

2013

# Industrialising Software Development in Systems Integration

Minich, Matthias Ernst

<http://hdl.handle.net/10026.1/2772>

---

<http://dx.doi.org/10.24382/4268>

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 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.



# **INDUSTRIALISING SOFTWARE DEVELOPMENT IN SYSTEMS INTEGRATION**

by

**MATTHIAS ERNST MINICH**

M.Sc. Inf.

A thesis submitted to the University of Plymouth

in partial fulfilment for the degree of

**DOCTOR OF PHILOSOPHY**

School of Computing and Mathematics

Faculty of Technology

**September 2013**

## Abstract

# Industrialising Software Development in Systems Integration

Matthias Ernst Minich

Compared to other disciplines, software engineering as of today is still dependent on craftsmanship of highly-skilled workers. However, with constantly increasing complexity and efforts, existing software engineering approaches appear more and more inefficient. A paradigm shift towards industrial production methods seems inevitable.

Recent advances in academia and practice have led to the availability of industrial key principles in software development as well. Specialization is represented in software product lines, standardization and systematic reuse are available with component-based development, and automation has become accessible through model-driven engineering. While each of the above is well researched in theory, only few cases of successful implementation in the industry are known. This becomes even more evident in specialized areas of software engineering such as systems integration.

Today's IT systems need to quickly adapt to new business requirements due to mergers and acquisitions and cooperations between enterprises. This certainly leads to integration efforts, i.e. joining different subsystems into a cohesive whole in order to provide new functionality. In such an environment, the application of industrial methods for software development seems even more important. Unfortunately, software development in this field is a highly complex and heterogeneous undertaking, as IT environments differ from customer to customer. In such settings, existing industrialization concepts would never break even due to one-time projects and thus insufficient economies of scale and scope. This present thesis, therefore, describes a novel approach for a more efficient implementation of prior key principles while considering the characteristics of software development for systems integration.

After identifying the characteristics of the field and their effects on currently-known industrialization concepts, an organizational model for industrialized systems integration has thus been developed. It takes software product lines and adapts them in a way feasible for a systems integrator active in several business domains. The result is a three-tiered model consolidating recurring activities and reducing the efforts for individual product lines. For the implementation of component-based development, the present thesis assesses current component approaches and applies an integration metamodel to the most suitable one. This ensures a common understanding of systems integration across different product lines and thus alleviates component reuse, even across product line boundaries. The approach is furthermore aligned with the organizational model to depict in which way component-based development may be applied in industrialized systems integration. Automating software development in systems integration with model-driven engineering was found to be insufficient in its current state. The reason hereof lies in insufficient tool chains and a lack of modelling standards. As an alternative, an XML-based configuration of products within a software product line has been developed. It models a product line and its products with the help of a domain-specific language and utilizes stylesheet transformations to generate compliant artefacts.

The approach has been tested for its feasibility within an exemplary implementation following a real-world scenario. As not all aspects of industrialized systems integration could be simulated in a laboratory environment, the concept was furthermore validated during several expert interviews with industry representatives. Here, it was also possible to assess cultural and economic aspects.

The thesis concludes with a detailed summary of the contributions to the field and suggests further areas of research in the context of industrialized systems integration.

---

# Table of Contents

<b>ABSTRACT</b> .....	<b>II</b>
<b>TABLE OF CONTENTS</b> .....	<b>III</b>
<b>LIST OF FIGURES</b> .....	<b>IX</b>
<b>LIST OF TABLES</b> .....	<b>XI</b>
<b>LIST OF ABBREVIATIONS</b> .....	<b>XII</b>
<b>ACKNOWLEDGEMENTS</b> .....	<b>XIII</b>
<b>AUTHOR’S DECLARATION</b> .....	<b>XIV</b>
<b>1 INTRODUCTION AND OVERVIEW</b> .....	<b>1</b>
1.1 INDUSTRIALIZATION AND ITS KEY CONCEPTS .....	2
1.1.1 Specialization .....	3
1.1.2 Standardization & Systematic Reuse.....	4
1.1.3 Automation.....	4
1.2 INDUSTRIALIZATION IN THE IT SECTOR .....	5
1.2.1 Hardware .....	6
1.2.2 Software .....	8
1.2.3 Services .....	11
1.2.4 Shortcomings of IT industrialisation .....	12
1.3 POSITION AND OBJECTIVES OF RESEARCH.....	14
1.4 RESEARCH METHODOLOGY .....	16
1.5 THESIS STRUCTURE .....	20
<b>2 LITERATURE REVIEW AND INTRODUCTION: SOFTWARE</b>	
<b>INDUSTRIALIZATION</b> .....	<b>22</b>
2.1 A PARADIGM SHIFT FOR SUPPLIERS AND CUSTOMERS .....	22
2.2 SPECIALIZATION – SOFTWARE PRODUCT LINES.....	25
2.2.1 Fundamental Concepts and Principles.....	25
2.2.2 Software Product Line Approaches and Recent Advances .....	29

2.2.3	Software Product Line Engineering .....	32
2.2.4	Application Engineering .....	35
2.2.5	Organizational Aspects .....	36
2.3	STANDARDIZATION – COMPONENT-BASED DEVELOPMENT .....	39
2.3.1	Fundamental Concepts and Recent Advances .....	40
2.3.2	Component Models and Platforms.....	43
2.3.3	Component Architectures and Frameworks.....	45
2.3.4	Component-Based Development Approaches and Recent Advances .....	48
2.4	AUTOMATION – MODEL-DRIVEN SOFTWARE ENGINEERING .....	52
2.4.1	Fundamental Concepts and Principles .....	53
2.4.2	Domain-Specific Languages .....	56
2.4.3	Model Transformation Engines and Code Generators .....	61
2.4.4	Model-Driven Engineering Approaches and Recent Advances .....	66
2.5	PREVAILING ISSUES AND SHORTCOMINGS.....	69
2.5.1	Development Culture .....	70
2.5.2	MDE Tool Support and Usability .....	71
2.5.3	Conclusion .....	73
<b>3</b>	<b>LITERATURE REVIEW AND INTRODUCTION: SYSTEMS</b>	
	<b>INTEGRATION.....</b>	<b>75</b>
3.1	ENTERPRISE APPLICATION INTEGRATION.....	76
3.1.1	Integration Dimensions .....	77
3.1.2	Drivers of Application Integration .....	79
3.1.3	Architectures and Techniques of Application Integration .....	81
3.2	THE INTEGRATION META MODEL .....	85
3.2.1	Model Overview and Applicability.....	86
3.2.2	Process Integration.....	88
3.2.3	Desktop Integration.....	89
3.2.4	Systems Integration.....	90
3.3	CHARACTERISTICS OF SOFTWARE DEVELOPMENT IN SYSTEMS INTEGRATION .....	91

---

3.3.1	Business Process Dimension .....	93
3.3.2	Workflow Dimension .....	94
3.3.3	Technology Dimension .....	95
3.3.4	Implications for Software Development in Systems Integration .....	96
3.3.5	Shortcomings of existing Industrialization Concepts .....	98
<b>4</b>	<b>THE INDUSTRIALIZATION OF SOFTWARE DEVELOPMENT IN SYSTEMS INTEGRATION .....</b>	<b>101</b>
4.1	AN ORGANISATIONAL MODEL FOR INDUSTRIALISED SYSTEMS INTEGRATION .....	101
4.1.1	Derivation of Organizational Structures for Software Product Lines .....	102
4.1.1.1	Strategy and Target Scenarios .....	102
4.1.1.2	Suitable Forms of Organization .....	107
4.1.2	A Three-Layered Model for Software Product Lines in Systems Integration .....	114
4.1.2.1	The Business Domain Layer .....	115
4.1.2.2	The Product Line Layer .....	119
4.1.2.3	The Production Layer .....	121
4.1.2.4	Organizational Overview of Industrialized Systems Integration .....	122
4.1.3	Conclusion and Coverage of Research Objectives .....	122
4.2	COMPONENT-BASED SYSTEMS INTEGRATION .....	124
4.2.1	Selection of a Suitable Component-Based Development Approach .....	125
4.2.1.1	Requirements from Systems Integration .....	125
4.2.1.2	Discussion of Component-Based Development Approaches .....	127
4.2.1.3	Approach Selection for Further Adaptation .....	129
4.2.2	The Business Component Factory .....	132
4.2.2.1	Levels of Component Granularity .....	132
4.2.2.2	Architectural Viewpoints .....	134
4.2.2.3	Development Process .....	135
4.2.2.4	Distribution Tiers .....	136
4.2.2.5	Functional Layers .....	137
4.2.3	Component-Based Development in Systems Integration .....	138

4.2.3.1	Architectural Viewpoint Alignment .....	138
4.2.3.2	Component Granularity Alignment .....	141
4.2.3.3	Development Process Alignment .....	144
4.2.3.4	Distribution Domain Alignment.....	146
4.2.3.5	Functional Category Alignment .....	149
4.2.4	Conclusion and Coverage of Research Objectives .....	152
4.3	MODEL-DRIVEN SYSTEMS INTEGRATION.....	153
4.3.1	Selection of a Suitable Model-Driven Engineering Approach.....	154
4.3.1.1	Requirements from Systems Integration .....	154
4.3.1.2	Discussion of Model-Driven Engineering Approaches .....	156
4.3.1.3	Approach Selection for Further Adaptation .....	158
4.3.2	Light Weight Model-Driven Systems Integration.....	159
4.3.2.1	Processes of Generative Programming.....	160
4.3.2.2	Alignment with the Organizational Model for Industrialized SI.....	164
4.3.2.3	Alignment with the Business Component Approach.....	171
4.3.2.4	Domain-Specific Languages and Generators .....	175
4.3.3	Conclusion and Coverage of Research Objectives .....	180
4.4	SUMMARIZED VIEW OF INDUSTRIALIZED SYSTEMS INTEGRATION .....	182
<b>5</b>	<b>EXEMPLARY IMPLEMENTATION: INDUSTRIALIZED SYSTEMS</b>	
	<b>INTEGRATION IN THE AUTOMOTIVE DOMAIN .....</b>	<b>187</b>
5.1	EXAMPLE DEFINITION: A SAMPLE SYSTEMS INTEGRATOR.....	189
5.1.1	Service-Offering Portfolio and Organizational Structure .....	189
5.1.2	Objective of strategic realignment .....	192
5.2	AUTOMOTIVE BUSINESS DOMAIN MANAGEMENT .....	193
5.2.1	Business Domain Analysis and Portfolio Definition .....	193
5.2.2	Business Domain Architecture and Roadmap Definition .....	197
5.2.3	Core Asset Development .....	201
5.2.3.1	Integration Framework and Infrastructure.....	203
5.2.3.2	Domain-Specific Components.....	204

---

5.2.3.3	Domain-Specific MDE Artefacts .....	208
5.3	SOFTWARE PRODUCT LINE ENGINEERING .....	216
5.3.1	Product Line Requirements Engineering .....	216
5.3.2	Architecture Design and Development .....	219
5.3.3	Core Asset Development .....	222
5.3.3.1	Product Line Specific Components .....	222
5.3.3.2	Product Line Specific MDE Artefacts .....	227
5.4	PRODUCT DEVELOPMENT .....	232
5.4.1	Requirements Definition and Application Design .....	233
5.4.2	Application Realisation and Testing .....	236
5.5	CONCLUSIONS FROM THE EXEMPLARY IMPLEMENTATION AND IMPLICATIONS ON THE OVERALL MODEL .....	243
<b>6</b>	<b>FEASIBILITY – A PRACTITIONER DISCUSSION .....</b>	<b>248</b>
6.1	INTERVIEW DESIGN .....	249
6.1.1	Interview guideline development .....	249
6.1.2	Expert selection .....	252
6.1.3	Interview process .....	253
6.2	INTERVIEW EVALUATION .....	255
6.2.1	Categorization and Definition of Variables and Indicators .....	256
6.2.2	Knowledge Extraction and Transcription .....	258
6.3	PRESENTATION AND INTERPRETATION OF RESULTS .....	261
6.3.1	Category 1: Readiness for a Paradigm Shift .....	261
6.3.2	Category 2: Product Lines in Systems Integration .....	264
6.3.3	Category 3: Component-Based Systems Integration .....	266
6.3.4	Category 4: Model-Driven Systems Integration .....	267
6.3.5	Category 5: Economic Feasibility of the Approach .....	270
6.3.6	Category 6: Additional Aspects to be Considered .....	271
6.4	SUMMARY OF THE PRACTITIONER DISCUSSION .....	273
<b>7</b>	<b>CONCLUSION, OUTLOOK, AND FURTHER RESEARCH .....</b>	<b>276</b>

---

7.1	CONTRIBUTIONS TO THE FIELD OF BUSINESS INFORMATICS .....	281
7.1.1	An Organizational Model for Industrialized Systems Integration .....	281
7.1.2	Component-Based Systems Integration .....	284
7.1.3	Model-Driven Systems Integration .....	286
7.2	RESEARCH LIMITATIONS .....	289
7.3	POSSIBLE FURTHER RESEARCH FOR SOFTWARE INDUSTRIALIZATION .....	292
7.4	COVERAGE OF RESEARCH OBJECTIVES .....	294
<b>APPENDICES.....</b>		<b>296</b>
A.1	LIST OF PUBLICATIONS .....	296
A.2	INTEGRATION METAMODEL ENTITIES .....	297
A.3	INTEGRATION METAMODEL – INFORMATION SYSTEM VIEWPOINT.....	302
A.4	EXPERT INTERVIEW PREPARATION PAPER.....	303
A.5	EXPERT INTERVIEW GUIDELINE .....	318
A.6	EXPERT INTERVIEW TRANSCRIPTION.....	321
<b>REFERENCES .....</b>		<b>330</b>
<b>PUBLISHED PAPERS .....</b>		<b>357</b>

## List of Figures

Figure 1-1: In-house production depth for IT Services in 2011.....	6
Figure 1-2 IT Design Science Research Cycle .....	17
Figure 1-3 Decomposition of the research aim into IT Design Science Research Cycles.....	18
Figure 2-1: An exemplary variability model.....	27
Figure 2-2: Software Development in a Software Product Line.....	28
Figure 2-3: Component Based Architecture Framework by Andresen.....	48
Figure 2-4: Models within an MDA approach.....	56
Figure 2-5: A Domain-specific concept in Eclipse GMF .....	61
Figure 2-6: Components of Intentional Programming .....	67
Figure 3-1: EAI drivers and their influence on each other.....	81
Figure 3-2: Point-to-point integration architecture .....	82
Figure 3-3: Federated Hub-and-spoke integration architecture .....	83
Figure 3-4: Integration bus architecture.....	83
Figure 3-5: Integration dimensions vs. integration metamodel viewpoints .....	87
Figure 3-6: Integration metamodel – an overview.....	88
Figure 3-7: Integration metamodel– process integration view .....	89
Figure 3-8: Integration metamodel– desktop integration view .....	90
Figure 3-9: Integration metamodel – systems integration view.....	91
Figure 4-1: Organizational Structure for Software Product Line Engineering .....	110
Figure 4-2: Organizational structure for product development.....	113
Figure 4-3: Three Layered Approach for Industrialized Systems Integration .....	122
Figure 4-4: Business Component Model – Component Granularity .....	134
Figure 4-5: Business Component Model – Distribution Domains.....	137
Figure 4-6: Architectural Viewpoint Alignment – Organizational Structure .....	141
Figure 4-7: Component Granularity Alignment – Organizational Structure .....	143
Figure 4-8: Development Process Alignment – Organizational Structure.....	145
Figure 4-9: Distribution Domain Alignment – Desktop Integration.....	147

---

Figure 4-10: Distribution Domain Alignment – Process Integration .....	148
Figure 4-11: Distribution Domain Alignment – Systems Integration .....	149
Figure 4-12: Functional Category Alignment – Desktop Integration.....	150
Figure 4-13: Functional Category Alignment – System Integration .....	151
Figure 4-14: Functional Category Alignment – Process Integration.....	152
Figure 4-15: Generative Domain Model .....	160
Figure 4-16: Mapping of Generative Programming Processes to Organizational Structure .....	170
Figure 4-17: XML based application modelling and code generation for industrialized SI .....	180
Figure 5-1: Exemplary Implementation Research cycle .....	188
Figure 5-2: ACME Integration Solutions Service-Offering Portfolio.....	190
Figure 5-3: ACME Integration Solutions Organizational Structure.....	191
Figure 5-4: Core processes of the automotive industry .....	194
Figure 5-5: Automotive Business Domain of ACME Integration Solutions.....	195
Figure 5-6: Automotive Service Bus Technical Architecture .....	199
Figure 5-7: ACME Integration Solutions automotive functional architecture .....	200
Figure 5-8: ACME Integration Solutions Automotive Service Bus technical roadmap.....	201
Figure 5-9: Typical core assets of a business domain .....	202
Figure 5-10: An example business component from the automotive business domain.....	206
Figure 5-11: XML schema example .....	212
Figure 5-12: Partial variability model of the supply chain management product line .....	218
Figure 5-13: Example of the OrderCreation Business Component .....	223
Figure 5-14: Rapid System Development process.....	226
Figure 5-15: Simplified feature model of the example customer product.....	235

---

## List of Tables

Table 1-1: Distribution of workforce in the three sector hypothesis.....	1
Table 4-1: Strategic alternatives for industrialized systems integration .....	105
Table 4-2: Comparison of CBD approaches with systems integration requirements .....	130
Table 6-1: Experts interviewed.....	255
Table 6-2: Extraction categories for qualitative content analysis .....	257
Table 6-3: Extraction rules for qualitative content analysis.....	259
Table 6-4: Exemplary knowledge extractions .....	261

## List of Abbreviations

ALM	Application Lifecycle Management
AM	Application Management
CASE	Computer Aided Software Engineering
CBD	Component Based Development
COM	Component Object Model
COTS	Commercial off the shelf
DCOM	Distributed Component Object Model
DSL	Domain Specific Language
EAI	Enterprise Application Integration
EJB	Enterprise Java Bean
JAVA EE	Java Enterprise Edition
JDK	Java Development Kit
JMS	Java Messaging Service
MDA	Model Driven Architecture
MDE	Model Driven Engineering
SI	Systems Integration
SPL	Software Product Line
UML	Unified Modelling Language
XML	Extensible Markup Language
XSD	XML Schema Definition
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformation

To my parents.

## **Author's Declaration**

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award without prior agreement of the Graduate Committee.

Relevant scientific seminars and conferences were regularly attended at which work was often presented; external institutions were visited for consultation purposes and several papers prepared for publication.

Bibliographic details of publications and presentations carried out during the research programme can be found in Appendix A.1 and are available as full-text at the end of this thesis.

Word count of main body of thesis: 84.683

Signed: 

Date: 31.08.2013

# 1 Introduction and Overview

According to the three-sector hypothesis by Fisher and Clark (1935; 1940), an economy can be divided into a primary sector producing raw materials and agricultural goods, a secondary sector manufacturing goods from raw materials, and a tertiary sector providing services. According to subsequent works by Jean Fourastié (1954, pp. 106ff.), over time an economy will shift the focus of its economic activities from agriculture and the extraction of raw materials via the production of goods towards a society based on the provision of services. The transition between each of the phases has a significant impact on the distribution of labour. During the first phase - the traditional civilization - the majority of the workforce works in the primary sector, while only an insignificant minority manufactures goods or provides services. In the second phase - the transitional period - industrial progress in the primary sector supports a growing population while demanding manufactured goods such as machinery. This demand is reinforced by further technological advances which lead to the majority of the workforce being employed in the secondary sector. Due to industrial production and saturated demands, the production of raw materials employs only one-fifth of the steadily increasing workforce. During the third and final phase - the tertiary civilization - the overall workforce is clearly distributed towards the provisioning of service, while only one-tenth work in the primary and secondary sector.

Table 1-1: Distribution of workforce in the three sector hypothesis (Fourastié, 1954, pp. 106ff.)

	<b>Traditional Civilization</b>	<b>Transitional Period</b>	<b>Tertiary Civilization</b>
Primary Sector	70 %	20 %	10 %
Secondary Sector	20 %	50 %	20 %
Tertiary Sector	10 %	30 %	70 %

The primary sector in its beginnings was characterized by individual craftsmanship and small trade. Technical innovations and institutionalized economic conditions allowed for improvements in productivity, which spawned the demand for the manufacturing of machinery and

other technical goods (Butschek, 2006, p. 115). A growing workforce and continuous technological advances provided the breakthrough for the industrial revolution and the initiation of the transitional period.

## 1.1 Industrialization and its Key Concepts

Relating to the production of goods, industrialisation is defined as the implementation of standardized and highly productive methods in order to increase efficiency and reduce cost (Encyclopaedia Britannica, 2005b, p. 304). It is characterized by an increasing division and specialization of labour, capital intensive technologies, mass production, rationalization and the application of new energy sources (Encyclopaedia Britannica, 2005c, p. 253). Furthermore, the availability of communication and transportation infrastructure leads to a partitioning of the value chain and thus subcontracting required manufacturing artefacts to external suppliers, those being more price-competitive than the parent company (Weiss, 2002, p. 146). Standardization and specialization advance the level of reuse and enable automation of rote and menial tasks, whereas creative tasks, such as product design, are still performed by highly-skilled workers. Further, efficiency gains may be achieved if simple but labour intensive tasks are automated.

Driven by an economically determined utilization of capital (Butschek, 2006, p. 116), industrialization is seen as a necessary step for economic growth, technological advances and increasing wealth. Only industrial production methods allow the production of a multiplicity of goods in a sufficient amount and quality (Encyclopaedia Britannica, 2005d, pp. 280 ff., 2005c, p. 254), and thus the transition to the secondary sector. To summarize, industrial methods can be categorized as follows:

- Specialization
- Standardization & systematic reuse
- Automation

As of today, the above principles can be found in almost all industries at different levels of penetration.

### **1.1.1 Specialization**

In the given context, the term specialization describes the concentration of an economic subject (worker, business, society, etc.) to a particular area within a larger scope, such as certain industries, product families, technologies and skills (Encyclopaedia Britannica, 2005e, p. 209). A production process is subdivided into less complex functions that can then be assigned to well-trained workers or purpose-built machinery. This division of labour allows for the specialization of individuals, expanding their knowledge and abilities in a particular area. In turn, a higher efficiency and quality along with an increase in economies of scale can be achieved. Specialization may also include production and product artefacts. The former include processes, tools and machinery, while the latter include standardized parts or platforms. Systematic reuse can only occur in a precisely delimited scope, defined by specialization and standardization.

The disadvantages of specialization lie in a reduced flexibility and thus the dependency on market demand of the area or skill in scope as well as the dependency on upstream production. A highly specialized economic subject cannot quickly change its area of focus. Thus a farsighted, strategic planning of specialization is mandatory.

Well-known implementations of specialization can, for instance, be found in the automotive sector. A whole industry subcontracting to automotive manufacturers emerged, specializing in certain product families such as engines, brake systems or electronic control units. Furthermore, employees specialize in particular skills and tasks in the production process, visible in innumerable specialized professions.

### **1.1.2 Standardization & Systematic Reuse**

Standardization describes the unification of specific attributes of production or product artefacts. The objective is to establish a common understanding of these attributes in order to exchange artefacts, integrate upstream work products, align production processes or simplify information exchange (Encyclopaedia Britannica, 2005e, p. 209; F. A. Brockhaus, 2005, p. 152). Together with specialization, standards provide the base for systematic reuse and thus a significant reduction of transaction costs. Only if an artefact follows clearly defined principles can it be reused as is in another product. Standards can be officially defined (by a binding regulation or contract), de facto (by market position or dominant usage), or voluntarily. They form the basis for a decomposition and reorganization of the value chain and thus an increase in efficiency of production.

With regard to market position or profit margin, standards can also be disadvantageous, as standards encourage competition between suppliers. Furthermore, they may require tradeoffs in functionality which may affect a unique selling point of one's own product or the lack of customization possibilities.

A good example of standardization can be found in the modular construction system of automotive manufacturers. Uniquely designed product artefacts such as axles or suspensions can be reused in many different models of a product family. Likewise, production artefacts such as assembly lines, tools or machinery, can be reutilized to produce many different products.

### **1.1.3 Automation**

The division of labour, standardization, and systematic reuse allows purpose built machinery to take over rote, menial, or dangerous tasks. The operational sequence, regulation and monitoring of the production process may also be performed by technical equipment (Encyclopaedia Britannica, 2005d, p. 288). Such machines often are more precise and time- and cost- efficient as compared to human workers. As machinery cannot solve unknown problems, specialization and standardization are especially important. In industrialized production, the worker's role shifts from manufacturing towards planning, monitoring and correction of the production process

(Encyclopaedia Britannica, 2005b, p. 305). The objective here also is to reduce cost and time and increase quality and quantity.

Drawbacks in automation are inherited from the previously mentioned principles. High upfront investments require a minimum utilization rate to break even, while reduced flexibility implicates a high market dependency of the segment in scope.

Automation is, as well, an important factor in the automotive sector. The industry heavily relies on automated production such as welding robots or automated assembly lines.

## **1.2 Industrialization in the IT Sector**

Today almost every economic sector shows signs of industrialization, although at different stages of maturity. A possible indicator of the degree of industrialisation can, for instance, be found in the level of in-house production. Automotive manufacturers, for example, reduced their in-house production depth, i.e. the ratio of a product manufactured in own factories, from almost 100% at the beginning of the 20<sup>th</sup> century to 35% in 2002, and are expected to reach 23% in 2015 (Becker, 2010, p. 13). Similar signs can be found in information technology. While in the beginning it often was a competitive advantage to invest in cutting-edge IT, it has now become more and more commoditized and easily available to everyone (Carr, 2003, pp. 44f; Taubner, 2005, p. 293). Beginning with mainframe timesharing, shared local area networks and eventually the internet, IT has evolved towards shared data centres and cloud-based infrastructure services. Each consolidation step was initiated by greater standardization, which in turn allowed service providers to specialize in certain areas and thus leverage further economies of scale. By using standardized products and data formats, it is possible to split up the value chain and outsource IT services to external providers. “More and more, companies will fulfil their IT requirements simply by purchasing fee-based ‘Web services’ from third parties – similar to the way they currently buy electric power or telecommunications services” (Carr, 2003, pp. 45f.).

Although not to the extent found in more mature industries, the in-house production depth for IT services has already decreased significantly. In a recent market survey of 156 enterprises in

Germany, Austria, and Switzerland, the in-house production depth was reported between 49.6% for application development, 60.1% for application management, and 59.3% for infrastructure management (Dumslaff and Lempp, 2012, p. 17).

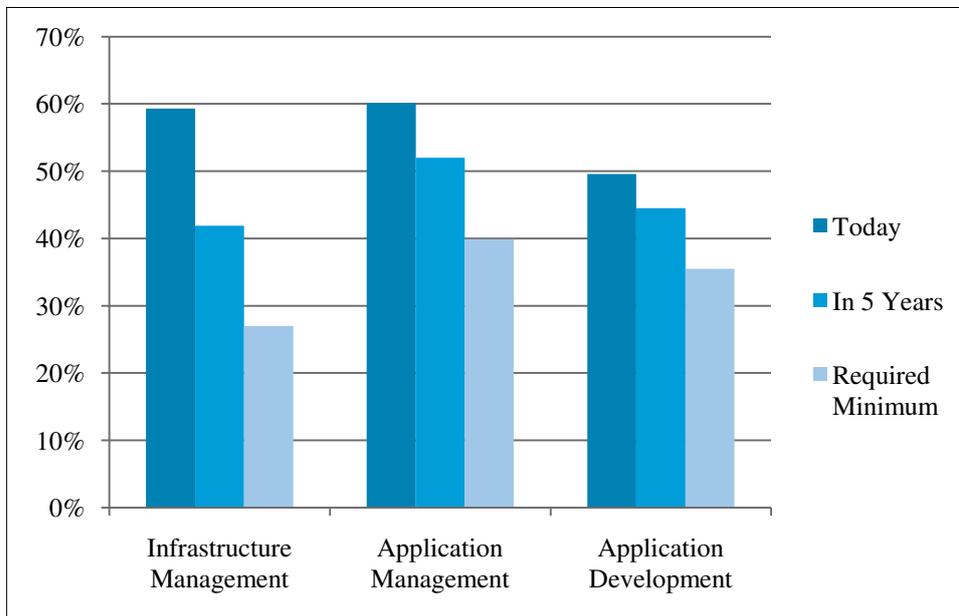


Figure 1-1: In-house production depth for IT Services in 2011

Furthermore, the “near-infinite scalability of many IT functions, when combined with technical standardization, dooms most proprietary applications to economic obsolescence” (Carr, 2003, pp. 45f.). This not only applies to applications but to any type of IT service. The result is an inevitable price deflation to which technology and service providers can only counteract by transforming themselves into utilities providers (Carr, 2003, pp. 46f.) and applying industrial methods in order to produce their services in a sufficient amount and quality at a reasonable price. Under the pressure of commoditization, several advances in IT industrialisation have recently been achieved.

### 1.2.1 Hardware

During the last decades, information technology hardware has become more and more standardized, especially with the introduction of the IBM Personal Computer, which can be seen as breakthrough in terms of compatibility and expandability. Different manufacturers started to produce compatible devices, which led to a price deflation on the developing market

and eventually a commoditization of personal computers. Proprietary architectures from Atari or Commodore could not keep up with increasing innovations and eventually disappeared. Similar effects can be observed in most other areas of information technology hardware, be it networking equipment, data centre storage facilities, or telephony devices. Individually developed hardware is only present in very rare and specialized application areas (Walter et al., 2007, p. 8). Even electronic control units in modern cars are becoming standardized, e.g. by adhering to the Controller Area Network standard to communicate with each other. Production of IT hardware has largely been industrialised, as for example automated and large scale production of integrated electronic components, circuit boards, and automated circuit board assembly.

After extensive industrialisation and commoditization of the IT hardware market, a constant development towards virtualization can be observed. What has been used to share and better utilize mainframe computing power is now the driving force of virtualization (Figueiredo et al., 2005, p. 30) and eventually that of cloud computing (Wang et al., 2010, pp. 139–140):

*“Cloud computing’s promise is that computer resources are now commodities that can be pooled and accessed on an as-needed basis. Economies of scale result from multiple users sharing those resources, and the commercial world now successfully offers cloud services at the infrastructure, platform, and application levels”* (Geller, 2012, p. 21).

However, cloud computing comes with certain challenges and obstacles. Processing confidential and business-critical data on someone else’s resources creates new security issues. Guaranteed performance and availability are additional threats (Dillon et al., 2010, p. 30), arising from sharing resources with others. To overcome these obstacles, major outsourcing providers offer private clouds to large enterprises or a closed group of customers not in competition with each other. By having all resources under their responsibility, providers can guarantee security, performance, and availability (Armbrust et al., 2010, p. 51).

From the above explanation and examples it can be assumed that the industrialisation of IT hardware production has largely been completed. Specialized suppliers produce standardized

components which can automatically be assembled to the final products. Industrial core principles are implemented and the industry has clearly moved to the secondary sector. Furthermore, with the emergence of cloud computing and virtualization services, a development towards the tertiary sector is becoming evident.

### **1.2.2 Software**

The effects of industrialisation on software development are significantly different as compared to hardware. Due to the fact that software is an intangible good and thus can be “produced” at virtually no cost, industrial key concepts must focus on software engineering instead. Still, the enormous cost for development can be shared across the high volume of produced goods (Taubner, 2005, p. 292), which can be observed with commodity software such as desktop operating systems or standard office suites. With industry-sector-specific software, however, it is completely different. Development costs are still very high but can only be shared among a comparably small customer base. Furthermore, business processes and thus the underlying software are expected to become more complex in the future (Greenfield et al., 2004, p. xvi; Taubner, 2005, p. 292), leading to even higher development costs. Applying industrial methods similar to other economic sectors appears inevitable.

Software development is still reliant on craftsmanship of highly skilled workers. The objectives of every software project form an antagonistic relationship between quality, quantity, time, and cost (Balzert, 2008, p. 196). As the available productivity of the performing organization is limited, tradeoffs between these objectives have to be made: Doing more work of a higher quality, for example, will result in higher costs and a longer development time. By applying industrial methods and thus enhancing an organization’s overall productivity, quality and product complexity can possibly be increased and at the same time cost and production time reduced.

Several efforts have been taken to apply such methods, and industrial key principles can now also be found in the field of software engineering: Specialization is represented by Software Product Lines (SPL), Standardization and systematic reuse may be found in Component Based Development (CBD), and Automation can be achieved with Model Driven Engineering (MDE).

An SPL is “a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” (Clements et al., 2007, p. 5). It exploits economies of scope rather than scale by reusing as many product and production artefacts as possible. CBD is an approach to exchange and systematically reuse software artefacts. “A software component is a unit of composition with contractually specified interfaces and context dependencies only” (Szyperski et al., 2002, p. 27). Components can be independently utilized and composed to applications and often represent a complete business concept. Using visual models as a description of software, MDE as the third industrial principle aims to raise the level of abstraction in order to fill the gap between the semantic problem solution and its technical implementation (Greenfield et al., 2004, p. 142). In contrast to the much more generic CASE approach, MDE reduces the degree of freedom and possible contexts by defining domain-specific languages and thus having much more precise and meaningful models. This allows using transformation engines and code generators to automatically advance the development process.

Unfortunately, the most important concept for industrialised software engineering, specialization, was invented last. Significant research on the topic was only begun with the first Software Product Line Conference in August 2000. Since that time, several institutional projects have been started, such as the AMPLE (Rashid, 2012) or FAMILIES (CAFÉ Consortium, 2012) project. Methodologies and Frameworks (Software Engineering Institute, 2012a; Bayer et al., 1999) for implementation are available, as are several books on the topic (Pohl et al., 2005; Linden, 2007; Clements et al., 2007; Greenfield et al., 2004). Adoption in the industry, however, is rather limited due to implementation cost, availability of tools, and significant changes in the development culture (Janßen, 2005; Selic, 2008, pp. 379 ff; Clements et al., 2007, p. xviii). Research literature offers several case study implementations, although it is unclear how successful they will be in the long term. Another indicator may be the Software Engineering Institute’s Software Product Line Hall of Fame: At the time of writing, only 16 companies were

listed as having successfully implemented a software product line from a technical and economical point of view (Software Engineering Institute, 2012b).

Without limiting the scope of one's production, subsequent industrial concepts, i.e. systematic reuse and automation, cannot successfully be implemented. As of the resulting universal generality, a large scale adoption of component-based development and model driven engineering in their initial occurrence possibly failed (Greenfield et al., 2004, pp. 21–23; Hahn and Turowski, 2003, pp. 128 ff; Selic, 2008, p. 382). Especially for CBD, the lack of standardization, insufficient component descriptions, and unknown intellectual property rights prevented a wide-spread inter-company adoption and the formation of market places (dos Santos and Werner, 2010, pp. 135 f.). Even within one company, components often had too broad a context (Greenfield et al., 2004, p. 21) or the responsible managers did not want to invest additional efforts in explicit development for reuse (Clements et al., 2007, p. xviii). MDE primarily suffers from usability problems and tool unavailability (Selic, 2008, p. 385; Hutchinson et al., 2011, p. 472), a lack of interoperability between different vendors, and scalability problems of large and very large models (Selic, 2008, pp. 385–387; Hutchinson et al., 2011, pp. 472–473).

Although industrialization in the software sector and thus advancement of the industry to the secondary sector has just begun, first traces of a tertiary sector can already be observed (Betz, 2007, p. 26). As license fees decline, software vendors try to compensate revenues by offering services to their customers. Since 2005, an average software firm has been selling more services than products (Cusumano, 2008, p. 24). Even more mature are concepts such as salesforce.com or Microsoft Office 365, providing software as a service at regular fees.

From above explanations and examples, it can be assumed that the industrialization of the software sector is largely immature. The majority of products are developed manually without any significant specialization, standardization, or automation. Commoditization of some products was only possible as a large amount of copies could be produced at virtually no cost. This mechanism only works with a very large consumer base and cannot be compared to commoditization in other industrial areas. It is believed that significant industrial software development

will only form in markets where a large number of copies can be sold. Only there can implementation costs of current concepts be justified with a long-term positive return on investment unless other, more efficient, methodologies are developed.

### **1.2.3 Services**

The industrialization of IT Services must be differentiated into services related to IT hardware and those related to software. The former can furthermore be separated into hardware-allocated services such as warranty and repair, and provisioning of certain functionality, such as a managed network port. Warranty and repair are mostly industrialized, especially with large hardware suppliers. Customers may choose from different support packages and receive their service according to a previously defined standard, extent, and service level. Large parts of the underlying support process are automated, e.g. ticket handling, notifications, or shipping of spare parts. A similar degree of industrialization can be found in service provisioning. Many outsourcing providers offer their customers a standardized service catalogue from which they can choose the most suitable features (Staines, 2011; Gonsalves, 2011; Walter et al., 2007, p. 9). Carr's assumption from 2003 about IT services becoming similar to utilities (Carr, 2003, pp. 45f.) has become reality. Consumers simply order a standard desktop PC, a managed LAN port, and a standard telephone. Services are being provided according to the Information Technology Infrastructure Library, which ensures standardized and repeatable service delivery (Walter et al., 2007, p. 9). Providers, on the other hand, specialize in certain areas and share their services among multiple customers. This includes not only hardware, but also personnel, service desk infrastructure, and logistics. Furthermore, by standardizing products, one can easily automate their provisioning. Good examples therefore are virtual servers in a cloud: a simple copy of an image file and some minor script-driven changes create a new server.

Software related IT Services can be separated into application management (AM) and application lifecycle management (ALM), including software development (Betz, 2007, pp. 23 f.). Application management refers to the provisioning and maintenance of standard software such as a database or an SAP installation. A service provider ensures an agreed-upon availability and

performance of the installation and performs regular maintenance activities as long as these don't require any additional development. Similar to hardware provisioning, such services are already available (Betz, 2007, p. 26; Walter et al., 2007, p. 9), but standardization and automation are not as far advanced. Application lifecycle management in turn refers to any activity that is required during the lifecycle of an application and includes bug fixing, update development, customization, or optimization of a previously developed application. These activities are predominantly related to software development and thus the industrialization aspects from section 1.2.2 apply.

With regard to services, it can be assumed that, besides hardware warranty and repair, IT infrastructure services in particular are largely industrialized. This includes the provisioning and management of applications, although not necessarily fully automated. Application lifecycle management in turn is still immature due to its close relationship to software development.

#### **1.2.4 Shortcomings of IT industrialisation**

In his article "IT doesn't matter", Nicholas Carr (2003) postulates the commoditization of IT and eventually an analogy to any other utilities such as water or electricity. Consumers buy what they need from a variety of different suppliers. Suppliers specialize in the industrial production of standardized products and offer them on the market.

With regard to IT hardware and infrastructure services, first signs of this development have become evident. Large outsourcing contracts are no longer based on a fixed infrastructure but on network ports, telephones, or storage space. Cloud computing and virtualization allow for continuous ordering and cancelling of resources as demanded by the business. By standardizing and automating their products, suppliers try to leverage economies of scale and thus gain competitive advantages. For application management of standard software, similar evidence can be found, although production is not as standardized and automated as with infrastructure services. As the industrial key concepts are generally in place, it is believed that further development of these is driven by market participants and their innovative efforts to increase efficiency.

As for software development and subsequent lifecycle management of customer-specific applications, services are still reliant on craftsmanship of highly skilled workers. Respective activities are far from industrialised production and thus also from becoming a commodity any time soon. Carr's assumption that most proprietary applications will become obsolete may yet become true (Carr, 2003, pp. 45f.), but must be postponed to the future. The industrial key concepts are still being researched and developed, which means that market participants do not sufficiently invest in their advancement due to uncertain return on invest. However, reducing development cost would be a big competitive advantage, especially as business processes become more and more complex.

In today's business world, IT faces high demands in quickly adapting to new requirements and business processes. The increasing complexity of IT systems and vast variety of technologies they may be implemented with are possibly the biggest obstacles for standardization. Furthermore, as legacy applications often do not offer the flexibility required, new customized systems are implemented which need to interact with the existing IT landscape. This situation inevitably leads to systems integration efforts, joining the different subsystems into a cohesive whole, in order to provide new business functionality or data access (Fischer, 1999, p. 86; Leser and Naumann, 2007, p. 3).

At this point, at the very latest, current industrialization concepts have reached their limits, i.e. they cannot be used to further simplify production. It is questionable whether Software Product Lines, Component-Based Development, and Model-Driven Engineering can be applied in the field of systems integration as well, and, even more important, if they can generate economic benefits to justify their implementation. Of course there are several other areas of software development or application lifecycle management where industrialisation is still immature, such as infrastructure software or embedded systems. Infrastructure software, including operating systems, databases, or office suites, has the advantage of a large customer base. Savings from industrialisation are expected to be smaller as development cost can be distributed across a comparably large user base. The same applies to embedded software, as found, for example, in

an integrated control unit in a car. Furthermore, both sectors are not subject to frequent changes as compared to enterprise applications.

It is therefore believed that software development of customized enterprise application solutions and their lifecycle management benefit the most from industrial production principles.

### **1.3 Position and Objectives of Research**

Industrialization is seen as the key driver to increase efficiency and reduce cost in most manufacturing enterprises (Encyclopaedia Britannica, 2005d, pp. 280–286, p.253; Butschek, 2006). Recent research also brought industrialization closer to software engineering by adapting the industrial key principles of specialization, standardization, and automation to the particular needs of the field. Each key principle is now represented in a specific development concept, i.e. software product lines for specialization, component-based development for standardization, and model-driven engineering for automation.

While these concepts are being developed and begin to find their way into practice for commodity software, it is not certain whether they can be applied to all areas of software engineering. One of these areas is systems integration. In an increasingly interconnected world, existing systems must implement new business processes, and new systems must smoothly be integrated into an existing IT landscape. As there are only very few greenfield approaches possible, systems integration becomes more and more important. Developing such solutions still relies on craftsmanship and highly-skilled workers (Kaib, 2004, pp. 77 f.). Industrial key principles are not extensively in place; only few integration approaches pick up component-based development (Conrad et al., 2006, pp. 232–236; Vogler, 2006, pp. 66–74) or domain-specific engineering (Conrad et al., 2006, pp. 236–242). Furthermore, high upfront investments to implement industrial methods may possibly never break even where only one or very few instances of a product are being sold.

This thesis, therefore, takes the position of a large systems integrator whose core business lies in developing complex software solutions to be integrated with each other, and in developing

software solutions to allow the integration of new or already existing IT systems into a given IT landscape. Such systems integrators are usually active in several industries, such as automotive, travel & transport, and public services (Pierre Audoin Consultants, 2009) and employ up to several thousand software developers. Leading enterprises in this area are, for instance, IBM, Accenture, CapGemini, CSC, and T-Systems. The underlying research can be classified into the scientific area of business informatics as it covers matters from business (organizational forms of enterprises and product family management) and computer sciences (implementation of CBD and MDE). It was conducted in close collaboration with the industry, meeting concerns about the fact that very little of software engineering research finds its way into actual application (Potts, 1993, p. 19).

The aim of the research described herein is to investigate whether techniques of software industrialisation can be applied to software systems integration within economic and technical constraints and, if feasible, to propose a means for doing so. As mentioned before, the adoption of such methods is still immature and, especially in the field of customer-specific software development, a positive return on investment is uncertain. The research in question analyses the strategy and objectives of a systems integration provider, identifies current industrialisation concepts, adapts them, and derives a suitable implementation for the given field. The key research objectives can be subsumed as follows:

1. Conduct a literature review to identify
  - a. the characteristics of software industrialisation and their prerequisites,
  - b. the characteristics of systems integration and their impacts on industrialised software development, and
  - c. how far current methods of software industrialisation have and can cope with industrialisation to determine their shortcomings in this respect.
2. Analyse existing methods of industrialisation to see which of them could be used or adapted to be used to overcome the previously identified problems relating to systems integration and from this propose (where possible) a viable means of applying industrialisation to systems integration.

3. Test the proposed methods of industrial software development in systems integration by examining how they could be applied in an example based on a typical real world scenario and update the proposal as necessary.
4. Test the proposal by interviewing experts in the field of industrialisation and systems integration to gauge their opinion on the viability of the proposal, and again, update the proposal as necessary.
5. Test the proposal by publishing articles on some of the ideas contained in the proposal in peer reviewed journals and conference proceedings.

The thesis picks up on further research suggested by Pohl et al. (2005, pp. 437) who identify the need to investigate the interaction of different software product lines and their products, which is exactly the case in enterprise systems integration: “Solutions for defining and managing variability across different product lines and across all software development artefacts is (sic) still immature” (Pohl et al., 2005, p. 437).

## 1.4 Research Methodology

This thesis and the underlying research is based on design science introduced by March et al. (1995, pp. 251-266). It “emphasizes the connection between knowledge and practice by showing that we can produce scientific knowledge by designing useful things” (Wieringa, 2009, p. 1). However, one has to carefully distinguish between solving practical problems and knowledge problems. The former change the research subject to better suit specific stakeholders’ needs, while the latter observe the research subject to better understand it (Wieringa, 2009, p. 1). IT research can be seen as the study of artefacts being adapted to their environment. It reflects both, practical problems in terms of design science and knowledge or theoretical problems in terms of natural science to explain how and why a designed solution works within its environment (March et al., 1995, p. 255). As IT artefacts always are produced and altered based on particular stakeholders’ needs, there can’t be fundamental and generalizable theories as in natural sciences, let alone persistent ones. As soon as the stakeholders’ needs change, the theory becomes invalid. March et al. (1995, p. 255) therefore suggest a research framework consisting

of two dimensions: The first is based on the design science outputs constructs, models, methods, and instantiations. The second is based on generic design science and natural science research activities and includes building, evaluating, theorizing, and justifying one or more of the previous artefacts. They furthermore postulate that these steps should form an iterative cycle until a satisfactory answer to the practical problem is found and scientifically justified.

