütte, "Finding Reusable Units of Modelling - an Ontology Approach", in *Proceedings of the 8th International Network Conference (INC'2010)*, Heidelberg, 2010, pp. 377–386.
16. M. Zinn, K. P. Fischer-Hellmann, and R. Schopp, "Reuseable Software Unit Knowledge for Device Deployment", in *Conception of complex automation systems*, Magdeburg, Germany, 2012.
17. B. P. Allen, "Case-based reasoning: business applications", *Communications of the ACM*, Bd. 37, Nr. 3, pp. 40–42, 1994.