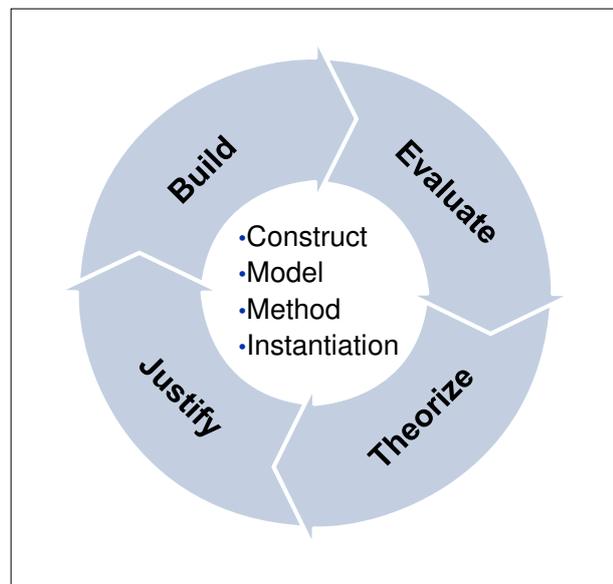


Figure 1-2 IT Design Science Research Cycle according to March et al. (1995)

The research cycle starts with the evaluation of the problem, theorizing a solution, justifying this solution, and finally building the solution. To select the correct methodology for these research activities, Wieringa (2009, p. 4) suggests to decompose the research question into practical problems and knowledge questions to avoid applying unsound research designs.

Applied to this thesis, the aim of the underlying research presented in section 1.3 can be decomposed into several nested research cycles as presented in the following figure.

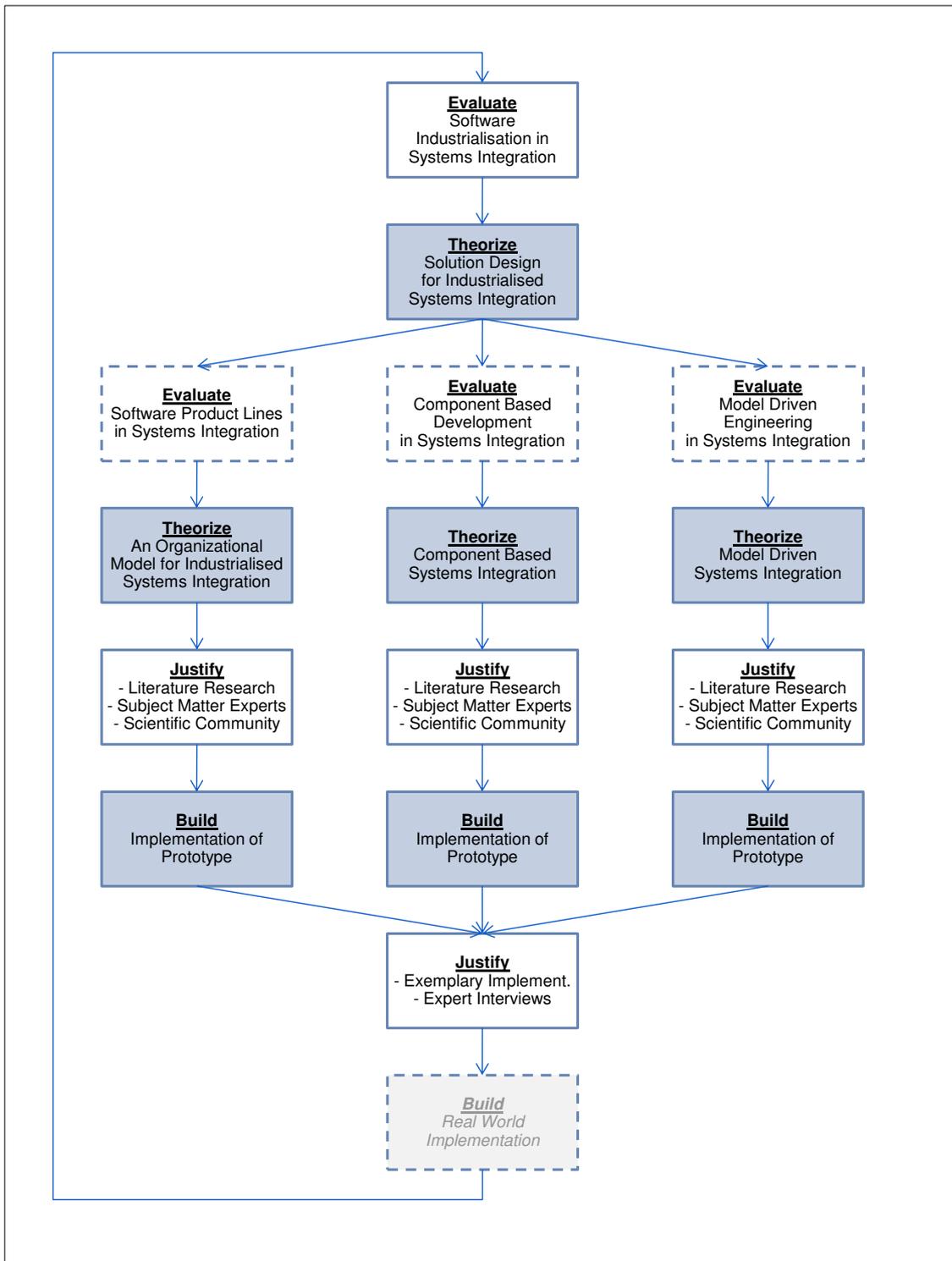


Figure 1-3 Decomposition of the research aim into IT Design Science Research Cycles

After identifying the overall problem it is further evaluated in chapters 2 (for software industrialization in general) and 3 (for systems integration in particular). As this evaluation represents solving a knowledge problem (i.e. identifying what the problem is), literature research and logical derivation of results was chosen as research methodology. The research furthermore depends

on the Author's professional experience in the field. A summary of the answer to this knowledge question is presented in sections 3.3.4 and 3.3.5.

As the evaluation of the problem resulted in three subproblems, a nested design science research cycle was applied to each of them in which the first stage was already completed by the previous evaluation. For Software Product Lines, Component Based Development, and Model Driven Engineering, sections 4.1, 4.2, and 4.3 solve a practical problem by theorizing (i.e. designing) a solution for particular stakeholders' needs. The subsequent knowledge question justifying (i.e. validating) the design was answered with literature research and a logical analysis of the design based on the author's own expertise from working in the field, as well as on expertise from subject matter experts. The concluding practical problem of building the design was solved by setting up prototypes. For each subproblem the results were furthermore published and presented to the scientific community. The respective papers can be found in the published papers section at the end of this thesis.

After theorizing a solution it must also be justified by ensuring that the proposed solution is valid in the environment it is intended for. In this research this was done in two steps. The first is described in chapter 5 and consists of an exemplary implementation of the overall solution. The implementation is based on the author's experience in the industry and personal discussions with subject matter experts. As not all aspects of the theorized solution could be implemented in a laboratory environment, an second validation based on qualitative interviews with additional subject matter experts was conducted.

Solving the final practical problem of the IT Design Science Research Cycle, building the overall solution, is left to the industry. As this step involves high risks and significant large scale changes to a company's methods of production and internal organization, it is seen as an individual research project. Although this limitation may seem as an incomplete research cycle it should be noted that complete cycles were executed for each subproblem of industrialised systems integration and that the overall design itself has been validated twice. The latter is also in

line with the overall research aim of (i.a.) proposing means to apply techniques of software industrialisation to systems integration within economic and technical constraints.

## 1.5 Thesis Structure

Chapter 2 examines currently-known concepts of software industrialization, reviews relevant literature, and depicts their implementation in practice. The concepts are considered with reference to the industrial key principles of specialization, standardization, and automation. It also describes a necessary paradigm shift of suppliers and customers due to less customer-specific development and more standardized products. Chapter 2 ends with a discussion of the prevailing issues and shortcomings and their approach in the scientific community.

Chapter 3 describes the fundamentals of systems integration with a focus on enterprise application integration. Based thereon, it discusses the elements of EAI within an integration meta model, which serves as the logical basis for the subsequent elaborations. The chapter concludes with the characteristics of systems integration and their affect on the application of industrial software development methods.

The literature review from the previous chapters along with practitioner discussions about practical issues and objectives were taken as the basis for the development of systems integration specific industrialisation concepts. Chapter 4 presents these findings in terms of a new organizational model, as well as specific approaches to component-based and model-driven systems integration. Close collaboration with practitioners ensures future relevancy and, together with the theoretical research work, concludes the initial development of the solution.

The reflection phase is conducted in Chapter 5 by utilizing an exemplary implementation as qualitative verification of the research. The previously developed concepts are being implemented at the example of a leading systems integrator active in the automotive domain. This includes a portfolio definition of the domain, a description of the required organizational model, an exemplary software product line including business domain and product-line specific core assets, as well as a description of the actual product development processes.

Chapter 6 discusses the research achievements from Chapters 4 and 5 with different subject matter experts from different industries. The chapter concludes with a summary of the interview results and their consequences for the research findings.

Chapter 7 reviews the research programme by discussing its achievements, describing its limitations, and pointing out further research required in the field of industrialized systems integration. The thesis concludes with an outlook on the possible future of industrialized systems integration.

## **2 Literature Review and Introduction: Software Industrialization**

As introduced in Chapter 1, concepts representing specialization, standardization, and automation are being researched and are beginning to find their way into practice. Before this can happen, a rethinking of the software development process in its entirety is necessary. This applies to both software suppliers and customers. Exactly what that means is described in the following, while Sections 2.2 to 2.4 explain the technical details behind these concepts. Section 2.5 provides possible reasons for a rather slow adoption on a larger scale in industry.

### **2.1 A Paradigm Shift for Suppliers and Customers**

When comparing the software industry to other industries, one may come to the conclusion that manufacturing software products is already industrialized. The duplication of software assets with the exact same quality, performance, and functionality can be done by clicking a button, exploiting economies of scale is a daily occurrence. The actual development of software may therefore be seen as a creative task, just like designing a car that cannot be standardized or even automated. However, exploiting economies of scale only works in mass markets at the expense of customization. Enterprise markets, where applications are being built upon customer-specific requirements, will never benefit thereof. Consequently, industrialization must go further by utilizing economies of scope instead of scale. Greenfield et al. (2004, p. 157) suggest to do so by using development assets, architectures and frameworks, as well as components with common functionality for the development of more than one product. They furthermore suggest that standardized platforms will appear and that products being based thereon will provide a customer specific collection of platform defined features. However, the approach must be limited as much as customers must understand that not every single detail can be requested for the final product. They may have to accept that, for example, a workflow system will authenticate users only against a Microsoft Active Directory Server, but not against a Novell system. Other platform limitations may mean that customers have to rearrange an existing business process. Halmans and Pohl (2003, p. 17) follow this assumption by stating that if they cannot agree to this

limitation, only a customized implementation at the respective cost is possible. They furthermore state that in the future, customers will have to decide if new components should be developed or if they can live with an 80% solution and benefit from a high degree of reuse leading to a lower purchasing price (Halmans and Pohl, 2003, p. 18, 2001, pp. 39 ff.). This assumption is backed by Clements et al. (2007, p. 237) who state that it is important to manage the experiences and requirements of the customers and guide them towards building a system that can be provided by an existing software product line. The author strongly agrees with these statements and sees them as a key prerequisite for industrialisation to become successful. A similar development can for instance be seen in SAP software. Here customers often follow predefined business processes and procedures in favour of lower cost.

While at first sight it may seem contradictory to shape the business according to the IT system, this paradigm shift comes with significant advantages. First and foremost, customers can expect a price decrease due to systematic reuse and more efficient development. Besides, products from a software product line are assumed to be of a higher quality since their components have been tested and used in other products and for other customers (Pohl et al., 2005, p. 13). Defects are more likely to be found and corrected, and improved components can be easily replaced. The collection of requirements from different customers and the resulting product features represent a “best of breed” approach from which new customers may benefit as well.

Besides the customers, software developing organizations have to go through an even bigger process of reconsideration. It seems to be a good idea to move away from a reactive customer-driven business model towards an anticipatory and market-driven one. Linden (2007, pp. 12 f.) suggests that industrial software suppliers shall no longer wait for their customers to define the requirements, but proactively analyse the customer’s industry. They identify the needs and issues in the field and build new functionality or innovative products for future market demands. Developing reusable assets can be seen as an investment which needs to be based on well-founded market research. If the assumptions were successful, systematic reuse within a product line will pay off after about three to four systems (Linden, 2007, p. 4; Pohl et al., 2005, p. 10). It must however be noticed that, especially in fast changing industries, assumptions about future

market demands come with a certain risk which sometimes may be higher than a potential benefit. When anticipating market demands and building applications accordingly, vendors must carefully specialize themselves to a discrete family of systems. As initially explained, specialization allows for more powerful tools and a better reusability of development artefacts.

To be able to systematically reuse product and production artefacts, software development should move away from monolithic construction in which constituent parts of a system are interwoven with each other and seen as a whole (Greenfield et al., 2004, p. 110). A software product should therefore be developed with reuse explicitly in mind, even if this may mean sacrificing certain functionality so that parts of it can be used in another context. A developer must not only have a particular customer's requirements in mind but also the scope of the product line. It may make sense to invest more time and budget in development for reuse if it fits into the overall strategy. In turn, developers must be aware of already existing assets which may be reused in a particular instance of a product. This, however, requires another rethinking and giving up the habit of assuming it is better to do things oneself than to rely on others. According to Greenfield et al. (2004, p. 114) the well known "not invented here syndrome" is one of the biggest obstacles in systematic reuse. This assumption is backed by Leitao (2009, p. 6) who state that developers often prefer to write functionality themselves, which in turn leads to a multiplicity of different implementations of the same concept, all prone to quality issues and incompatibilities. A company selling ten different products with a slightly different database connector faces ten times the risk of defects as compared to reusing the initial connector.

An internal paradigm shift is introduced by Clements et al. (2007, pp. 29 f.) and Linden (2007, p.7): Software production will be split into two major processes, designing and developing the overall architecture and reusable assets of a product family; and assembling the final products from these reusable assets. Developers responsible for the latter may feel displeased because of no longer "creating" something but assembling applications from other people's work. Allocating personnel to the engineering and production departments requires thorough change management. Developers will also no longer have the satisfaction of immediately seeing if their ideas successfully translate into executable code. This satisfaction is assumed to be the reason

that software developers identify themselves with mastering a specific programming technique instead of mastering e.g. financial systems (Selic, 2008, p. 388). In an environment where the majority of problems can be solved by reusing existing artefacts and only small functional parts or glue codes are being developed by hand, the self-concept of a developer may become constricted.

## **2.2 Specialization – Software Product Lines**

It seems to be very difficult or even impossible to determine how mechanisms for reuse or automation should be implemented in an arbitrary context. Systematic reuse must be planned for and cannot occur coincidentally. A software product line, therefore, defines “a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” (Clements et al., 2007, p. 5). The concept requires separating product development from product line development to utilize economies of scope for production artefacts such as processes, tools, but also executable program codes. By concentrating on a clearly delimited scope, production assets can be much more powerful.

### **2.2.1 Fundamental Concepts and Principles**

The introductory chapter separated software development into standard software and customer-specific individual software. Both have their advantages and disadvantages: standard software is inexpensive due to a large number of sold copies, but lacks diversification. Individual software is highly customizable, but at significantly higher cost. The combination of both, producing a large number of products tailored to individual needs, seems most favourable. However, the more standardized a product, the less attractive it may seem for potential customers: In this context Piller (2006, p. 114) defines the distance between the preferences of a customer and the actual product as the probability of buying. It is therefore important to find the feature combination that attracts the largest segment of customers to maximize product sales. In contrast to Piller’s statement, s software buyer’s preferences may be very specific or the required features

mutually exclusive; thus finding a feasible combination may be impossible. One way out of this dilemma is mass customization, i.e. “the large-scale production of goods tailored to individual customers’ needs” (Davis, 1987, p. 183).

Similar to individual software, more customization means higher technological investments, leading to higher prices and lower profit margins, which are both undesirable (Pohl et al., 2005, p. 5). Thus development of product platforms emerged, providing a technological basis for individually-developed products. Probably most advanced in this context is the automotive industry: nowadays cars are being built from a modular platform which serves as the foundation for, e.g., all models with a transverse engine (Volkswagen AG, 2010). Different customers’ needs are being satisfied with over 30 different models. Furthermore, each model may be individualized with a large number of extras. The platform approach offers a large variety of different products and thus a larger customer base, and at the same time reduces development and production cost. Axles, engines, electronic control units and other not directly visible parts are identical for a large number of models. Some features may even be implemented in software only. The author agrees with this analogy although the actual production is somewhat different. While in the automotive industry the production process eventually takes up higher efforts than the engineering process, with software engineering it is vice-versa. For a family of related products, a common software framework or architecture is being developed. First, the commonalities of the related products, including any anticipated future ones, are being defined and implemented as core assets. They can be developed from scratch or derived from earlier projects or products (Pohl et al., 2005, p. 7). Subsequently, individual features are being identified and mapped to variation points in the underlying architecture. Developing the framework or architecture and variation points is the effort intensive part of the process. The result is a variability model describing mandatory and optional or alternative features as presented by Pohl et al. (2005, pp. 7f.) or Lee and Muthig (2006, p. 57). The feature model is then instantiated in customer specific products. New feature requests are being analyzed and either included into the product line architecture or remain an individually developed extension. Figure 2-1 shows an example of a variability model. Although it seems as the process introduced by Pohl (2005) and

Clements et al. (2007) allows to implement any customer specific requirement if it is only added as a feature to the variability model it has to be noted that this is not necessarily feasible. Software product lines live from a clearly delimited context in which reusable assets can be most powerful. The broader a product line is, however, the more complex its assets will be.

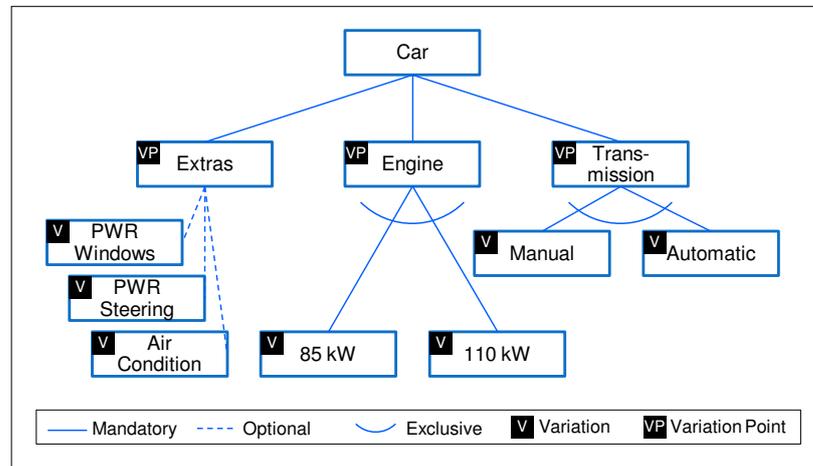


Figure 2-1: An exemplary variability model

It seems obvious that a shared platform needs to be maintained centrally; thus product line development and product development are separated. Software product line engineering produces product and production assets (also known as core assets), utilized by application engineering to produce a particular family member. Clements et al. (2007, p. 33) suggest that core assets not only include executable code artefacts but also shared architectures, reusable software components, development patterns, test cases, coding and documentation guidelines, development and project management tools and processes, as well as any other artefact being used for development. This clearly makes sense as source code represents only a small part of the work products in a development project. Also, new assets being developed during application engineering, e.g., due to specific customer requirements, are fed back into the product line and may become a core asset. It is therefore very important that customer-specific variability is developed with a potential reuse in mind. Besides developing new and deriving from already existing core assets of actual products, product line engineering also includes product and requirement management to identify any future needs of the markets the product line is serving. Figure 2-2 illustrates the

overall concept. It must however be noted that development for reuse does make sense for every feature as not all of them will be required by other customers as well.

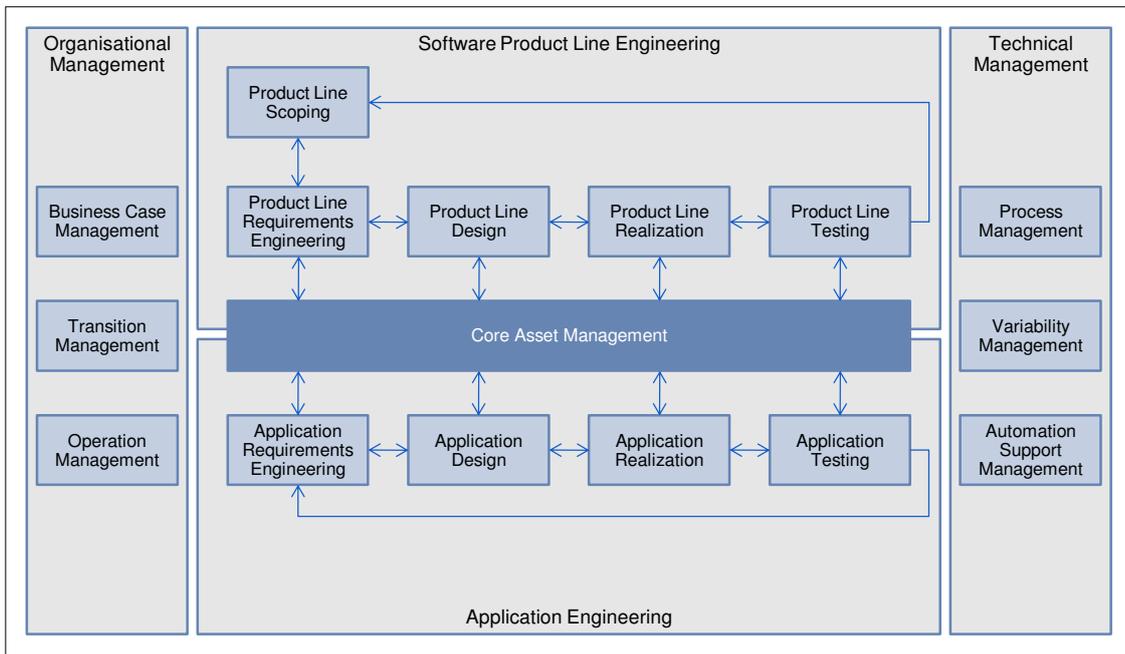


Figure 2-2: Software Development in a Software Product Line (Käkölä, 2010, p. 6)

The advantages of software product lines are manifold – first and foremost the reduction of development cost, which also is the driving force behind software industrialization (Pohl et al., 2005, p. 9). Despite upfront investments for setting up the software product line and developing core assets, an SPL is assumed to break even after three to four products (Leitner and Kreiner, 2010, p. 4; Pech et al., 2009, p. 293; Pohl et al., 2005, p. 10). Another advantage is founded in frequent reuse and quality assurance of core assets: Potential defects are much more likely to be detected and, once a solution is found, it can be distributed to all other products containing the same core asset (Clements et al., 2007, p. 20; Pohl et al., 2005, p. 10). This also results in reduced maintenance efforts as the defect for other products and its solution is known already. Due to reduced development time, frequent reuse also results in a reduction of time to market, at least for products making extensive use of existing core assets (Pohl et al., 2005, p. 11). The first few products, however, will most likely take slightly longer due to additional efforts for development for reuse. Other benefits are improved cost estimates, reduced cost for the cus-

tomers, and a reduced demand for highly skilled engineers (Clements et al., 2007, pp. 17–27; Pohl et al., 2005, pp. 11–13).

Concluding the above it can be said that the literature in the field of software product lines is mostly consistent with regards to the separation of engineering and production layers, as well as providing a framework for systematic reuse. Their estimations about the number of products required for a product line to break even seems to be too optimistic, though. From the author's experience in the field of software development in an industrial environment it seems unrealistic to achieve a positive return on invest within three to four products as suggested in the literature.

## **2.2.2 Software Product Line Approaches and Recent Advances**

The fundamental concepts and principles of software product lines have remained largely unchanged since their emergence about a decade ago. The first comprehensive approach was PuLSE, a methodology to develop software product lines, developed by the Fraunhofer Institute for Experimental Software Engineering (Bayer et al., 1999). It is defined around three main elements which are: deployment phases, technical components, and support components. Deployment phases describe initialization of a product line, the construction of the product line infrastructure, and the usage of the latter to produce applications. Technical components define the technical know-how of setting up and using the product line, while support components enable adaptation, evolution, and deployment of a product line (Bayer et al., 1999, p. 123). Similar to PuLSE, Clements et al. divide software product lines into a software development, technical and organizational practice area (Clements et al., 2007). The processes of each practice area and their implementation within a software product line are described, including implementation patterns to put theory into practice. Pohl, Böckle, and van der Linden follow a vertical approach by describing software product lines in the sequence they are most likely to be implemented, i.e., variability management, domain engineering, application engineering, and organizational aspects (Pohl et al., 2005). Both works differ only marginally in the description of the core processes of a software product line which are further detailed in the next section. Käkölä and Duenas (2006) draw on the works of Pohl et al. by presenting research results on

economic considerations, requirement engineering and variability management including legacy systems, architectural matters with regard to reference architectures and architecture adherence, system and system integration tests, as well as engineering issues in terms of tools and processes. A best-practice approach gathered from eight different companies during product line implementation was written by van der Linden, Schmidt, and Rommes (2007). In contrast to the previous, they focus on business-related aspects of software product-line engineering, such as markets, economics, or processes, and keep the technical details to the minimum required for clarity. In their book *Software Factories*, Greenfield et al. (2004) take software product lines as one building block for industrialized software development. Together with models and patterns, domain specific languages, components and services, and eventually code generators, they present their concept of a software factory which is “a software product line that configures extensible tools, processes, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework based components” (Greenfield et al., 2004, p. 163). For the latter the advantages of industrialization remain unclear in the context of the technical requirements Greenfield et al. propose. Their approach is still very technology centric and on a low level in terms of implementation. The advantages achieved on the one reuse side are given up with the additional implementation efforts.

In recent times, additional research has been conducted in various areas of software product lines. Most prominently, variability management and feature representation were in the focus of the literature. Chen et al. present an extensive overview of different variability management approaches, although they criticize the lack of experimental verification and comparison (Chen et al., 2009). Results are available on very specific research issues but cannot be generalized to guide practitioners in day-to-day business. Besides, Chen et al. claim a shortage of variability management processes, weak scalability of methods and tools, and a lack of test methods for variability-induced defects. Czarnecki et al. (2012) pick up some of these issues and provide a comparison of different variability approaches in their recent work. They differentiate between feature modelling approaches (capturing features from an end-user’s point of view) and decision

modelling approaches (a set of decisions to identify a product during application engineering), whereas feature modelling approaches are identified as beneficial in terms of generality and the ability to model commonalities as well. In addition to anticipated variability within the products of a product line, Galster and Avgeriou identify architecture variability as another important research question. Non-functional requirements, such as performance or security, may not only require changes to a few components but to the whole architecture, which still is an unsolved problem (Galster and Avgeriou, 2011). Lence, Fuentes et al. (2011) approach this issue by utilizing aspect-oriented programming. However, significant changes to the fundamental application architecture still require a manual redesign of the latter. As not all software product lines may be based on a green field approach, deriving product line architecture from existing assets, projects, or legacy systems has also been researched. Torres, Kulesza, et al. present an empirical assessment of six product derivation tools based on modularity, complexity, and stability, which leads them to suggest three of these as being most beneficial for product derivation (Torres et al., 2010). The fundamentals of a reverse engineering approach for legacy code that was not explicitly developed for systematic reuse was presented by She et al. (2011) based on the example of the Linux Kernel and FreeBSD. They claim that, although theoretical fundamentals are available and proven, reverse engineering of feature models is still far from being mature and readily available in tools. Similar experience is reported in works by Ryssel et al. (2011) and Hubaux et al. (2008), especially concerning feature combinations that require the manual definition of constraints to be unambiguous. Further research is being conducted on the application of model-driven engineering (Magalhaes et al., 2011; Istoan et al., 2011), aspect-oriented development (Tizzei and Rubira, 2011; Groher et al., 2008; Voelter and Groher, 2007; Conejero and Hernández, 2008), and agile-development methodologies (Blau and Hildenbrand, 2011; Mohan et al., 2010; Babar et al., 2009; Ghanam et al., 2009). Except for aspect-oriented development which is sometimes being used during application engineering within a software product line, these approaches are still under investigation and not yet seem mature enough for a broader adoption in the industry.

Most recent research is being conducted on software product-line evolution (Wu et al., 2011; Tizzei and Rubira, 2011; Estublier et al., 2010) and software ecosystems (Nöbauer et al., 2012; McGregor, 2010; Bosch, 2009) or both (Brummermann et al., 2012). Product line evolutions put great emphasis on the impacts of changing demands, new technologies and resulting changes of the architecture and feature models and, in consequence, the influence of already existing products derived from that architecture. Software ecosystems suggest conjointly advancing a software product line together with users, customers, and providers. The benefits are assumed to be a larger base for research & development activities by allowing external suppliers on the product line platform and quickly building a broad customer base to gain competitive advantage. From the author's point of view it is questionable if such a market will evolve for business software systems. The efforts and risks involved probably outweigh the benefits of sharing own developments.

To align the different directions of research and give practitioners and tool vendors a reliable basis, the International Organization for Standardization is working on a reference model for product line engineering and management, which is planned to be released in May 2014 (International Organization for Standardization, 2012; Käkölä, 2010). Until the latter has been released, the present work will rely on the descriptions of software product line and software product line engineering processes in the following sections.

### **2.2.3 Software Product Line Engineering**

The product line development process creates the platform for each software product line. This platform serves as the basis for all future products of the product line by providing specific core assets, such as joint feature specifications, architectures, reusable software components, design patterns, or process frameworks (Pohl et al., 2005, p. 22). The primary goals of product line development are to analyse the market, define the scope of the product line, its commonalities and variabilities, and reusable artefacts for product development (Pohl et al., 2005, p. 24). Considering literature on software product line engineering the processes are more or less equal. Pohl et al. (2005, pp. 21–23), Clements et al. (2007, pp. 56 ff), Linden (2007, pp. 59 ff), and

Balzert (2008, pp. 548 ff.) all differentiate between some sort of product management, requirements engineering, architecture, development, test, and integration. The individual processes can be summarized as follows:

- **Product Management** analyses the business domain a product line is active in and defines the products, their commonalities and variable features to be provided. It thereby checks the overall viability in terms of market stability, availability of resources, and organizational constraints. Product Management identifies typical business processes, associated problems and solutions, and evaluates and prepares this knowledge for further processing within the software product line. The process also analyses already existing assets and legacy systems and develops a long-term strategy with a product and technology roadmap describing future versions and their release dates.
- **Product Line Requirements Engineering** defines the scope of the intended software product line by specifying functional and non-functional requirements for the reference architecture and its implementation. The requirements should include all foreseeable products to be built within the product line. They are based on the product and technology roadmap and result in “reusable, textual and model-based requirements and, in particular, the variability model of the product line” (Pohl et al., 2005, p. 25). The scope evolves over time due to changing business domain requirements, as well as system decisions within the product line (Clements et al., 2007, p. 111). It is important to note that product line requirement engineering does not specify a single product but the requirements of the product line as a whole, comprising all derivable products.
- **Architecture Design & Development** transforms the scope defined in requirement engineering into a technical architecture for the product line and its products. The architecture decomposes a software system into common and variable functional parts, defines relationships and interfaces, and establishes rules for their implementation. The process therefore takes the product line requirements and the variability model and produces the technical reference architecture and a “refined variability model that includes so-called internal variability (e.g. variability that is necessary for technical reasons)” (Pohl et al., 2005, p. 26). It will

also call for a set of components to be defined, implemented, and integrated (Linden, 2007, p. 58). At this point it may make sense to identify suitable commercial off-the-shelf (COTS) or legacy components and align the architecture accordingly, saving time and effort in subsequent core asset development.

- **Core Asset Development** provides the “detailed design and the implementation of reusable software assets, based on the reference architecture” (Pohl et al., 2005, p. 242). It includes software components, glue code, variability mechanisms, common processes, development tools, and any other reusable assets required for product development. Core asset development may also mean buying commercial off-the-shelf components and adapting or mining already existing ones. Core components with built-in variability must also come with a definition on how they can be instantiated in a specific context (Linden, 2007, p. 85). The result of core asset development is a collection of loosely coupled, configurable components, not a running application (Pohl et al., 2005, p. 27). Also, all other artefacts required to build a product from the product line are created in this process. At this point it seems questionable if really all product artefacts can be produced in advance or if it rather is an iterative process.
- **Domain Testing** develops test cases and inspects all core assets and their interactions against the requirements and contexts defined by the product line architecture. Domain testing also includes validation of non-software core assets, such as business processes, design patterns, product line architecture or development policies. It only tests reusable core assets but not complete applications, as during the software product line engineering processes no actual products are built. Although testing the individual components is an important step which helps with quality assurance the issue here is that most defects only occur in an integrated environment which can be found in the final application only.
- **Software Integration** in the context of product line development occurs during pre-integration of several software components. They form blocks of functionality common to all products and contexts of a product line, e.g., a certain business process consisting of several components and a database. Furthermore, the integration process ensures the inter-

operability of all reusable assets and provides the required integration mechanisms. This process must not be confused with systems integration as defined in Chapter 3.

Software product line engineering can be compared to building a production plant in other industries, including tools, machinery, and prefabricated assembly parts. The primary artefacts are a product roadmap, the domain requirements, the product line architecture and variability model, the domain realisation artefacts, and the domain test artefacts (Pohl et al., 2005, pp. 28–30).

#### **2.2.4 Application Engineering**

During actual product development, knowledge and reusable assets, such as business functionality components, are captured to include them in the Software Product Line for future products. The primary objective is to build applications by exploiting commonalities and variabilities while reusing as many product line assets as possible. According to literature (Pohl et al., 2005, pp. 21–23; Clements et al., 2007, pp. 56 ff; Linden, 2007, pp. 59 ff; Balzert, 2008, pp. 548 ff.), the processes of product development can be summarized as follows:

- **Application Requirements Engineering** collects customer expectations and derives the requirements of an application to be developed within the software product line. It analyses variances from the product line’s core assets and decides together with the customer whether to implement application-specific assets or to accept a functional trade-off. The success of the overall product line heavily depends on application-requirements engineering, as here the amount of reuse and thus efficiency of the product line architecture is decided. Potentially recurring requirements and thus application specific components should be fed back to the product line requirements engineering process of software product line engineering to integrate them into the software product line for further reuse.
- **Application Design** translates the customer requirements into particular application architecture from the overall product line architecture and its reusable core assets. Abstract variation points are instantiated and product specific requirements and components added. Application design defines how and with which reusable core assets the product will be

realised and may be compared to a detailed technical concept in single system development. It thereby has to adhere to the rules and regulations set forth in the reference architecture (Pohl et al., 2005, p. 32).

- **Application Realisation** is the process of assembling the application from core assets within the application architecture, binding their variability points according to requirements and design, and implementing application-specific assets. To cover similar products in the future, development of the latter should always occur with reuse in mind and must follow the boundaries set by the product line architecture. Compared to single system development, integration efforts are decreased due to predefined architectures and integration mechanisms (Clements et al., 2007, p. 122).
- **Application Testing** ensures sufficient quality of the end product. Although the components have been tested during product line development, instantiated variability points and interaction with other components must also be covered. This is necessary as during domain testing it is impossible to cover all potential combinations of core assets. The product testing process is also responsible for ensuring proper alignment with the requirements imposed by the software product line, as well as testing application-specific components not being part of the product line.

### 2.2.5 Organizational Aspects

With increasing size and complexity of an enterprise, the division of labour and its assignment to organizational units becomes increasingly important (Klimmer, 2007, p. 22). The same principle applies to software vendors, providing different and highly complex solutions within software product lines. Literature depicts several forms of organization applicable to Software Product Lines. Bosch, for example, suggests development departments, business units, and (hierarchical) domain engineering units (Bosch, 2001, pp. 91–97). Clements & Northrop (2007, pp. 314–320) pick up Bosch’s suggestions and differentiate between core asset and product development, suggesting an evolving structure. In their book, “Software Product Lines in Action”, van der Linden et al. (2007, pp. 66-76) present divisional, functional, and matrix structures, which are well known from manufacturing industries and match some of Bosch’s sugges-

tions. Subsuming the above, the following points further explain currently-known forms of organization:

- The **development department organization** does not instantiate any permanent organizational structure. Members of the department are dynamically organized in projects as required and may work in domain engineering, application engineering, or supportive tasks. The advantages are that developers work where they are needed most and efficient communications between each others. After completing one project they move on to the next. The disadvantage is a lack of consistency and the fact that developers may not be available in case of subsequently discovered defects or necessary enhancements. The model, furthermore, is suggested for only up to 30 developers due to a lack of scalability (Bosch, 2001, p. 92).
- In a **functional organization**, work is divided by phases of software development, i.e. requirements engineering, design, implementation, and testing. Applied to Software Product Lines, an additional unit for domain engineering is implemented. Developing a particular product involves all units and requires thorough communication. The advantage lies in the resource flexibility for different projects within a functional business unit. According to Linden (2007, p. 70), this leads to two important effects: Firstly, the integrity of architecture is more likely to be ensured, as the person responsible for a certain part also makes the changes to this part. And secondly, working on multiple projects within one functional unit encourages reusability, as engineers and departments may experience a personal benefit. However, one has to keep the cultural challenges in mind not yet considered by Bosch: Developers primarily working for product development and not product line engineering may have a problem with constantly reusing other people's work instead of being creative themselves.
- A **divisional organization** divides the work by objects of work, e.g. different products, services, or markets. With respect to Software Product Lines, there would be a domain engineering department and several application engineering departments. According to Linden et al. (2007, p. 67), this is "the most common way to structure the organisation for

software product line engineering”. The advantages are that responsibilities are clearly assigned and that all activities required for developing a product are located in the same unit. The two major challenges for this form of organization are funding the domain engineering unit, as application development units may not obtain a direct benefit, and a potential lack of functional interaction between domain and application engineering units, as additional alignment efforts may be considered unnecessary overhead. Here again cultural issues must be taken care of in order to be most efficient.

- Similar to the divisional organization, a **business unit organization** is “responsible for the development and evolution of one or a few products in the software product line” (Bosch, 2001, p. 92). Several related business units share reusable assets in a distributed manner, i.e. each unit may adapt, enhance and make new versions of an available asset. There is no central domain engineering unit; the initial set of shared assets is usually developed in an initiation project. Asset ownership may be unconstrained, fixed, or based, depending on usage. According to Bosch, this form of organization is reasonable for up to 100 developers. The primary advantage is an increased control over the product line evolution. However, as no dedicated domain engineering unit is in charge, conflicts may occur between the different business units regarding further evolution.
- The **domain engineering unit organization** separates the development of products from the development of product line assets. The model is applicable for larger organizations in which a software product line typically employs around 100 or more engineers (Bosch, 2001, p. 95). Bosch further distinguishes two alternatives, i.e. one in which a domain engineering unit is responsible for all product line assets, and one in which multiple domain engineering units exist, each responsible for a subset of assets. He suggests implementing the latter if the domain engineering unit outnumbered about 30 software developers. Advantages are reduced communication overhead in addition to application developing business units not being able to add product-specific features to shared assets. The disadvantages are an increased management effort and personnel overhead.
- A **hierarchical domain engineering organization** aims at large or very large software product lines, exceeding by far 100 software developers in product developing business

units and over 30 in the domain engineering unit (Bosch, 2001, p. 96). It separates the management of shared assets into a hierarchical structure. Application-developing business units may add additional shared assets which are required by a subset of the family members only. The advantages of hierarchical domain engineering organization are the ability to cover very large and complex software product lines and its ability to scale up to several hundred software developers. The downside is the considerable overhead introduced by several domain engineering units, as well as a potential inflexibility and a reduced reaction time to changing market requirements.

- The **matrix organization** combines the separation of functions and the division of work by objects through overlapping a horizontal and a vertical structure (Linden, 2007, p. 70). Specialists from different functions form a virtual unit to develop a particular product. In this form of organization, fundamental or strategic decisions are made on the functional level, while operational decisions are made on the divisional level, i.e. per product to be developed. The structure supports the interaction between domain engineering and application engineering, as both objectives must be considered by the respective engineers. However, the management structure becomes very complex and may lead to conflicts due to deviating objectives between horizontal and vertical management.

### **2.3 Standardization – Component-Based Development**

One of the first ideas of using industrial principles in software development came up in October 1986 at the NATO conference on software engineering. In his contribution “Mass produced Software Components” (McIlroy, 1969, pp. 138–155), McIlroy suggested developing applications by assembling previously produced components, as most of a software’s functionality has already been developed or will be required in many other applications as well. This idea can be mapped to the industrial principle of standardization which is the foundation for the exchange of artefacts and systematic reuse (see section 1.1.2).

### 2.3.1 Fundamental Concepts and Recent Advances

In his book about component software, Szyperski (2002, p. 27) defines a software component as follows:

*“A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”*

A similar, yet broader, interpretation is available in Sametinger’s book about software engineering with reusable components (Sametinger, 1997, p. 68):

*“Reusable software components are self-contained, clearly identifiable artefacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status.”*

Further definitions of the term can be found in literature. Most of them are similar to the above but with a wider scope, i.e. they also include macros, templates, processes or other development-related artefacts (Nierstrasz and Tschritzis, 1995, p. 5; Jacobson, 1998, p. 166; Orfali et al., 1996, p. 34). As in the context of this work, such artefacts are covered within the core assets of software product lines (see section 2.2.3); the present work will adhere to the delineation of the term from Szyperski.

Similar to manufacturing industries, systematic reuse requires a clearly delimited context (Andresen, 2004, p. 293). It is, for example, much easier to build graphical user-interface components for Microsoft’s .NET platform than within an arbitrary context (Greenfield et al., 2004, p. 125). By using current component standards, it is possible to encapsulate business logic within reusable software building blocks. The context in which this occurs can be set by architecture frameworks and Software Product Lines. The latter also define a delimited scope to employing reusable components for the development of new product line members. According to Andresen (2004, p. 2651), current Component Based Development (CBD) standards define the requirements a component has to fulfil from a syntactic and semantic point of view. They fur-

thermore define interface specifications, component allocation and component interaction across different programming languages and platforms. Greenfield et al. (2004, pp. 246 f.) extend the term by including the underlying architecture which can also be provided in a way that it becomes possible to completely implement the commonalities of a certain product line, while allowing the “plugging in” of customer specific requirements and variability. From the author’s point of view the latter is mandatory as only architectures and frameworks allow a systematic reuse of components.

The term “component” is used in a variety of ways in software development and should therefore be defined in further detail. What is a component? In addition to the definition by Szyperski (2002, p. 27), the following attributes (see Andresen, 2004, pp. 20–21) of a software component can be identified in the literature:

- It is a discrete software artefact with clearly defined properties and interfaces which can be embedded within a specific software environment. An example would be an Enterprise Java Bean (EJB), being deployed in an EJB container and providing its services to other EJBs.
- It possesses specific (business) knowledge required to complete a certain business process or task. An example would be a component verifying a payment against a credit card institute for an online shop.
- It provides this knowledge via well-defined interfaces. To be sufficiently reused, this knowledge or functionality must be available via clearly documented interfaces. For example, payment verification may define credit card number, card holder name, validity, and amount as input variables, and transaction status and number as return values. Also pre- and postconditions, as well as error handling or other non-functional aspects, are important.
- It comes with a precise specification detailing the action that will be carried out, including performance, timing, and interaction with the framework or other components. The specification also includes the requirements of ensuring that the component reacts in the specified manner. The specification usually does not include any information about its internal implementation.

- It communicates and interacts with other components and systems. Without such interaction or communication, a system could not be built by assembling different components. A component must therefore at least be invocable from an external source.
- It can be independently packaged and reused, meaning that for either the component itself or an interacting set of components, it must be possible to package and distribute them as an autonomous, reusable unit. The internals of this package do not need to be visible to the utilizing system or organization.
- It adheres to a specific component model, which “defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model” (Councill and Heineman, 2001, p. 7). In addition, the component infrastructure is “a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly-defined performance specifications” (Councill and Heineman, 2001, p. 7).

Now that a component is technically defined, the question is how big or small it should be in terms of functionality. To be on the safe side, developers could implement all required dependencies into the component itself, making it able to run on every machine without any prerequisites (except for processor architecture, maybe). This, however, would lead to very heavy components with a great deal of assistive functionality being redundantly included at the price of lower performance and an increased risk of defects. Szyperski et al. (2002, p. 45) point out that very lean components may lead to an “explosion of context dependencies” as a significant amount of assistive functionality is expected from an external framework. This statement appears to be even more valid if standardized architectures or frameworks are not present. Szyperski et al. (2002, p.45) furthermore state that if the deployment environments were stable, this would not be a problem; however, as technology advances, deployment environments advance as well and may not be compatible for all times. A very lean component with many external dependencies and interfaces may therefore only be reused in very few contexts: “Maximizing reuse minimizes use” (Szyperski et al., 2002, p. 45). In conclusion it can be said that when de-

veloping within a software product line, an organization should carefully decide about component granularity, utilized component models and platforms, and component architecture. There is no general rule, but a long term strategy may prevent frequent technology changes and therefore reuse of rather lean components becomes economically viable. If markets demand up-to-date technology, a heavier implementation may ensure a longer reuse, even in changing contexts.

### 2.3.2 Component Models and Platforms

The IT landscape provides several implementations of component based development which are primarily concerned with the technical mechanisms of enabling components to communicate with each other and providing a framework for their implementation. The most prominent representatives are the CORBA Component Model, Microsoft's Component Object Model and .NET Framework, and Sun's Enterprise Java Beans.

- The **Corba Component Model** is based on the Common Object Request Broker Architecture defined by the Object Management Group. It provides a framework for distributed object- and component-oriented applications and consists of four building blocks (Andresen, 2004, pp. 273 f.). The Component Model defines the basic structure, properties, interfaces, references, assembly, and deployment of a component (Andresen, 2004, p. 273). The Component Implementation Definition Language describes structure and state of implemented components. Developers may use it to automatically derive skeleton code out of it. The CORBA Implementation Framework as the third building block provides a framework for component implementations. The framework uses the interface and component definitions written in Component Implementation Definition Language to generate skeleton code for their navigation, identity, instantiation, and lifecycle management, for instance. Together with the component server, the Corba Implementation Framework provides the required infrastructure. By using a generic interface definition language, CORBA components are also platform and language independent. The Container Programming Model represents the runtime environment for components and is implemented in a deployment platform, such as an

application server. It provides internal and external APIs to interact with the components and describes how these can be included in other applications and how components may interact with the CORBA infrastructure. Besides these four building blocks of the Corba Component Model, CORBA itself provides further infrastructure such as brokerage, messaging services, and transaction handling, for instance.

- The **Component Object Model (COM)** is an architecture developed by Microsoft for the communication between components based on a Windows operating platform. They can be used by multiple applications through standardized interfaces and are based on dynamic link libraries (via memory pointers) or executables (via local procedure calls). To allow the usage of remote components, Distributed COM (DCOM) also allows invoking components through remote procedure calls (Schryen and Bastian, 2001, p. 66). With COM+, it was further enhanced and is now based on the Windows Distributed interNetwork Applications Architecture, aiming at scalable, distributed systems (Andresen, 2004, p. 279), which are based on client server concepts and web services. Similar to CORBA, interfaces of components are described platform independently with the Microsoft Interface Definition Language. To allow interoperability between different programming languages, any data is converted into a normalized format. However, support for proxies and component discovery is primarily available for windows platforms (Schryen and Bastian, 2001, p. 69).
- The **.NET Framework** was developed by Microsoft to develop stand-alone, component, and web-based applications. Similar to Java, it is based on an intermediate language, which is then executed just in time by the Common Language Runtime (similar to the Java Virtual Machine). It furthermore offers a Common Type System, allowing the implementation of components or applications in almost any programming language capable of being compiled to the Microsoft Intermediate Language. .NET-Components support the same means of interaction as DCOM/COM+ as well as web services. In addition, Microsoft .NET provides a wide variety of supporting services and tools (Andresen, 2004, p. 282) such as the BizTalk Server for integration and orchestration of web services and components, several programming languages (C#, Visual Basic, C++, etc.), programming and component libraries, and integrated development environments.

- The **Enterprise Java Beans** model is a component-oriented framework for distributed information systems. It is based on the Java 2 Enterprise Edition library and allows the invocation of remote methods via Web Services, Internet Inter ORB Protocol based on CORBA, Native Java, and JMS (Java Message Service). While Java and JMS require a Java implementation on both sides of the connection, Web Services and Internet Inter ORB Protocol allow platform-independent communication between different (also non-Java) components. It furthermore offers a great reduction of complexity by providing services such as transaction handling, distribution, and security to the developers. The infrastructure of EJBs primarily consists of an EJB Server, EJB Containers, and EJB Components. The server provides the runtime environment for one or more containers and shields them from the characteristics of the operating system. It is therefore not platform-independent. An EJB Container provides the infrastructure for exactly one EJB Component and its instances and provides it with the supporting services as previously mentioned. Any interaction of the component with the outside world is also handled by the container. Enterprise Java Beans are thus platform- (as long as an EJB Server is available), but not language-independent. They can, however, interact with components of other languages or frameworks on a binary level.

### **2.3.3 Component Architectures and Frameworks**

As for all software development projects, a proper architecture is indispensable for success (Vogel et al., 2009, p. 32). According to Vogler (2009, p. 11) it defines the fundamentals of any application to be developed and provides an overview about otherwise very complex requirements definitions. Vogler's statement is backed by the literature introduced in the previous section: To enable proper reuse of components without frequently over- or underlapping functionality, common and aligned component based architecture must be established. It should be aligned to component-based development and different component standards (such as CORBA, EJBs, or Microsoft .Net).

Unfortunately, only very little of the literature covering architectures specifically designed for component-based development is available. Most only discuss frameworks of current imple-

mentations such as CORBA, EJBs, or .Net (Schryen and Bastian, 2001, pp. 52 ff; Sametinger, 1997, p. 121; Zwintzsch, 2005, p. 80). Not an architecture itself, but an approach to define such architectures from scratch and across multiple corporate application systems can be found in Robra's book about modelling component-based software architectures (Robra, 2007). In his book on component-based software development with MDA, UML 2 and XML, Andresen (2004, p. 77) inter alia develops a generic component-based development framework. It is aligned with ANSI/IEEE 1471, a recommendation for architecture definitions of software systems. From the author's point of view, it is the currently most tangible approach available in the field. It will therefore be reviewed in further detail.

Andresen's Framework primarily consists of four sub-architectures, each covering a different level of abstraction (Andresen, 2004, p. 72). This partitioning equates to the concept of Model-Driven Architecture (MDA), which raises the level of abstraction to bridge the gap between the actual problem and the implementation of its solution and is part of the discussion in section 2.4. The levels of abstraction used by Andresen are defined as follows:

- **Business Architecture:** The topmost layer provides an overview of the system to be developed. It represents the required business logic and serves as a starting point for the identification and development of components (Andresen, 2004, p. 105). It is further subdivided into a business, requirements, process, and conceptional view, each representing a particular context. The artefacts of each view can be anything from business cases, functional requirements, process models, or UML dataflow charts. They all contribute to obtaining a holistic picture of the system in scope. The Business Architecture maps to the Computation Independent Model in a Model-Driven Architecture.
- **Reference Architecture:** The second layer represents a logical, platform and technology independent model of a system's building blocks. It depicts the flow of information between these components in terms of interfaces, dependencies, and contractual agreements (Andresen, 2004, p. 133). The Reference Architecture is further subdivided into a system layer, interaction, and specification view. The first view, for instance, describes process, entity, and service components in terms of UML diagrams. The second view describes the ele-

ments required to interact with the system, such as interfaces, hierarchies, user interfaces, e.g. with UML interaction diagrams. The specification view covers the interaction of a system's components by defining patterns, rules and regulations, and architectural aspects. The Reference Architecture can be equated to the platform independent model in a Model-Driven Architecture.

- **Application Architecture:** On layer 3, the reference architecture is transformed into a platform-specific one, based on a particular component standard such as Corba, EJBs, COM+, or .Net. Andresen further subdivides it into a layer, integration, and implementation view (Andresen, 2004, p. 175). The first one defines the layered architecture of the system together with the layer interactions. The second view defines how certain components are supposed to interact, for instance via web services or middleware solutions. The third view, implementation, provides the detailed and platform-dependent specification of components, services, files, or similar items required to implement the system.
- **System Architecture:** The lowest layer addresses the actual implementation of the components specified on the layer above, the physical interconnection of existing systems, and the distribution among the IT infrastructure. It is therefore subdivided into an infrastructure, system, distribution, and runtime view (Andresen, 2004, p. 227). The infrastructure view provides all artefacts required to run and deploy components such as containers, servlet engines, or middleware. On the system view, these artefacts are utilized to generate source code of components, define deployment descriptors and to create additional artefacts required to interconnect to other systems. The distribution view allocates the previously-created components and artefacts to the physical hardware and deploys them there. The final view, runtime, defines the state of the system during start-up and runtime and includes installation scripts, start-up procedures, performance specifications and others. The System Architecture can be mapped to the Enterprise Deployment Model in a Model-Driven Architecture.

The following figure provides an overview of Andresen's overall framework, consisting of the four sub-architectures and their respective views.

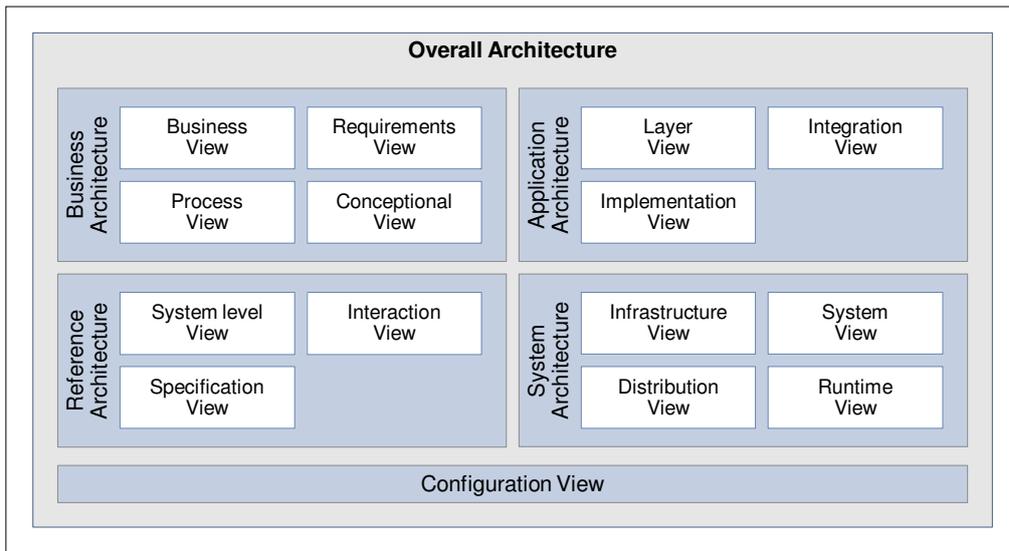


Figure 2-3: Component Based Architecture Framework by Andresen

An architecture framework similar to the one previously shown constitutes a fundamental building block for Component-Based Development (Andresen, 2004, p. 67; Vogel et al., 2009, p. 32). Without clear structures and guidelines, it must be assumed that component development will lead to a proliferation of functionality and features, which makes a systematic reuse more and more complicated, yet not impossible.

### 2.3.4 Component-Based Development Approaches and Recent Advances

After having introduced various component models and frameworks, the present section introduces component-based development approaches utilizing such models and frameworks to build business applications.

One of the first approaches was developed by Brown at the Software Engineering Institute at Carnegie Mellon University and adapted by Haines et al. (1997). According to their work “Component Based Software Development / COTS Integration” (Haines et al., 1997), component-based development can be subdivided into four major steps. During the first (component qualification), existing components are discovered and evaluated against their potential to be deployed in another context. The result of the qualification defines whether certain functionality can be integrated from existing artefacts or must be manually developed. Component qualification may include functional and non-functional requirements such as algorithms or interfaces

and quality or performance. If required, suitable components can be adapted in the second step. Adaptations could be wrappers for underlying platforms or the integration of certain aspects as security concepts, for instance. Components can be categorized into white-box, grey-box and black-box ones (Haines et al., 1997) as follows: The former allow significant changes to the component at the cost of compatibility and ability to replace. Adaptations to the latter have very little negative side effects, but may not allow the required flexibility. Grey-Box components do not allow changes to their source code but provide extension languages or APIs to adapt them to specific requirements. In a third step, the previously qualified and adapted components are assembled to a new application. This assembly is usually built on frameworks which provide the implementation base for the components. The third step of Haines et al. is backed by Crnkovic et al. (2002, p. 38) who state that it is „[...] very important that there exists a context in which [...] [components] can be used.“ Frameworks, furthermore, overlap with patterns, which „[...] define a recurring solution to a recurring problem“ (Crnkovic et al., 2002, p. 38). The final step focuses on maintenance and enhancement. Components are replaced with their improved or debugged versions or with totally new ones, combining the functionality of previously existing ones (Haines et al., 1997).

One of the most comprehensive books on building large-scale component systems is Herzum and Sims “Business Component Factory” (Herzum and Sims, 2000). It is a methodology to model, analyse, design, construct, validate, deploy, customize, and maintain large-scale distributed systems. One of its principles is “that any software artefact in a system should be defined in one and only one place, and it should be reused as many times as necessary” (Herzum and Sims, 2000, p. 71). From the author’s point of view this principle perfectly fits the concept of Software Product Lines, in which a distinctive unit or team is responsible for a particular set of core assets, such as reusable business components. The approach of Herzum and Sims consists of five dimensions: the level of component granularity, architectural viewpoints, the development process, distribution tiers, and functional layers. Component granularity subdivides reusable components into distributed components, providing simple tasks and functionality, business components, implementing an autonomous business concept or business process, and busi-

ness component systems, representing a complete system out of several business components and additional resources (Herzum and Sims, 2000, p. 72). Architectural viewpoints constitute the conceptual framework required to develop a component-based application into technical, application architectural, organizational, and functional aspects (Herzum and Sims, 2000, p. 73). The development process viewpoint consists of rapid component development for designing, building, and testing an individual business component; system architecture and assembly for architecting, assembling, and testing a complete system; as well as federation architecture and assembly for architecting, assembling, and testing a federation of systems (Herzum and Sims, 2000, p. 247). The distribution tier allocates components to a user, workspace, enterprise, and resource tier, depending on their functionality within the system (Herzum and Sims, 2000, p. 118). The fifth dimension defines utility business components, entity business components, process business components, and auxiliary business components as four broad functional layers, depending on the services a component is providing (Herzum and Sims, 2000, p. 50).

Andresen suggests a component-based development model utilizing model driven architecture (MDA), unified modelling language (UML) and the extensible markup language (XML). The author describes it as architecture centric-, quality- and communication-oriented, and is characterized by an iterative and incremental approach, and agile modelling and development (Andresen, 2004, p. 12). The architecture centricity is obtained by developing a system from three different viewpoints (Andresen, 2004, pp. 175 ff.): The layer view describes components based on their services and differentiates between components for presentation (e.g. user interfaces or session handling), controlling (e.g. workflow, process, or queue controller), business logic execution (e.g. process, entity, or service components) and integration (e.g. wrapper, connector or data access components). The integration view is concerned with the integration of existing components and systems, mostly by utilizing wrapper or connector components and XML as the data exchange format. The implementation view is responsible for the actual implementation of the different components from previously defined models, based on the model-driven architecture. In Andresen's approach, the model-driven architecture is used to separate the business logic from system-specific implementations by using textual or visual

models on different levels of abstractions. Once a problem solution is defined on a higher level, it is transformed into a more specific model by adding technical details. Transformations continue until eventually executable artefacts are derived. Andresen applies agile development principles to architecture and component modelling and implementation to quickly obtain the system artefacts, react to changing requirements or unforeseen issues and simplify communication within the development teams. Quality centricity is achieved by using conventional quality assurance principles and metrics, applied to the respective components. The approach uses MDA to design and model the overall system architecture and individual components, and UML for visual representation. It does not cover model transformation engines or code generators, or round trip engineering of the individual models.

In their work “Building Systems from Commercial Components”, Wallnau et al. (2002) claim that “the principal source of risk in component-based design is a lack of knowledge about how components should be integrated and how they behave when integrated” and furthermore state that “all component-based roads lead to the commercial component marketplace” (Wallnau et al., 2002, p. xvi). Thus their approach clearly focuses on design and modelling component-based systems based on commercially-available components (Wallnau et al., 2002, pp. 30 ff.). They do so by introducing component ensembles as abstractions exposing interactions and dependencies between sets of components, similar to patterns in programming languages. The second concept contains blackboards, semantically instantiating a previously defined component ensemble and documenting current knowledge and remaining design questions and risks. A component ensemble may, for instance, generically define a secure web browser and web server interaction, while a blackboard instantiates an actual business process on this ensemble, such as booking a flight. Both tasks are iteratively performed and feedback on each other within a risk-driven discovery process, called  $R^3$  (risk analysis, realize model problem, repair residual risk). The process inter alia results in prototypes called model problems. In case of several different component ensembles collaborating within a system, a design space is introduced, defining ensemble interactions and dependencies. The design space furthermore covers future market trends, such as new technologies or component releases. The extensive design artefacts along

with the model problems are being used by software engineers to either mine commercially-available components or develop new ones, resulting in model solutions, defining the blueprint for the final system assembly. Unfortunately, component marketplaces as anticipated by many authors at the beginning of component-based software engineering have not been formed for a variety of reasons (dos Santos and Werner, 2010, p. 135). However, the approach presented by Wallnau et al. remains promising, especially in large software developing enterprises with a potentially large component base. A similar method is pursued by Lingyun et al., utilizing a specification language to describe components and interaction graphs and isomorphic trees of interaction to simulate component assembly and integration (Lingyun et al., 2010).

Recent work in the field concentrates on self-management or self-configuration of component-based systems, as for instance in Ehlers et al. (2011), Adler et al (2010), Rudametkin (2010). The objective is to automatically take over functionality of unavailable components by reconfiguring the remaining ones according to predefined rules. Another focus area is quality assurance, performance prediction, and non-functional requirements (Bertolino et al., 2011; Roy et al., 2011; Meng and Barbosa, 2010). Recently, research on component-based development has also shifted towards services and service-oriented architectures (Cubo and Pimentel, 2011; Ruz et al., 2010; Rudametkin et al., 2010). Similar to components, services provide clearly defined interfaces and functionality and are made available over the (inter)net (Crnkovic et al., 2011, p. 22). The advantages are that with standardized and widely available technologies, such as HTTP and XML, orchestration of systems utilizing such services became much easier. Disadvantages can be found in low performance or unknown data security and privacy, especially when being used over the internet.

## **2.4 Automation – Model-Driven Software Engineering**

The final aspect of industrialization, automating repetitive and simple tasks, may be achieved with Model-Driven Engineering (MDE) and was initiated by Computer-Aided Software Engineering (CASE) in the 1980s (France et al., 2007, p. 6). It aims to raise the level of abstraction of software engineering to fill the gap between the problem solution to be implemented and the

actual technology utilized to do so. Once a suitable level of abstraction is found, the description of the solution has to be refined by adding previously-omitted detail until an executable implementation is available. The distance between the description and technical implementation characterizes what is commonly referred to as abstraction gap (Selic, 2008, p. 381; Pham et al., p. 4; Frankel, 2003, p.8, 60). Once a context-free description of this gap is found, model transformation engines and code generators can possibly create an executable solution.

### **2.4.1 Fundamental Concepts and Principles**

CASE as the first approach towards model-driven engineering was too generic to precisely describe a solution and mapped poorly to the underlying technologies. Graphic representations were too complex and could not always precisely describe what was needed. The result was very complex source code which had to be altered by hand (Selic, 2008, p. 382). The corresponding models were out of date very soon, as the CASE tools could hardly depict manual changes to the code (Stahl and Bettin, 2007, p. 44).

Model-Driven Engineering today is much more advanced, overcoming the problems discovered with CASE tools. Using visually-represented models as a description of software, it aims to raise the level of abstraction in order to fill the gap between the problem solution and the technical implementation (Greenfield et al., 2004, p. 142). According to literature it can be further subdivided in two major streams: Model-Driven Architecture (MDA) and Model-Driven Development. The former is a specification of the Object Management Group and describes a framework for a model-driven approach, separating the business and application logic of software systems. The latter deals with the automated creation of software which implies that a preferably large number of artefacts is derived from formal models (Beltran et al., 2007, p. 11). Both approaches overlap but also complement each other. MDA focuses on the overall architecture and a framework, while Model Driven Development focuses on the actual implementation of models without limiting itself to any existing standards. To prevent possible misunderstandings, it should be noted that the Object Management Group developed the Query View Transforma-

tion standard for model transformations, which was initially released in 2008 and updated in January 2011 (Object Management Group, 2011), as part of their Meta Object Facility.

The Object Management Group's Model-Driven Architecture aims at the separation of the business logic from the application logic and platform technology. It is therefore based on several different models, each representing a specific logic or layer in the concept. However, to be more successful than the previously mentioned CASE tools, Petrasch and Meimberg (2006, p. 37) suggest that model-driven engineering along with model-driven architectures must occur in a particular context. Their suggestion is supported by literature such as Greenfield et al. (2004) who provide this context within a certain domain, often represented by a software product line. In order to make automated model-driven engineering possible, the respective models must be very precise and formal. Such a clearly defined context is provided by domain-specific languages. Based on the three Object Management Group standards Unified Modelling Language (UML) for visual specification, Meta Object Facility (which includes the earlier mentioned Query View Transformation) as the basis for Domain Specific Languages and Common Warehouse Metamodel for data structure specifications, MDA specifies four different core models. Each more precise than the previous one, they allow the business, application, and technology layers to evolve independently.

- **Computation Independent Model:** Also known as the Business Model, it describes the required systems from a hard- and software independent point of view. A joint understanding of the required system and its intended use are the major focus of this model. It can be represented as a high-level UML class diagram for example, containing the key concepts and terms of the respective domain. The definition of a Computation Independent Model can be alleviated by certain analysis patterns such as Party, Organization Structure, Transactions, Contract, or Product (Fowler and Rice, 2008; Petrasch and Meimberg, 2006, p. 101). However, the Computation Independent Model is not more exact than a description in a natural language (Petrasch and Meimberg, 2006, p. 103).
- **Platform Independent Model:** After defining the business need within a Computation Independent Model, it is further elaborated with conceptual information and details into a

platform independent model. The platform independent model describes the business domain and required system on a formal and precise level, i.e. it already contains elements like entities, attributes, or data types, for instance (Petrasch and Meimberg, 2006, p. 103). The platform independent model is the first model which may automatically be transformed by transformation engines or code generators and thus needs to be as precise as possible (Singh and Sood, 2009, p. 1648).

- **Platform Model:** Also known as Platform Description Model, the Platform Model formally describes the future platform on which the application in scope is supposed to run (Petrasch and Meimberg, 2006, p. 103). Typical elements of such a platform model would be an operating system API, a formal description of the database server or programming language to be used, or any additional services like middleware or transaction handling. These Platform Models do not have to be developed separately for every new application, but can be reused. Together with metamodels, mapping rules or profiles, for instance, the platform independent model and the Platform Model serve as the foundation for the underlying Platform-Specific Model (Petrasch and Meimberg, 2006, p. 107).
- **Platform-Specific Model:** The result of the model transformation from platform independent model and platform model is the Platform-Specific Model. It formally describes the application for the platform specified in the platform model. Of course, such a transformation does not necessarily need to be all-inclusive: The platform specific model may also be taken as a new platform independent model if a more detailed transformation is to take place afterwards, e.g. by specifying a particular version of a middleware or operating system. This new platform, however, must also be formally described in a new platform model.
- **Platform-Specific Implementation:** The final result of the transformation process is the Platform-Specific Implementation, i.e. an executable artefact reflecting the requirements previously depicted in the Computation Independent Model. It may be enhanced with additional code or components, for example, if not every required functionality could be transformed from the higher-level models.

With the help of the previously-mentioned models, a stepwise transformation of the initial business need into platform-specific models and implementations becomes possible. It helps to close the abstraction gap between subject matter experts and software developers. The following figure illustrates the concept:

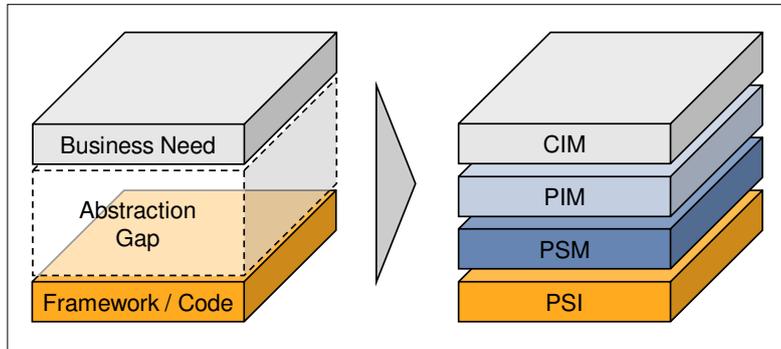


Figure 2-4: Models within an MDA approach

Omitted details are subsequently added until executable software is available. Of course, this process is not trivial by far. The extensive degree of freedom and context sensitivity becomes an issue if the model is to be interpreted by a code generator.

It is important to note that model-driven engineering may occur not only within the boundaries set forth by the Object Management Group and their MDA standard. Any artefact “that captures metadata in a form that can be interpreted by humans and processed by tools” (Greenfield et al., 2004, p. 217) represents a formal model and is thus qualified for model-driven engineering. Such formal artefacts do also include textual descriptions, as a model does not necessarily need to be graphical.

## 2.4.2 Domain-Specific Languages

To reduce the afore-mentioned context sensitivity and degree of freedom, a domain specific language (DSL) is used to formally describe concepts found in a specific domain, such as financial online services, e-commerce applications, Customer Relationship Management systems, or anything else clearly delimited. Due to its formal character, anything described with a DSL can be understood by both man and machine. Schmidt (2006, p. 27), for instance, states that the

characteristics of a specific domain are represented by metamodels, precisely specifying semantics and constraints associated with this particular domain. Greenfield et al. support this specification, although they are limiting modelling languages to visual type systems:

*“A modelling language is a visual type system for specifying model-based programs. It raises the level of abstraction, bringing the implementation closer to the vocabulary understood by subject matter experts, domain experts, engineers and end-users.” (Greenfield et al., 2004, p. 142).*

One of the most successful examples of a Domain Specific Language can be found in what you see is what you get Editors for graphical user interfaces. While in the beginning Graphical User Interfaces could only be built by highly-skilled developers, today’s wizards and code generators allow almost everyone to develop powerful user interfaces. What made this possible was the definition of a highly specialized, domain-specific language, implemented in graphical design tools. Its language elements (buttons, panes, text fields, etc.) can be combined based on a clearly specified grammar (e.g. buttons can only appear within panes or windows, etc.). Other well-known examples are Event Driven Process Chains or the Entity Relationship Model (Beltran et al., 2007, pp. 50 f.). Despite the previous examples and the definitions of Schmidt and Greenfield et al. it should be noted that domain specific languages do not necessarily need to be of a graphical nature. Even plain text can represent a DSL as shall be seen in the following. With DSLs, it should, for instance, be possible to assemble an online shopping system with credit approval, product catalogue and payment system without having to worry about the particular implementation and interaction of the components. Concluding the above, DSLs have several important advantages (Beltran et al., 2007, pp. 59 f.):

- Specifications can be described faster and more precisely with DSLs
- Change requests can be captured precisely and unambiguously with DSLs
- Specifications are context-free and leave no room for interpretations
- Code generators can be built for a specific domain and are thus more powerful and easier to handle than, e.g., former CASE tools

- Transforming a model to source code by a generator is less error-prone than manual implementation for each product

To efficiently utilize model-driven engineering with its model transformation engines and code generators, it is inevitable that a domain-specific language for model creation be defined. However, domain-specific languages do not necessarily need to be graphical; a textual implementation may also be appropriate. Imagining an automotive ordering system, a fragment being written in a DSL may look like this:

```
Define MYCARORDER PrivateOrder
  Customer=13357
  CarModel=ModelT
  EngineType=ABC
End MYCARORDER
```

For a computer to process this language, a context-free grammar is needed. Based on this grammar, a parser is required to interpret the text and build a syntax tree which is then translated into native program code. One could do this from scratch with, depending on the language complexity, significant efforts, or utilize a parser generator such as YACC, which takes a meta language describing this grammar. For the above example, part of the grammar representation on the basis of Extended Backus-Naur Form would look like the following:

```
Definitions ::= Definition*
  Definition ::= "Define" Identifier Order
  Customer ::= 5 * Numeral
  CarModel ::= "ModelA" | "ModelT"
  EngineType ::= "ABC" | "AAH" | "ACK"
End Identifier

Identifier ::= Character{Character}
Order ::= "PrivateOrder" | "BusinessOrder"
Character ::= "A" | "B" | "C" | "D" | "E" |
             "F" | "G" | "H" | "I" | "J" |
             "K" | "L" | "M" | "N" | "O" |
             "P" | "Q" | "R" | "S" | "T" |
             "U" | "V" | "W" | "X" | "Y" | "Z" ;
Numeral ::= "0" | "1" | "2" | "3" | "4" |
           "5" | "6" | "7" | "8" | "9" | ;
```

From the example above and according to Cook (2007, p. 16 f.), implementing a DSL in this manner is very difficult and error-prone and requires significant expertise, not to mention developing a comfortable integrated development environment with syntax checking, text highlight-

ing and auto completion. An alternative way is to use already-existing possibilities of a modern third generation programming language. In particular, object orientation allows defining classes, structures, enumerations or configurable syntax, which may easily embed domain-specific concepts (Cook, 2007, p. 17). Another alternative would be an XML representation of the domain concepts, while the required grammar is described in an XML schema. The advantage lies in the availability and integration of tools and integrated development environments, although implementing the grammar requires a bit more effort as compared to Extended Backus-Naur Form solutions (Cook, 2007, p. 19). The above order written in an xml-based DSL could look follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<Orders>
  <PrivateOrder name="MYPRIVATEORDER">
    <Customer>13357</Customer>
    <CarModel>ModelT</CarModel>
    <EngineType>ABC</EngineType>
  </PrivateOrder>
</Orders>
```

The according xml schema representing parts of the grammar could look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema
  xmlns="http://foo.bar/orderexample"
  attributeFormDefault="unqualified"
  elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace=" http://foo.bar/orderexample ">
  <xs:element name="Orders">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded"
          name="PrivateOrder">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Customer" type="xs:CID" />
              <xs:element name="CarModel" type="xs:CM" />
              <xs:element name="EngineType" type="xs:ET" />
            </xs:sequence>
            <xs:attribute name="name" type="xs:string"
              use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

```
<xs:simpleType name="CID">
  <xs:restriction base="xs:string">
    <xs:minLength value="5" />
    <xs:maxLength value="5" />
    <xs:pattern value="[0-9]" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="CM">
  <xs:restriction base="xs:string">
    <xs:enumeration value="ModelA" />
    <xs:enumeration value="ModelT" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="ET">
  <xs:restriction base="xs:string">
    <xs:enumeration value="ABC" />
    <xs:enumeration value="AAH" />
    <xs:enumeration value="ACK" />
  </xs:restriction>
</xs:simpleType>

</xs:schema>
```

Implementing the previous in a graphical domain-specific language is even more complex. One would need conventions for representing structure such as containments, objects, classes, entities, relationships, associations, connections, or dependencies, and behaviour, such as lifelines and arrows to indicate messages and data flows, graphical elements for flow charts, state machines, and use cases (Cook, 2007, pp. 20–26). Once created, these elements must be imbued with their own grammar, indicating how different elements may and may not interact with each other on the user interface, not to mention implementing all these within a suitable tool. Subsequently, the domain-specific grammar (see the above xml schema) must be applied to this tool to eventually describe domain-specific concepts in a graphical model. To the author doing all this seems almost impossible for an organization aiming to apply MDE to gain efficiency benefits from industrial software engineering. Similar to using parser generator for textual DSLs, graphical DSLs should be bootstrapped with an appropriate integrated development environment providing the fundamental aspects of graphical languages. Other approaches suggested in the literature seem too effort intensive to start with and are often based on very simple implementations of IDE prototypes developed at universities. In addition to some stand-alone tools, the most promising approaches can be found in the Graphical Modelling Framework for Eclipse

(Eclipse Foundation, 2012) and Microsoft's Visual Studio DSL Tools (Cook, 2007, p. 27); other examples are MetaEdit+ (MetaCase, 2012), or the Tiger Project (TU Berlin, 2012). These tools usually accept an XML representation of the domain specific language and, along with the necessary adaptations, allow model-domain specific concepts in a graphical editor, which may then again be serialized within an XML file.

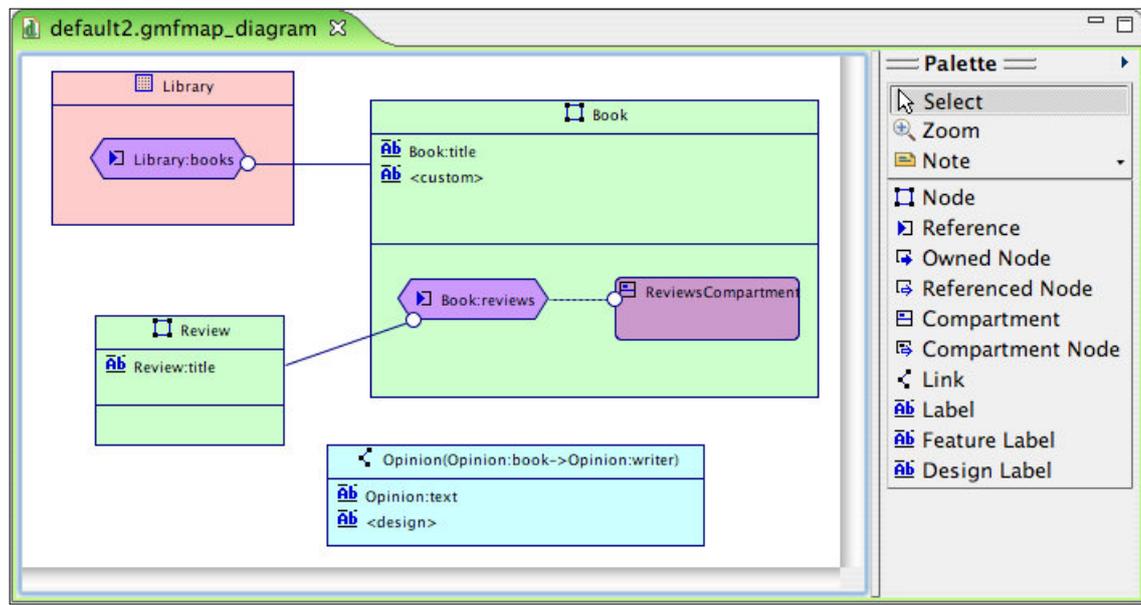


Figure 2-5: A Domain-specific concept in Eclipse GMF (The Eclipse Foundation, 2012)

In order to conceptually design an application, Domain Specific Languages are used to build the required models. The initially-mentioned separation between business logic, application logic and platform technology is achieved by subdividing the architecture into several models, each adding previously-omitted detail until the business logic has been transformed to a platform- and technology-specific implementation concept. This separation of concerns must also be represented by the grammar of the domain-specific language, which may for instance be done by declaring lower-level details as optional or by defining separate grammars, one for each modeling layer.

### 2.4.3 Model Transformation Engines and Code Generators

Once a model of an application is completed, it must either be transformed into a lower level model by translating it and adding additional detail, or, in case of low level models, used to

generate source code from it. Therefore, model-driven engineering combines Domain-Specific Languages with Transformation Engines & Generators (Schmidt, 2006, p. 27). Both the DSLs and the transformers and generators are uniquely designed for a particular application domain, reducing the degree of freedom and possible contexts.

Transformations between models become necessary if the abstraction gap between a problem domain and the technical implementation capabilities is too large, for instance, if a specified system is supposed to run on different platforms. Intermediate models would then take care of the particular requirements of these platforms. To generate subsequent artefacts out of models, transformation engines or code generators need to be provided together with meta-models of the source and the target model, as well as a set of mapping rules between them. While the meta-models are already available with the definition of the Domain-Specific Languages, the transformation rules must be expressed within a transformation language (Pham et al., 2007, p. 3).

Literature distinguishes between three different model transformations (Brown et al., 2005, p. 11; Beltran et al., 2007, pp. 70 f; Petrasch and Meimberg, 2006, pp. 125 ff; Stahl and Bettin, 2007, pp. 199 f.), i.e. model-to-model transformations, refactoring transformations, and model-to-code transformations:

- **Model-to-Model Transformations:** These transformations convert the information of one model to another, more specific model. An example would be the transformation of a platform-independent model into a platform-specific model with the help of transformation rules and a platform description model. Both source and target models are instances of different metamodels. The source model remains unchanged.
- **Refactoring Transformations:** This type of transformation occurs if an pre-existing model is modified or enhanced with new or adapted functionality. Afterwards, the model contains new instances of already known or changed meta classes. It is similar to a model-to-model transformation, but with an equal source and target model.
- **Model-to-Code Transformations:** The third type probably is the most complex one. “These transformations convert a model element into a code fragment. This is not limited to

[...] programming languages: configuration, deployment, data definitions, message schemas, and others kinds of files can also be generated” (Brown et al., 2005, pp. 11 f.). Similar to model to model, such transformations require certain rules or templates about how a specific model element is to be converted into e.g. executable code. The closer the source model is to the executing platform, the more precise such transformations can be.

A frequently-mentioned concept for transformation and generation is that of generators and cartridges (Beltran et al., 2007, pp. 121 ff; Stahl and Bettin, 2007, pp. 139 ff; Cook, 2007, pp. 309 ff.). The generator itself can be compared to a workflow engine providing fundamental infrastructure equally for any kind of transformation or generation, while the cartridges provide specific, model-related knowledge. The latter may include model parsers, model validators, transformation and mapping rules, or code generators (Stahl and Bettin, 2007, p. 140). The actual transformation or generation activities can be implemented in various ways. The most commonly-used technique is the application of templates. A template consists of predefined source code with place holders to be automatically filled in depending on model artefacts (Stahl and Bettin, 2007, p. 146). This concept is known from dynamic web applications implemented in ASP, JSP, or PHP. Another template alternative is Extensible Stylesheet Transformation Languages (XSLT). An XML parser creates a Document Object Model, while the XSLT processor executes the appropriate template specified for each node in the model (Cook, 2007, p. 313; Stahl and Bettin, 2007, p. 147). To define a class for the order specification from above (which was slightly adapted to represent the actual class of a private order), the model and required xslt could look like this:

XML-Model for PrivateOrder class:

```
<Class name="PrivateOrder">
  <attribute name="Customer" type="String"/>
  <attribute name="CarModel" type="String"/>
  <attribute name="EngineType" type="String"/>
</Class>
```

XSLT Stylesheet for Java Transformation including getter and setter methods:

```
<xsl:template match="Class">
  private class <xsl:value-of select="@name"/> {
```

```
        <xsl:apply-templates select="@attribute"/>
    }
</xsl:template>

<xsl:template match="attribute">
    private <xsl:value-of select="@type"/>
    <xsl:value-of select="@name"/>;
    public <xsl:value-of select="@type"/> get_
    <xsl:value-of select="@name"/>() {
        return <xsl:value-of select="@name"/>;
    }
    public set_<xsl:value-of select="@name"/>(<
    <xsl:value-of select="@type"/>
    <xsl:value-of select="@name"/>
    ) {
        this.<xsl:value-of select="@name"/> =
        <xsl:value-of select="@name"/>;
    }
</xsl:template>
```

Java code being generated:

```
private class PrivateOrder{
    private String Customer;
    private String CarModel;
    private String EngineType;
    public String get_Customer() {
        Return Customer;
    }
    public String get_CarModel() {
        Return CarModel;
    }
    public String get_EngineType() {
        Return EngineType;
    }
    public set_Customer(String Customer){
        this.Customer=Customer
    }
    public set_CarModel(String CarModel){
        this.CarModel=CarModel
    }
    public set_EngineType(String EngineType){
        this.EngineType=EngineType
    }
}
```

The above example works with any class modelled according to the xml model, no matter how many attributes may exist. For more advanced implementations, the xslt stylesheet needs to be extended to provide the appropriate templates.

According to Stahl et al. (2007, p. 149), an alternative way of generating code after parsing a model is to use control structures, typing, and modularization from a pre-existing language such

as C# or Java and concatenate the source code within a String Buffer. Implemented in Java their approach would look as follows:

```
public Class JavaClass {
    private String name;
    private List<JavaMember> members;

    public JavaClass(String name; List<JavaMember> members){
        this.name = name;
        this.members = members;
    }

    @Override
    Public String toString() {
        String buff = "public class "+name" {\n";
        For (Member m : members)
            buff += m;
        buff += "}\n";
        return buff;
    }
}
```

Apparently, model transformations presented so far may only occur in one direction, i.e. from a generic to a more specific one. Indeed, Object Management Group's MDA approach does not say anything about bi-directional transformations, as it would contravene with the concept of modelling architecture or design decisions on the source model only (Stahl et al., 2007, p. 204). Unfortunately, this is not always possible, especially as not every single aspect or feature can be represented in a model. Certain functionality will probably always be directly implemented in the source code. To overcome this issue, two approaches exist: The first defines protected regions within the code that a code generator will not overwrite. However, this method may also lead to problems, as the code may become barely readable, or protected regions may simply "disappear" if the embracing model element is deleted (Petrasch et al., 2006, p. 137). Another approach suggested in the literature is the separation of generated and manually written code in separate files. Here the object oriented-concept of inheritance is very helpful: generated code contains abstract calls to the classes being implemented manually. Another option is to use import mechanisms in the target languages (Beltran et al., 2007, pp. 175 f.).

Either way, round trip engineering of models is not yet satisfactorily solved. Meanwhile literature suggest workarounds with protected regions or separate code files for manual implementations.

#### **2.4.4 Model-Driven Engineering Approaches and Recent Advances**

Model Driven Engineering Approaches have existed for several years now. In the following, the most comprehensive ones are introduced. Apart from these, advances in research concentrate on different aspects of model-driven engineering, which will subsequently be presented.

Czarnecki et al. present a “software engineering paradigm based on modelling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge” (Czarnecki et al., 2000, p. 5). Besides extensively utilizing the concept of software families, they also describe the application of model-driven software engineering to their concept of generative programming (Czarnecki et al., 2000). It therefore defines a problem space expressed by a Domain-Specific Language, defining domain specific abstractions and concepts in order to describe and specify the desired family member. Contrary to the problem space, the solution space “consists of implementation-oriented abstractions, which can be instantiated to create implementations of the specifications expressed using the domain-specific abstractions from the problem space” (Czarnecki, 2005b, p. 5). It is usually expressed by an implementation language. The mapping between both, which is the key concept of Generative Programming, contains the configuration knowledge such as illegal feature combinations, default settings, default dependencies, construction rules and grammar, and optimizations. These mapping rules are implemented within a generator returning the solution space, which may either be an intermediate model or executable program code. Their generator concept is based on intentional programming, “an extendible programming and metaprogramming environment based on active source, that is, a source that may provide its own editing, rendering, compiling, debugging, and versioning behaviour” (Czarnecki and Eisenecker, 2000, p. 503) developed several years ago by Charles Simonyi at Microsoft.

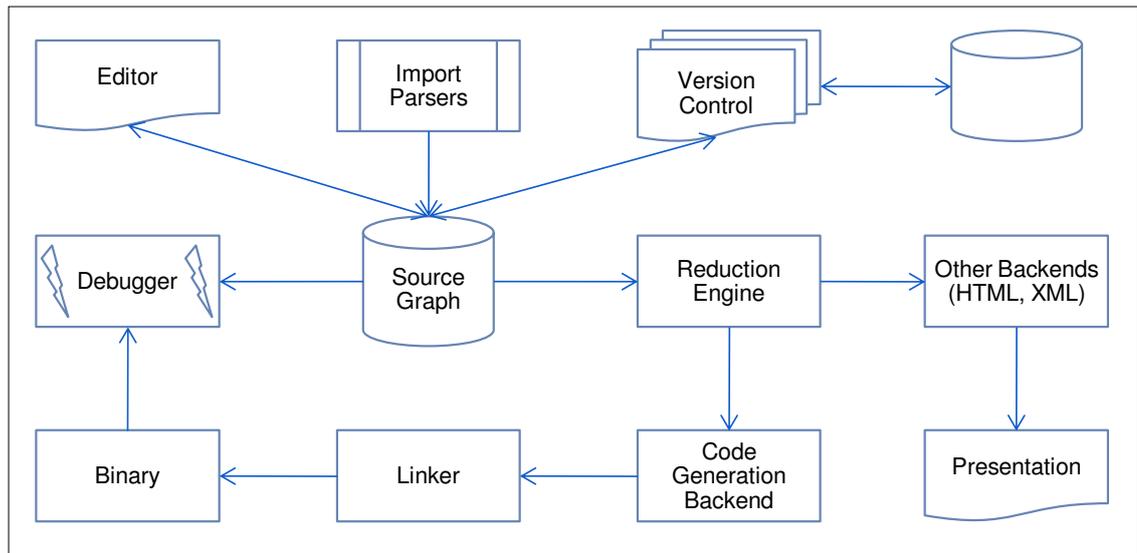


Figure 2-6: Components of Intentional Programming (Czarnecki and Eisenecker, 2000, p. 503)

The active source with all its artefacts is represented as a source graph in a database, which can be modelled by various editing interfaces (Simonyi et al., 2006, p. 452). The source graph captures the actual intention of the developer (e.g. print characters a-z) instead of capturing the implementation (in this case e.g. a for-while-construct). Knowing what was intended to be achieved is assumed to make later understanding and maintenance, but also code generation, much easier. An additional advantage stated by Simonyi et al. is the refinement to a high level intentional model in several iterations until a very detailed technical specification is available, which may eventually be translated into executable source code (Simonyi et al., 2006, p. 454). Software Developers are supposed to take the high level intentions of the domain engineers expressed in a domain-specific concept and build a generator accordingly. Although being theoretically researched, the intentional programming is still rather immature in terms of integrated development environments (Freeman and Webb, 2004, p. 199) and broad applicability in the industry and assumed to take several years to break through (Kamp, 2012, p. 4). Some proof of concept tools are currently being developed, for example Cedalion by Lorenz and Rosenan (2011), a language, editor and compiler for intentional programming. Simonyi meanwhile left Microsoft and founded a company providing tools and consultancy to enterprises wishing to start an intentional programming approach. Unfortunately, no further details on concepts, tools, or processes are available from Cedalion (Simonyi, 2012). For the present context, i.e. the in-

ustrialization of systems integration as an already existing business, generative programming is assumed to be too immature to be used in a commercial day to day context.

Another approach is Software Factories, introduced at Microsoft by Greenfield et al. (2004) which, similar to Generative Programming, utilizes Software Product Lines and Component-Based Development, along with a highly customized integrated development environment. It is based on Software Factory Schemes which model certain viewpoints required to develop a system. Such viewpoints express concerns regarding the business logic and workflows, data model and data messaging, application architecture, and technology, and may be present on all levels of abstraction. Altogether the schemes with their viewpoints exactly define what needs to be done and how to manufacture a family member. In order to provide a customized integrated development environment, the schema with its viewpoints is represented by a Software Factory Template. The template can be loaded into an integrated development environment, providing wizards, patterns, frameworks, templates, domain-specific languages, and editors. Complete definitions of domain-specific languages furthermore allow (semi-) automatic model-to-model transformations and code generation. Greenfield et al. basically utilize the concepts presented in the previous sections about domain-specific languages and model transformation and code generation. In their approach, they describe how these can be implemented within the Microsoft Visual Studio integrated development environment, including the definition of DSLs and the development of transformers and generators. Compared to Generative Programming and especially Intentional Programming, they utilize conventional, well-known techniques which make their concept realistic to practice. However, the approach of Greenfield et al. is rather complex and requires significant efforts before the first artefact is generated. It is furthermore dependent on Microsoft Visual Studio, although it is conceivable that the general principle may be used for other integrated development environments as well.

Recent research focuses on more detailed concepts, such as model-driven testing in which an application is being modelled from different perspectives and subsequently test scenarios and executable test cases are derived (Amrani et al., 2012; Ridene and Barbier, 2011), even including modelling model-based testing (Kanstren et al., 2012). The testing aspect is especially

important as even small changes to models or domain-specific languages may lead to major changes in the source code to be generated (Rapos and Dingel, 2012, p. 962). Referring back to section 2.3, model-based techniques are also being investigated for their applicability upon composition of software components (Lau et al., 2011; Parra et al., 2011) or the orchestration of web services (Sabraoui et al., 2012; Zhang et al., 2012). Here, similar principles as introduced above are being employed with the objective of automated system assembly utilizing reusable software components.

## **2.5 Prevailing Issues and Shortcomings**

Software product lines and component-based development, representing the industrial key concepts of specialization and standardization, are well researched and have found their way into practice for conventional software development. Linden et al. (2007, pp. 121–265), Clements et al. (2007, pp. 349 ff., 417 ff., 443 ff., 485 ff.), Pohl et al. (2005, pp. 413–433), and Cagatay (2009) present numerous experience reports on the application of software product lines in practice and present common pitfalls and challenges. However, during the course of the present research, no experience reports were found from software product lines interacting with other product lines, as would be the case in a large systems integrator working for customers from different areas. For instance, a product line producing logistics systems is likely to interface with products from a finance systems or shop floor systems product line, but not with a banking systems product line. Furthermore, while the question of the breakeven point has largely been answered, i.e., after two to four products being produced, it remains unclear how to proceed in cases where this breakeven point cannot be reached. Are software product lines still the right approach or should development of customer-specific systems generally occur outside a product line?

For component-based development, experience reports are not as extensive as for software product lines, presumably due to the concept not being as revolutionary as software product lines or model-driven engineering and its ability to be adopted with pre-existing infrastructure. One of the earliest ones was presented by Sparling (2000), describing the lessons learned during

six years of component based development. Crnkovic and Larsson (2002, 2000) present some more experience reports from the industry, as do Adamek and Hnetynka (2008) and Tracz (2001). In their works, the authors mention similar issues as pointed out for software product lines, especially the achievement of a return on investment and the question of how to tailor components to increase their potential for reuse. Besides several technical challenges, all of them mention development culture as one of the biggest issues of component-based development. Reusing other people's work seems to be a major challenge for developers and will be outlined in the following section.

The third industrial concept, model-driven engineering, shows a rather different picture: Comparing MDE with previous advances of software development, such as compilation technology or 3<sup>rd</sup> generation languages, further advancing the level of abstraction and thus increasing automation, seems obvious. However, even after almost 30 years of research in Computer-Aided Software Engineering (CASE) and similar approaches as the ones introduced above, this has not yet happened. In an article on automation and model-based software engineering (Selic, 2008), Selic names some of the most significant reasons for the lack of acceptance of automated software development in the industry: Foremost, the biggest advantage of fourth-generation programming languages (i.e. Domain-Specific Languages) is also their biggest drawback: A limited scope makes them very powerful, but also reduces the economies of scale for any infrastructure development such as integrated development environments, transformation engines, or code generators. Development tools are either developed in-house and commercially hardly ever break even, or built by a very small number of vendors, leading to a vendor lock-in and the risk of tools being discontinued. Besides the lack of a sufficient tool chain, development culture is, as for component-based development, a major issue in model-driven engineering.

### **2.5.1 Development Culture**

According to Clements et al. (2007, p. 29) and Linden (2007, p. 7), working in a software product line with reusable components involves two major functions: Designing and developing the overall architecture and reusable assets of a product family; and assembling the final products

from these reusable assets (Clements et al., 2007, pp. 29 f; Linden, 2007, p. 7). Developers responsible for the latter may feel displeased in that they no longer are “creating” something but assembling applications from other people’s work. Allocating personnel to the engineering and production departments requires thorough change management. Developers will also no longer have the satisfaction of immediately seeing if their ideas successfully translate into executable code. This satisfaction is assumed to be the reason that software developers identify themselves with mastering a specific programming technique instead of mastering financial systems (Selic, 2008, p. 388), for instance. In an environment where the majority of problems can be solved by reusing existing artefacts and only small functional parts or glue codes are being developed by hand, the self-concept of a developer may become constricted.

The same applies to model-driven engineering: Depending on the layer on which a system is being modelled, the developer may be working on aspects far away from source code. Furthermore, the time until first results become visible is significantly longer, and thus modelling requires a highly disciplined process which may be discouraging for traditional software developers. Modelling large parts of a system without instant verification, as known from traditional development, requires very thorough analysis of modelling decisions to identify possible consequences (Selic, 2008, p. 12). In MDE, it is not possible to consecutively alter source code until the requirements are met. A possible defect on a high-level model may only be detected after several transformations and eventually code generations have been completed. Iterative approximation is hardly possible here. Model-driven engineering thus needs to get away from hacking solutions into an integrated development environment and recompiling them until they work; a more design- and engineering-oriented approach seems inevitable.

### **2.5.2 MDE Tool Support and Usability**

Graphical representations of domain-specific languages and eventually a system described therein inevitably take up a large amount of space within an integrated development environment. As the number of DSL elements, rules, and relationships can be quite large and will mostly be unknown to the tool vendor, intuitive user interfaces seem very difficult to develop.

Given that MDE aims to bridge the gap between business experts and software developers, intuitive user interfaces are even more important. Compared to traditional text-based integrated development environments, current MDE tools may even reduce productivity (Selic, 2008, p. 9) as their handling is ambiguous if used for different domain specific languages (e.g. symbols, workflows, auto completion) and they do not provide the usability developers are currently used to. This also applies to text-based integrated development environments, where automatic code completion, or type and syntax checking, are often not available for domain specific languages. In addition, current tools in the field of MDE are hardly customizable, i.e. icons, menus, shapes, or modelling concepts cannot be changed according to the business domain they are used in (Selic, 2008, p. 10). Business experts as well as software developers have to work with largely the same user interface, although their tasks are completely different. The business expert may expect something more related to “the real world” while the software developer needs access to implementation (Selic, 2008, p. 10). Given the expectation that model-based engineering aims to describe a complete system capable of being queried at any time (although on a higher abstraction level), the corresponding model must represent a complex network of links and relationships between different modules and subsystems, as well as newly created language artefacts. Partitioning such a model and thus allowing different developers to advance it at the same time by branching and merging it seems extremely difficult. The level of difficulty is even increased when one developer changes elements required by another. It is not yet known how graphical models can be represented in version control systems and how changes to the same model can be merged. Besides usability, currently there is no such thing as a common modelling standard vendors are adhering to. It “is rarely possible to effectively exchange models from equivalent tools from different vendors” or to exchange models between corresponding tools, such as security or performance analysis tools (Selic, 2008, p. 10). Besides the lack of standards, the reason for this can also be seen in a too small and fragmented user base for which it is economically not feasible to provide support. In addition, as long as there is no “de-facto standard” or clear market leader, tool vendors will try to bind customers to their own (even incomplete) tool suite and thus do not proactively promote interoperability.

These and similar issues were also reported in several case studies (Staron, 2006, pp. 68–69; Shirtz et al., 2007, p. 181; MacDonald et al., 2005, pp. 18–193) on the application of MDE in industry. With Model Driven Architecture, Generative and Intentional Programming, and Software Factories, there are some interesting and promising approaches being developed. However, their way into industrial practice is still prone to “a great deal of improvisation, invention, and experimentation and still carries with it significant risk” (Selic, 2008, p. 16). Major improvements in standardization and availability of tools must be made to further advance model-driven engineering beyond academia. It is therefore not believed that for the time being a full-fledged model-driven engineering approach in an industrial setting is feasible.

### **2.5.3 Conclusion**

Summarizing literature findings it can be said that software product lines and component-based development are comparably well researched and are slowly finding their way into practice. Several industry projects and case studies were started and companies begin to adopt the principle in their day-to-day development. However, studies analyzing the long-term success of software product lines are not yet available but would be very helpful in making valid investment decisions. Similarly, component-based development can be seen as a mature technology that has found its way into practice. After the rise and fall of CORBA (Henning, 2008), it is mostly Sun/Oracle’s Enterprise Java Beans and Microsoft’s .Net component model which is being used in an enterprise context. What has not yet been achieved is the success of commercial components. Most reuse occurs within the boundaries of the enterprise in question; only very little is sourced from external vendors. Thus component market places have not yet formed. However, there are some areas in the field of software development where industrialization based on these two concepts has not yet been researched. One of these is systems integration, the focus area of the present work. Systems integration comes with certain characteristics distinguishing it from single-system development which will be detailed in the next chapter. It thus cannot be guaranteed that the principles of specialization and standardization can seamlessly be applied. For model-driven engineering, the situation is different. Research is still in progress and major issues are yet to be resolved. Some early adopters started first experiments in practice, but

throughout report the aforementioned issues. Chapter 2 concludes the first part of the evaluate phase of the design research cycle by answering the question what the problem, i.e. the research gap actually is. The second part of this question is discussed in the following chapter: In the context of systems integration in conjunction with software product lines and component-based development, it must be evaluated where the issues are, and if the course for future model-driven engineering can already be set today.

### **3 Literature Review and Introduction: Systems Integration**

As previously introduced, it is questionable if current concepts of software industrialization may be applied seamlessly to the field of systems integration as well. Compared to conventional software development of commodity applications, systems integration comes with certain characteristics requiring special consideration. As the underlying research explicitly focuses on this field of software development, this chapter provides a detailed introduction into the different forms of integration and their distinctive features which will eventually require a rethinking of the current industrialization concepts previously introduced.

According to Conrad et al. (2006, p.11), two major directions of systems integration can be distinguished: One is to maintain multiple similar or equal data sources. Here integration solutions are used to avoid redundancy and inconsistency of information and present it in a uniform way. The second direction aims at a generic and global access to different information systems. Systems integration solutions support the business processes of an enterprise while providing access to all relevant data sources and IT systems. Fischer (1999, p.86) and Riehm (1997, p. 10) further differentiates the term either as a state in which entities continue to exist after being integrated, or as the process of integrating them into a larger entity. Integration as a state defines classes by which the degree of integration of IT systems can be differentiated and evaluated. Integration as a process deals with the steps required to move an IT system from a given degree of integration to a higher one, which is done by merging distinct entities into a cohesive whole or integrating them into already existing systems (Riehm, 1997, p. 10; Fischer, 1999, p. 86). Although the major directions defined by Conrad, as well as the differentiation into state and process are valid, integration as a process of integrating distinct entities into a cohesive whole appears to be most relevant in practice. The present research therefore follows the latter definition of the term integration.

The process of integration can be even further divided into information integration and application integration (Conrad et al., 2006, p. 11; Leser and Naumann, 2007, pp. 3–5). Information

integration concentrates on the integration of different data sources, for instance, by data consolidation or data warehousing. Application integration in turn covers the combination of different software systems that support certain business processes. The integration of such systems is also referred to as Enterprise Application Integration (EAI) and depicts a core area in today's information systems engineering. As the scope of this thesis lies in the latter, the following section describes EAI in further detail.

### 3.1 Enterprise Application Integration

In contrast to data integration, EAI focuses on the integration of software-based business processes which are usually spread over several different and heterogeneous IT systems. Enterprise Application Integration allows utilizing different already-existing business processes across the enterprise to form new ones. However, several interpretations of the term exist in the literature (Vogler, 2006, p. 52):

- EAI as an integration middleware solution. The integration aspect is limited regarding the implementation of a middleware solution with its supportive services (Kloppmann et al., 2000, p. 23). Here, EAI is defined as “the creation of business solutions by combining applications using common middleware” (Ruh et al., 2001, p. 2).
- EAI as a high level integration of different business information systems, also focussing on semantic requirements of integration (Hasselbring, 2000).
- EAI as an integration framework architecture (Liebhart et al., 2008, p. 13; Longo, 2001, p. 56).
- EAI as an approach to implement business requirements, including strategic and process-related considerations, utilizing different integration techniques for their implementation (Conrad et al., 2006, pp. 11 ff; Vogler, 2006, p. 53). In this definition, EAI provides the “unrestricted sharing of data and business processes among any connected application and data source in the enterprise” (Linthicum, 2000, p. 3).

The present research adheres to the fourth definition of the term, i.e. Enterprise Application Integration as an integration approach from a strategy, process and technology related perspective. It does so because each dimension of integration (see sections 3.1.1) may have a severe influence on its neighbouring one and thus may not be considered in isolation. Following the definition Kloppman et al. and Ruh would limit Enterprise Application Engineering to the utilization of common middleware, which is not always the case. Although the definition of Hasselbring fits the author's understanding, it seems to be too broad and unspecific to base further research on. Liebhart et al. and Longo's definition in turn sees it as an architecture only, which also limits the understanding of EAI to less than it actually represents in practice.

The term Enterprise Application Integration within the context of the present work is therefore defined as follows:

Enterprise Application Integration is the development of business solutions or tools implementing customer-specific business processes, utilizing arbitrary internal or external information systems by means of information technology. Integration activities are based on strategic decisions and include business process integration, presentation layer integration, and information systems integration.

### **3.1.1 Integration Dimensions**

Within the previously adopted definition of the term EAI, literature usually defines several layers or dimensions of integration. They start from a strategic and business process point of view, through process partitioning for different systems, to the actual data and functionality management on an implementation level.

In their book on Enterprise Application Integration, Ruh, Magginnis and Brown identify presentation integration, data integration, and functional integration as the main dimensions for EAI (Ruh et al., 2001, pp. 19–20). Vogler, for example, defines process, desktop and systems as the three subdomains of integration (Vogler, 2006, pp. 53–57). The process domain defines how business processes are depicted onto the IT landscape and how they support the overall work-

flow from a more strategic point of view. The second domain (desktop) defines when and how different (heterogeneous) applications are involved, and how they exchange information with the user (e.g., via a common user interface) or with each other. The underlying systems domain then defines which application accesses which data, how data exchange takes place, and how data redundancy is managed. Hasselbring (2000) offers a similar classification by defining a business, application and technology architecture. Hasselbring limits the term EAI to the second layer only, while applying interorganisational process engineering and middleware integration to the first and last layer, respectively. Despite the different interpretation of the EAI term, Hasselbring indeed considers the remaining aspects in his paper. In a later work with Conrad et al. all three dimensions are considered (2006, p. 4). A comparable classification can be obtained from Fischer (1999, p. 90), who identifies a business, organizational, functional and technical dimension. The business dimension defines which IT systems are required based on the strategic business needs. The organizational dimension aligns IT systems and workflows, and optionally adapts either. Data collection and storage of information (data integration), as well as controlling intermeshing activities (process integration) is done within the functional dimension. The fourth dimension (technology) aims at proper coupling of the different IT systems, independent of their location or underlying technology (systems interconnection).

Taking the previous definitions and explanations into consideration, the following three dimensions of integration are defined for the present work and will be taken as reference in the following chapters:

- **Business process dimension:** On the business process dimension, the organizational objectives, structure and core business processes of an enterprise are characterized. They define which business functionality and information is required and how the involved IT systems must interact from a semantic point of view.
- **Workflow dimension:** The workflow dimension subdivides a business process into distinct activities and maps these to the different IT systems. It defines the data sources and functionality required from the available IT systems from a technical point of view, as well as

the interaction among each other and with the end users. On the strategy dimension, these data sources and functionalities map to the semantic steps of the business process.

- **Technology dimension:** Information and communication infrastructure of an integrated systems landscape is implemented at the technology domain. It defines which applications may access which data or functionality, how this is done, and how data management (e.g., redundancy) takes place.

### 3.1.2 Drivers of Application Integration

Situations from which Enterprise Application Integration efforts arise are manifold and extend from new business models over mergers and acquisitions to phasing out legacy systems. According to Vogler (2006, p. 19), each of these leads to reengineering business processes, integrating new data sources, or developing new applications. Categorized according to previously introduced dimensions of integration, the drivers of EAI can be subsumed as follows:

- **Business process dimension:** Strategic decisions, driven internally or externally, often lead to changes in the underlying business processes. Literature identifies several reasons for business process dimension driven integration. Bahli et al. (2007, p. 112), Volger (2006, p. 20), and Puschman et al. (2001, p. 1), for instance, identify the cooperation with other companies as one major reason. It requires a frequent exchange of information, such as bills of material, production line data, or financial transactions. Potentially self-contained production support systems need to be interfaced with external partner systems. Bahli et al. (2007, p. 113) and Linthicum (2000, p. 17) name the offering of new services to customers, such as online order tracking or online billing, which eventually lead to new business processes and information systems being implemented. Another common reason for significant integration efforts mentioned by Tanriverdi et al. (2001, p. 705) and Puschmann et al. (2001, p. 1) are mergers and acquisitions between enterprises. Many general management processes and systems exist twice and need to be reconciled in a joint system. Due to changes in the business processes and possibly the integration of new information systems, the underlying workflow and technology dimension are influenced as well.

- **Workflow dimension:** For integration efforts on the workflow dimension, Vogler (2006, p. 21) and Puschmann et al. (2001, p.1) identify changes to an already implemented business process as a major source. No matter how small or large the process change has been, there will most likely be changes to the underlying IT systems. Should the change affect other than process-exclusive IT systems, changes to even other business processes cannot be ruled out (Vogler, 2006, p. 21; Puschmann and Alt, 2001, p. 1). While these drivers are certainly right, one should consider that they may lead to a loop back to the business process domain. As information systems may have limitations prohibiting the exact depiction of a modelled business process, the process may have to be changed (Vogler, 2006, p. 21). Other drivers of application integration may result from regulatory necessities, such as data preservation for telecommunications provider.
- **Technology dimension:** Integration decisions on the technology dimension are inherited from higher dimensions or result from technology-driven necessities. Conrad et al. (2006, p. 203), Vogler (2006, p.21), and Linthicum (2000, p. 12), for instance, identify legacy systems which may run out of service and thus need to be replaced. Frank (2001, p. 283) adds changes to the EAI architecture or technological base resulting from integration decisions. The reasons for the latter can often be found in a historically-grown integration landscape which lacks a clear and future-proof architecture. Again, changes to the technology dimension cannot be viewed in isolation. As it represents the technical foundation for all workflows and business processes, it may influence upper layers and induce changes on these.

The following figure illustrates the most common drivers of enterprise application integration for each integration dimension and their influence on each other (see Vogler, 2006, p. 22):

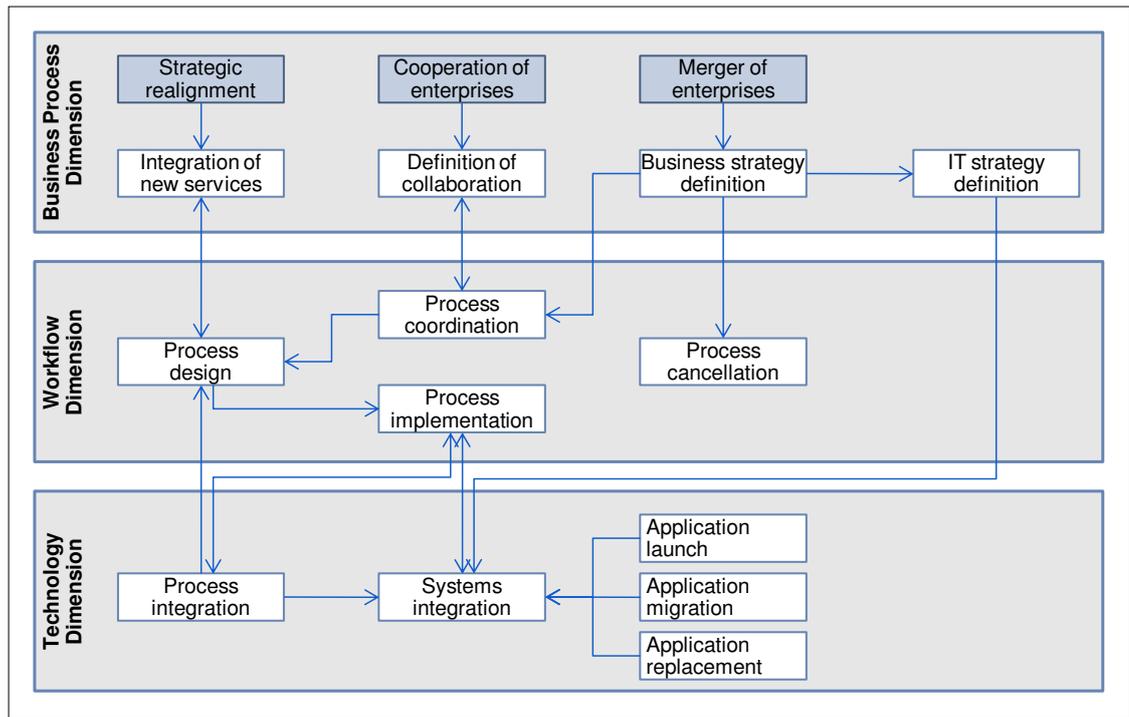


Figure 3-1: EAI drivers and their influence on each other

### 3.1.3 Architectures and Techniques of Application Integration

To integrate different business processes on heterogeneous platforms with each other, several architectures exist. The simple-most architecture is a message-based Point-to-Point Architecture, which is probably the foremost reason for integration efforts on the technology dimension (Vogler, 2006, pp. 19,21; Linthicum, 2000, p. 7). In this, configuration systems are interconnected with each other in a meshed way, leading to a multiplicity of different communication channels. They grow exponentially with each additional system and quickly become highly complex. The advantage clearly lies in its low cost and the ability to quickly integrate two systems with each other. The disadvantage is its high complexity evolving over time and thus tremendous efforts when changing a tightly-interwoven system with a large number of interfaces (Conrad et al., 2006, p. 84; Linthicum, 2000, p. 135). Furthermore, there is no platform providing fundamental services such as transaction monitoring or messaging (Conrad et al., 2006, p. 135). Each system is responsible for its own integration relationship and the required services.

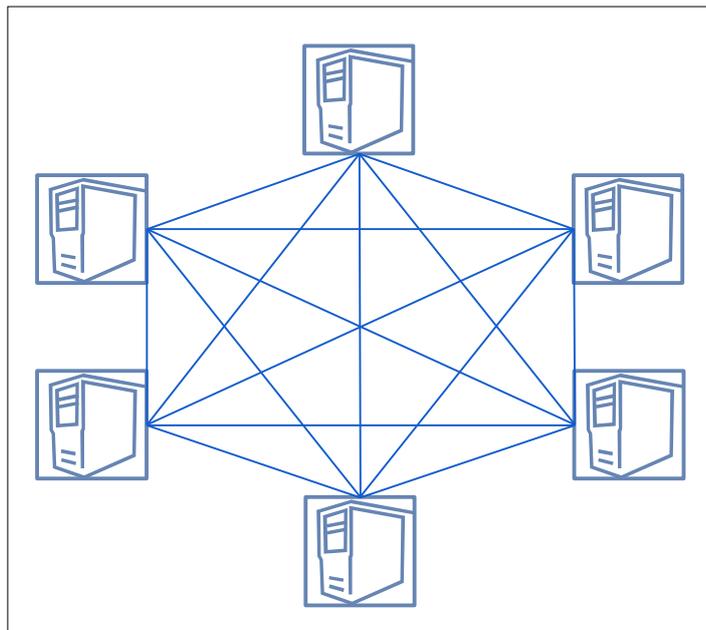


Figure 3-2: Point-to-point integration architecture

To minimize the interface problem, a Hub-and-Spoke Architecture may be implemented (Liebhart et al., 2008, p. 25; Conrad et al., 2006, p. 84). It provides a centralized integration system through which messages can be sent, translated, transformed, and routed. Adapters on the hub or the integrated systems are used to access the different sources of information (Linthicum, 2000, p. 136). An API for the hub allows software programmers to access the services of the source and target applications in a standardized way. Given that the functionality does not change, changing or replacing an integrated system now only requires rewriting the respective adapter. It greatly reduces the number of interfaces but may lead to a performance bottleneck and single point of failure. To overcome the latter, a Federated Hub-and-Spoke Architecture may be implemented. It basically consists of several hub-and-spoke architectures which themselves are integrated with each other (Soomro et al., 2012, p. 43). Performance and reliability depend on the number of hubs implemented.

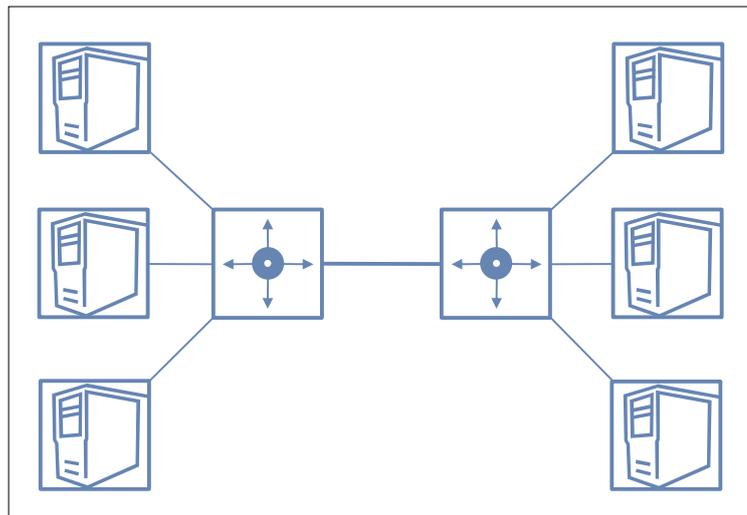


Figure 3-3: Federated Hub-and-spoke integration architecture

Spreading the hub and its services across different sites and systems leads to a Pipeline Architecture as i.a. suggested by Liebhart et al. (2008, pp. 26,27). Systems are connected to an integration bus via local interfaces and one or more message brokers manage information exchange between them. As more systems are added to the integration bus and coordination load on the broker increases, Soomro et al. (2012, p. 43) or Linthicum (2000, p. 315), for instance, suggest to add additional brokers to increase performance. Other functionality like security, error handling, or logging may also be separated from the broker and distributed across the network (Soomro and Awan, 2012, p. 43). In contrast to a hub-and-spoke architecture, this helps to alleviate performance and reliability issues.

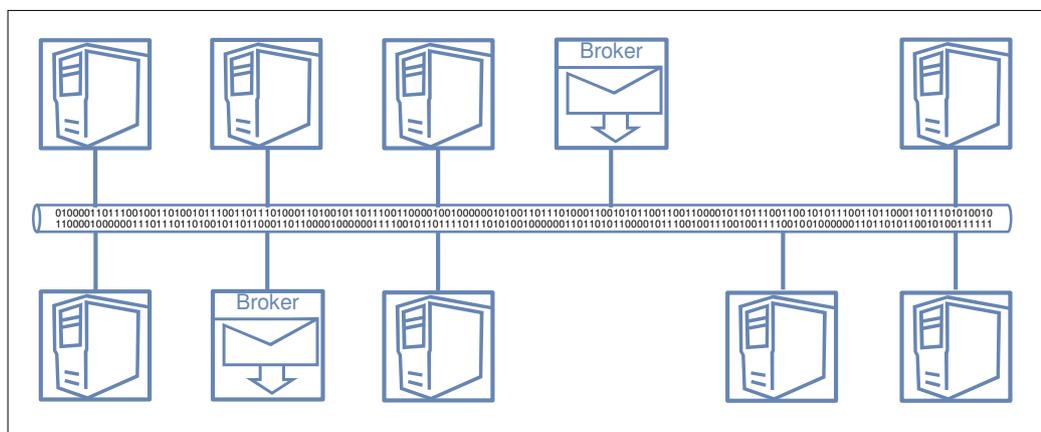


Figure 3-4: Integration bus architecture

Previously described integration architectures can be implemented with different technologies. Liebhart et al. (2008, p. 16), for instance, suggests a Message Broker or message oriented middleware as a central component responsible for asynchronous message routing and guaranteed delivery. It utilizes several interfaces and may also transcribe messages between sender and receiver and is most commonly used in a Hub-and-Spoke architecture. Extensions of some commercial products also provide process management and supervision (Conrad et al., 2006, p. 84).

Another possibility proposed in the literature (Liebhart et al., 2008, p. 16; Papazoglou et al., 2007, p. 393) is the Enterprise Service Bus. It provides an integration infrastructure which separates client applications from the service they are using. Core functionalities here are routing and messaging, transformation and mapping, mechanisms for orchestration of different services, and monitoring of the provided services. It can often be found as part of a Hub-and-Spoke or pipeline architecture (Liebhart et al., 2008, p. 18). “The ESB [Enterprise Service Bus] is designed to provide interoperability between large-grained applications and other components via standards-based adapters and interfaces. The bus functions as both transport and transformation facilitator to allow distribution of these services over disparate systems and computing environments” (Papazoglou et al., 2007, p. 393).

Transaction Process Monitors represent another important integration methodology (Conrad et al., 2006, p. 150). They ensure that a transaction consisting of multiple process steps will either be fully completed or be rolled back if one or more process steps failed. Their core functionality includes provision of client interfaces, authentication, load balancing, logging, exception handling, and workflow management (Conrad et al., 2006, pp. 150–152). A transaction process monitor in an integrated environment accepts a transaction, splits it up into feasible parts, coordinates and monitors their execution, and in the end either guarantees success or failure towards the invoking system (Conrad et al., 2006, p. 79). In case of a failed transaction, the transaction monitor also ensures that the systems are not left in an inconsistent state.

As a more recent development, Web Services represent a technology to provide business services over the internet. They are based on standardized, xml based protocols, i.e. the Simple Object Access Protocol for invocation of services, the Web Service Description Language for a formal description of the service, and Universal Description Discovery and Integration for registration and localization of available services (Conrad et al., 2006, pp. 187–189). Although initially developed to be used over the public internet, web services appear reasonable to be used in an integration architecture. However, without further development they only represent a peer-to-peer based architecture in their initial form. For more advanced architectures as introduced above, additional functionality like transaction handling or messaging seems inevitable.

Besides Web Services, Component Based Middleware may also be used to provide business processes and services to different consumers in the enterprise. The most prominent representatives are CORBA and JAVA EE as introduced in section 2.3.2 of the present work. They provide their services as components via preferably platform-independent, object-oriented communication protocols (Ruh et al., 2001, p. 84). Similar to web services, software components may also be used in any of the above integration architectures but possibly require additional functionality. In addition, software components may be implemented on different application layers, which according to Conrad et al. (2006, p. 86) are the presentation layer, the application layer, and the database layer. This allows, for example, the creating of user interface integration without having to develop a complex interface to the business logic.

## **3.2 The Integration Meta Model**

The above drivers, architectures, and techniques allow for an enormous number of different integration methodologies. Combinations thereof, often even within the same enterprise, inevitably lead to high complexity and heterogeneity. The result of such uncontrolled growth will eventually lead to new integration efforts, resulting in a vicious circle (Ruh et al., 2001, pp. 12-13; Linthicum, 2000, p. 6). A possible way out may be found with the help of an integration metamodel, clearly describing how the development of enterprise application integration solutions on the different dimensions works. Such a model was developed by Vogler in

2006 and is so far the only all-embracing model available in literature. Other works in this field usually focus on modelling the direct integration relationships between one or more systems (Berger et al., 2010; Grossmann et al., 2008; Zhu and Zhang, 2006), but not on independently modelling the EAI system itself on the different integration domains. As for its extensiveness and independence of any implementation technology, Vogler's model will be taken as the basis for applying industrial methods to systems integration. The present section will present the model in more detail.

### **3.2.1 Model Overview and Applicability**

Based on previous work by Gassner (1996), Derungs (1997), and Riehm (1997), Vogler developed an integration metamodel, describing the objects and relationships of enterprise application integration in general (Vogler, 2006). Its notation follows entity relationship diagrams known from relational databases. It consists of entities representing individual real-world objects (including objects required in an IT system), relationships, establishing a logic connection or relationship between entities, and attributes, identifying context relevant information about an entity. A relationship also contains the cardinalities on each end, with '1' standing for exactly one, 'c' for none, 'cn' for none, one, or multiple, and 'n' standing for one or more instances. The model also uses XOR relationships, represented by an arc intersecting the respective relationships.

Vogler sub-divides her model into four different viewpoints which are process integration, desktop integration, systems integration, and information systems. With exception of the information system viewpoint, the integration dimensions defined in section 3.1.1 can be matched to the viewpoints.

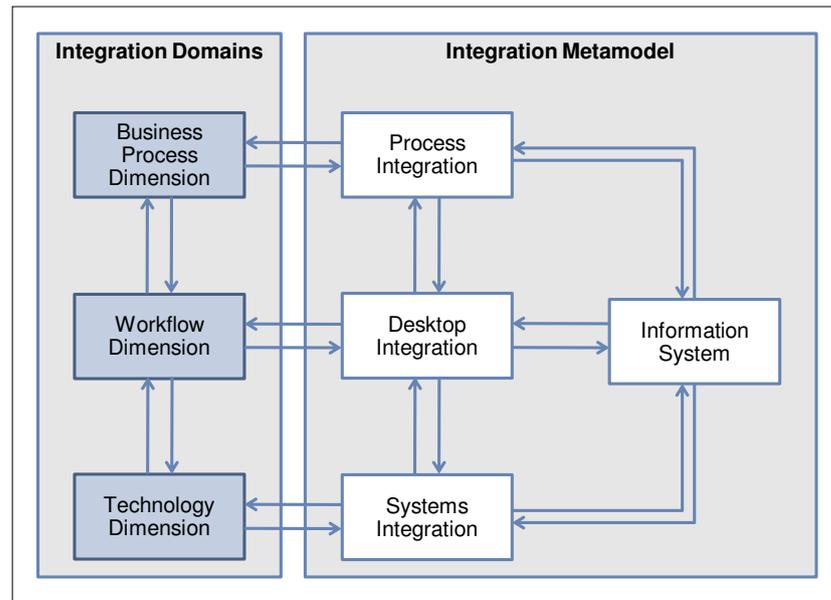


Figure 3-5: Integration dimensions vs. integration metamodel viewpoints

The viewpoints allow seeing the model from different problem-oriented perspectives (Vogler, 2006, p. 83): From a process integration (see business process dimension in 3.3.1) perspective, business processes are modelled and automatic workflow control is established. Desktop integration (see workflow dimension in 3.3.2) in turn is responsible for one or more workflows of a business process and makes different enterprise applications available at a user's workstation. These applications may also be made available in the form of a single application invoking tasks on the different enterprise systems. Systems integration (see technology dimension in 3.3.3) is responsible for realising the integration relationships defined by process and desktop integration on the system level. It defines data and functionality exchange on a technical level and is especially important when replacing or enhancing existing systems.

The integration metamodel as defined by Vogler from a generic viewpoint is represented in Figure 3-6 (Vogler, 2006, p. 82). It allows describing an integration relationship independently from a specific architecture or technology and can also be used to describe any integration product developed by a systems integrator as defined in the positioning section of this work (see page 14). The model supports other author's findings of the literature on integration dimensions and drivers of application integration, and also supports the different architectures and techniques of application integration. It therefore represents an ideal basis to apply methodologies of

software industrialization and shows how the respective aspects of component-based development and model-driven engineering map to systems integration.

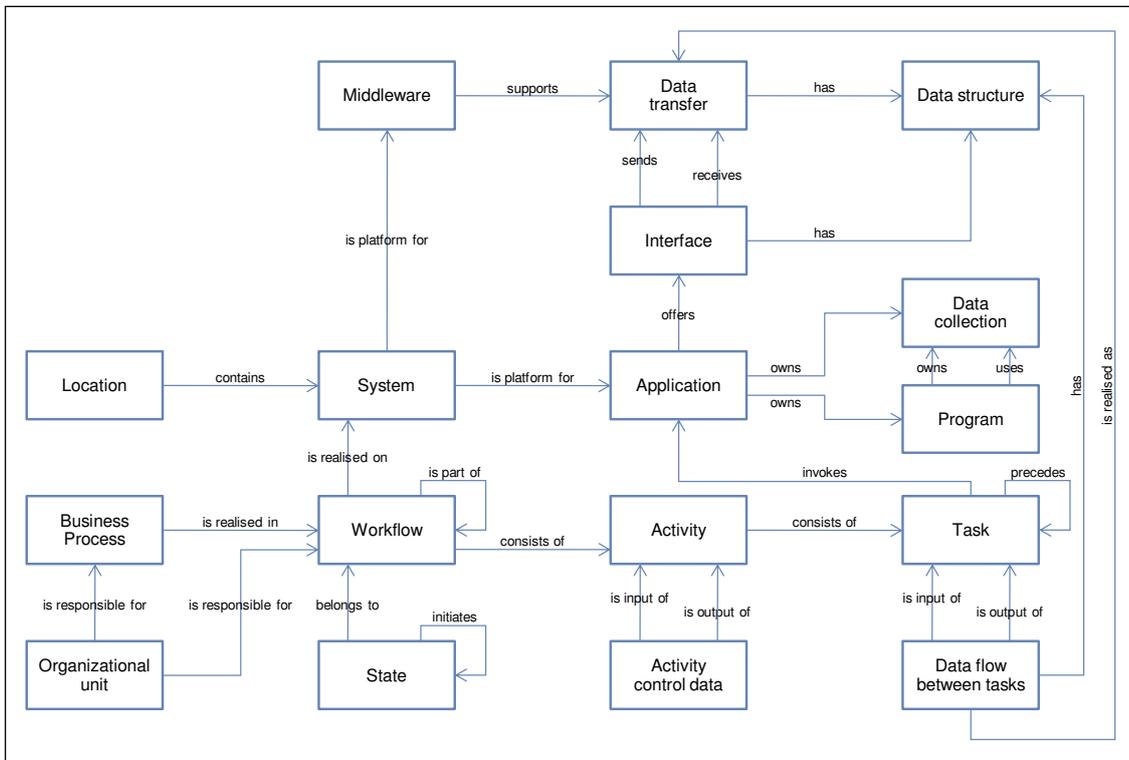


Figure 3-6: Integration metamodel – an overview (Vogler, 2006, p. 82)

The following sections provide a more detailed picture of the integration viewpoints and thereby detail the overall integration meta model. The information system viewpoint, however, is omitted at this point. In Vogler’s meta model, it formally represents the applications to be integrated with each other by themselves. An (partial) example would be a database schema or the architecture of an enterprise resource planning system. Interesting in this context are the integration relationships of such applications which are, though, already covered by the first three viewpoints.

### 3.2.2 Process Integration

The central element in the process integration view is the business process. It is implemented as one or more workflows and realised on one or more IT systems on a particular site. The responsibility of the workflow lies with one organizational unit. The workflow itself is represented as a finite state machine, moving from one state into one or more subsequent states. These state tran-

sitions initiate one activity consisting of one or multiple tasks and data sources available from enterprise applications. Execution of activities and underlying tasks depends on none, one, or multiple execution conditions and none, one, or more authorizations provided by the responsible organizational unit. Vogler also defines the organizational structure in more detail which is necessary for the overall integrity of the model, but can be neglected in the context of the present work. Figure 3-7 illustrates the process integration viewpoint (Vogler, 2006, p. 85).

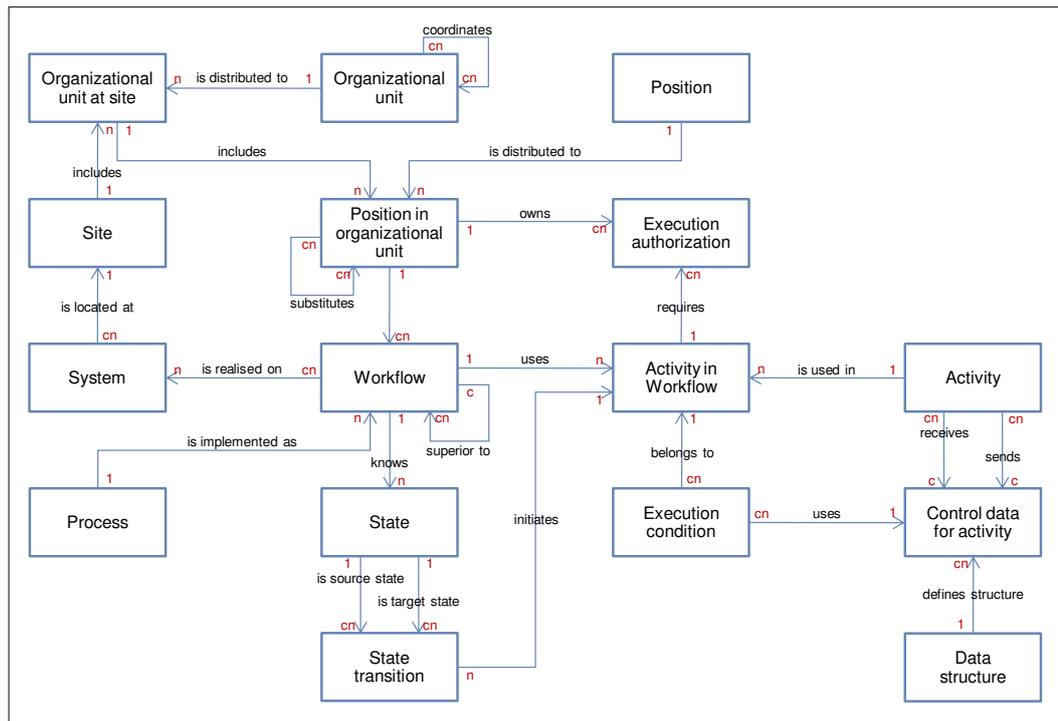


Figure 3-7: Integration metamodel– process integration view (Vogler, 2006, p. 85)

### 3.2.3 Desktop Integration

As defined within the process integration view, an activity bundles one or more tasks which may initiate exactly one application or none at all. An application in the metamodel context is defined as a collection of one or more programs used to provide data access and complex business functionality through clearly defined interfaces. Programs in turn are responsible for data storage and management and provide functional building blocks for more complex business applications (Vogler, 2006, p. 89). Based on a data structure, tasks may also send and receive information to or from other tasks. The task itself can be realized with one or more user-

interface elements, but may also be executed without any user interaction. Figure 3-8 illustrates the desktop integration view (Vogler, 2006, p. 86).

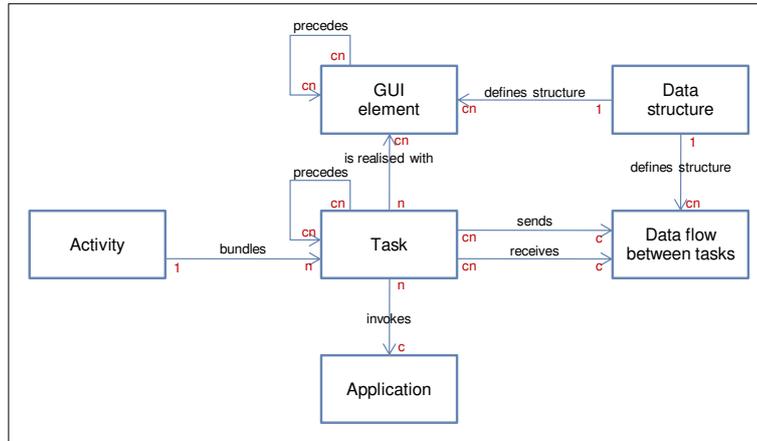


Figure 3-8: Integration metamodel– desktop integration view (Vogler, 2006, p. 86)

### 3.2.4 Systems Integration

Any integrated enterprise application requires an integration relationship with one or more other applications or data sources. The systems integration viewpoint describes these relationships which are not visible from the process or desktop point of view. This is due to the fact that logical entities used in business processes do not necessarily map 1:1 to information systems, i.e. one IT system may represent several process or desktop entities. Changes to the previous two viewpoints will therefore also cause changes to the systems integration viewpoint which is one of the reasons why enterprise application integration projects are far from trivial.

According to Vogler's model (2006, p. 88), to establish the required integration relationships, data transfer between different applications is required. This data transfer is based on interfaces which may either be program- or data-collection interfaces. A program interface provides a variety of business methods or functions belonging to one particular program. The program in turn is part of the underlying application responsible for more complex data access and business functionality. A data collection interface belongs to a particular data collection which is defined by one or more data structures. This data collection belongs to one particular program which utilizes this and (or) other data collections to fulfil its tasks. A middleware ensures the technical

data transfer between the different enterprise applications and their program- and data-collection interfaces.

Based on the respective requirements, systems integration chooses from different architectures and techniques presented in section 3.1.3 to implement the required relationship. Figure 3-9 illustrates the viewpoint (Vogler, 2006, p. 88).

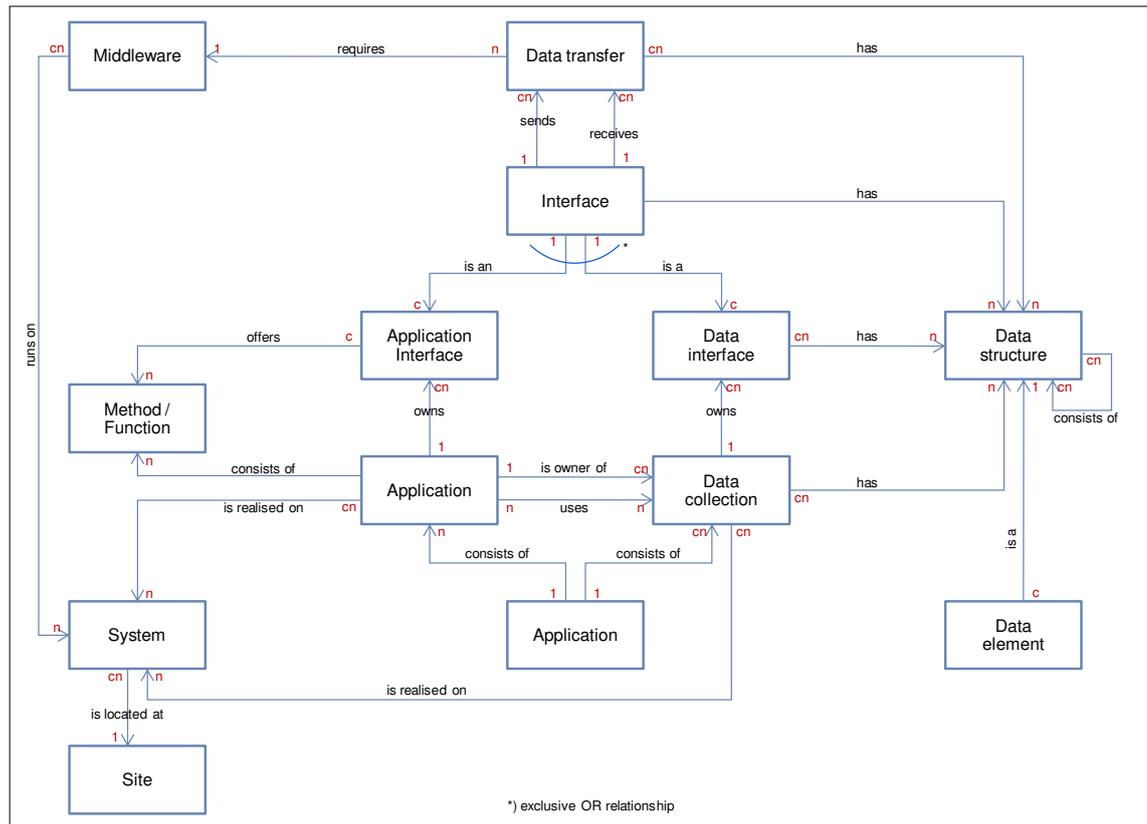


Figure 3-9: Integration metamodel – systems integration view (Vogler, 2006, p. 88)

For further reference, appendices A.2 and A.3 contain a detailed description of all meta entity types as well as the omitted information system viewpoint as defined by Vogler.

### 3.3 Characteristics of Software Development in Systems Integration

Integration activities and thus the development of software systems to provide appropriate solutions results from an ever-increasing complexity in today’s enterprise information systems. According to literature (e.g. Linthicum 2000, p. 6; Bahli et al., p. 111), in the pioneering time of

information technology, most applications were centralized on mainframes with homogeneous architectures and data sources. As technology advanced and client server platforms became available, enterprise applications started to spread across different systems (Bahli et al., 2007, p. 111). With an increasing market for software, new vendors and software products emerged, resulting in an ever-increasing heterogeneity in technology and data storage. Ruh et al. (2001, pp. 12-13) and Linthicum (2000, p. 7) point out that this was fortified by implementation of cutting-edge technology without considering its suitability for the particular situation. A lack of architectural foresight, which, thinking of the internet, was impossible to fully achieve, led to ad-hoc point-to-point integration of enterprise applications where necessary (Linthicum, 2000, p. 7). From the author's personal experience in the industry their statements can only be emphasised. As of today, the number of different systems and technologies has increased in such a way that it seems impossible to further implement ad-hoc relationships between different enterprise applications. A more organized and architecturally-structured approach to connecting systems with each other is inevitable, which eventually results in enterprise application integration.

Although literature states that EAI supports and simplifies the execution of business processes, during implementation it involves several prerequisites and challenges. The multiplicity of possible integration architectures and techniques in combination with unique IT landscapes present in each customer's system make enterprise application integration a comparably special area of software development. Linthicum (2000, p. 7) acknowledges this fact by mentioning the lack of a clear structure or integration architecture according to which development could occur.

Under these circumstances it seems questionable if current concepts of software industrialization, which highly rely on standardization and predictability, can be applied to the field of systems integration as well. The situation in each customer's system seems to be too diverse to allow for increasing the benefits of specialization, standardization, and automation. To obtain a better overview of these challenges, the following sections will detail the most relevant differences to conventional development of commercial off-the-shelf products.

### 3.3.1 Business Process Dimension

New business processes are often defined from a semantic point of view or are based on preconceptions from earlier projects regarding their representation in IT systems. According to Vogler (2006, p. 24), the IT landscape has indirect implications on the business process. It is important to understand the integration relationships in order to identify and choose an optimal solution (Vogler, 2006, p. 24). A business process may be implemented in a large variety of ways; however, from this point of view it is hard to choose the optimal one. ‘Optimal’ may have different meanings depending on the particular situation. It may stand for low implementation cost, low impact on other business processes, high performance, or high adaptability. Most of the time, it will be a trade off. This, however, is a major challenge as indicated by Ruh et al. (2001, p. 12): In many companies the particular situation is hardly known. The effect of one solution over another cannot be estimated. Furthermore, business processes designers may not know about the solutions available on the market and from the internal IT and often do not have the knowledge to design an overall concept (Ruh et al., 2001, p. 12). A sound decision and thus business process design should only be made with a complete picture of the IT landscape.

Similarly, the unawareness of the particular IT landscape may lead to unforeseen consequences of integration decisions (Vogler, 2006, p. 25). Only minor changes in a business process may lead to adaptations of the underlying systems, which in turn may require the adaptation of other business processes due to changed interfaces or data structures. The same issue was already reported by literature examined on the integration dimensions. Here Conrad et al. (2006, p. 14) stated that it is important to know if an information system, or parts thereof, are autonomous in terms of design, execution, and communication. Design autonomy is achieved if a particular part of an information system does not share any data model with other parts or other systems. This means that it is the only consumer of its data and thus solely responsible. Execution autonomy means that it can independently schedule and execute incoming requests. A communication-autonomous part or system can join or leave an IT landscape without affecting its own or other functionality. If these consequences are known early enough, business processes can be designed around them. Otherwise it may be a trial-and-error approach until a business process

implementation is found that does not affect other processes. Planning the according development efforts thus comes with significant uncertainties.

### **3.3.2 Workflow Dimension**

Linthicum (2000, p. 8) identifies a suboptimal degree of integration resulting from subdividing a business process into workflows and depicting them ad hoc on different IT systems as one of the issues in the workflow dimension. Due to time or cost constraints, these systems are often interconnected on a point-to-point basis instead of using shared integration architecture. In extreme circumstances, this leads to  $n*(n-1)$  relationships, making later changes more and more complex and expensive.

Another issue in this dimension was identified by Vogler (2006, p. 27) as part of her integration metamodel research. According to her, the integration relationships may be unknown due to insufficient documentation and the lack of an overall architecture. New implementations may be redundant and the consistency and integrity of interfaces cannot be ensured. A customer dataset from a customer-relationship-management system may be queried multiple times via multiple interfaces. In combination with insufficient documentation of integration relationships, changing aspects of the source system inevitably leads to unforeseen consequences for other systems. Unknown integration relationships thus also lead to uncertainties with regards to implementation efforts when planning a new integration project (Vogler, 2006, p. 27).

The root cause of a suboptimal degree of implementation and unknown integration relationships identified by Smojver et al. (2009, pp. 1-2), Gorton et al. (2003, p. 3), and Gassner (1996, p. 32), and lies in the lack of a methodological approach. Best practices or standardized processes for systems integration projects only partially exist. However, suitable methodology has been defined in literature during the last years but has not yet been made known or adopted in the industry (Gorton et al., 2003, p. 1). This is also becoming evident in that systems integration is not sufficiently considered in current software development models (Smojver et al., 2009, pp. 1-2; Gassner, 1996, p. 32). Integration projects are often performed ad hoc and for a single purpose only that leads to the initially-mentioned suboptimal degree of integration.

Heterogeneity caused by the previous problems prevents the implementation of holistic integration platforms or architectures within enterprises (Vogler, 2006, p.27). Based in Conrad et al. (2006, p. 14) it is reinforced by the fact that autonomous system development always results in different approaches to related problems. His observation is backed by Lui et al. (2011, p.2) and Linthicum (2000, p. 11) who state that such autonomous system development often results in stovepipe applications or information silos. Heterogeneity on the workflow dimension is caused in particular by different semantics and representations of data. A customer in one system may relate to the end user of a product; in another system it may refer to a reseller. Without further documentation, integration becomes very difficult. Even if the different semantics are known, transformations or creation of new temporary data becomes necessary (Gorton et al., 2003, p. 1).

### **3.3.3 Technology Dimension**

From a technical dimension point of view, heterogeneity is the major issue in systems integration. Gorton et al (2003, p. 1) and Linthicum (2000, p. 8) for instance name different hardware platforms, operating systems, or database systems which evolved over time in the enterprise as major reasons. Accessing an interface of a particular system usually requires a particular programming language or technique. Integrating an SAP system with, for example, PeopleSoft may mean having to write a remote function call in ABAP/4 on the SAP system (Linthicum, 2000, p. 249) and an appropriate receiver invoking the PeopleSoft Message Agent API (Linthicum, 2000, p. 263). In between, data transformation to overcome structural logical heterogeneity needs to take place. Imagining several such relationships makes changing a single, tightly-interwoven system almost impossible. The required implementation effort disproportionally rises with the number of systems to be integrated, unless a common architecture or platform is used. Furthermore, from a systems integrator's point of view, every customer is different, thus reusing a previously implemented solution is not an option.

Another big problem is the integration with legacy applications. These were often designed as standalone solutions with no integration in mind. Obsolete data management, interfaces, or a

lack of documentation or maintenance make integration extremely difficult (Themistocleous et al., 2001, p. 7). Due to their age and initial scope, such systems are often monolithic constructions without any application programming interfaces and do not offer a redesign of their implemented workflows (Vogler, 2006, p. 29). Outdated technology such as data stored in flat files and being manipulated via ALGOL or FORTRAN make concurring access very difficult. Furthermore, these systems often cannot be replaced. They often contain information and functionality not available otherwise and therefore impose restrictions on the overall integration concept.

The final issue of systems integration identified in literature lies in the redundancy of data (Lui et al., 2011, p.2; Vogler, 2006, p. 29; Linthicum, 2000, p. 11). In integrated environments, it becomes difficult to define which data resides where, how it is accessed and how redundancy is managed. Without such management, information may easily become outdated and inconsistent, leading to serious issues in business process execution. The reason for such redundancy of data is again found in autonomously-developed applications (Lui et al., 2011, p. 2; Linthicum, 2000, p. 11). It is further reinforced if data is concurrently accessed and altered within different systems.

### **3.3.4 Implications for Software Development in Systems Integration**

Recapitulating previously described characteristics and challenges, it can safely be assumed that systems-integration projects differ from traditional stand-alone software development. The literature examined expressed the typical problems faced in systems integration projects in the industry. Today's economy is in a permanent change, resulting from merger and acquisitions, split-ups, and co-operations between enterprises. Consequential process adaptations and realignments result in systems-integration and software-development projects. Furthermore, new or advanced information technology or the necessity to replace legacy systems may be a driving force. By means of traditional stand-alone software development, such integration requirements often cannot be met. This is especially true in fast-changing environments. Successfully implementing new processes or updated requirements is hard to achieve without a clear strategy and highly flexible software development approaches. In addition, enterprises usually have a past in

which autonomous systems and information silos evolved in an uncontrolled way. The result is a highly complex and heterogeneous IT landscape with redundant and inconsistent data sources. Software development in systems integration thus has to give consideration to all of the above without adding new complexity or heterogeneity.

Based on the author's professional experience in the industry, systems integration solutions are still implemented from scratch by utilizing traditional software development methods such as the Waterfall Model or the V-Model. These, however, were designed with monolithic systems in mind, as integration was not of interest at the time of their development. More recent works such as the V-Model XT briefly reference integration with external environments (Balzert, 2008, p. 622) but still do not pursue a standardized and methodological approach. The result may be an "integrated monolithic system" with highly complex dependencies. Moreover, these development models do not incorporate the basic principles of industrialization and thus may not leverage potential improvements in cost, efficiency and quality as initially stated.

As discussed in chapter 2, Software Product Lines, Component-Based Development and Model-Driven Engineering represent specialization, standardization, and automation for software development. The respective concepts are well understood and first literature is available on combining them in factory-like development environments, as for example in Greenfield and Short's book on Software Factories (2004) or Czarnecki and Eisenecker's Generative Programming (2005b; 2000).

Based on the the literature reviewd, confirming the experience of the author being a generic problem throughout the industry, software development in this context has to challenge a multiplicity of technologies, inflexible legacy systems, once-only technology combinations and a very high complexity. It appears debatable as to whether the concepts for industrialized software development in their original form can be applied to the field of systems integration, which will be further detailed in the following section.

### 3.3.5 Shortcomings of existing Industrialization Concepts

In systems integration, the multiplicity of different technologies, caused by high heterogeneity, inflexible legacy systems and different data sources, seems to be a major drawback to the definition of distinct product lines and subsequent technologies such as CBD and MDE:

- **Software Product Lines:** In Software Product Lines, design and development occur in a particular context, sharing common features and solving common problems. Product families may either be tailored around complete business solutions or a series of related products. They concentrate on reusable implementation artefacts as well as frameworks, processes and tools.

With reference to systems integration, the multiplicity of different technologies, caused by high heterogeneity, inflexible legacy systems and different data sources, seems to be a major drawback to the definition of distinguished product lines. In a product line covering Customer Relationship Management systems, for example, products may be highly integrated with third party logistics and finance systems. Including support for any potentially attached systems undermines the advantages of a delimited context, while excluding them will force development to occur outside the industrialized concepts. An additional drawback is the de-facto development of one-off solutions per customer. Barely any solution operates in the same environment or is interconnected with the same type of systems. The initial set-up cost for software product lines may therefore be contraindicative as the return of investment cannot be ensured.

- **Component-Based Development:** According to Greenfield et al. (2004, p. 130), development by assembly with software components has certain requirements that must be met: Platform independent protocols (e.g., XML), self-description of components (formalized and enhanced meta-data within components), deferred encapsulation (allowing the interweaving of additional functionality), assembly by orchestration (machine-controlled interaction and management of components), and architecture-driven development (to promote the availability of well-matched components).

With regard to systems integration, major difficulties to technically applying development by assembly cannot be found. However, the assembly approach relies on systematic reuse and thus on a methodical approach in a clearly-delimited context that may not be easy to define in a constantly changing environment encompassing different customers. This context also has an influence on the availability of predefined software architectures as well as the number of reusable components. Furthermore, as stated by Vogler (2006, p. 146) and Gorton et al. (2003, p.1), systems integration standards and overall IT architectures are not common as of today. The most important challenge to be met is the definition of a component-based systems integration architecture in which development by assembly may occur. Ideally, this architecture can be reused on a constant basis to ensure a positive return on investment.

- **Model-Driven Development:** The concept raises the level of abstraction to reduce complexity and express business concepts more efficiently. It consists of domain-specific modelling languages and model transformation engines and code generators. The former allow a context-free description of the intended products of a product line, whereas the latter provide model transformation to a lower, more specific model or possibly the generation of source code.

For systems integration, the efforts required to define a domain-specific language could become an obstacle, especially if applied to product lines with a limited number of expected products. With reference to Software Product Lines, the scope of a DSL cannot be clearly delimited as each product may need to be integrated with other external systems. Furthermore, to automate the development process by transforming models to a lower level or generating source code, transformation engines and code generators have to be implemented which also impose high set-up costs and are only beneficial if reused on a constant basis.

In conclusion, it can be said that eventually it all comes down to a positive return on investment. Implementing concepts of software industrialization requires significant upfront expenditures. This investment must pay off within a reasonable amount of time. For the time being, industri-

alization concepts in their current form do not seem to do so in a complex, heterogeneous, and constantly changing environment.

Chapters 2 and 3 are pursuing the first objective of the present research, which is examining the literature to identify the characteristics of software industrialisation and their prerequisites, pinpoint the particularities of systems integration, and isolate how far current methods of software industrialisation can cope with systems integration. The chapters thus conclude the evaluation phase of the research cycle.

Chapter 4 analyses each concept of software industrialisation detail and develops an alternative way for its successful implementation within a systems integration context. It does so by adding new features to alleviate implementation, omitting those that can't be achieved, and aligning them with the previously introduced integration metamodel. The result is an approach towards industrialized systems integration which allows for a positive return on investment and a reduced complexity in integrated IT landscapes.

## **4 The Industrialization of Software Development in Systems Integration**

The previous three chapters set the basis for industrialisation in general and the application of industrial principles to software engineering in particular. Chapter 3 has furthermore pointed out that, despite most concepts being well researched, industrialization in the field of systems integration is still immature. Software product lines, component based development and model-driven engineering in their original form are far too complex to be applied in a heterogeneous and customer-centric environment.

This chapter therefore introduces a novel approach to the above industrial concepts. It starts with an organizational model for software product lines in systems integration, which serves as the basis for the underlying component-based development and model-driven engineering. The developed model carefully considers the requirements and characteristics of systems integration while retaining specialization as the fundamental idea of software product lines. Subsequently, a new method of implementing component-based development, aligned with the integration metamodel, is presented. The combination of both ensures compatibility between different products of a product line and even across product lines. For model-driven engineering, the situation is not as promising. Due to its prevailing issues and shortcomings described in section 2.5, MDE is not yet believed to be mature enough for implementation in its entirety. However, expecting future advancements from academia and practice, fundamental aspects such as basic domain-specific languages and simple code generation are being applied where possible in the context of this work.

### **4.1 An Organisational Model for Industrialised Systems Integration**

Product lines combine the development of closely related products and introduce formal and repeatable processes. They can be found in many different industries and implicitly enforce specialization across the involved parties (Clements et al., 2007, p. 6). Coordinating and struc-

turing the production of such complex goods and services in order to maximize efficiency is commonly referred to as ‘organisation or work’ (Wöhe and Döring, 2008, p. 113). The constitution of an organizational structure depends on the economic objective of the enterprise as well as the products offered. In the context of the present work, it must be assumed that a systems integrator’s objectives and products differ significantly from those offering independent and mostly standardized software (see section 3.3). Currently available implementation approaches for software product lines are therefore regarded insufficient for this specific field. Thus an alternative organizational model for industrialized systems integration has been developed.<sup>1</sup> It allows specialization with the help of software product lines based on the specific needs of a large systems integrator. It starts with the identification of organizational structures for software product lines, puts them in contrast to the requirements of systems integration, and results in a three-layered model for software product lines in systems integration.

#### **4.1.1 Derivation of Organizational Structures for Software Product Lines**

To identify the organizational needs of software product lines, possible strategies and target scenarios of industrial software development are identified based on an approach developed by Lang (2004, pp. 200 ff.). Originally aimed at software development in general, it has been adjusted for industrialized systems integration in particular. The resulting strategy and target scenarios then lead to an organizational structure for industrialized software product line engineering and application engineering.

##### **4.1.1.1 Strategy and Target Scenarios**

The strategy of an enterprise describes the overall objectives of a business venture (Lang, 2004, p. 203), i.e. its generic goals. For an economically performing enterprise, literature identifies profit, growth, competitiveness, and product quality as the most important strategic variables (Wöhe and Döring, 2008, pp. 75 f; Töpfer, 1985, pp. 245 f; Fritz et al., 1988, pp. 573 ff.). Fritz

---

<sup>1</sup> Parts of section 4.1 were published by the author in Minich et al. (2010) during the course of research.

et al. and Töpfer furthermore assume a hierarchical dependency with competitiveness as the super ordinate strategic objective, consisting of solvency and increasing profit; which is a direct result of growth and product quality (Lang, 2004, p. 207).

Formal goals of an enterprise describe operations and methods to reach the generic goals previously identified. In the context of software industry, the generic goals can be operationalized with software development, software maintenance, software integration, software consulting, and software training (Hansen and Neumann, 2001, pp. 533 ff; Baaken and Launen, 1993, pp. 58–60). For the present work, the subject matter is limited to software development and software integration in the context of the three industrial key principles introduced above. With reference to the definition of EAI in section 3.1, it is assumed that software integration occurs within or as part of software development. It will therefore not be considered separately. In favour of a wider applicability of the approach, a more detailed specification of the formal goals will be omitted at this time. Their embodiment can be observed in sections 4.2 and 4.3.

To derive an organizational structure, target scenarios supporting the strategy are defined. As the super ordinate objective of an enterprise is found in its competitiveness on the market, these scenarios are accumulated from external and internal influencing factors (Frese and Noetel, 1992, p. 76), i.e. aspects implied by the market and those an enterprise can influence itself.

According to Frese and Noetel (1992, pp. 77–82), external factors influencing the production process are the degree of product standardization, from stock or individual orders and customer influence during production. In the context of software development, product standardization can be mapped directly. Whether from stock order (commercial off-the-shelf) or individual order (customer specific developments) will not be delved into in this work, as, in the context of this work, it resembles the degree of standardization, i.e. the applicability to a range of customers or only one. The third variable, customer influence, will also be mapped to the degree of standardization: On commercial off-the-shelf products, customers usually have no influence; they have medium influence on customized products, and high influence on individual devel-

opments. The external influencing factors of software development thus allow the following three strategic directions (Lang, 2004, p. 250):

- Standard software
- Customized software
- Individual software

Internal influencing factors of an enterprise include cost, product quality, and quality of delivery (i.e. delivery of manufactured products to the customer on time and in sufficient quantities) (Frese and Noetel, 1992, p. 81). Projected on the software industry, product quality can be applied as well. According to several studies on buying behaviour in the software market, cost is not decisive for purchase decisions. It does, however, influence the decision between standard and individual software (Lang, 2004, p. 240; Balzert, 1996, p. 32). However, standardization is a key objective in industrialized software development and thus part of the competitive advantage. In contrast to Lang, it is therefore still considered as an influencing factor. Quality of delivery must be separated into delivery on time and delivery in sufficient quantities. The latter can be omitted due to the intangible nature and simple reproduction of software. Delivery on time may also be omitted for software development in general (Lang, 2004, p. 251) as no complex logistics as in manufacturing industries are needed. However, the author believes that for standard software, it may indeed be relevant. In competitive and volatile markets, suppliers may benefit from quickly reacting to market demands and providing respective products with short time-to-market. This applies in particular to software product lines which rely on a product portfolio derived from market demands (see product management and product line requirements engineering in section 2.2.3). Combining previous internal influencing factors, the following strategic approaches may be identified:

- Time-to-market and quality leadership at the cost of higher outlays
- Cost and time-to-market leadership at the cost of lower quality
- Quality leadership at the cost of higher expenditures and longer time-to-market
- Cost leadership resulting in longer time-to-market and lower expenditures

- Time-to-market leadership at the cost of higher outlays and lower quality

As quality and cost are diametrically opposed to each other (Balzert, 2008, p. 196), a quality and cost leadership cannot be achieved, regardless of any time-to-market strategies.

Combining external and internal influencing factors shows the available strategic alternatives for system integrators in the context of industrialization as follows:

Table 4-1: Strategic alternatives for industrialized systems integration

	<b>Standard Software</b>	<b>Customized Software</b>	<b>Individual Software</b>
<b>Time-to-market and quality leadership</b>	I	II	III
<b>Cost and time-to-market leadership</b>	IV	V	VI
<b>Quality leadership</b>	VII	VIII	IX
<b>Cost leadership</b>	X	XI	XII
<b>time-to-market leadership</b>	XIII	XIV	XV

By identifying the major strategic alternatives for economically performing system integrators, a first step towards an organizational structure is made. However, the strategic alternatives may not be sufficient to provide a sound basis for organizational structures. This is because every enterprise faces a given level of uncertainty in its business (Frese, 1998, pp. 73 ff., 113 ff.). Projected on software development, the major sources of uncertainty can be found in the following (DeMarco and Lister, 2003, p. 20):

- Unclear requirements of the product
- Interaction of the software with other systems and end users
- Changes in the environment such as objectives and requirements of the software
- Resource availability within the developing enterprise
- Management capabilities of the developing enterprise
- Reliability of the supply chain

- Political interests within the developing enterprise
- Potential conflicts between project stakeholders and their differing approaches to solutions
- Technical innovations in the market and development project
- Scalability due to additional functionality

In the following, only those sources directly influential to the process of software development within the strategic alternatives introduced above are considered. It is assumed that the remaining sources have no influence on the organizational structure, nor can they be alleviated by altering it. The relevant ones are combined to the following three factors of uncertainty:

- **Complexity**, influenced by unclear product requirements, interaction with other systems and end users, and scalability of the software.
- **Dynamic**, influenced by changes and innovations in the environment and the development project.
- **Novelty**, influenced by novel requirements of the product to be developed, i.e. requirements or technologies not previously covered in any other product.

As introduced in section 2.2, software product lines can be distinguished between product line engineering and application engineering. The former provides a basis in terms of architectures, reusable components, process frameworks and other development artefacts. The latter utilizes these artefacts to develop products on top of the product line infrastructure. Based on this differentiation and the previously derived strategic alternatives and factors of uncertainty, the following two scenarios of industrialized software development are identified:

- a) Quality oriented development of standard software (strategic alternative VII) in a complex environment with a high degree of novelty. Scenario a) applies to software product line engineering. Considering the impact of a defect introduced into the product line (and thus all subsequent products), a high level of quality outweighs the objective of lower cost and shorter time-to-market. It is also assumed that developing a new software product line is highly complex (Linden, 2007, p. 278) and comes with a high degree of novelty. As of a product line's fundamental and long lasting nature, a low dynamic is assumed.

b) Cost and time-to-market oriented development of customized software (strategic scenario V) in an established environment with low dynamics and low complexity. Scenario b) applies to application engineering within a software product line. As the primary motivation for software product lines is based on economic considerations (Linden, 2007, p. 3; Clements et al., 2007, p. 17; Pohl et al., 2005, p. 9), lower cost and shorter time-to-market outweigh quality in this scenario. This is only possible as the required level of quality is automatically ensured by utilizing artefacts of the respective product line. As products are developed within the boundaries of a well-known environment, novelty and dynamics can be considered as low. Likewise, complexity is reduced due to the reuse of common parts in a predefined platform and architecture.

Besides scenarios a) and b), enhancement of software product lines and further development of products also are conceivable scenarios. For the latter, the author assumes that it will only occur within the boundaries of the product line as otherwise the benefits of systematic reuse will not be reached and new products also could no longer benefit from already existing core assets. For product line enhancement in turn, scenario a) applies. Both will therefore be omitted for the development of organizational structures.

#### **4.1.1.2 Suitable Forms of Organization**

The decomposition and rearrangement of super ordinate tasks into smaller, more manageable ones is the key principle of organization. According to Wöhe et al. (2008, p. 118) and Grochla (1995, pp. 96 ff.), work can essentially be decomposed based on activities and objects of work. Activity-based decomposition creates functional units, adequate for homogeneous and constant tasks on multiple work objects. Object-based decomposition creates divisional units, responsible for all tasks required to create or modify a heterogeneous and potentially dynamic work object (Wöhe and Döring, 2008, p. 118; Grochla, 1995, pp. 97 f.). The degree of either decomposition is limited by the dependencies and interactions between the prospective functional or divisional units (Grochla, 1995, p. 96). In the following, the typical processes and work objects of the two

scenarios introduced above are discussed, their interdependences described, and a suitable form of organization for each derived.

## Software Product-Line Engineering

Software product-line engineering, reflected in scenario a), consists of several processes to define its scope and develop the infrastructure and core assets to be utilized in future products.

As described in detail in section 2.2.3, the primary processes can be subsumed as follows:

- Product Management
- Domain Requirements Engineering
- Architecture Design & Development
- Core Asset Development
- Domain Testing
- Software Integration

Developing and implementing a delimited software product line is a singular undertaking for an enterprise, although maintenance and optimization are necessary during its lifetime. While the primary processes remain the same throughout the development of other domains within the same enterprise, work objects (architecture or core assets, for instance) are unique and require different production steps for their completion, often involving complex or novel technologies. A decomposition based on work objects thus seems more appropriate than an activity-based breakdown. With regard to software development, it has been observed that independent and small teams most efficiently complete complex tasks due to easier collaboration and more intense communications (Balzert, 2008, pp. 81–84). It is therefore assumed that software product-line development, as a complex form of software development, must be broken up into small and individual packages. This decomposition, however, is limited by the interdependencies of the underlying work objects (Grochla, 1995, p. 96). It must be decided if interacting objects should be merged and represented in a single organizational unit or if they can remain separate in favour of smaller and more efficient units.

As the present work aims at a generic organizational structure, decomposition is limited to the level of primary processes. For very large products, however, it may be feasible to further divide activities on a functional or technical basis if they can be modularised. For software product line engineering, decomposition of work objects is suggested as follows:

- **Product Management** has an observing and strategic focus. It cannot for instance initiate any changes to the business domain because of technical requirements. As it defines the overall strategy of the software product line (i.e. its generic goals), it is independent from subsequent processes and has only limited interaction with them. Product Management can therefore be implemented in its own organizational unit.
- Software product-line development occurs in a complex environment with a high degree of novelty. Functional and non-functional requirements as well as variability points and commonalities are highly dependent on their technical feasibility. In a new environment, however, technical feasibility cannot be assured from the very first beginning. It is therefore assumed that requirements definition is an iterative activity between the **product line requirements engineering** and the subsequent **architecture design & development** process. Due to their intense interaction and thus increased communication requirements, both processes should be internalized into one organizational unit. The work objects of this unit are a requirements model for functional and non-functional core assets and common product line architecture, defining the technical requirements and specifications for the core assets to be implemented.
- Once the requirements and architecture of a software product line are defined, **core asset development** may occur by applying a conventional software development process. The problems and their conceptual solutions are defined, and a basic, technical feasibility is ensured. Consequently, core asset development is expected to have only little interdependencies with the Architecture Design & Development process. The Develop Core Assets process is therefore implemented in its own organizational unit. Based on the nature of the core assets to be developed, it is also conceivable to break it down into different teams, developing subsets of software components, development tools, test cases, or any other core

asset. Further subdivision of core asset development would also have the advantage that distinct teams are responsible for particular assets throughout their lifecycle and may also take over maintenance later on. Each organizational unit thus has one or more core assets as work objects it is responsible for.

- The **Domain Testing** process is responsible for testing all core assets within the context of the software product line. While unit testing of the core assets can be done within the Core Asset Development process, domain testing requires a certain degree of integration in order to identify defects originating from the interaction of core assets with each other and the overall product-line architecture. In turn, software integration may require additional tests to ensure the correct interaction of integrated software. It is therefore assumed that both processes, Domain Testing and Software Integration, interact closely with each other and should be internalized in a joint organizational unit.

Based on above explanations, the division of work for scenario a) can be depicted as follows:

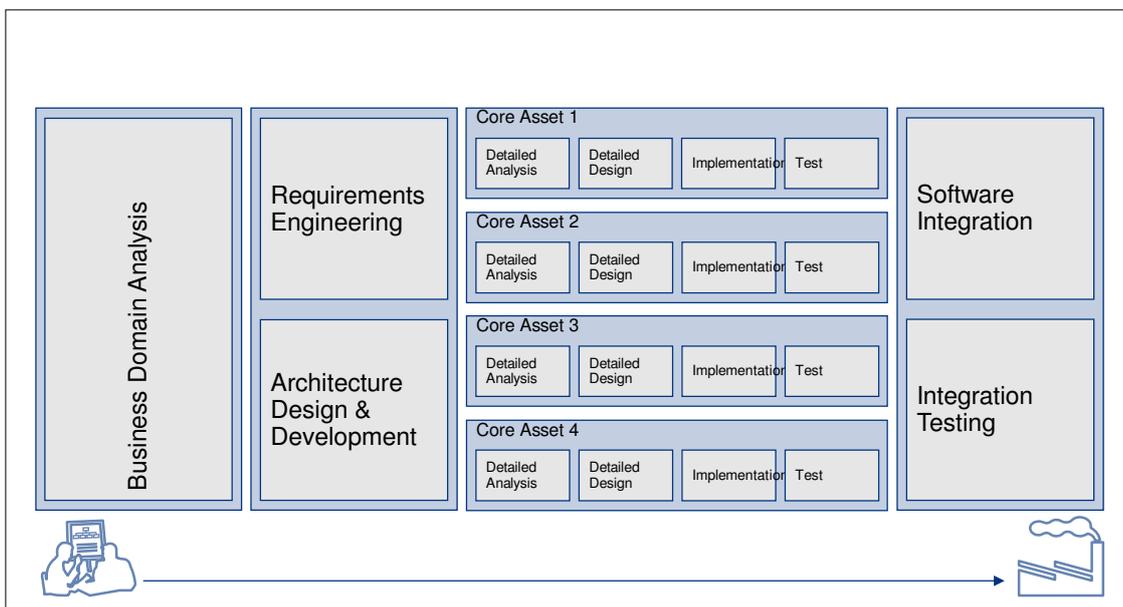


Figure 4-1: Organizational Structure for Software Product Line Engineering

## Application Engineering

Reflecting scenario b), in application engineering the products of the product line are built by utilizing the architecture, core assets, and commonalities developed in software product line

engineering. As discussed in detail in section 2.2.4, the primary processes of application engineering can be subsumed as follows:

- Application Requirements Engineering
- Application Design
- Application Realisation
- Application Testing

Contrary to implementing a software product line, application engineering is a recurring activity. Although each application is slightly different, large parts of their functionality remain identical. Variability points are regarded as customer-specific configurations of standardized products, i.e. the core principle of software product lines. The objects of work (i.e. products) are therefore considered homogeneous and stable. Similar products have been previously developed and product architecture and technology are well understood. Interdependencies between the primary processes are therefore expected to be low. Compared to scenario a), application engineering is characterised by low dynamics and low complexity. Consequently, an activity-based decomposition of work is more appropriate than an object-based one (Wöhe and Döring, 2008, p. 118; Grochla, 1995, pp. 97 f.). The degree of decomposition is determined by a preferably small team size in software development and the required interaction between the resulting functional units. It must furthermore be decided if interacting functional units should be merged and represented in a single one, or if they can remain separate in favour of smaller and more efficient teams. As the present work aims at a generic organizational structure, decomposition is limited to the level of primary processes. For very large products, however, it may be feasible to divide activities on a more detailed technical basis. For application engineering, decomposition by activities is suggested as follows:

- **Application-Requirements Engineering** “reuses the domain requirements artefacts to define the [product] requirements artefacts” (Pohl et al., 2005, p. 308). It does not need to be aligned with the subsequent product design process, as the technical feasibility of the requirements is already assured by the software product line architecture. Although cus-

customer specific assets may require a closer interaction, the author believes that their rather small number in industrial development does not justify a general internalization of the design process.

- For **application design**, similar principles as those for application-requirement engineering apply. Technical architecture and variability model are predefined by the software product line and thus assumed to reflect the product requirements. The design is based on predefined and tested architectures and core assets and follows the joint commonality and variability model. Towards product realization, the utilization of standardized and reusable core assets ensures only little interaction between the implementation and the design team. As stated before, only a small number of customer-specific assets is assumed which will only be implemented at predefined variation points. Although this may lead to higher interaction as compared to requirements engineering, the advantages of two separate units outweigh the disadvantages of a joined one.
- Similar to core asset development, the implementation activities in **application realization** are known, well defined, and as they are based on a standardized architecture, do not hold any uncertainties. However, as the product line cannot anticipate all possible or customer-specific combinations or adaptations of reusable assets, product specific tests are required. In general, testing can be divided into component and integration tests (Balzert, 2008, p. 552). Component tests apply to adaptations or new combinations of reusable assets, while integration tests are performed on the new product as a whole. To allow an early removal of potential defects, partial internalization of the product testing process is suggested for component tests. Besides yet unknown component combinations, this allows the application developers to test customer-specific implementations or extensions which are not part of the software product line.
- After internalizing quality assurance on the component level, the **application testing** process remains responsible for the proper interaction of all components and customer-specific extensions of the application and the alignment with the requirements imposed by the software product line. As such, integration tests can only be conducted after completing the application or at least parts of it. Further internalization is not possible.

This decomposition of work also separates knowledge-intensive and creative work, i.e. application-requirement engineering and application design from the rather technical implementation work. Such separation also allows utilizing external resources from low-wage countries without exposing valuable business and product line know-how. A separate test team may then take the implemented applications and test them according to product-line standards. This principle is rather common in software developing companies and known as offshore or nearshore outsourcing. However, the application of both industrialization and outsourcing has not yet explicitly been reported in literature.

Based on above explanations, the division of work for scenario b) is presented in Figure 4-2. In contrast to the organization of software product-line engineering, the different teams in application engineering specialize on certain functions within the product line instead of a particular product itself. It has to be noted that the organizational model has nothing to do with the actual development process being used (e.g. waterfall model, Rational Unified Process, or agile programming), although it may have an influence on it.

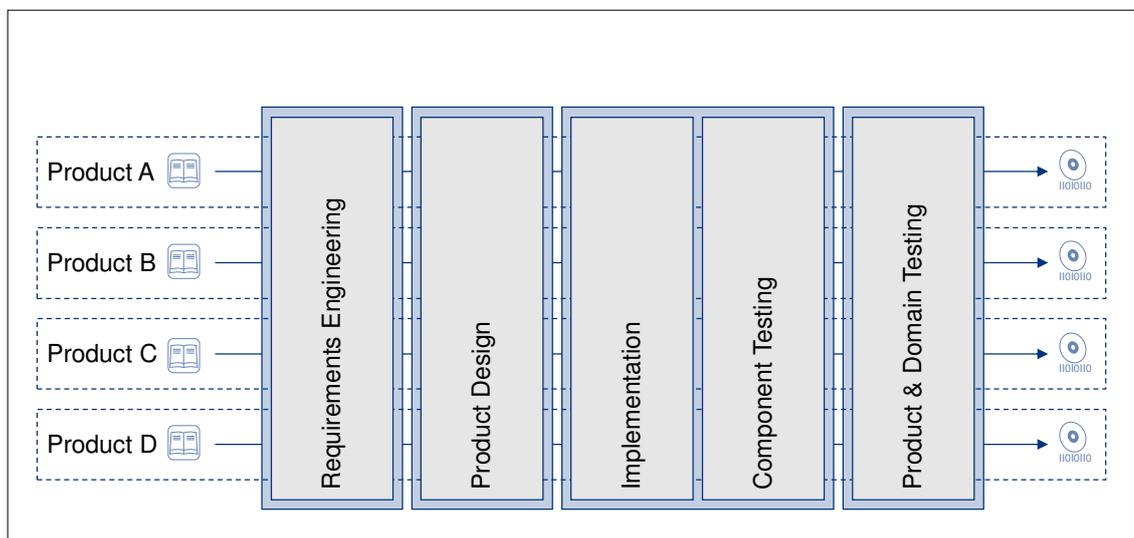


Figure 4-2: Organizational structure for product development

### 4.1.2 A Three-Layered Model for Software Product Lines in Systems Integration

As previously described, it seems disputable as to whether the concept of software product lines in its original form can be applied to the field of systems integration. Revisiting section 3.3, the major issues can be subsumed as follows:

- **Integration across software product line boundaries:** Conventional software product lines have a preferably narrow scope to be most powerful. Systems integration however requires considering a wide variety of different products with a very different scope. If the software product lines of a system integrator are not compatible with each other, integration will inevitably occur outside their boundaries.
- **Multiplicity of technologies:** With reference to systems integration, the multiplicity of different technologies, caused by high heterogeneity, inflexible legacy systems, and different data sources, seems to be a major drawback to the definition of distinct product lines. Too much scope which will most likely be used only once would have to be added.
- **Uncertain return of investment:** As systems integration produces extremely customer-specific solutions, the minimum number of product instances to break even cannot be ensured. The cost for a substantial software product line may outweigh its potential savings.

Considering the above, it can be assumed that systems integration occurs in a complex and dynamic environment. Its work products are heterogeneous and novel, which suggests a divisional form of organization (see section 4.1.1.1). This statement is backed by current organizational structures of major systems integration companies such as Accenture, Capgemini, HCL, IBM and T-Systems. All of them are organized in a vertical structure based on industries such as automotive, aerospace, public, travel & transport and finance (Pierre Audoin Consultants, 2009). This form of organization helps to better understand a customer's requirements and provide him with specialized solutions. Furthermore, integration of different IT systems mostly occurs within the boundaries of a particular industry. An automotive supplier, for instance, will hardly need to integrate any of the systems with an e-government solution from the public sec-

tor. Yet integrating an SAP accounting system with a logistics application from one of the suppliers may be necessary.

Any integration model or architecture should therefore at least support the typical systems of the respective industry. Implementing such architecture within a software product line, however, would broaden its scope far beyond being efficient and thus feasible for industrialization. This especially applies to reusable core assets which would be too generic to provide any benefit. To overcome these problems, a novel three-tiered approach for software product lines in systems integration has been developed. It adds a layer of abstraction on top of the software product lines which spans across a strategic business segment the system integrator is active in. The additional layer is responsible for compatibility across products from different product lines, provides joint core assets, abstract specifications and architectures, and defines which software product lines will exist. By combining conceptual efforts required in every software product line of a business segment, the major issues shown above can be partially alleviated. Due to the size of such a business domain, which, after all, consists of several interrelated product lines, the concept is only feasible for large systems integrators as defined in the scope of this work.

#### **4.1.2.1 The Business Domain Layer**

The Business Domain Layer is a new super ordinate layer that spans over a complete division or business segment within a system integrator's organizational structure. It identifies the major requirements of the business domain in scope and conceptually defines fundamental core assets, technologies, and systems typically used therein. The development of an abstract system landscape and integration architecture ensures the interoperability of different systems and product lines within the business domain.

#### **Processes of the Business Domain Layer**

To ensure interoperability and continuous reuse across different product lines, the new Business Domain Layer consists of the following four core processes:

- **Business Domain Analysis** explores the typical IT landscape of the business domain in scope and identifies areas of expertise required to develop and provide the products and services under consideration. Similarly to software product lines but on a higher level, it identifies the recurring problems and known solutions. The derived information results in a domain model describing the business segment and its typical IT landscape as a whole.
- **Portfolio Definition** evaluates the information from the domain model and develops a product portfolio for the particular business segment. The portfolio covers typical applications and solutions for the most important business services of the segment and identifies the portfolio elements to be supported. Based on these portfolio elements, the system integrator can define distinct software product lines which will produce the applications required by the customers. It is important to note that these product lines are not related to each other with regard to contents. They rather reflect the typical and most important systems required by a customer of a certain business domain.
- **Architecture & Roadmap Definition.** Once the scope is defined by Business Segment Analysis and Portfolio Definition, an integration architecture and basic product line definitions as well as a component framework applicable for all product lines must be developed. As different product lines have different functional and technical requirements, this architecture will mainly exist in an abstract form. It defines standardized structures and features, allowing the later integration of products from different product lines within the same business domain. An alignment with the integration metamodel introduced in section 3.2 ensures a common understanding of integration across all underlying product lines. To ensure an aligned advancement throughout their lifecycle, the process also defines a technology roadmap applicable to all software product lines.
- **Core Asset Development** develops reusable assets, applicable to all or many software product lines within the business segment. Such joint core assets may, for instance, be development tools and processes or joint software development patterns. Core Asset Development may also include the production of reusable software components equal to each product line. A typical example therefore would be an interface component to a particular

technology or an integration middleware, as there is a high chance that these will be required in more than one product line. Alternatively, assets may be harvested from already existing software development projects and adapted to the needs of the business domain. Such core assets explicitly include the required integration infrastructure such as middleware, message brokers, or databases as explained in section 3.1.3. In addition, more sophisticated concepts of software industrialization such as domain specific languages (DSLs) and their respective transformation engines and code generators may be developed. They are applicable to the overall business domain and may be inherited to satisfy the needs of the particular software product line.

The work objects of the above processes can be combined into a product line skeleton which will be instantiated by a particular software product line. It is believed that this new layer for software product lines will have a positive effect on the previously mentioned major issues of industrialized systems integration. The first concern, integrating products from different product lines, may be solved by the Portfolio Definition and Architecture & Roadmap Definition processes. Furthermore, the core asset development process provides a joint integration infrastructure usable by all underlying product lines. The compatibility is ensured by an abstract architecture applicable to all software product lines within the business domain. The second concern, multiplicity of technologies, can be alleviated by a joint technology roadmap. It will limit the number of utilized technologies within the software product lines and thus reduce their heterogeneity. This does not reduce the heterogeneity introduced by legacy systems or third party applications. It may, however, be alleviated by joint interface components to such systems across multiple product lines. The third concern, ensuring the return of investment, can also be attenuated. Software product-line engineering may instantiate a predefined skeleton which results in a greatly reduced effort in business domain analysis, product line requirement engineering, architecture design & development, and core asset development. Due to reduced efforts and thus cost, the breakeven point of a software product line may be reached earlier. Although this approach may not be as efficient as traditional software product lines, it still helps to

advance the breakeven point in systems integration and to have a positive effect on the overall product quality due to reusing quality-proven components.

## Organizational Structure of the Business Domain Layer

Revisiting section 4.1.1, a quality-oriented development of standard software in a complex environment with a high degree of novelty is assumed for the implementation of the business domain layer. As it is a singular and novel undertaking for an enterprise, a decomposition based on work objects again seems most appropriate. It is limited by the degree of interaction between adjacent processes as follows:

- **Business Domain Analysis** and **Portfolio Definition** are closely related. Potential products are derived out of the market requirements identified during business domain analysis. Subsequently, a product portfolio is developed and the market continuously observed to adapt the portfolio to potential changes (Bliemel and Fassott, 2006, pp. 4724 ff.). A joint organizational unit for Business Domain Analysis and Portfolio Definition, especially in the initial set up of the Business Domain Layer, therefore seems most appropriate.
- The **Architecture & Roadmap Definition** process is based on a previously-defined product portfolio. As the latter is driven by market demands, it only has a weak dependency on the underlying architecture and technology roadmap. Vice versa, product line and integration architectures are defined on an abstract level only and thus leave enough room for proper adaptation during software product line engineering in the product line layer. In favour of small and efficient teams, further internalization into the previous organizational unit is therefore not necessary.
- Once the architecture of the business domain layer is defined, **core asset development** may occur independently. The problems and their conceptual solutions are defined, and a basic, technical feasibility is ensured. Consequently, core asset development is expected to have only little interdependencies with the Architecture & Roadmap Definition process. The Core Assets Development process can therefore be implemented in its own organizational unit. Based on the nature of the core assets to be developed, it is also conceivable to break it

down into different teams. These teams may then be responsible for particular assets throughout their lifetime and take over their maintenance.

With the business domain layer spanning several software product lines, it becomes possible to serve a customer's integration needs of typical software systems in such a business domain. Needless to say, a reallocation of tasks to the new layer also has implications on the software product-line engineering processes. These are described in the following.

#### **4.1.2.2 The Product Line Layer**

After implementing the Business Domain Layer, the Product Line Layer must be established. It consists essentially of several software product lines identified in the Portfolio Definition process of the Business Domain Layer. The Engineering processes of these software product lines differ only marginally. The most obvious variance to conventional software product-line engineering is the lack of the Business Domain Analysis process and a reduced product-line requirement engineering process. These functions are now incorporated in the Business Domain Layer and provide their findings to the subsequent product lines. All other processes remain the same but must adhere to the specifications and utilize the provided core assets from the business domain layer.

Implementing a software product line of a specific business domain as well as an integration infrastructure is a singular undertaking for an enterprise. Despite being aligned with the layer above, most work objects (e.g. architectures, reusable software components, instantiated abstract assets) are unique and require an activity-based breakdown. Derived from traditional software product-line engineering, it must be decomposed into small and individual packages. It is therefore assumed that the organizational structure will remain the same. The omitted Business Domain Analysis process was implemented in its own organizational unit due to very few interactions with other processes. The product-line requirement engineering process, on the contrary, was internalized with Architecture Design & Development. This internalization remains the same, as only product identification and definition are moved into the Business Domain Layer. Technical scope definition, such as variability management, for instance,

remains and requires close interaction with the Core Asset Development process. Its organizational structure therefore will not be changed. The new software product line engineering processes are defined as follows:

- **Product-Line Requirement Engineering** inherits the generic product and technology roadmap as well as functional and non-functional requirements which are defined by the business domain analysis and portfolio definition processes within the business domain layer. The requirements for each product are further elaborated and an extended commonality and variability model of the product line is derived. Within the boundaries set forth by the business domain layer, product-line requirement engineering also defines the tools and technologies utilized for production. Necessary changes to the generic product and technology roadmap are being fed back to the business domain layer.
- **Architecture Design & Development** transforms the scope defined in requirement engineering into a technical architecture for the product line and its products. As in conventional product line architecture design & development, this architecture describes the functional parts, defines relationships and interfaces, and establishes rules for their implementation. It thereby follows the architectural requirements inherited from the business domain layer to ensure the ability to integrate its products with those from other product lines. The result is an extended variability model including technical (Pohl et al., 2005, p. 26) and integration related variation points, as well as a detailed allocation of the architecture to the integration metamodel introduced in section 3.2. Unless already existing from the business domain layer, the architecture design & development process will also identify the required core assets for product-line operation (Linden, 2007, p. 58). These may either be developed in the subsequent process or purchased from external suppliers. With completion of this process, proper integration between products of different product lines within the same business domain can be ensured.
- **Core Asset Development** designs and implements the reusable artefacts required by the product line. These may either be defined by the business domain layer or the previous architecture design & development process. Alternatively, assets may be harvested from

already existing software development projects and adapted to the product line's needs. The type of assets does not differ from conventional software product line engineering and includes software components, glue code, variability mechanisms, common processes, design patterns, development tools, domain specific languages, model transformation engines and code generators, and any other reusable asset required for product development. The result of core asset development is a collection of loosely coupled, configurable components, not of a running application (Pohl et al., 2005, p. 27). Adherence to the standards set forth by the business domain layer ensures easy integration with products from other product lines.

The remaining two processes, i.e. domain testing and software integration, are not subject to any changes. As their activities focus on testing and manipulating assets created within the previous three processes, adherence to the requirements from the business domain layer is ensured.

#### **4.1.2.3 The Production Layer**

The actual development of a product within a software product line is reflected in the third layer of the approach which consists of the already known application engineering processes. The latter do not differ from conventional application engineering as described above. The actual integration requirements with other products or external or legacy systems are gathered during application-requirement engineering. Detailed interaction on a functional and technical basis is characterized during application design, and the respective implementation is covered by application implementation. As with other functionality, the latter uses predefined core assets or, if not yet existing, implements new functionality which may be fed back into the product line. Due to the additional functionality and integration relationships with other systems, the application testing process requires additional efforts.

With exception of the above, the processes of the production layer are equal to application engineering in conventional software product lines. A repeated definition of those is therefore omitted.

#### 4.1.2.4 Organizational Overview of Industrialized Systems Integration

Outlining sections 4.1.2.1 and 4.1.2.2, a three-layered approach to specialization in industrialized systems integration was developed. The primary layer needs to be implemented for every business segment a systems integrating enterprise is active in. The subsequent ones reflect the intended software product lines therein. The overall structure can be depicted as follows:

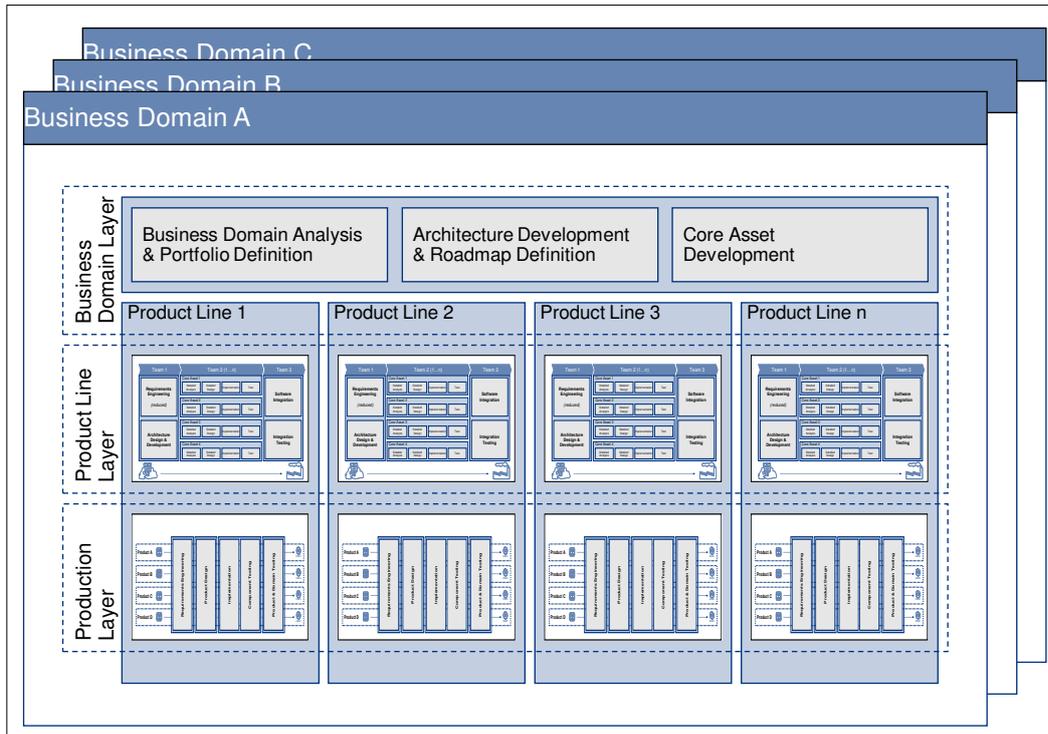


Figure 4-3: Three Layered Approach for Industrialized Systems Integration

#### 4.1.3 Conclusion and Coverage of Research Objectives

Section 4.1 presented a solution to implementing specialization as the first industrial key principle in systems integration. It introduced the general concepts of software product lines and showed that in their existing form they cannot be applied to the field of systems integration. The most important reasons therefore are

- an integration of products across different product lines cannot be ensured,
- a high heterogeneity broadens the scope of a product line at the cost of efficiency, and
- an uncertain return of investment due to too few products being produced.

In spite of these issues, an alternative approach to implementing software product lines as one of the industrial key concepts in the field of systems integration was developed. To do so, a business strategy for systems integration providers was defined and two target scenarios reflecting software product lines identified. Based on these scenarios, the primary processes of product-line engineering and product development were decomposed according to their objects of work and degree of interaction. Out of these, two organizational structures suitable for the implementation of software product lines were derived. In order to reflect the characteristics of systems integration, they were embedded in the three-layered approach as follows:

- **Business Domain Layer:** The uppermost layer spans over a complete division or business segment within a systems integrator's organizational structure. It identifies the major requirements of the business domain in scope and conceptually defines fundamental core assets, technologies, and systems typically used therein. The development of an abstract system landscape, an integration architecture, and integration infrastructure ensures the interoperability of different systems and product lines within the business domain.
- **Product Line Layer:** The product line layer implements the previously-defined product portfolio within several related software product lines. It aligns them with the business domain layer by instantiating abstract core assets and joint architecture frameworks, as well as using technologies defined in the technology roadmap.
- **Production Layer:** The third layer is responsible for implementing the actual application based on customer requirements. This should, as much as possible, occur within the boundaries of the software product line and includes integration requirements with other products or external or legacy systems.

Figure 4-3 depicts the overall structure of the three-layered approach developed. The primary advantage of this concept lies in the consolidation of similar tasks for all software product lines within a given business segment. By moving them to an abstract level, their range of application can be broadened and only product-line specific core assets need to be instantiated and enhanced when applicable. Projected on the characteristics of systems integration, this will lead to the following advantages:

- An integration of products across different product lines becomes possible due to joint architectures and core assets within a given business segment.
- The issue of high heterogeneity will be at least partially resolved due to a joint technology roadmap and compatible core assets. It has, however, no effect on heterogeneity introduced by legacy or third party systems.
- The return on investment can be achieved more easily as large parts of software product-line engineering have been consolidated to a higher layer and only need to be implemented once. This reduces the effort and thus allows breaking even after fewer products have been developed.

With the adoption of software product lines to systems integration, standardization & systematic reuse as well as automated production as the remaining two industrial key principles still need to be implemented. Sections 4.2 and 4.3 demonstrate how this may be achieved in a systems integration context while considering the previously developed organizational structure.

## 4.2 Component-Based Systems Integration

During the introductory chapters, component-based development was identified as the software-related representation of the second industrial key principle, i.e. standardization and systematic reuse. Similar to software product lines, component-based development may not be easily applied to the field of systems integration for various reasons described in chapter 3. Characteristics to be considered can be summarized as high technical and functional heterogeneity, unique IT environments and technology combinations, as well as an uncertain return on invest due to high upfront investments and a low number of very similar products. To overcome these issues, a solution by adapting an already existing and proven approach to the requirements of systems integration has been developed.<sup>2</sup>

---

<sup>2</sup> Parts of this section were published by the author in Minich et al. (2011) during the course of research.

## **4.2.1 Selection of a Suitable Component-Based Development Approach**

Before choosing any existing methodology, the most suitable one has to be identified. This is especially important with regards to the characteristics of systems integration. Currently-known approaches vary in comprehensiveness, strategic alignment (e.g. COTS vs. internal development), and internal structure. Thus the following sections briefly summarize the CBD approaches from chapter 2, present the requirements from systems integration on component-based development, and conclude with the selection of the most suitable approach for further alignment.

### **4.2.1.1 Requirements from Systems Integration**

From a technical point of view, there is no reason why component-based development may not be utilized in systems integration. Platform-independent protocols, assembly by orchestration, or architecture-driven development, to name a few (Greenfield et al., 2004, p. 130), may also be implemented in systems facing integration with each other.

To be most efficient and to achieve a positive return on investment, systematic reuse is inevitable. In a highly volatile and heterogeneous environment, however, reusing a component several times cannot be ensured. The situation from customer to customer is too different to allow for components being constantly reused. This very same situation was found at the beginning of industrialization – unless manufacturers specialized in a specific area of work, reusing product or production artefacts such as tools, machinery, or standardized parts, in other words industrialization – was not possible (Encyclopaedia Britannica, 2005e, p. 209). This results in the question as to how systems integration can be further subdivided to benefit from specialization and how this affects component-based development. The first part of the question has already been answered in section 4.1. A super ordinate layer concentrating on one specific business domain sets the technical and functional constraints for each product being developed including most common integration relationships. Yet these constraints may not yet be sufficient. Even within a certain domain, business processes are too distinct and require further specialization. However, further specialization from a functional point of view defies the objective of including just as

much functionality as required for the majority of integration relationships within a given business domain. In turn, limiting the scope too much forces development to occur outside the software product line.

A possible alternative can be found in an additional technical specialization. This technical specialization does not refer to the actual technology a system is developed in, but the typical artefacts it consists of. Vogler's previously presented integration metamodel (see Vogler, 2006) with its different views on process, desktop, and systems integration provides a reasonable breakdown of typical entities within systems integration. The advantage over a technology-related subdivision lies in the fact that the major differences between customer-specific implementations originate from the process layer (see section 3.1.2). Underlying entities such as integration middleware, message brokers, data bases and structures or application interfaces may often be the same, or at least be standardized for a large variety of customers. The same applies to architectures and technological frameworks. The biggest difference is found in components representing and controlling business logic or workflows; here systematic reuse is rather unlikely unless processes are standardized within the business domain.

Besides systematic reuse, development would also benefit from systems integration standards and architectures which are not yet common (Vogler, 2006, p. 146; Gorton et al., 2003, p. 1). By adhering to such standards, heterogeneity and complexity in the field can be greatly reduced. Similar effects could be observed across various other industries (Encyclopaedia Britannica, 2005e, p. 209; F. A. Brockhaus, 2005, p. 152) and led to a simplified exchange of production artefacts, alignment of production processes, and simplified information exchange.

Concluding the above, the requirements from systems integration for component-based development lie in a further specialization and standardization from a technical point of view so that reuse of components not representing business processes can be maximized. Furthermore, an approach providing standardization and an architectural context to the field would be beneficial, as it reduces heterogeneity and complexity of integration relationships.

#### 4.2.1.2 Discussion of Component-Based Development Approaches

In 2.3.4, currently available component development approaches were presented in more detail. The present section revisits these and discusses whether they match the requirements implied by systems integration.

- **Component-Based Development with COTS Integration** by Haines et al. (1997) as one of the first approaches basically defines a development approach tailored to the particular needs of component-based software development. They qualify existing components in a first step, adapt them with wrappers or additional code if necessary in a second one, and assemble applications based on component frameworks in a third step. During the fourth step, maintenance and enhancement of already deployed components is carried out. Haines et al. also include well-known software development patterns, offering proven solutions to recurring problems. In their approach, they do not offer any detail on the internal structure or architecture of a component-based application but leave this to the individual software development project.

With regard to the requirements of systems integration, Haines et al. limit their approach to generic steps required to identify, adapt, and implement components, either from external suppliers or those internally developed. They do not break them down any further, which means that in an integration context, most of them could not be reused. Also the approach does not provide any architecture which could be beneficial to systems integration. It must, however, be said that their work was developed during the early years of CBD and thus can't be as advanced as others.

- Herzum and Sims' **Business Component Factory** (Herzum and Sims, 2000) depicts a methodology to model, analyse, design, construct, validate, deploy, customize, and maintain a large-scale distributed system. It consists of five dimensions describing component granularity (distributed components, business components, business component systems), architectural viewpoints (technology, application architecture, functionality, and project management), a component development process, distribution tiers (user, workspace, enterprise, resource), and functional layers (utility, entity, process, and auxiliary components).

With regard to systems integration, Herzum and Sims' approach offers some promising aspects. Partitioning components based on their size (distributed, business, and system components), application area (workspace, enterprise, and resource tier), and functional category (utility, entity, process, and auxiliary components) offers sufficient potential for the required technical specialization. It is underpinned by different architectural viewpoints representing a conceptual framework for the overall application. As they also see complete systems as reusable component and consequently apply their methodology on these, the approach may serve as the foundation of an integration architecture. It is, however, not extensive enough compared to Vogler's integration metamodel.

- **Component-Based Software Development with MDA, UML2 and XML** by Andresen (2004) develops a component-based system from three different viewpoints. The layer view differentiates between presentation, controlling, and business logic; the integration view is concerned with integrating the different components with each other by wrappers or XML; the implementation view covers the actual development of software artefacts by means of model-driven architecture. The latter is used to separate business logic from technical implementation. Instead of model transformation engines or code generators, Andresen uses agile development to advance the models until an executable system is completed.

With regards to systems integration, Andresen also offers a subdivision by developing systems from three different viewpoints. They subdivide components based on the technical services they provide, interaction with other components, and business process representation. It thus allows further specialization from a technical point of view. The approach does not offer an architecture suitable for distributed systems or systems integration in general. When extending the integration viewpoint to handle complete systems as components, it may however be possible to serve as a foundation as in Herzum and Sims' Business Component Factory.

- **Building Systems from Commercial Components** by Wallnau et al. (2002) introduces component ensembles and blackboards as abstractions for component-based development similar to patterns in conventional programming languages. An ensemble describes logically-connected components providing a self-contained and reusable business concept. The

components are obtained from commercial markets or, if not available, are being implemented. Blackboards instantiate an actual business process utilizing several of these ensembles, including a definition of data transfer between them. The combination of both is referred to as model problems and the implementation as the respective solutions. During actual system development, model problems are evaluated against the customer's requirements and the solutions utilized.

Although not obviously supported, the concept of component ensembles may be applied to technically subdivide reusable components. Given that business logic and technical services will be separated in different ensembles, systematic reuse may also occur in a systems-integration context. The blackboard concept of the approach allows the selecting of an already-existing technical infrastructure and implementing customer-specific business processes on top of them. The approach does not provide any architecture which could be beneficial to systems integration.

#### **4.2.1.3 Approach Selection for Further Adaptation**

From the above methodologies, only Herzum and Sims' Business Component Factory and Andresen's Component Based Development with MDA, UML2 and XML offer the foundation and enough flexibility to be adapted to the requirements of systems integration. They both allow further division of components into business logic and technical services, increasing the potential for systematic reuse within a product line. Not surprisingly, a generic architecture suitable for systems integration (i.e. the architecture for the system to be integrated itself independently) is not offered, although both approaches do provide concepts to be drawn on. The following table gives an overview of their support of SI requirements as well as their support of software product lines and model-driven engineering in general.

Table 4-2: Comparison of CBD approaches with systems integration requirements

	<b>Andresen</b>	<b>Herzum &amp; Sims</b>
Means of component subdivision	<ul style="list-style-type: none"> <li>• Layer based (system dependent, e.g. presentation, controlling, business logic, integration)</li> </ul>	<ul style="list-style-type: none"> <li>• Granularity based (distributed, business logic, system, federation)</li> <li>• Distribution tier-based (user, workspace, enterprise, resource)</li> <li>• Functionality based (process, entity, utility, auxiliary)</li> </ul>
Means of business logic separation	<ul style="list-style-type: none"> <li>• Layer based (see above)</li> </ul>	<ul style="list-style-type: none"> <li>• Functionality based (see above)</li> </ul>
Architectural component separation	<ul style="list-style-type: none"> <li>• Business architecture</li> <li>• Reference architecture</li> <li>• Application architecture</li> <li>• System architecture</li> </ul>	<ul style="list-style-type: none"> <li>• Technical architecture</li> <li>• Application architecture</li> <li>• Functional architecture</li> </ul>
Support of Software Product Lines	No. Only a brief reference within the reuse chapter.	Partially. Suggests similar own concepts but not as extensive as current SPL approaches.
Support of Model-Driven Engineering	Partially. Extensive use of modelling systems and model-driven architectures. No support for transformation engines and code generators.	No. Only a brief reference to UML as a generic modelling language.
Support of integrating larger systems	Yes. Knows wrapper, connector and data access components.	Yes. Treats systems as components themselves and applies the same integration methods as for smaller components.

From the above comparison, it can be seen that Herzum and Sims' approach offers more possibilities to further subdivide component-based development within a software product line

(Herzum and Sims, 2000, p. 59). Both allow separating business logic from technical functionality (Herzum and Sims, 2000, p. 157; Andresen, 2004, p. 25), and both allow structuring the architecture of component-based systems around this separation (Herzum and Sims, 2000, p. 285; Andresen, 2004, p. 77). The support for development within a software product line is insufficient. While Andresen offers only a brief reference (2004, p. 304), Herzum and Sims suggest their own concepts which are, however, far from the extensiveness introduced in section 2.2 (Herzum and Sims, 2000, p. 536). It stands in contrast to model-driven engineering. While Herzum and Sims only suggest UML as a generic modelling language (2000, p. 533), Andresen conforms with Object Management Group's model-driven architecture. Components and systems are modelled in UML2 (Andresen, 2004, p. 27), and the overall application architecture reflects the MDA standards (Andresen, 2004, pp. 79 ff.). However, instead of using model transformation engines and code generators, Andresen suggests agile development to advance abstract models to implementation specific ones and executable code (Andresen, 2004, p. 83). Both approaches support the integration of large-scale components or systems by providing wrappers, connectors and data access components (Herzum and Sims, 2000, pp. 181 ff.), while in addition, Herzum and Sims treat complete systems as reusable components and integrate them into their overall approach (Herzum and Sims, 2000, pp. 193 f.). Beyond that, they do not offer any integration architecture beneficial to systems integration.

Considering above characteristics and differences, Herzum and Sims' Business Component Factory was chosen for further advancement of industrialized systems integration. The ability to separate between business and technical functionality and further subdivide the respective components into standardized building blocks offers an increased potential for reuse, even in very specific and heterogeneous environments. In addition, the Business Component Factory provides a comprehensive framework reflecting different views on the applications to be developed. The lack of support for model-driven engineering is considered negligible, as at the time of writing, model-driven and automated software development is far from being mature (Selic, 2008, p. 16). However, this does not mean that MDE is not possible.

For a smooth integration of component-based development in the overall approach towards industrialized systems integration, Herzum and Sims' Business Component Factory must be aligned with the organizational model developed in 4.1. This ensures functional and technical specialization during development and provides the basis for systematic reuse. As systems integration requires an additional specialization apart from the business logic, the methodology of the Business Component Factory is also being aligned with Vogler's integration metamodel. By developing components in alignment with the model entities, business logic can easily be separated and the technical foundation frequently reused.

## **4.2.2 The Business Component Factory**

After selecting the component based development model to progress with, it will now be explained in more detail. Herzum and Sims' Business Component Factory is a methodology to model, analyse, design, construct, validate, deploy, customize, and maintain large scale distributed systems. One of its principles is "that any software artefact in a system should be defined in one and only one place, and it should be reused as many times as necessary" (Herzum and Sims, 2000, p. 71). This principle perfectly fits the concept of Software Product Lines, in which a distinctive unit or team is responsible for a particular set of core assets, such as reusable business components. The approach of Herzum and Sims has five dimensions, describing a component-based system and its development process from different viewpoints. As an introduction, the level of detail is kept rather small in the following. More detailed explanations are made in 4.2.3, in which these dimensions are aligned with the organizational model for industrialized systems integration and the integration metamodel.

### **4.2.2.1 Levels of Component Granularity**

The model differentiates five different levels of component granularity. The smallest one is the language class. As it does not have a run-time interface, cannot be deployed independently, and is not network addressable, it is not considered a component in itself (Herzum and Sims, 2000, p. 38).

Next to the language class is the distributed component, which forms the lowest level of component granularity. It has well-defined build- and run-time interfaces, can be independently deployed, and is network addressable (Herzum and Sims, 2000, p. 39). Distributed components may, for instance, be implemented as an Enterprise Java Bean, a CORBA, or a DCOM component, and represent simple functionality such as database connectivity or an input dialogue.

Several distributed components, glue code, resources, documentation, and other deliverables form a business component (Herzum and Sims, 2000, pp. 40 f.). It represents a complete implementation of an autonomous business concept (entity or process), such as a sales order or a credit verification process. As distributed components, business components have well-defined interfaces, can be independently deployed, and are network addressable.

The fourth level of granularity, the business component system, is represented by a “set of business components collaborating together to deliver a solution to a business problem” (Herzum and Sims, 2000, p. 157). Such a solution may, for instance, be an order-management process consisting of the business components process management, purchase order, vendor, and item. A business component system is also provided with a well-defined interface, representing itself as a component (i.e. a system-level component).

The highest level of granularity is the federation of system-level components and consists of “several system-level components federated to address the information processing needs of multiple end users perhaps belonging to different organizations” (Herzum and Sims, 2000, p. 193). This level of granularity is more concerned with interoperability concepts between independent systems than with component technology in its original sense (Herzum and Sims, 2000, p. 238).

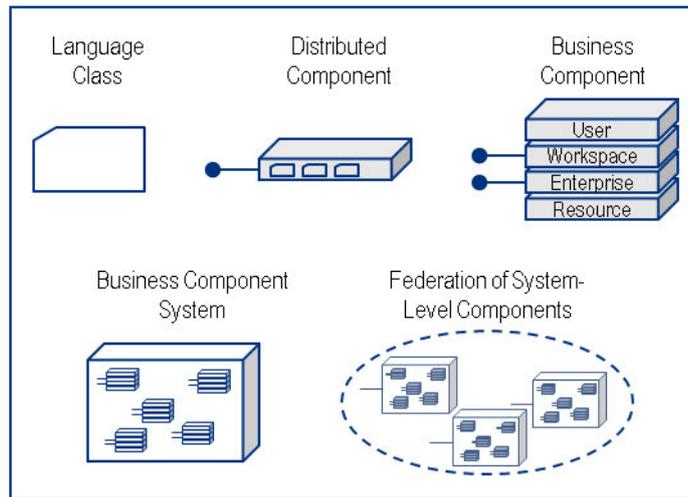


Figure 4-4: Business Component Model – Component Granularity  
(Herzum and Sims, 2000, p. 59)

#### 4.2.2.2 Architectural Viewpoints

The second dimension consists of four architectural viewpoints. They allow defining the approach from a technical, application, project management, and functional point of view.

The technical architecture is concerned with the component execution environment and fundamental services (Herzum and Sims, 2000, pp. 285 ff.) such as activation, deactivation, persistence, error handling, transaction services, and messaging within a component-based application. It furthermore describes an extended integrated development environment, including development, testing, version management, and other tools required during development (Herzum and Sims, 2000, p. 322). The technical architecture is application independent and may be used for a variety of different products.

The application architecture in turn defines the specific characteristics of the application to be built, such as design principles for a high-performance distributed system or a workstation-based single user application (Herzum and Sims, 2000, p. 331). The viewpoint also names more specific patterns and principles such as the layering principle or collaboration patterns. Additionally, programming standards and naming conventions may also be outlined in the application architecture (Herzum and Sims, 2000, p. 376).

A less obvious viewpoint is the project management architecture. It “[...] addresses the concepts, principles, guidelines, policies, and tools required to cost-effectively build a scalable and high performance system with a large team [...]” (Herzum and Sims, 2000, p. 423). These include traditional software configuration and release management, but also dependency management as well as general principles on how to organize and structure the development process or how to measure team efficiency.

The previously mentioned architectural viewpoints can be seen as a “factory setup” for industrialized component-based development. Based here on, the functional architecture defines the actual application to be built. It contains component-based business modelling during which components are identified, business processes and entities are modelled, and reusable components selected (Herzum and Sims, 2000, p. 427). The functional architecture also contains component-based design in which not-yet existing components are technically designed, including interfaces between components and the underlying architecture, as well as persistence of component and business data (Herzum and Sims, 2000, p. 477).

#### **4.2.2.3 Development Process**

An aligned development process is the third dimension of the business component approach. It contains three basic manufacturing processes corresponding to the three levels of component granularity. They are rapid component development for designing, building, and testing an individual business component; system architecture and assembly for architecting, assembling, and testing a complete system using business components; as well as federation architecture and assembly for architecting, assembling, and testing a federation of systems using system-level components (Herzum and Sims, 2000, p. 247). The process is based on the well-known V-Model and consists of requirements, analysis, design, implementation, and several testing and validation phases (see figure below). It is iteratively executed and follows ten characteristics (Herzum and Sims, 2000, p. 283):

1. Component-centric
2. Architecture-centric
3. Component autonomy
4. Collaborations and dependency management
5. Iterative development
6. High level of development concurrency
7. Continuous integration
8. Risk-driven development
9. Strong focus on reuse
10. Mindset of good product development

All phases and activities are strictly component centric, i.e. each functionality or other aspect of the system belongs to one and only one business component.

#### **4.2.2.4 Distribution Tiers**

The anatomy of a component is separated into user, workspace, enterprise, and resource tier. The user tier presents the component on the screen and communicates with the user. It may be stand-alone, plug in, or non-existent at all, and typically has no business logic (Herzum and Sims, 2000, p. 119). The workspace tier implements local business logic and is responsible for interacting with the enterprise tier, serving as a broker for the user tier. Business logic therein may, for instance, include transaction management for more complex transactions initiated by one user but utilizing several enterprise-level resources (Herzum and Sims, 2000, p. 121). The enterprise tier implements enterprise-level business rules, validation, interaction between enterprise components, as well as data integrity. It typically forms the core functionality of business components of a complex, large-scale component system (Herzum and Sims, 2000, p. 122). The resource tier exclusively manages access to shared resources such as databases or files, and shields all higher layers from the technical implementation. It is also responsible for mapping the logical model within the enterprise tier to the actual database implementation, often as its

own “island of data” and only accessible from its own enterprise tier (Herzum and Sims, 2000, p. 123).

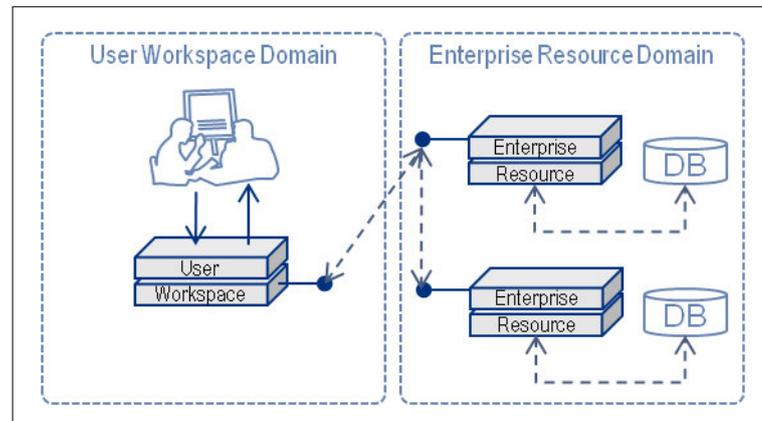


Figure 4-5: Business Component Model – Distribution Domains

#### 4.2.2.5 Functional Layers

The fifth dimension defines four broad functional categories which are utility business components, entity business components, process business components, and auxiliary business components. Utility components can most generally be reused and represent quite autonomous concepts such as unique number generators, currency converters, or an address book (Herzum and Sims, 2000, p. 177). Entity business components represent the logical entities on which a business process operates and is rather specific to a particular business domain. Examples are item, invoice, address, or customer (Herzum and Sims, 2000, p. 178). The actual business process is implemented within a process business component. It is usually unique from one industry or corporation to another and hardly reusable. Based on business process descriptions or use cases, they implement an actual process by utilizing utility, entity, and auxiliary business components (Herzum and Sims, 2000, p. 180). The last category, auxiliary business components, provides services usually not found within a business process description. Such services may be performance monitoring, messaging, or middleware services (Herzum and Sims, 2000, p. 181).

### **4.2.3 Component-Based Development in Systems Integration**

While each of the previous sections describes the process of systems integration and component-based development individually, it is currently unknown how they can be combined to achieve component-based systems integration. The present work contributes to this question by organizationally and conceptually aligning their different aspects and viewpoints, resulting in a novel approach to component-based systems integration.

This is achieved by taking the previously introduced organizational model as well as Vogler's metamodel for systems integration as a foundation. Depending on their intent, the five dimensions of the business component approach will be aligned to either one of the two models. The result will illustrate how component-based systems integration should be organized and how the characteristics and advantages of component-based development can be maintained.

#### **4.2.3.1 Architectural Viewpoint Alignment**

Herzum and Sims suggest four architectural viewpoints in their approach which define the execution environment, development patterns and standards, functional design and scope, as well as organizational decisions, including tools and guidelines. In the context of systematic reuse, these viewpoints are defined once and are then applied to different products. It is therefore most feasible to align them with the different layers of the organizational model for industrialized systems integration. Furthermore, this allows distinguishing architectures based on business domains, software product lines, and eventually customer-specific products.

The project management architecture is “concerned with the set of architectural and organizational decisions, and associated tools and guidelines, required to scale a development to more than 10 developers working together” (Herzum and Sims, 2000, p. 73). Such processes, tools and guidelines are also part of software product lines and considered core assets in software product line development (Clements et al., 2007, p. 33). As suggested in 4.1.2.1, these joint core assets should be implemented within the business domain layer, which ensures the interoperability of different systems and product lines within the business domain. With respect to systems integration, it furthermore allows for consolidation efforts from different product lines of a cer-

tain business domain. Such joint-core assets also ensure the return of investment, even in a volatile and heterogeneous environment. The project management architecture is therefore aligned with the business domain layer. Within the project management architecture, the business domain layer defines mandatory development tools, processes, and guidelines for all underlying software product lines.

The second architectural viewpoint, the technical architecture, is “concerned with the component execution environment, the set of development tools, the user-interface framework, and any other technical service and facility required to develop and run a component-based system” (Herzum and Sims, 2000, p. 73). The technical architecture does not yet cover any functional aspects; it rather defines the technical infrastructure required to implement and operate a component-based application. With reference to software product line engineering, very similar activities that can be summarized under the term architecture design & development (Pohl et al., 2005, pp. 21–23; Clements et al., 2007, pp. 56 ff; Linden, 2007, pp. 59 ff.) can be found. Although an SPL architecture is more comprehensive than what is described within a technical architecture, the technical architecture’s reduced scope perfectly fits into the organizational model for industrialized systems integration: The business domain layer, as a super ordinate instance above software product lines, defines mandatory technologies, architectures, and systems. These artefacts will then be further refined within the actual product line. Having a joint technical architecture ensures the interoperability of different systems and product lines, reduces technical heterogeneity, and helps to achieve a positive return on investment by consolidating architectural efforts. Of course, such architecture would support different technologies as needed; however, it still ensures interoperability from a technical point of view. The technical architecture of the component-based approach is therefore aligned with the business domain and SPL engineering layer of the organizational model for industrialized SI. The business domain layer defines cross product line requirements and standards, whereas the software product line layer defines more detailed technical concepts aligned with the requirements of the products to be developed therein.

The third viewpoint is the application architecture. It is “concerned with the set of architectural decisions, patterns, guidelines, and standards required to build a component-based system that conforms to the extra-functional requirements” (Herzum and Sims, 2000, p. 73). Such can be architectural principles (e.g. noncircularity) and styles (e.g. type-based vs. instance-based), collaboration patterns for transactions, or a system-wide error handling mechanism (Herzum and Sims, 2000, p. 332). With regard to the organizational model for industrialized SI, an application architecture is too specific for a whole business domain due to the variety of many different product lines and products. There may be, for instance, shop floor systems in a factory requiring almost real-time performance, whereas an order management system within the same business domain may be focussed on high scalability. An application architecture only makes sense for applications within a clearly delimited scope which can be found in a software product line. The application architecture is therefore aligned with the SPL Engineering layer of the organizational model. Breaking it further down to the production layer of a single family member would undermine the principles of standardization and systematic reuse and also reduce economies of scope.

The functional architecture is the most detailed architectural viewpoint and is “concerned with the functional aspects of the system, including the actual specification and implementation of a system that satisfies the functional requirements” (Herzum and Sims, 2000, p. 73). It consists of two key processes: component-based business modelling and component-based design. The former can be further broken down into business modelling and functional modelling: “The business modeller’s objective is to produce a model of the problem space that can help in identifying business challenges and business opportunities.” (Herzum and Sims, 2000, pp. 428 f.). “The functional architect, on the other hand, when modelling the business, aims to support the production of a software application” (Herzum and Sims, 2000, p. 429). Comparing this separation of concerns with the organizational model for industrialized SI, business modelling represents the business domain analysis & portfolio definition process within the business domain layer. Functional modelling in turn fits into the architecture design & development process with the SPL engineering layer, as does the second core process of the functional architecture, i.e.

component-based design. Business modelling is therefore separated from the rest of the functional architecture and aligned with the business domain layer of the organizational model. Functional modelling and component-based design should be aligned with the SPL engineering layer to define the functional architecture of the software product line based on the business model developed in the business domain layer.

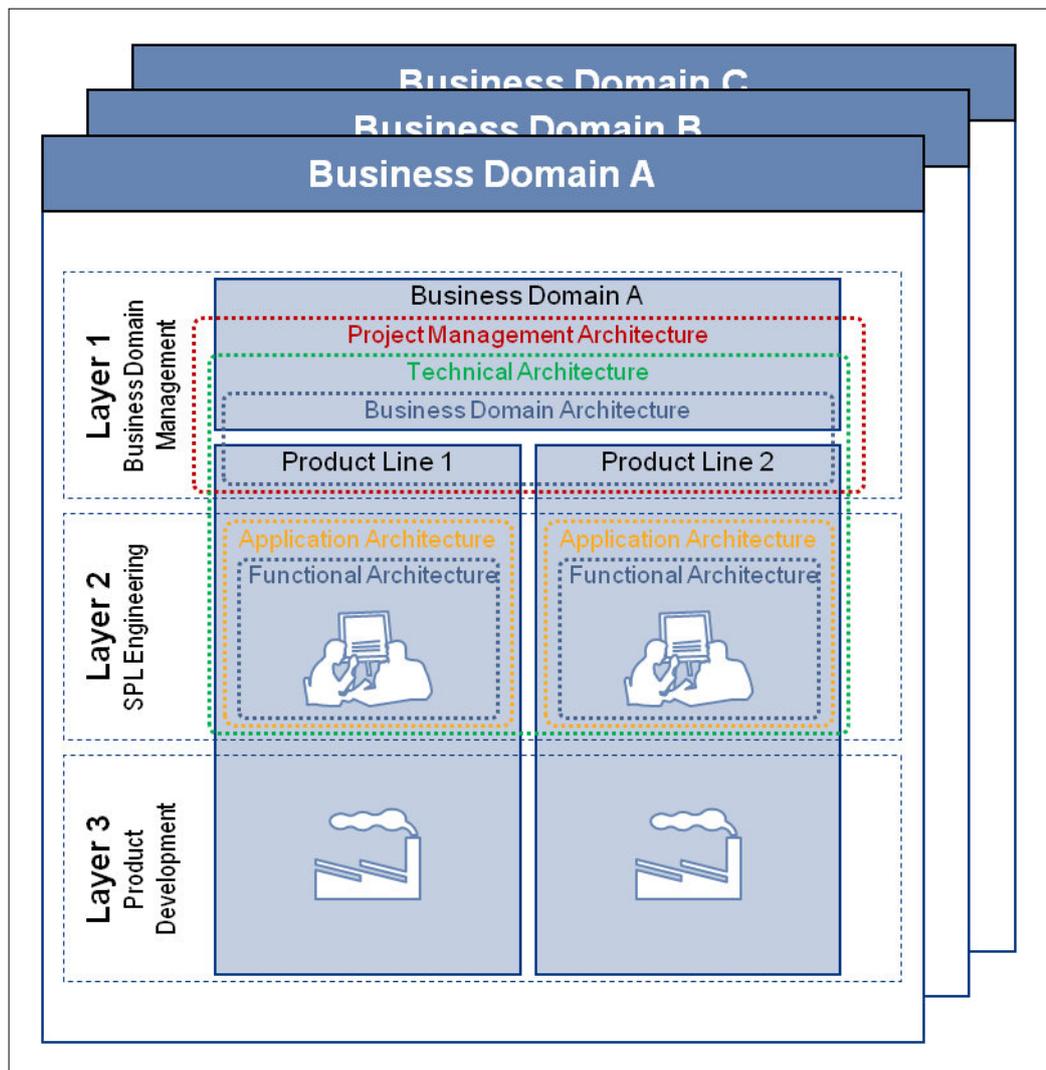


Figure 4-6: Architectural Viewpoint Alignment – Organizational Structure

#### 4.2.3.2 Component Granularity Alignment

The business component factory approach is based on five levels of component granularity. For the present research, the smallest and largest ones, i.e. the language class and the federation of system-level components, are omitted. Even though the former has defined boundaries and is reusable, it does not have a run-time interface, cannot be deployed independently, and is not

network-addressable. The latter, a federation of system-level components, usually represents a whole landscape of component-based systems and cannot be treated as a distinct component due merely to its size.

Differentiating the remaining three levels based on the entity types of the integration metamodel does not make sense. A distributed or business component may be used in any of these entity types such as middleware, applications, or workflows. The granularity levels are therefore aligned with the organisational model for industrialised systems integration. This allows for clear responsibilities for each component as demanded by the business component approach (Herzum and Sims, 2000, pp. 382 f.) and core asset development in software product lines (Pohl et al., 2005, pp. 243 f.). Additionally, definition and refinement of reusable components can be applied throughout the hierarchical structure of software product lines. To do so, Herzum and Sims' approach is being enhanced by differentiating between global distributed components and global business components and local distributed components and local business components.

Global components are developed and maintained on the Business Domain Layer. They provide reusable functionality for all or some of the underlying software product lines. Good examples for global business components are components representing logic entities of a certain business domain such as invoices, orders, bills of materials, or products. In addition, global distributed components may provide standardized interfaces to other systems outside of the product line such as an SAP installation or an enterprise resource-planning tool. Based on the technical or application architecture introduced in 4.2.3.1, some of these global distributed components or global business components can be made mandatory in underlying SPLs to ensure compatibility between products from different product lines of a given business domain.

Local distributed components and local business components are developed and maintained on the software product-line level. As shown in Figure 4-7, they are either developed individually or partially inherited from the business domain. In both cases, they represent a business concept (for local business components) or functionality (for local distributed components) which is unique for the respective product line. Among other non tangible assets, an appropriate choice

of local and global distributed components and business components represent the reusable core assets of a software product line.

These core assets, together with other artefacts like glue code or additional resources, can then be used in the product development layer to actually produce an application within a software product line. With reference to component-based development, this application represents a business component system and therefore the third level of component granularity. As every other component, a business component system must also provide run-time interfaces, be independently deployable, and network addressable.

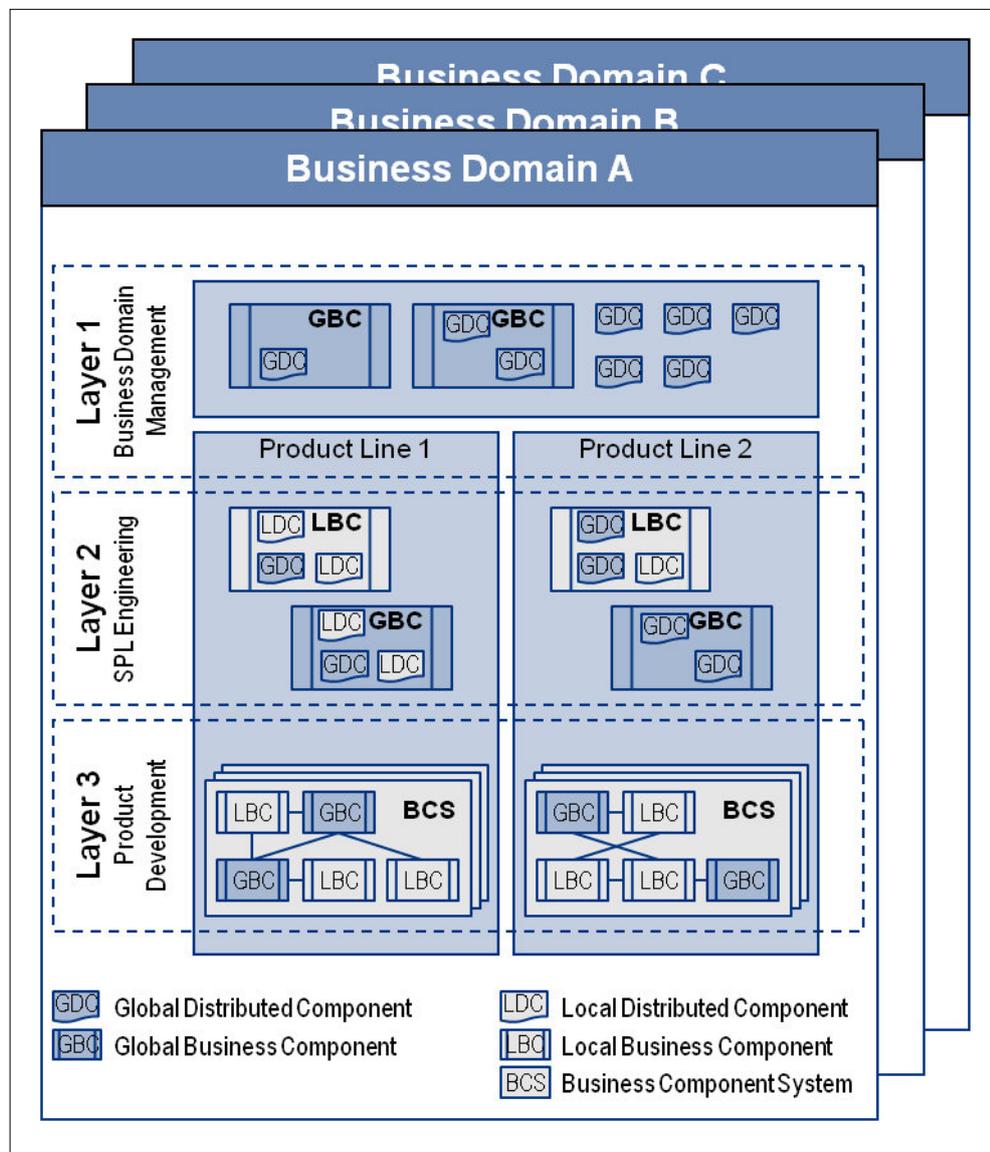


Figure 4-7: Component Granularity Alignment – Organizational Structure

### 4.2.3.3 Development Process Alignment

The development process dimension of the business component approach defines the chronological sequence in which the activities of the other four dimensions are carried out. This development process is strictly “component-centric, architecture-centric, with a strong emphasis on component autonomy at the various granularity levels, factored by a precise collaboration model” (Herzum and Sims, 2000, p. 72) and allows for high levels of reuse and concurrency. It consists of three basic manufacturing processes which will be aligned with the organisational model for industrialized SI.

The first process, rapid component development, covers definition, building, and testing of individual business components (Herzum and Sims, 2000, p. 247). As shown in section 4.2.3.2, such components may be developed on the business domain as well as the SPL engineering layer of the organizational model. The former covers domain-specific global distributed components and global business components, and will be built within the core asset development process of the business domain layer. The latter, local distributed components and local business components, cover product-line specific functionality. They will be built in the core asset development process of the respective software product line. Rapid component development is therefore aligned with both the business domain layer as well as the SPL engineering layer.

System architecture and assembly represents the second manufacturing process. It covers “architecting, assembling, and testing a system using business components” (Herzum and Sims, 2000, p. 247). Compared to the organizational model, these activities occur within the product development layer processes which are product requirements engineering, product design, product realisation, and product testing (Minich et al., 2010, p. 402). It is important to notice that the system architecture and assembly is not related to the architectural viewpoints discussed in 4.2.3.1. System architecture and assembly rather selects, adapts, and deploys already-existing distributed and business components according to customer-specific requirements. It is the actual manufacturing process in its original sense. It is therefore aligned with the product development layer of the organizational model.

The third development process, federation architecture and assembly, is the most advanced activity in business component-based development. It designs, assembles, and tests a federation of system level components, i.e. complete business component systems. It is assumed that such federations can be built after two or three years of experience with business components (Herzum and Sims, 2000, p. 247). However, as a business component system represents a business component itself, federation architecture and assembly selects, adapts, and deploys already existing business component systems according to customer specific requirements. Federation architecture and assembly is therefore also aligned with the product development layer of the organizational model, although it is far from being trivial and depends on extensive experience and supporting architectures as described in section 4.2.3.1.

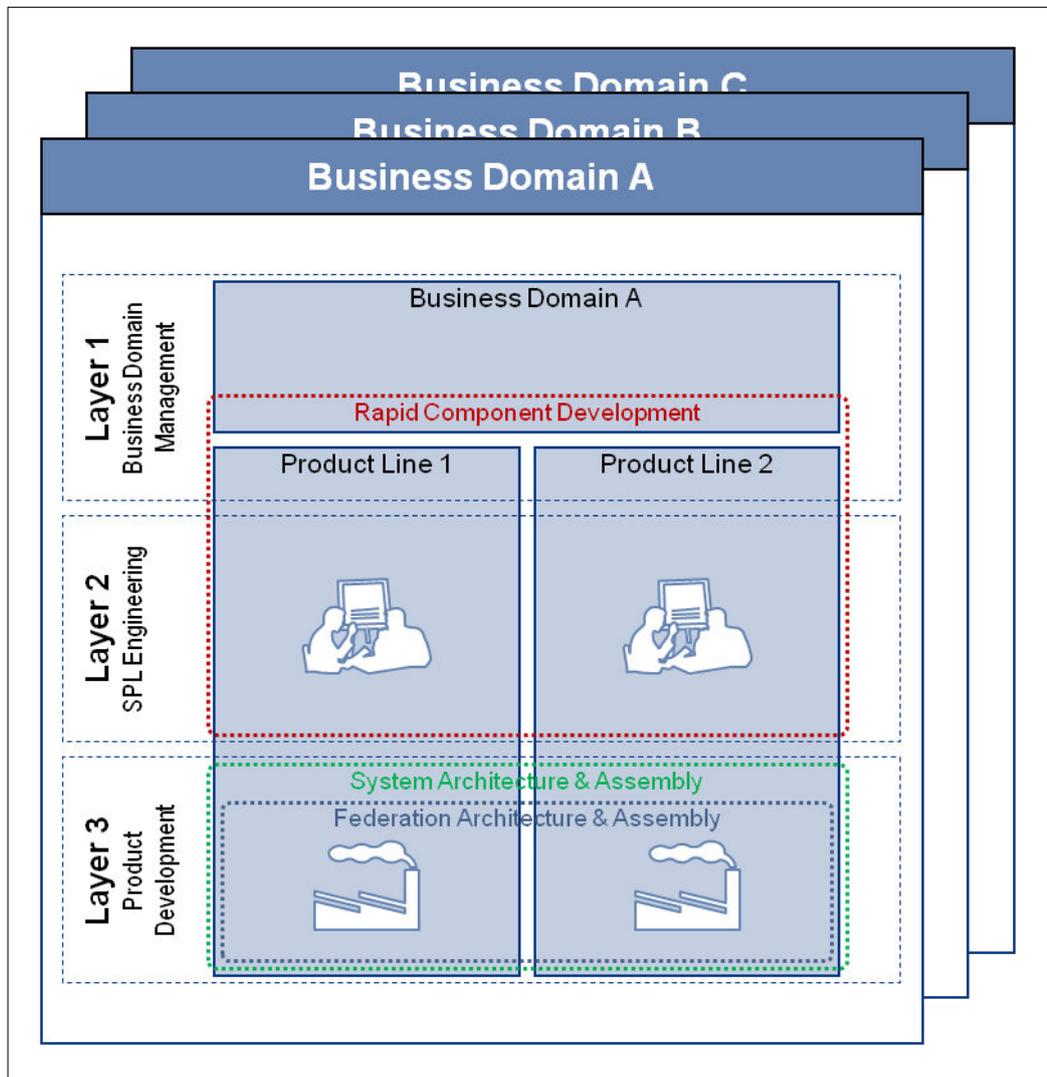


Figure 4-8: Development Process Alignment – Organizational Structure

#### 4.2.3.4 Distribution Domain Alignment

Herzum and Sims differentiate between four different distribution tiers which are the user, workspace, enterprise, and resource tier. “Each tier corresponds to a different logical area of responsibility [...], and each addresses a separate area of concern” (Herzum and Sims, 2000, p. 119). They are furthermore grouped into the user workspace and the enterprise resource distribution domain. This is because local and enterprise-wide functionality is usually separated from each other and treated differently in large-scale systems.

The user workspace domain is responsible to “support a single human being’s view of system facilities through some user interaction/interface technology” (Herzum and Sims, 2000, p. 128). It represents a typical client application that may also contain locally delimited functionality such as interaction with other local tasks. The enterprise-resource domain, in turn, implements “a set of computing facilities within which state changes to important (probably concurrently shared) resources can reliably be made” (Herzum and Sims, 2000, p. 128). It represents a typical server-based application and data source which is used by multiple client applications and users (the user workspace domain), but also other business components from the enterprise resource domain.

As this architectural viewpoint is more focussed on the logical structure of a component-based system and how to scope and distribute functionality, aligning it with the organizational model would not be appropriate. Thus Vogler’s integration metamodel is taken as a reference, serving as a foundation when planning and designing an integrated system (Vogler, 2006, pp. 82; 298). This alignment will thus help to decide how certain entities of the integration metamodel may be implemented within a business component approach and vice versa. The integration metamodel defines similar layers or tiers which are desktop, process, and systems integration.

Desktop integration takes distinct tasks as its reference point. It is responsible for presenting them to the users, interacting with them, exchanging data with other tasks, and providing an interface to enterprise applications and resources where required (Vogler, 2006, pp. 85 f.). This exactly describes, in other words, the responsibilities of the user workspace domain which is

also responsible for interacting with the user and providing local business functionality by presenting single tasks to the user. During development of a component-based integration solution, the entities referenced in Figure 4-10 will be implemented in the user workspace domain. However, there may be circumstances in which an activity also finds its way into the user workspace domain. This is the case if an end-user application independently represents a complete business process or workflow, including local process integration activities. This situation is depicted with a dotted line. It becomes more evident in the process integration view, as most process related metamodel entities are presented there.

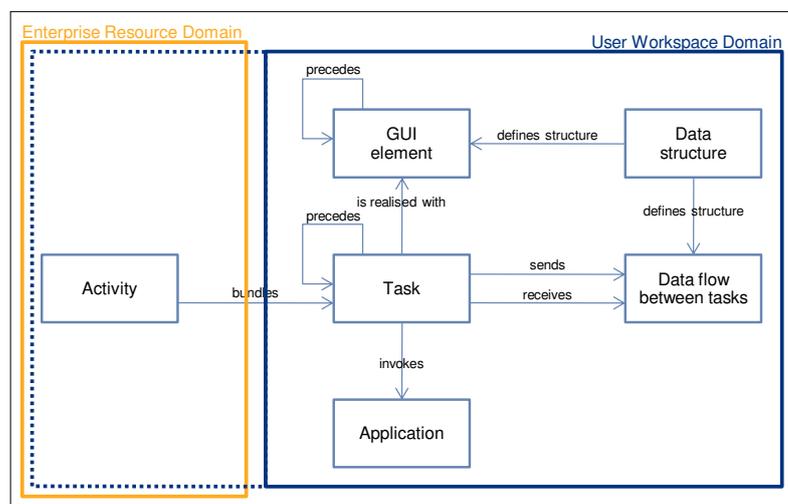


Figure 4-9: Distribution Domain Alignment – Desktop Integration

The process integration layer of the integration metamodel is concerned with process control by defining one or more workflows which are hierarchically structurable and consist of different activities and transactions (Vogler, 2006, p. 84). Such processes usually involve multiple other enterprise resources and end users. With reference to the business component model, the enterprise tier implements “enterprise-level business rules, validation and interaction between enterprise components, and it also manages the business aspects of data integrity” (Herzum and Sims, 2000, p. 122). These activities reflect those of the process integration tier of the integration metamodel. Systems (being a component in their own sense) as well as process related entities with their corresponding execution and control mechanisms do not have any direct interaction with the end user. They are therefore implemented as components of the enterprise resource

domain. Entities not covered by either one are organizational ones necessary for understanding the metamodel which will not be implemented in an actual system (e.g. site or organizational unit).

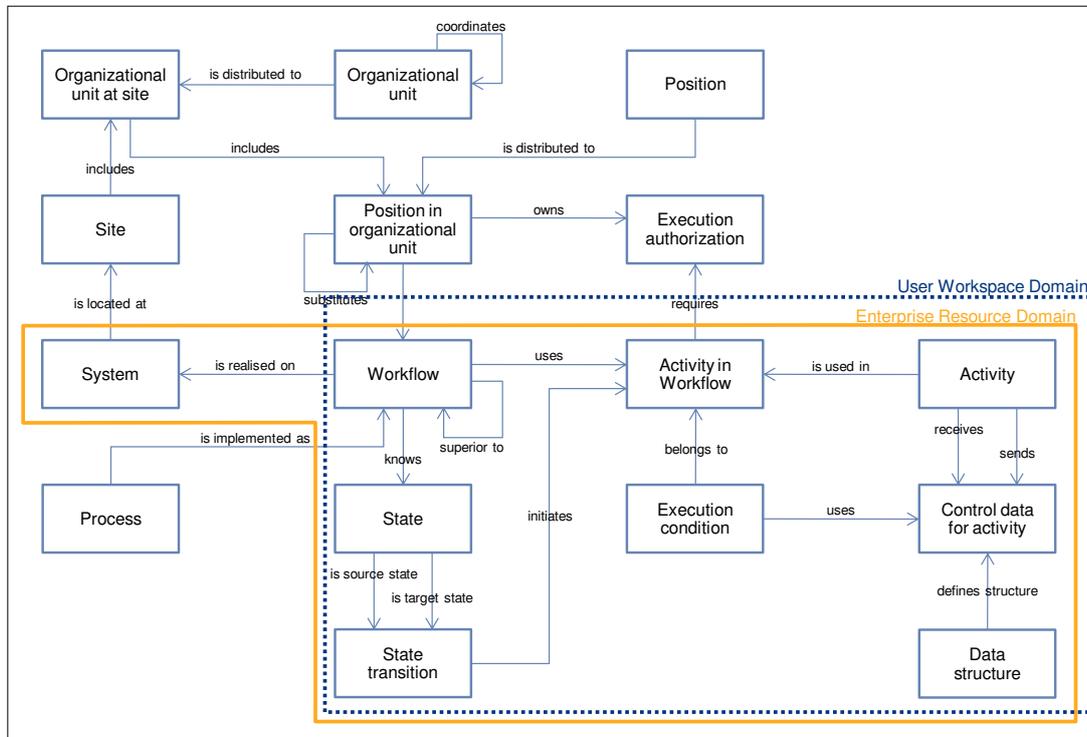


Figure 4-10: Distribution Domain Alignment – Process Integration

The systems integration layer is concerned with interfaces and data transfers between different applications and resources. It may also use a middleware, for instance, to support such activities (Vogler, 2006, p. 87). The layer furthermore provides data and application interfaces utilized by applications to interact with each other. Similarly, the enterprise resource tier of the business component factory “manages the physical access to shared resources” (Herzum and Sims, 2000, p. 122) and shields the business logic from technical aspects. This includes the provisioning of interfaces to data and functionality for components from the user workspace domain for direct representation of a business process, or for other components from the enterprise resource domain assembling a more complex business process. Both situations represent activities of the systems integration tier. The enterprise resource domain of the business component model is therefore aligned with the process and systems integration layer of the integration metamodel.

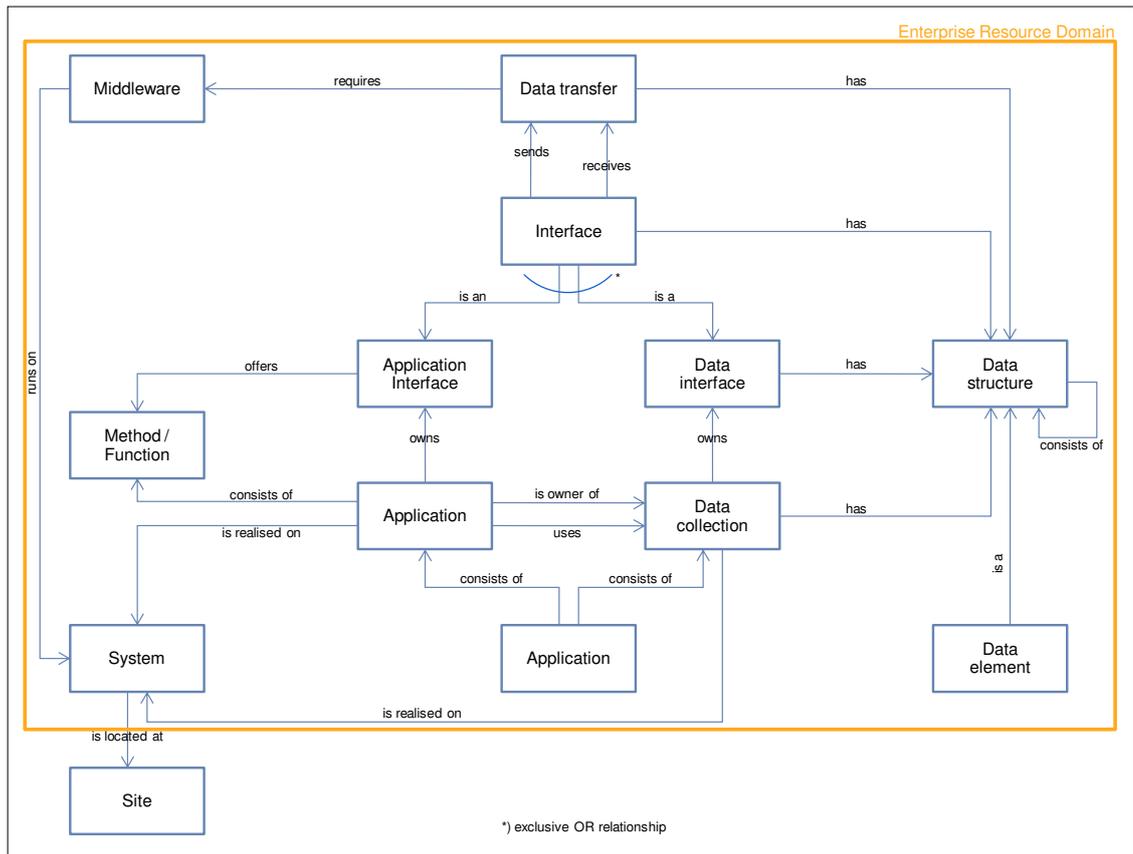


Figure 4-11: Distribution Domain Alignment – Systems Integration

The following figures show how metamodel entities for desktop, system, and process integration are represented within the business component factory methodology.

#### 4.2.3.5 Functional Category Alignment

As with distribution domains, functional categories describe a concept realized separately for each product line and product. They are “concerned with the functional aspects of the system, including the actual specification and implementation of a system that satisfies the functional requirements” (Herzum and Sims, 2000, p. 73). In the following, they will therefore also be aligned with Vogler’s integration metamodel.

Herzum and Sims define three broad functional categories for business components which are process business components, entity business components, and utility business components. “Process business components represent business processes and business activities” (Herzum and Sims, 2000, p. 180). They define a process as a workflow with different activities and tasks

to be executed, and also control these to ensure overall process completion. Such business processes are very customer-specific and can hardly be reused. Examples for process business components are order management or payment processing. They are aligned with the metamodel entities of the desktop and process integration layer. These metamodel entities together represent exactly the responsibilities of the process business component category by defining workflows, activities, and tasks together with their respective control data and data flows. Figures 4-12 and 4-14 show how metamodel entities for desktop and process integration are represented within the process component category of the business component factory.

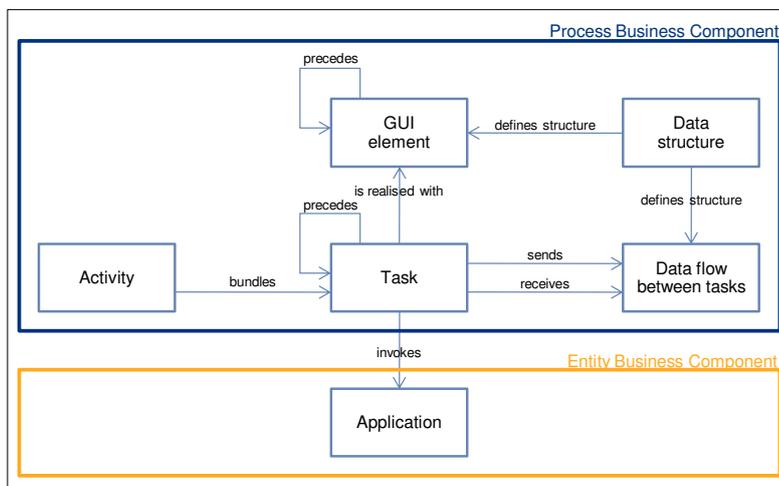


Figure 4-12: Functional Category Alignment – Desktop Integration

Utility business components describe supporting concepts that are generally available to other components and may be used in a variety of systems and product lines. Examples for utility business components are, for instance, a middleware service, print services, or a currency converter. As such, the utility functional category is aligned with parts of the systems integration layer of the integration metamodel as shown in figure 4-13. As the model does not differentiate between technical and functional aspects, this functional category is mapped to the system and middleware entities, although a wide variety of other utility business components is conceivable (but not represented in the integration metamodel). Depending on the functional architecture of a system, utility business components may also provide data transfer and interfacing services independently from entity business components. This may be the case if legacy systems are to

be integrated but cannot be aligned with the integration metamodel. In Figure 4-13 this situation is represented with a dotted line.

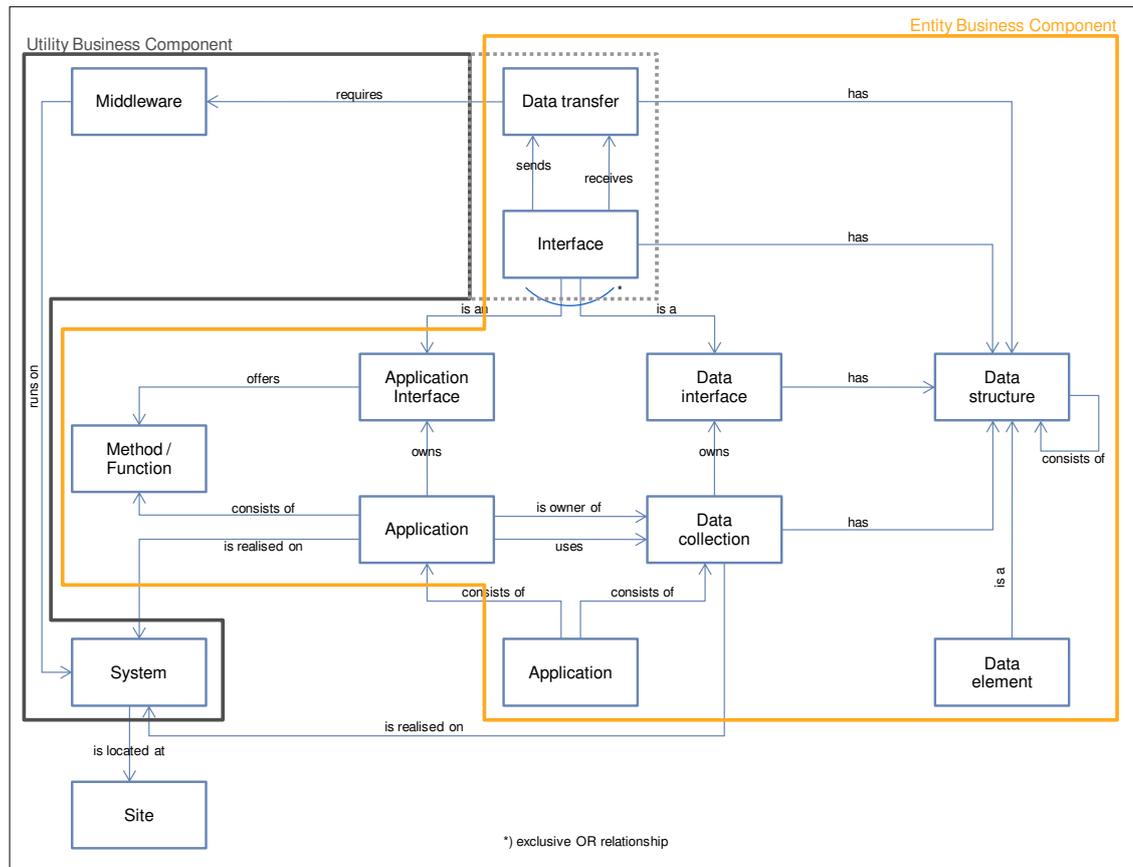


Figure 4-13: Functional Category Alignment – System Integration

Entity business components represent logical concepts and entities within a system including interfaces to manipulate them. They often contain persistent information which may be altered by tasks. Such concepts are usually specific to a business domain (e.g. the automotive industry) or a product line (e.g. a product line for shop floor solutions). They are most likely to be reused in “software systems and component frameworks aimed at supporting a particular collection of business activities” (Herzum and Sims, 2000, p. 179). Examples for entity business components are a work order, a bill of materials, or a customer. With regards to the integration metamodel, a precise point of demarcation cannot be found. It is therefore assumed that applications and programs (together with their data collections and interfaces) can be seen as system level components as described in section 4.2.2.1. Currently, the sole utility business component can be found in the information system where a particular solution is implemented. For new systems,

however, these would be replaced by corresponding entity business components which rely on utility components for data transfer, being utilized by process business components.

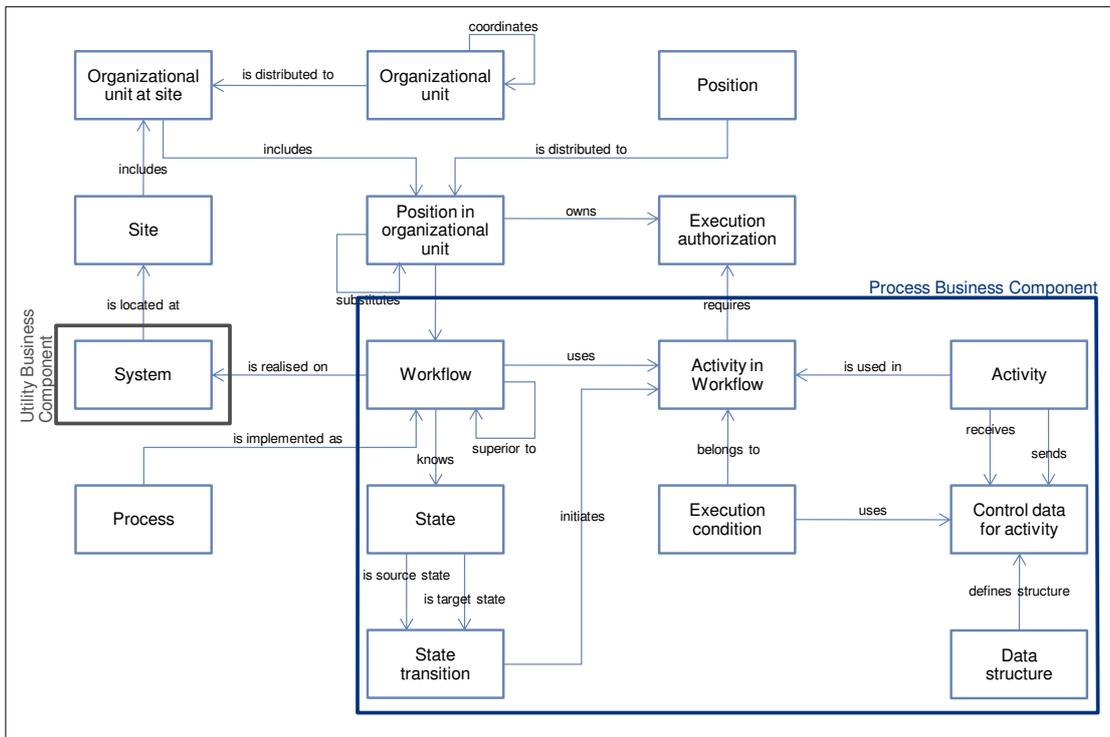


Figure 4-14: Functional Category Alignment – Process Integration

#### 4.2.4 Conclusion and Coverage of Research Objectives

During the course of research, the business component approach developed by Herzum and Sims was identified as a feasible concept for component-based development of systems integration solutions on an industrial scale. In its original form, however, the approach would be negatively affected by the characteristics of systems integration. To overcome these, Herzum and Sims’ approach was aligned with the previously developed organizational model for software product lines in systems integration as well as the integration metamodel developed by Vogler. The former allowed consolidating recurring tasks of CBD into the business domain layer of the organizational model and thus reducing the overall efforts required for implementation of the second industrial key principle, i.e. standardization and systematic reuse. It was furthermore shown how the different viewpoints of the business component factory approach must be aligned with the organizational units of a software product line in systems integration.

Additionally, alignment with the integration metamodel allowed defining how the entities of the model can be represented by business components (e.g. utility, process, or entity) and how different entities can be distributed in a large-scale system (i.e. user workspace domain, and the enterprise resource domain).

The present section has shown that, with slight adaptations, the business component factory approach can be applied to the field of software development in systems integration while retaining the requirements of specialization as the first industrial key principle. It also has demonstrated how the integration metamodel may serve as a generic integration framework and thus help to reduce complexity and heterogeneity mid to long term.

After describing how specialization and standardization & systematic reuse can be implemented in systems integration, automation as the third key principle must be evaluated upon suitability in the context of systems integration. Similar to the previous two, section 4.3 describes current approaches of model-driven engineering in further depth, analyses their specific limitations, and adopts the most suitable one to industrialized systems integration.

### **4.3 Model-Driven Systems Integration**

With software product lines and component-based development in place, the efficiency of software development in systems integration can already be increased significantly. As previously in many other industries, specialization and standardization yield powerful and reusable product and production assets (Encyclopaedia Britannica, 2005d, pp. 280ff., 2005c, p. 254). Such assets “allow the automation of software development as opposed to the creation of custom ‘one of a kind’ software from scratch” (Rashid et al., 2011, p. 3). Specialization and especially standardization are necessary, as machinery cannot solve unknown problems.

As with the first two concepts of industrialization, implementing model-driven engineering from the beginning comes with certain upfront investments (France and Rumpe, 2007, p. 5) which may inhibit the adoption in the field of systems integration. The implementation cost of domain-specific languages, model transformation engines, or code generators may never pay off

in a customer-specific, one of a kind setting. Thus, to implement MDE as the final concept of industrialized systems integration, the present section identifies an approach suitable for adaptation to the particular requirements of systems integration.<sup>3</sup>

### **4.3.1 Selection of a Suitable Model-Driven Engineering Approach**

As with the second industrial key principle, the most suitable methodology towards model-driven engineering has to be identified and subsequently adjusted to the characteristics of systems integration. As can be seen in the following, current model-driven engineering approaches are not as mature as other concepts of software industrialization (Selic, 2008). However, as for systems integration with its high heterogeneity and low number of products within a software product line, MDE must be as efficient as possible to yield an economic benefit. The following section explains these requirements in more detail with current limitations of MDE in mind. Afterwards, the model-driven engineering approaches from 2.4.4 are discussed regarding their suitability for systems integration, and an approach for further adaptation selected.

#### **4.3.1.1 Requirements from Systems Integration**

As with component-based development, there is no technical reason why model-driven engineering may not be utilized in systems integration. Bridging the abstraction gap between high-level business process planning and low-level systems integration is regarded as being highly beneficial. Models would allow approaching some of the biggest challenges, such as obtaining an overview of integration relationships and consequences of later changes (Afonso et al., 2006, p. 132).

However, also model-driven engineering has to cope with the characteristics of systems integration. This especially applies to its high heterogeneity, complexity, and very customer-specific products. All these result in a low number of products developed within a software product line. In consequence, domain-specific languages, model transformation engines and code generators

---

<sup>3</sup> Parts of this section were published by the author in Minich et al. (2012) during the course of research.

as the primary MDE artefacts must be developed efficiently and with broad reuse in mind. They furthermore must be aligned with the implementation of the first two industrial key concepts, i.e. the organizational model and component-based development.

To maximize potential reuse while remaining specialized enough to speed up the development process, MDE fundamentals need to be implemented at the previously introduced business domain level. Only here may a domain-specific language suitable for more than one software product line be established. This ensures a broader area of application and thus reuse, while maintaining compatibility between the different product lines of a business domain. A DSL defined at this level is still assumed specific enough to avoid facing the same consequences as CASE tools before (Selic, 2008). Where necessary, domain-specific languages may be extended within a particular software product line to represent more specialized concepts. Of course, this implies that at the same time model transformation engines and code generators are being extended to cover these new DSL elements.

Another important aspect arising from model-driven engineering in very large organizations is the possibility of modularization and concurrent work. For conventional text-based languages, modularization, versioning control, and code merging are well understood and supported by respective tools. For model-driven engineering, however, this is not yet the case, as tool suites are still lacking important functionality to represent the complete development lifecycle (Tepola et al., 2009, p. 19; Selic, 2008, p. 386; Afonso et al., 2006, p. 127). A model-driven approach for a large systems integrator thus must be able to be implemented in a gradual manner, adding functionality as more powerful tools become available. To be the most future-proof, open standards should be preferred.

Concluding the above, the requirements from systems integration for model-driven engineering lie in domain-specific languages, model transformers, and code generators developed for a particular business domain and being extendable by the underlying software product lines. Furthermore, the chosen approach must allow an iterative implementation based on open standards.

#### 4.3.1.2 Discussion of Model-Driven Engineering Approaches

In 2.4.4, model-driven engineering approaches available at the time of writing were presented. The present section revisits these and discusses whether they match the requirements implied by systems integration.

- An initiative from the Object Management Group, **Model-Driven Architecture** defines a model-driven development approach which is based on a separation of functional and technical concerns (Object Management Group, 2003). It specifies UML as its modelling language, and the Meta Object Facility as the meta model for all specification artefacts. These are used to create the Computation Independent Model the platform independent model, the platform specific model, and the platform specific implementation. Beginning with the computation independent model, details are being added with each transformation until executable source code is available (Petrasch and Meimberg, 2006, p. 98).

With regard to systems integration, an MDA implementation is not necessarily limited to a particular business domain or software product line (Czarnecki, 2005a, p. 333). Nevertheless, the computation independent model in combination with domain-specific languages may be used to limit the scope as necessary (Petrasch and Meimberg, 2006, p. 130). Also, hierarchical development of domain-specific languages and transformation engines are conceivable, depending on the technology they are implemented with. In general, MDA is capable of supporting industrialized systems integration as described above. However, due to its very conceptual and theoretic specification, it does not seem well-suited to being implemented in a fast-changing and efficiency-driven environment. This situation is reinforced by the lack of powerful tools to support the modelling and development process (Teppola et al., 2009, p. 19).

- Based on the work of Czarnecki and Eisenecker (2005b), **Generative Programming** aims at automating the development of a family member within a clearly delimited context. It therefore defines a problem space expressed by a Domain Specific Language and the solution space consisting of “implementation-oriented abstractions, which can be instantiated to create implementations of the specifications expressed using the domain-specific abstrac-

tions from the problem space” (Czarnecki, 2005b, p. 5). The mapping between both contains the configuration knowledge such as illegal feature combinations, default settings, default dependencies, construction rules and grammar, or optimizations. These mapping rules are implemented within a generator returning the solution space, which may either be an intermediate model or executable program code.

With regard to systems integration, Czarnecki and Eisenecker’s approach explicitly asks for a delimited scope and is comparable to software product lines. In their book they offer various techniques of domain-specific languages and generators, including extendible ones (Czarnecki, 2005b, p. 139). Generative programming furthermore allows for the creating of DSL, generator, and other artefacts required ‘on the fly’ during regular software development (Czarnecki, 2005b, p. 16). This reduces the necessity of high upfront investments and leads to artefacts tailored exactly to the needs of the implementing company. It is also up to the company to decide to which extent Generative Programming should be implemented. Open standards are not explicitly mentioned but may be followed during DSL and generator development. Their examples are based on XML or common programming languages.

- **Software Factories** is an approach introduced at Microsoft by Greenfield and Short (2004) which, relies on Software Product Lines and Component-Based Development. It is based on software factory schemes, describing the systems to be developed from different viewpoints. These express business logic and workflows, data model and data messaging, application architecture, technology, and variability and may be present on different levels of abstraction. Altogether the schemes with their viewpoints exactly define what needs to be done and how (Greenfield et al., 2004, p. 164). In order to provide a customized integrated development environment, the schema with its viewpoints is represented by a software factory template. The template can be loaded into an integrated development environment, providing wizards, patterns, frameworks, templates, domain-specific languages, generators and editors. Complete definitions of domain specific languages furthermore allow (semi-) automatic model to model transformations and code generation.

With regard to systems integration, software factories also support specialization and standardization & systematic reuse. The approach allows delimiting the scope and building

respective domain-specific languages. As the schemes and templates representing a software factory must be developed in advance (Greenfield et al., 2004, p. 164), software factories do not seem to be suited for hierarchical extension, nor does the approach offer any higher layer to enable integration of systems developed in different factories. Furthermore, implementation guidelines and examples are currently aimed at proprietary integrated development environments.

#### **4.3.1.3 Approach Selection for Further Adaptation**

From above approaches, only Czarnecki and Eisenecker's Generative Programming (2000) has proven to be flexible enough to cover the requirements of systems integration on a short term basis. The processes described are very much in line with software product lines (Czarnecki, 2005b, p. 21) and may easily be adapted. Instead of coarse-grained business components, it aims at reusable functionality at source code or programming library level (Czarnecki, 2005b, pp. 165 ff.). As long as reusable components are assembled on the source code level or at least assembly instructions are generated accordingly, this does not impose any constraints. The development of domain-specific languages and model transformation engines and code generators occurs as needed during the actual implementation and thus allows an incremental implementation resulting in lower upfront investments. In addition, as the DSL and its elements are implemented as needed, there is no reason why this could not be done in a hierarchical manner between a business domain and underlying software product line.

Applying software factories may be possible as well. However, the necessity of implementing the complete software factory schema including DSLs, generators, and tools before starting with product development (Greenfield et al., 2004, p. 164) seems contradictory. A hierarchical separation in development of these artefacts, i.e. business domain vs. product line specific assets, is not possible without a significant redesign of the overall process. Although flexible enough to use other environments, the guidelines and actual examples of the approach from Greenfield et al. (2004) are tailored around Microsoft's Visual Studio.

Compared to generative programming and software factories, MDA does not necessarily rely on a clearly delimited problem domain. Although it does offer everything necessary, the approach itself is too generic and excessively theoretical. Although practical guidelines such as Frankel (2003), Beltran et al. (2007) or Petrasch and Meimberg (2006) contain everything necessary for developing software with MDA, they only briefly touch on software product lines or component-based development.

For the time being, a full-fledged model-driven engineering approach in an industrial setting does not seem possible. The reasons therefore can be found in a lack of experience, immature processes, lack of modelling standards, and especially the lack of tools covering the complete development process (Teppola et al., 2009, p. 19; Selic, 2008, p. 386; Afonso et al., 2006, p. 127). This especially applies to the field of systems integration with characteristics like one-off development, high heterogeneity, and multiple systems to be integrated. However, with an incremental approach, those aspects of MDE possible as of today may already be implemented. The remaining and more complex features will be implemented once further advances from integrated development environment solution providers or standard setting bodies are available and proven in practice.

With regards to the industrialization of systems integration, automation as the third industrial key principle cannot be completely achieved. Therefore, generative programming as the most suitable approach is selected and adapted to the requirements of systems integration as far as possible. As soon as technology advances, it may be completely implemented during further research.

### **4.3.2 Light Weight Model-Driven Systems Integration**

The present section presents the main processes of generative programming and subsequently aligns them with the organizational model and component-based development approach previously adapted. During the course of the alignment, it can be seen that some processes such as domain scoping and engineering, can be adapted very well. Others, such as domain specific language or configuration knowledge related, only allow a rudimentary implementation. For

those, an alternative route is suggested offering fundamental modelling and code generation based on the techniques introduced in 2.4.2 and 2.4.3.

### 4.3.2.1 Processes of Generative Programming

Czarnecki and Eisenecker's approach is based on a generative domain model which describes the problems to be solved on one hand and their respective solutions in terms of elementary components on the other hand. Once the solution space is sufficiently populated, unknown but similar problems may be solved as well. Creating a solution requires certain configuration knowledge, such as illegal feature combinations, default settings, component dependencies, etc.

Figure 4-15 illustrates the concept.

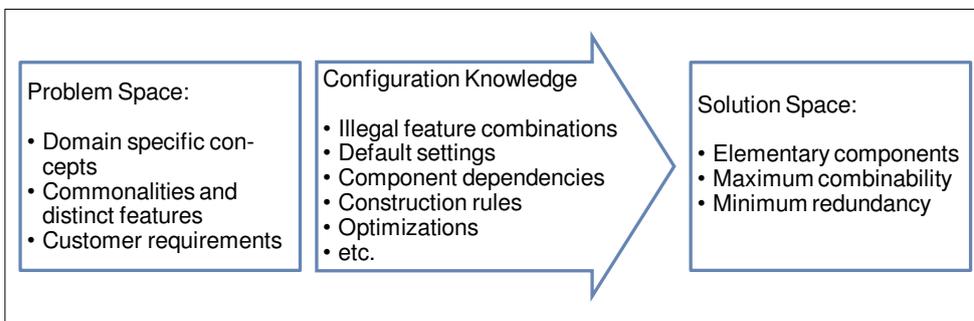


Figure 4-15: Generative Domain Model (Czarnecki and Eisenecker, 2000, p. 132)

To populate the problem and solution space and specify the configuration knowledge, generative programming includes the following eight processes (Czarnecki and Eisenecker, 2000, p. 134):

1. **Domain Scoping** identifies the domain of interest, stakeholders, goals, and defines the scope of the Generative Programming approach. It is influenced by e.g. the stability and maturity of potential solutions, available resources to implement them, and the potential for re-use during production. A good balance between these factors ensures business success once products of the domain are being developed. It is important to notice that the domain must constantly be maintained, i.e. stakeholders, goals, market developments, or technical trends must be monitored and the scope updated accordingly (Czarnecki and Eisenecker, 2000, p. 37). Depending on the domain of interest, several models for domain analysis are available.

The most prominent ones are Feature-Oriented Domain Analysis, Organization Domain Modelling, Draco, Capture, Domain Analysis and Reuse Environment, and the Domain-Specific Software Architecture Approach. An extensive description and comparison of current domain scoping approaches may be found in Moreno-Rivera and Navarro (2011) or Jatain and Goel (2009).

2. **Feature & concept modelling** identifies the distinguishable characteristics of a system within a certain domain and models them within a feature model (Czarnecki and Eisenecker, 2000, p. 38). Commonalities and variabilities identified during the domain scoping process are documented in a feature model. These can be services, operations, non-functional characteristics, or technologies (Lee and Muthig, 2006, p. 57). Features and concepts can be separated into those visible by the end user and those describing internal aspects of the software relevant to some stakeholders of the concept (Czarnecki and Eisenecker, 2000, p. 38). The output of the feature & concept modelling process is a feature diagram representing a hierarchical decomposition of features, feature definitions describing their functionality, composition rules indicating valid and invalid combinations, and rationales for features “indicating reasons for choosing or not choosing a given feature” (Czarnecki and Eisenecker, 2000, p. 39).
3. **Common architecture & component definition** depends on the previously developed feature model, their descriptions, composition rules, and rationales and identifies key areas of functionality. With systematic reuse in mind, the process defines how features and commonalities will be represented by components and what the overall system architecture will look like. The representation of the components will depend on the particular architecture and component framework selected (Czarnecki and Eisenecker, 2000, p. 156). The process results in an architecture applicable for the complete software product family, including functional component categories with technical specifications of the respective implementation components (Czarnecki and Eisenecker, 2000, p. 8).
4. **Domain-Specific Language design** specifies a language based on the previously-developed feature model, common architecture, and component specification. This may be done in different ways, ranging from simple translational semantics (i.e. defining a translation scheme

to an implementation language) to complex axiomatic semantics (i.e. defining a mathematical theory for checking on programs written in a given programming language). At this time, the process limits itself to an abstract syntax, i.e. the formal DSL specification by defining syntax trees (Czarnecki and Eisenecker, 2000, pp. 157–158). The concrete syntax used to specify a product in an integrated development environment will be defined in the DSL implementation process. This has the advantage that for one abstract syntax, multiple concrete ones may be developed, such as graphical or textual ones (Czarnecki and Eisenecker, 2000, p. 157).

5. **Specification of configuration knowledge** defines “which combinations of components satisfy which configurations of features” (Czarnecki and Eisenecker, 2000, p. 754) utilizing the feature models and architectures identified above. The separation of both allows evolving the solution space independently (Czarnecki and Eisenecker, 2000, p. 132) without having to realign requirements from the problem space with e.g. the customer. For instance, once a better-performing solution to a particular problem is available, the configuration knowledge can be adapted accordingly. New applications will automatically use the new solution, while older ones may remain as they are or be updated. Furthermore, the configuration knowledge shields the developer from knowing all potential components and features by specifying illegal combinations, default settings, dependencies, or construction rules (Czarnecki and Eisenecker, 2000, p. 14).
6. **Architecture & component implementation** is concerned with the implementation of the solution space defined by the common architecture & component definition process. Development occurs independently of actual customer projects based on the requirements of the business domain in general (Czarnecki and Eisenecker, 2000, p. 7). Once the solution space is completed, it must be maintained and may also be updated to the latest technical standards. Czarnecki and Eisenecker suggest several technologies to implement architecture and components. These are generic programming with conventional 3G languages, aspect oriented programming, transformational generators, static metaprogramming, or intentional programming (Czarnecki and Eisenecker, 2000, pp. 165 ff.). Each of them exists legitimately and their usage depends on the actual software family to be implemented.

7. **Domain-Specific Language implementation** takes the DSL specification from the DSL design process and derives a concrete implementation. Here generative programming differentiates between separate DSLs (e.g. SQL or T<sub>E</sub>X), embedded DSLs (e.g. template meta programming in C++), and modularly composable DSLs (e.g. embedded SQL, or aspect-oriented programming) (Czarnecki and Eisenecker, 2000, pp. 137–139). Several technologies for DSL implementation are available, either text-based or graphically represented. Examples of the former ones are template metaprogramming in C++, Template Haskell, OpenC++, OpenJava, or Metaborg (Czarnecki, 2005a, p. 335). Examples of graphical DSL tools are UML profiles, Generic Modelling Environment, MetaEdit+, or Microsoft’s DSL Tools (Czarnecki, 2005a, p. 335).
8. **Configuration knowledge implementation** in generators allows for advancing the problem specified with the help of a Domain-Specific Language into executable program code. To do so, generators apply validation of the input specification, complete a given specification with default settings, perform optimizations, and subsequently generate the implementation. Generators may be implemented as stand-alone programs, using built-in meta programming capabilities of a programming language, or by using a predefined generator infrastructure (Czarnecki and Eisenecker, 2000, pp. 339–340). Stand-alone tools require the most effort but may be an option if the configuration knowledge is not too complex. Metaprogramming as available in C++ may “generate code by composing functions and class templates” (Czarnecki, 2005a, p. 340) at compile time. Although infrastructure and knowledge being broadly available, generators based on metaprogramming are limited to the host language and do not offer any debugging facilities (Czarnecki, 2005a, p. 340). A common generator infrastructure allows for the defining of “a common format for source representations and standard sets of operations for encoding transformations on the representation” (Czarnecki, 2005a, p. 340). An example of such an infrastructure was described with intentional programming in Figure 2-6. While the first two approaches are viable for the time being, the third will prevail in the future and require further research and tool development (Czarnecki and Eisenecker, 2000, p. 341; Kamp, 2012, p. 5).

Comparing the eight process steps with the concepts of Software Product Line and Component-Based Development, Generative Programming can be clearly allocated to the industrial key principles. Domain scoping and feature & concept modelling reflect specialization by limiting the scope to a particular business domain and product portfolio, similar to software product lines. Common architecture & component definition, specification of configuration knowledge, and architecture & component implementation refer to standardization and systematic reuse. They can be best compared to the processes of component-based development. Domain-specific language design, its implementation, and the implementation of configuration knowledge reflect the industrial key concept of automation, i.e. developing purpose-built 'machinery' fulfilling repetitive tasks.

Considering above separation, generative programming may be implemented with only one-third of the initial effort, as large parts of domain-, component-, and architecture-related activities are already covered in software product lines and component based programming. The respective process allocation combining all three industrial key concepts is described in the next section.

#### **4.3.2.2 Alignment with the Organizational Model for Industrialized SI**

Defining the domain and product portfolio of a generative programming approach reflects large parts of software product line engineering. The artefacts to be developed are similar to those to be created within the business domain and software product line layer of the organizational model. Including the necessary changes introduced by Generative Programming, the following two sections describe the final processes of the business domain and software product line layer. These now include the requirements from software product lines, business components, and generative programming and thus resemble all three industrial key principles applied to the particular field of systems integration.

## The Business Domain Layer

The Business Domain Layer was developed to align domain-wide functionality and utilize economies of scope due to similar concepts and core assets among different product lines of a given domain. It therefore contains the Software Product Line processes domain analysis & portfolio definition, architecture development & roadmap definition, and core asset development.

The processes of the business domain layer already cover the Generative Programming processes 1 and 2 which includes, for instance, the development of a domain or feature model (see 4.1.2.1). Furthermore, the activities of Generative Programming processes 3 and 4 are already enclosed in Architecture Development & Roadmap Definition and Core Asset Development. However, as the Business Domain Layer only features concepts suitable for more than one product line, a differentiation between global (business domain wide) and local (product line specific) aspects of Generative Programming is necessary. This means that there will, for instance, be DSL design activities in both the Business Domain and the Software Product Line Layer. In the former, the overall structure and domain-wide syntax and semantics are defined, whereas the latter covers product line specific syntax and semantics such as “bill of materials” for a shop floor system produced in a particular software product line. The distribution is illustrated in Figure 4-16 on page 170. Combining the activities of the business domain introduced in 4.1.2.1 with the respective ones from Generative Programming, the Business Domain Layer in its final stage consists of the following core processes:

- **Business Domain Analysis** explores the typical IT landscape of the business domain in scope and identifies areas of expertise required to develop and provide the products and services under consideration. Similar to software product lines but on a higher level, it identifies the recurring problems and known solutions. The information derived results in a domain model describing the business segment and its typical IT landscape as a whole.

Generative programming adds formal domain analysis methods such as Feature Oriented Domain Analysis, Organization Domain Modelling or Domain Analysis and Reuse Environment to the process, although these could have been used without Generative Program-

ming as well. All other requirements of the domain scoping process of Generative Programming are already covered.

- **Portfolio Definition** evaluates the information from the domain model and develops a product portfolio for the particular business segment. The portfolio covers typical applications and solutions for the most important business services of the segment and identifies the portfolio elements to be supported. Based on these portfolio elements, the system integrator can define distinct software product lines which will produce the applications required by the customers. It is important to note that these product lines are not related to each other with regards to functionality. Rather, they reflect the typical and most important systems required by a customer of a certain business domain.

The feature & concept modelling process of generative programming can only be partially covered by portfolio definition. Although some generic and business domain-wide features may be defined on the business domain layer, the majority is defined in the underlying software product lines. The portfolio definition process may, for instance, specify information exchange features applicable to all underlying software product lines, but not functionality required for a particular one. This is subsequently done in the software product line layer.

- **Architecture & Roadmap Definition.** Once the scope is defined by Business Segment Analysis and Portfolio Definition, an integration architecture and basic product line definitions as well as a component framework applicable for all product lines must be developed. As different product lines have different functional and technical requirements, this architecture will mainly exist in an abstract form. It defines standardized structures and functionality, allowing for subsequent integration of products from different product lines within the same business domain. To ensure an aligned advancement throughout their lifecycle, the process also defines a technology roadmap applicable to all software product lines.

When defining the overall architecture and technologies to be used in the business domain, it is important to consider the requirements on systems modelling, componentization and code generation from the respective generative programming process. This decision directly affects the ability to automatically transform models or generate code as described in 2.4.3.

- **Core Asset Development** develops reusable assets applicable to all or many software product lines within the business segment. Such joint core assets may, for instance, be development tools and processes or joint software development patterns. Core Asset Development may also include the production of reusable software components equal to each product line. A typical example therefore would be an interface component to a particular technology or an integration middleware, as there is a high chance that these will be required in more than one product line. Such core assets explicitly include the required integration infrastructure such as middleware, message brokers, or databases as explained in section 3.1.3.

Generative programming adds the fundamentals of some more sophisticated concepts of software industrialization, i.e. domain-specific languages (DSLs) and their respective transformation engines and code generators. Any specifications made at this time are applicable to the overall business domain and may be inherited to satisfy the needs of the particular software product line. It is therefore important not to be too descriptive and to leave enough room for later extension. This may, for instance, be achieved with XML-based domain-specific languages and generators (see 2.4.2 and 2.4.3) which can be extended by adding product line specific language entities in the XML schema definition (XSD) and the respective transformation functionality in the XSL Transformation document (XSLT).

## The Software Product Line Layer

The Software Product Line Layer consists of several software product lines identified in business domain analysis and portfolio definition processes of the business domain layer. The most obvious variance to a conventional software product line is the lack of the business domain analysis process, and a simplified domain requirements engineering process. These functions are now incorporated in the business domain layer and provide their findings to the subsequent product lines. All other processes remain the same but must adhere to the specifications and utilize the provided core assets from the business domain layer.

As to generative programming, all but the first development process within the Software Product Line Layer can be found. However, due to the separation of domain-wide and product line specific concerns, the Generative Programming processes 2 and 3 (see section 4.3.2.1) are required in both business domain and software product line engineering. The former has already been described. For product line engineering, only product line-related concerns are dealt with. A systems integrator's feature model for the automotive industry (i.e. the business domain) may, for instance, define the entity car with several features such as model, engine, transmission, colour, price, owner, and so on. These features exist in all products of the underlying product lines. A product line for shop floor systems may, however, extend this feature model by adding features like electronic control unit type, brake type, or parts list. As this has no implication on the functionality of the car itself or the customer, it is not necessary that these features be known in other product lines. A financial system does not need to know what type of Engine Control Unit is built into a car, but it does need to know the price and the owner of the car. This same principle applies to feature & concept modelling, common architecture & component definition as well as domain-specific language design of generative programming. These three processes are shared between the business domain and software product line layer. All remaining processes are carried out in the software product line layer only. Combining the activities introduced in the organizational model for industrialized systems integration with the respective ones from Generative Programming, the product line layer in its final stage consists of the following core processes:

- **Product Line Requirements Engineering** inherits the generic product and technology roadmap as well as functional and non functional requirements defined by the business domain layer. The requirements for each product are further elaborated and an extended commonality and variability model of the product line is derived. Within the boundaries from the business domain layer, product line requirements engineering also defines the tools and technologies utilized for production.

Generative programming further develops the variability model into a detailed feature model, including external (visible), internal (functional) and non-functional features. The

feature model should be extensive enough to completely describe customer requirements as long as they are to be built by the software product line.

- **Architecture Design & Development** transforms scope and features defined in product line requirements engineering into a technical architecture and specification for the product line and its products. The architecture decomposes a software system into common and variable functional parts, defines relationships and interfaces and establishes rules for their implementation. It follows the requirements inherited from the business domain layer to ensure later integration with products from other software product lines. Unless already existing in the business domain layer, the architecture design & development process will also identify the core assets required for product line operation (Linden, 2007, p. 58). With completion of this process, proper integration between products from different product lines of one business domain can be ensured.

With regard to generative programming, the configuration knowledge in terms of component dependencies, default configurations, construction rules, illegal combinations, and rules for their implementation is specified. Each identified area of functionality requires one or more components with an architecture-specific component model, interaction scheme, and distribution mechanism. The domain-specific language defined by the business domain layer is extended with the product line-specific programming artefacts.

- **Core Asset Development** provides the design and the implementation of reusable software assets (Pohl et al., 2005, p. 242). This implementation includes the overall framework, software components, glue code, configuration and variability mechanisms, common processes, development tools, executable code, and other product line assets. The result is a collection of loosely coupled configurable components, not of a running application (Pohl et al., 2005, p. 27).

In terms of Generative Programming, core asset development is also responsible for the implementation of the domain-specific language as specified by the previous process. In a later and more mature stage, Core Asset Development will also implement the configuration knowledge within generators to advance the system specified with the help of a DSL into intermediate models or executable code. As this can be extremely complex, postponing this

activity until reasonable experience with the DSL and the product lines has been gathered is suggested (Czarnecki and Eisenecker, 2000, p. 135). Also, manual assembly based on pre-defined models may be feasible to ensure practicality of the DSL.

- **Domain Testing** develops test cases and inspects all core assets and their interactions against the requirements and contexts defined by the product line architecture. Domain testing also includes validation of non-software core assets such as business processes, product line architecture or development policies.

As generative programming does not state any particular requirements on testing, no changes to this process are necessary.

- **Software Integration** in the context of product line development occurs during pre-integration of several software components. They form blocks of functionality common to all products and contexts of a product line. Furthermore, the integration process ensures the interoperability of all reusable assets and provides the required integration mechanisms.

As generative programming does not state any particular requirements on software integration, no changes to this process are necessary.

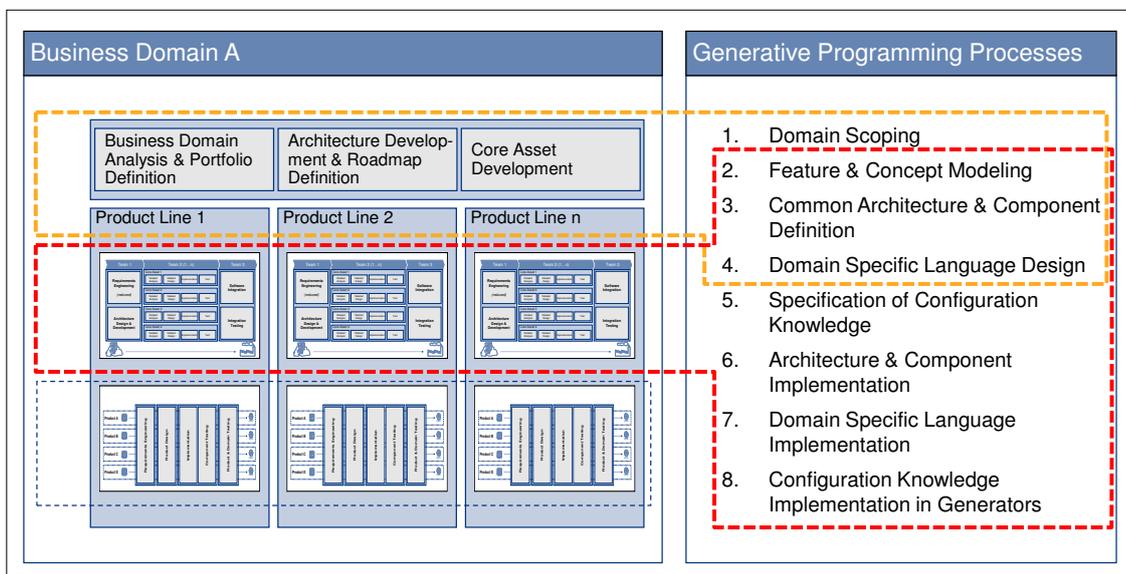


Figure 4-16: Mapping of Generative Programming Processes to Organizational Structure

### 4.3.2.3 Alignment with the Business Component Approach

After presenting the applicability of generative programming to the first industrial key principle, the present section will align it with the business component approach ensuring standardization and systematic reuse. It will therefore discuss the implications of Generative Programming to each of the five domains introduced in 4.2.2.

#### Component Granularity

Generative Programming does not explicitly refer to well-defined components as known from Enterprise Java Beans or Corba, for instance. Also it doesn't conceptually concentrate on business processes and therefore does not know reasonable levels of granularity. An artefact may, for instance, be a generic and reusable data container for C++, allowing for the handling of domain-specific types of information. It may also be a reusable programming library providing a complex business concept like a bank account. Generative Programming, on the other hand, rather concentrates on technologies and means to develop reusable artefacts of variable sizes, depending on the intended usage. This way of partitioning a problem into reusable artefacts is known as continuous recursion. One iteratively partitions a problem into different but reasonable granularities. The Business Component model in turn follows a discrete recursion approach (Herzum and Sims, 2000, p. 38). It therefore defines five levels of granularity: the language class, the distributed component (a component in its common sense, e.g. an EJB or CORBA component), the business component (still independently deployable, consisting of distributed components and glue code, representing a business process), and the system level component (a set of business components providing business functionality). The highest level of granularity is the federation of system-level components (i.e. system-level components federated to provide multiple complex business services).

It is assumed that discrete recursion and thus partitioning of the problem is more beneficial in an environment with systematic reuse. For each layer of recursion, a developer has to define scope, characteristics, packaging, and deployment (Herzum and Sims, 2000, p. 38). In an environment where components are to be reused as much as possible, it seems more beneficial to define these

layers of recursion on a common basis. A middleware messaging adaptor for a specific enterprise resource planning system will most likely exist as a distributed component as introduced above. A developer can rely on this concept and build his application accordingly. It is therefore suggested to introduce discrete recursion to the Generative Programming approach if it is to be used within component-based development and systematic reuse. Furthermore, discrete recursion allows adhering to the integration metamodel. Matching reusable components with discrete metamodel entities ensures a common structure among all product line architectures and products. This allows for more systematic reuse and also alleviates later integration efforts.

### Architectural Viewpoints

The second dimension consists of four architectural viewpoints which are the Project Management Architecture, concerned with organizational decisions, tools, and guidelines), the Technical Architecture (defining the execution environment, component and user interface frameworks, and other technical facilities), the Application Architecture (describing development patterns, guidelines, or standards), as well as the Functional Architecture (identifying the features and functional aspects of a system and their relationships).

With regard to the Project Management Architecture, Generative Programming does not make any statements about the organization or structure of a development project within its processes. The project management architecture from the Business Component Model is therefore regarded as being beneficial to the Generative Programming approach. In the organizational model for industrialized systems integration, the project management architecture is found in the Business Domain Layer, whose organizational decisions, tools, and guidelines will influence the development in Generative Programming. The remaining three rather technical viewpoints are concerned with the execution infrastructure and programming frameworks (Technical Architecture), development patterns, guidelines, and programming standards (Application Architecture), as well as the functional aspects of a system including its implementation (Functional Architecture). Generative Programming in turn only offers the generic process common architecture & component definition. It is therefore suggested to replace the respective Generative

Programming process with the actual implementation of the much more detailed architectural viewpoints from the Business Component Model. For Generative Programming, this replacement offers a more comprehensive view on different aspects of the architecture, while for CBD it ensures coverage of more component-related artefacts such as the component infrastructure or execution environment.

## Development Process

The Business Component Model encompasses a set of manufacturing processes which support component, system, and federation of systems development. However, as most organizations are in a transitive state towards component-based development, Herzum and Sims suggest a process called rapid system development. It follows the well-known V-Model, wherein requirements to implementation denote the left and component systems, and acceptance testing the right side of the V (Herzum and Sims, 2000, p. 248). Rapid system development subsequently allows engineering reusable artefacts based on customer-specific requirements and building the respective end product. The advantage is that reusable artefacts evolve on the fly. The disadvantage is that, beginning with the requirements of one specific customer, one may easily miss important variation points or even make architectural decisions which may conflict with the overall scope of the product line. Generative programming in turn focuses much more on domain engineering activities and the technical implementation of reusable artefacts rather than development of the end product. It puts explicit focus on feature modelling processes such as Feature Oriented Domain Analysis or FeatuRSEB (Czarnecki and Eisenecker, 2000, pp. 69 ff.), as all Generative Programming artefacts rely on a detailed domain model. As research in the field has progressed, Product Line Use Case Modelling for Systems and Software engineering (Eriksson et al., 2006) may also be a viable option for precise domain modelling. The advantage over Feature Oriented Domain Analysis or FeatuRSEB is that besides a feature model it also allows for the allocation of use cases, use case variations, and cross-cutting concerns to each feature. In the context of the present work, the rationale of Generative Programming to define a precise model of the product domain before implementing any reusable artefacts is followed. This seems especially important if domain-specific languages and generators are to be

built, although they will be rather simple in the beginning. The Requirements, Analysis, and Design activities process of Herzum and Sims' rapid system development process is therefore extended with Feature Modelling and Use Case Development of Eriksson et al.'s approach (2005, pp. 56–60). The result will be a detailed feature model, including a variety of use cases for the required feature combinations. Based on these artefacts, the customer-specific application can be built and reusable components derived.

### Distribution Tier

In their model, Herzum and Sims distinguish between user, workspace, enterprise, and resource tier. The user tier presents the component on the screen and communicates with the user. It may be stand-alone, plug in, or completely non-existent. The local business logic is implemented by the workspace tier which will interact with the enterprise tier. Typical business logic may, for instance, include transaction management utilizing several enterprise-level resources. The latter are implemented by the enterprise tier, providing business rules, validation, and interaction between components. It typically forms the core functionality of business components of a complex, large-scale component-based system. The resource tier manages access to shared resources such as databases, files, or communication infrastructures and shields all higher layers from their technical implementation.

Such detailed differentiation of reusable components and their internal structure is not provided by the Generative Programming approach. Being more generic, Generative Programming leaves such decisions on the target architecture of the product line, which in turn depends on the overall feature model (Czarnecki and Eisenecker, 2000, p. 156). With regard to the Business Component model, feature model and architecture will already be available and are furthermore influenced by the conceptual structure of business components. In combination with Generative Programming, no issues are expected when implementing the four distribution tiers with the means of Generative Programming.

## Functional Categories

The final dimension defines utility, entity, process, and auxiliary business components (Herzum and Sims, 2000, pp. 118 ff.). Utility components can most generally be reused and represent autonomous concepts such as unique number generators, currency converters, and an address book. Entity business components represent the logical entities on which a business process operates and are specific to a particular business domain. Examples are item, invoice, address, or customer. The actual business process is implemented within a process business component. Usually unique for one industry or customer, it is hardly reusable. The fourth category, auxiliary business components, provides services usually not found within a process description. Such may be performance monitoring, messaging, or middleware services.

As with the distribution tier above, Generative Programming does not know any functional categories. However, a detailed feature model in connection with component granularity, distribution tiers, and functional categories will provide a structured and standardized approach to generative development of business components. As such, it is more likely to yield systematic reuse from one or more software product lines than if the structure of reusable artefacts is flexible from component to component.

### **4.3.2.4 Domain-Specific Languages and Generators**

As introduced in the respective sections of chapter 2, model-driven engineering is still far from being mature (Kamp, 2012, p. 5) and providing powerful and comprehensive tools for software development in an industrial setting. The foremost reason therefore is the characteristic of domain-specific languages. Limiting their scope makes them very powerful and expressive, but only for a limited user group (Selic, 2008, p. 381). This results in mediocre tools which has been observed in several case studies and was mentioned as a major drawback (Staron, 2006, pp. 68-69; Shirtz et al., 2007, p. 181; MacDonald et al., 2005, pp. 18-193). For the time being, the implementation of domain-specific languages and model transformation engines and code generators is too difficult on an industrial scale. This applies in particular to a highly complex and hardly predictable environment like systems integration.

This limitation leaves little room for automation according to the third industrial key principle. Until more powerful concepts and tools are available, industrialized systems integration has to adopt already existing and well-proven technologies to the requirements of automation. Sections 2.4.2 and 2.4.3 have introduced some of the most common approaches to DSLs and transformation engines and generators. Given that systems integration is a very heterogeneous field of software engineering, a platform-independent approach must be chosen. This is especially important as according to the organizational model, a business domain may define domain-specific language constructs to be adhered to by the underlying software product lines. If such language constructs are being provided as static C++ metaprograms, all underlying product lines must use C++ as their implementation language. Such restrictions are not acceptable in a business domain comprising several independent product lines. Another approach would be the definition of an independent language on the basis of the Enhanced Backus-Naur Form. This, however, means that one not only has to define the language itself, but also parsers, linkers and compilers, let alone implementing an efficient integrated development environment. In the economically-sensitive context of systems integration, this is not an option.

One possible approach is the adoption of XML as a modelling and transformation / generation technology. The advantages over other languages are as follows:

- XML is a well-known and broadly available technology. Textual and graphical editors are widely available and the majority of integrated development environments supports the manipulation of XML files.
- XML is text based and thus allows the use of conventional tools for version management, including branching and merging.
- XML offers partitioning of larger models into pieces by using XML entities. This allows concurrent manipulation of the model by different developers.
- XML allows a document (i.e. a model represented in an XML file) be checked against a formal specification (i.e. the metamodel represented in an XML Schema Definition), similar to a grammar.

- XML supports the transformation of XML documents by using Extensible Stylesheet Language Transformations (XSLT). XSLT is turing complete (Lyons, 2001) and defines rules to transform a given XML document into any other textual document.
- XML is (as its name suggests) extensible and thus supports the concept of a superordinate business domain layer. Language elements or constraints defined there can be extended on the software product line layer. The adherence of product-line specific models to the business domain-specific language elements or constraints can be assured by the respective schema definition. The extendibility also applies to the XSL Transformation Language.

With XML documents, XML Schema Definitions, and XSL Transformations, it becomes possible to implement a simple domain-specific language including model transformation engines and code generators. In the beginning this language should only be used to create valid feature models and application specifications based on these feature models. This limitation reduces the amount of domain-specific language elements and thus also the complexity of the transformation engines and code generators. Applied to the organizational model of industrialized systems integration, several instances of XML, XSD, and XSLT files are required. The overall structure is presented in Figure 4-17.

On the business domain layer, a root schema definition representing the overall metamodel of the domain-specific language for the business domain is established. This XSD file contains the generic language elements allowed in the DSL and its extensions. As the overall objective of the business domain is systems integration, Vogler's metamodel of integration shall be the basis for the DSL metamodel. Additional language-relevant elements must, of course, be added. Defining integration as a whole on this rather high level ensures that all following extensions will conform to it and thus any product line or application-specific concept is based on the same integration model. In a second step, the feature model of the overall business domain is established. Features to be defined on this layer include business-domain specific entities such as vendor, customer, and product. It also defines means of integration such as middleware, messaging, or data exchange formats. The feature model also represents business domain-specific core assets (reusable business components suitable for all underlying software product lines) as defined in

4.2.3. With the help of XSLT and rules set in accordance thereto, the feature model is then transformed into the business domain metamodel represented as an XML Schema Definition. In modern integrated development environments or XML tools (e.g. XMLSpy), XSD files can automatically be derived, although they need to be manually adjusted. Subsequently, constraints may be defined in the XSD, as these cannot be represented in an XML document. As the business domain layer may also specify reusable software components, means for their generation become necessary as well. In the beginning of model-driven systems integration, generation should be limited to substitution of XML elements with the respective component or glue code. In Java applications with EJB, components would, for instance, result in source code or JAR file, metadata files, or deployment descriptors.

The software product line layer consists of similar steps. The business domain metamodel extended according to the specific requirements of the software product line. A product line covering customer relationship management solutions may, for instance, extend the element customer with some additional marketing-relevant fields. Additional extensions necessary to describe the complete feature model are conceivable. Once the XSD of the product line is complete, its detailed feature model is developed within an XML file conforming to the product line metamodel. In the beginning of model-driven engineering in systems integration, features either specify reusable software components (from the business domain or product line layer) or generic containers for customer specific implementations. This limitation allows for quickly setting up model-driven engineering and advancing the level of detail as the product line grows and further experience is gained. In the beginning it may very well look like a configuration tool for products of a product line. As the product line metamodel (i.e. its XSD file) was extended from the business domain metamodel, it automatically conforms to domain-wide requirements. The feature model now describes all possible products from this particular software product line. As in the business domain layer, the XML feature model is transformed into a software product line metamodel represented by an XML schema definition file. After adding the necessary constraints, it can be published as the basis for application development in the production layer. In addition to that, a generation facility for the product line's core assets is required. As

with the business domain layer, this generation facility should be implemented as an XSLT document substituting features defined in an application model with the respective software components. The XSLT documents from both the business domain as well as the product line layer must include translations for all possible features and combinations thereof. This also explains why for the first generation it should be limited to rather large parts. Too many would exponentially increase complexity and increase the risk of failure.

On the application layer, the customer's requirements are analyzed and the respective features of the product line are selected. This results in a definition of the final product within an XML file, adhering to the specifications from the software product line described in the metamodel. This can be seen as the instantiation of a feature model with all variability removed. Once completely specified and verified, the application layer uses the XSLT files provided by the business domain and software product line layer to automatically produce either intermediate models or executable code based on the product line's core assets. In the beginning it is suggested to generate text-based artefacts such as source code or deployment information. This can easily be achieved with substitution mechanisms broadly available. Any customer-specific implementations will subsequently be added to the generated code. This can either happen within the generated files directly or within placeholders to avoid overwriting customized code upon regeneration. Depending on the generated artefacts, a final compilation or assembly may still be necessary before application delivery.

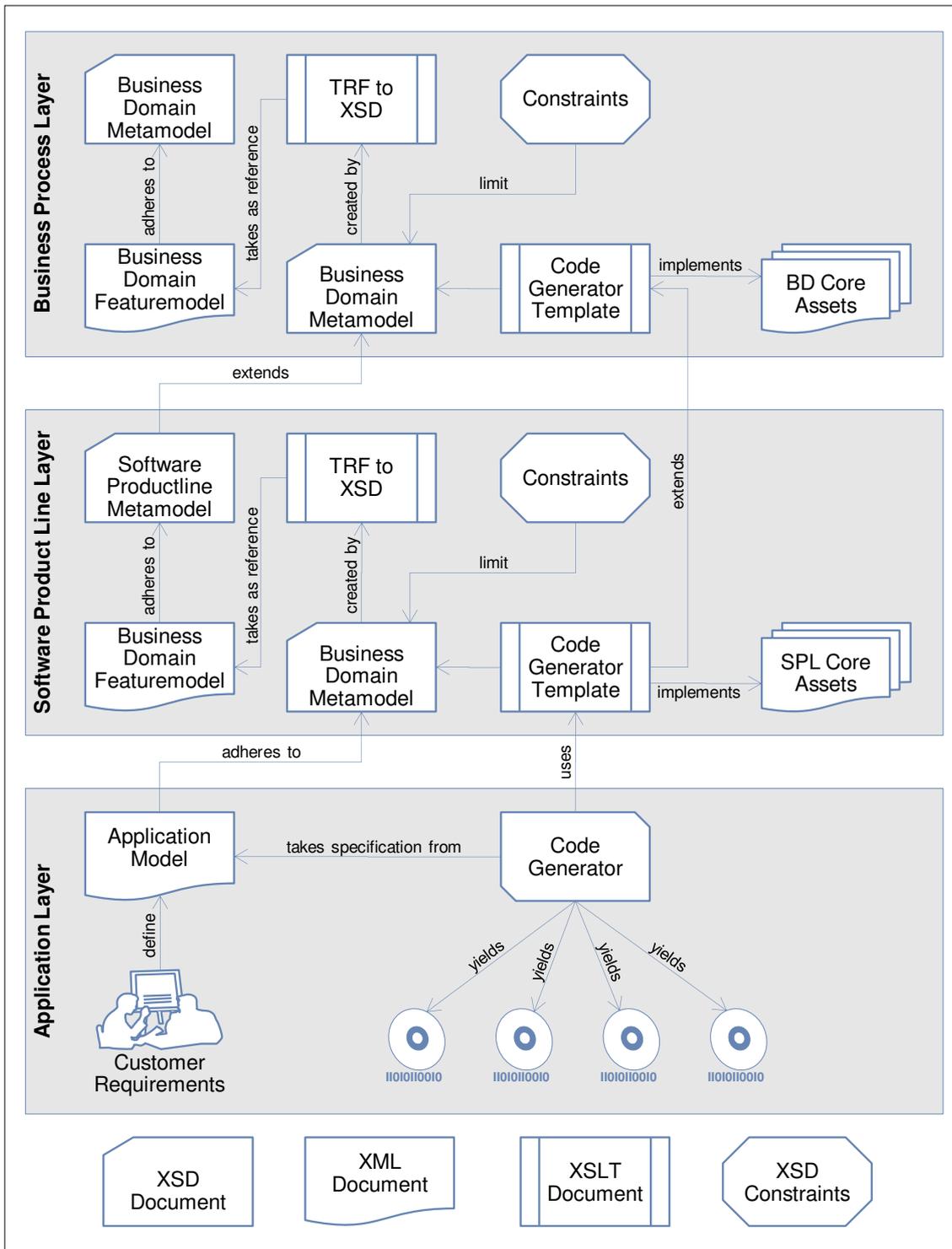


Figure 4-17: XML based application modelling and code generation for industrialized SI

### 4.3.3 Conclusion and Coverage of Research Objectives

During the course of research it became evident that model-driven engineering in its current state seems not mature enough to be practical in an industrial context (Kamp, 2012, p. 5). This especially applies to a complex and volatile field such as systems integration. The primary rea-

son therefore can be found in an insufficient tool chain and thus high efforts for developing domain-specific languages, model transformation engines, and code generators (Staron, 2006, pp. 68-69; Shirtz et al., 2007, p. 181; MacDonald et al., 2005, pp. 18-193). Despite promising concepts for a variety of issues, practical problems remain unsolved. These especially affect usability, interoperability, and scalability (Selic, 2008, pp. 385-387): Visual models of large or highly-integrated systems become extremely large and are hard to oversee in their entirety. Feasible partitioning mechanisms for concurrent model manipulation are not offered. Context-based modelling support for developers, such as syntax completion in C++ or Java integrated development environments, is also not available. Interoperability issues intensify the problem. Exchanging model artefacts between different tools is hardly possible, although the MDA standard may alleviate this issue midterm.

Although several approaches towards the implementation of model-driven engineering are available, only Czarnecki and Eisenecker's Generative Programming (2005a; 2000), Object Management Group's Model Driven Architecture (2003), and Software Factories by Greenfield et al. (2004) were found to be suitable for an industrialized approach. Out of these, Generative Programming was chosen as most suitable for industrialized systems integration, especially as large parts of its processes have already been covered in the first two concepts of industrialization, which are software product lines for specialization and component based development for standardization and systematic reuse.

In the course of section 4.3, Generative Programming was analyzed and its processes have been allocated to the organizational model for industrialized systems integration. This allocation consolidates recurring efforts which would otherwise be necessary for each software product line of a given business domain. The results are lower set up costs for software automation. The processes were furthermore set in conjunction with those from the business component approach chosen for standardization and systematic reuse, which shows that both benefit from each other. Previously-defined processes of the business domain and software product line layer were adapted accordingly. For model transformation engines and code generators the situation was more difficult. Although Czarnecki and Eisenecker did suggest modelling and generation

techniques, these were not suitable for the overall concept presented in this work due to limitations of a certain programming language (3GL template metaprogramming), the need for extreme efforts (custom DSL and generators), or immaturity (intentional programming). To overcome these limitations, an XML-based approach has been developed. Its advantages lie in the ability to be hierarchically organized, its foundation on a well-known and broadly available technology, its text-based orientation for easy usability and scalability, as well as ability to be used with widely-available tools and integrated development environments. Furthermore, the approach allows starting with a small and simple domain-specific language and simple XSLT code generators based on content substitution. Both can be incrementally advanced as the product line develops. To ensure compatibility between different software product lines within a business domain, it is suggested that the top level metamodel of the domain specific language be aligned with Vogler's metamodel for systems integration (Vogler, 2006). This alignment consistently bridges the gap between the business components defined during component-based development and model-driven engineering, as both root in the metamodel for systems integration.

#### **4.4 Summarized View of Industrialized Systems Integration**

In the present chapter, a novel approach to industrialized systems integration was presented. It originates from the attempt to increase efficiency in the field by applying industrial key principles to software development. These are specialization, represented by software product lines, standardization and systematic reuse, represented by component-based development, and automation, represented by model-driven engineering. As discussed in chapter 3, systems integration comes with certain characteristics distinguishing it from conventional software development such as developing a commercial off-the-shelf office suite. These characteristics also inhibit the seamless implementation of software product lines, component-based development, and model-driven engineering. The objective of the presented approach was to find a way by which the three key principles can be utilized in systems integration despite its characteristics. This means that each concept must be easy to implement, require little upfront investments and

break even after few products have been developed. Furthermore, the overall approach must allow for the incremental implementation, with each concept offering an increase in efficiency.

The first industrial key principle aims at specialization of the production process, i.e. the implementation of **software product lines**. However, in a systems integration context, a highly heterogeneous, complex and volatile environment (Vogler, 2006, pp. 24 ff.) may prevent sufficient specialization. The most common issues can be summarized as high technical and functional heterogeneity, unique IT environments and technology combinations, as well as an uncertain return on invest due to high upfront investments and a low number of very similar products. A product line which is too broad may not yield efficiency gains through economies of scope. One that is too strict, however, often forces development to occur outside the product line's boundaries (Greenfield et al., 2004, pp. 21–23). Furthermore, in the present context integration across software product line boundaries is inevitable. The question arising from these issues is how to scope and organize software development in systems integration. To answer this question, section 4.1.1 identified possible forms of organization for software product lines in general, while section 4.1.2 matched them to the particular needs of systems integration. The considerations result in a novel three-layered organizational model which combines the advantages of specialization with the necessities of systems integration. The new superordinate layer, i.e. the business domain layer, depicts a complete industry segment a systems integrator is doing business with. Within such a business domain, several software product lines are established, each producing software for a particular area. In an automotive domain, for example, product lines for shop floor systems, customer relationship management, or logistics are conceivable. This segmentation is backed by the current organizational structures of major systems integrators. The advantage of such a structure over previously known product line concepts is that fundamental core assets, technologies and systems may be defined and developed for all underlying product lines. These may, for instance, be common middleware, logical entities and data structures, or domain-specific language and code generator frameworks. These centrally-provided core assets allow for the removal of significant implementation efforts from the product lines and at the same time ensure consistency in case of integration needs. The developed

organizational model for industrialized systems integration thus defines the relevant processes of the business domain layer, simplifies software product line implementation in the product line layer, and defines how products are built at the production layer. The result is depicted in Figure 4-3.

The second industrial key principle is represented by **component-based development** and is affected by the same characteristics as software product lines. To overcome these, an already existing and proven approach to CBD was adapted. A suitable approach was to be identified according to the ability to be spread across the newly developed organizational model of industrialized systems integration, the adaptability to Vogler's metamodel of integration (2006), technology independence, and the ability to be incrementally developed as the product line advances. Among the examined approaches, Herzum and Sims' Business Component Factory offered the foundation and enough flexibility to be adapted to the requirements of systems integration. It describes component-based development from various viewpoints, each being able to be aligned either to the organizational model or the integration metamodel. The former allows joining CBD with software product lines as the first industrial key principle. This is achieved by defining which architectures and component types of the business component factory approach are being defined by which organizational layer and how the development process is distributed (see Figure 4-6, Figure 4-7 and Figure 4-8). In addition to Herzum and Sims' approach, component granularity was further subdivided to allow a hierarchical decomposition between the business domain and its software product lines. Alignment with the organizational metamodel, in addition, allows for the allocation of distribution domains and functional categories of business components to clearly defined metamodel entities. This allocation allows for a very similar structure of component-based development across the product lines of a business domain. Access to enterprise-wide data sources, for instance, will always occur with components from the enterprise resource domain. Controlling a business process will always be the responsibility of a process business component, and so on. Software developers can be sure that systems from another domain will adhere to this concept and in this example, process control is carried out by no other components. Figure 4-9 to Figure 4-14 illustrates this in more detail.

For **model-driven engineering** representing the third industrial key principle, a less advanced approach had to be developed. This was due to a lack of experience, modelling standards, and especially tools covering the complete development process (Teppola et al., 2009, p. 19; Selic, 2008, p. 386; Afonso et al., 2006, p. 127). This situation is especially hindering in a heterogeneous and volatile field such as systems integration. The present work has therefore adapted Czarnecki and Eisenecker's Generative Programming (2005b; 2000) to the particular needs of industrialized systems integration. Advantageous in this approach was the fact that the majority of Generative Programming processes were already available with software product lines and component-based development. Where this was not the case, the respective approaches have been extended to support model-driven engineering based on Generative Programming. Due to the afore-mentioned issues of MDE, developing a domain-specific language and the according model transformation engines and code generators must be based on a well-known and broadly available technology. Existing approaches like template metaprogramming, visual modelling tools or intentional programming are not mature enough to be applied in an industrial setting, especially with regard to the lack of a tool landscape. To still benefit from modelling and code generation in systems integration, a novel XML-based approach was developed. It specifies a metamodel at the business domain layer mandatory for all underlying software product lines. These may, however, extend (but not curtail) the model to their needs by defining detailed feature models for their intended products. The production layer may then model a particular customer's requirements into a product specification. XSLT documents are furthermore used to translate the chosen features into executable code by substituting them with core asset implementations. The overall process is presented in Figure 4-17 in further detail.

In conclusion it can be said that a selective combination of existing technologies, tailored to the field of systems integration, as well as new developments alleviating their overall implementation have presented a viable approach towards industrialization of the field. Chapter 4 pursues the second objective of this thesis' aim by analysing the existing methods of software industrialisation to see which ones could be used in systems integration and how they need to be adapted where necessary. It also partially completes three nested cycles of design research to

theorize each individual solution. Their justification was then done by presenting each concept described with a peer reviewed research paper to the scientific community and discussing them with subject matter experts from the industry.

As the concepts developed were presented from a generic point of view, chapter 5 will present their major artefacts on a real-life example. It will start with the development of an organizational structure and the processes required in the business domain layer. On the next level, a product line is exemplarily defined including typical software components. For one of these examples, the following chapter also presents its representation within the domain-specific language and the artefacts required to translate it into executable code.

Subsequent to chapter 5, an expert interview with representatives from large international systems integration enterprises will evaluate the overall feasibility and discuss possible areas of improvement.

## 5 Exemplary Implementation: Industrialized Systems Integration in the Automotive Domain

The previous chapters described the fundamental concepts of industrialization, their implementation in software development, as well as the challenges to be solved when doing so in a very complex and volatile environment, i.e. systems integration. To overcome these challenges, chapter 4 presented an approach towards industrialized systems integration, i.e. applying software product lines, component-based development, and model-driven engineering. In order to provide an approach beneficial to various systems integrators in the industry, it was not tailored to the needs of a particular one. For better understanding and illustration, however, a concrete example is regarded to be helpful. The present chapter will present such within an exemplary implementation based on a real world example and the author's professional experience in the field. It is constructed similar to a case study approach, which allows testing and verifying new theories and describing the processes therein (Johnston et al., 1999; Eisenhardt, 1989, pp. 534f.). They are "particularly well-suited to new research areas or research areas for which existing theory seems inadequate" (Eisenhardt, 1989, p. 548). Therefore the current chapter presents the key elements of the developed approach at the example of a fictitious systems integrator, following an anonymised real-world example. Each concept developed in chapter 4 will be tested against the example of the fictitious systems integrator. Due to space restrictions, only one instance of each concept will be presented instead of an all-embracing industrialization approach (which moreover would take several months, if not years, to fully complete). The goals for the exemplary implementation are therefore defined as follows:

E1: Can the newly developed Organizational Model for Industrialized Systems Integration be implemented at a large scale systems integrator?

E2: Is the newly developed approach for Component-Based Systems Integration able to represent typical products of a large scale systems integrator?

E3: Is the newly developed approach for Model Driven Systems Integration able to represent typical products of a large scale systems integrator?

The implementation is conducted as an instrumental experiment in which the environment of the experiment is given, i.e. the situation of a large systems integrator serving several customers from different industries. The objective here is not to understand the systems integrator itself, but to use this particular example as an instrument to test the theoretical concepts developed in the previous chapters. The sample systems integrator itself thus is of secondary interest and plays a supporting role. It does however define the applicability of the theoretical concepts being tested. Adding additional instances of the experiment may limit or broaden the validity of the results.

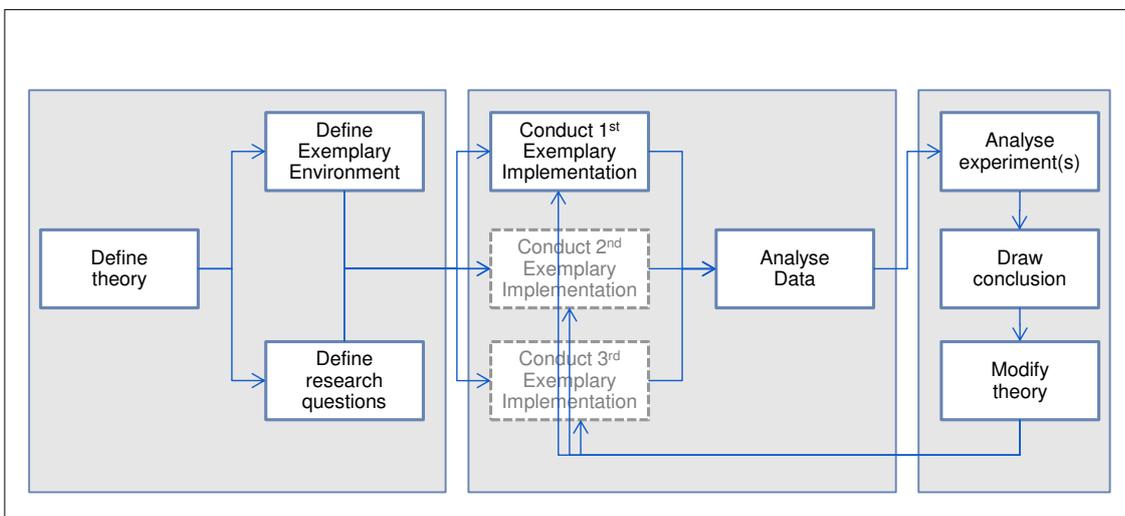


Figure 5-1: Exemplary Implementation Research cycle

The exemplary implementation has taken place in three phases: a preliminary stage in which the theory has been developed, the surrounding conditions defined, and the research questions formulated. During the fieldwork and analysis stage, the research questions were tested and the results analysed. The final stage drew conclusions drawn and modified the theories where necessary. Phases two and three were conducted in an iterative manner, i.e. the theory was constantly modified until it was able to answer the research questions from the preliminary stage. This means that, especially in the case of model driven systems integration, several cycles and theory modifications were necessary until a feasible solution was found. Further details on the approach's initial weaknesses and implications from the implementation can be found in section 5.5.

## **5.1 Example Definition: A Sample Systems Integrator**

The object of research is a fictitious systems integrator on the European market, competing with well-known enterprises such as IBM, Accenture, or CapGemini. For the present thesis it shall be called ACME Integration Solutions. The main customers are assumed to be several large original equipment manufacturers as well as third party suppliers in the automotive sector. Other customers are located in utilities sector, travel and transportation industry, public services, and banking and insurance.

To retain the current market position and be able to react to customer requirements more flexible and dynamic, several strategic initiatives have been put in place. One of these is concerned with the application of industrial methods to software development in systems integration. To achieve this, ACME Integration Solutions has chosen to implement the approach developed in chapter 4 of the present work.

### **5.1.1 Service-Offering Portfolio and Organizational Structure**

As of today, ACME Integration Solutions is active in several different industries and technological areas. The service-offering portfolio is separated into industry-specific solutions such as product lifecycle management systems for manufacturing or automotive industries and generic solutions such as enterprise resource management solutions, independent of any industry. An overview of the service offering portfolio is given in the following figure.

Automotive OEM & Suppliers	PLM	SCM	Sales & After Sales	Dealer Mgmt. Systems	Embedded Systems	Digital Factory
Public Sector	eGovernment solutions	Public Safety & Security	Document Mgmt. Solutions	eHealth Solutions	Tele-medicine	Road Toll Solutions
Telecommunication Industries	NGN Products & Services	NGN Network Mgmt.	Operation Support Systems	Voice & IP Integration	Operation Support Systems	Billing Services & Solutions
Insurance & Banking	Mainframe Legacy Solutions	Identity & Access Mgmt.	Financial Online Services	Business Intelligence Solutions	Risk Mgmt. Solutions	Secure Transact. Mgmt.
General Services	CRM	HR Mgmt.	Finance & Accounting	Accounts Receivable	Managed Document Solutions	Business Process Consulting
	Web Application Solutions	Transact. Mgmt.	Enterprise Security Mgmt.	Business Reporting	Data Warehouse	Workflow Mgmt. Systems
	C++	Java/J2EE	.Net	Visual Basic	ABAP	Database Services

Figure 5-2: ACME Integration Solutions Service-Offering Portfolio

The tasks provided under the service-offering portfolio include the development, integration and management of standardized, industry-specific and customer-specific solutions as well as the necessary consulting and testing. For each industry, specific solutions are offered. On the next level of detail, each industry and subsequently each service offering element is described. In the present example, the automotive original equipment manufacturer & suppliers industry segment was chosen for further itemisation.

The industry segment automotive original equipment manufacturer & suppliers covers four major groups of customers: the original equipment manufacturers (e.g. Daimler, BMW, MAN), automotive industry suppliers (e.g. Bosch, Continental), car and motor vehicle parts retailers (e.g. ATU, Autoparts24), and dealers and body shops. For these customers, six industry-specific service-offering elements have been defined (see Doe, 2012, p. 17). Customer requirements not covered by industry segments are being developed within the general services segment, depending on the required technology.

Once a customer request is received, sales representatives together with technical experts identify which service-offering element is best suited to fulfil this request. After specifying the exact customer requirements, the organizational unit responsible for the particular service offering element then begins software development. However, due to historic growth, the allocation of an service offering element to an organizational unit is not always exclusive, resulting in increased communication and alignment efforts.

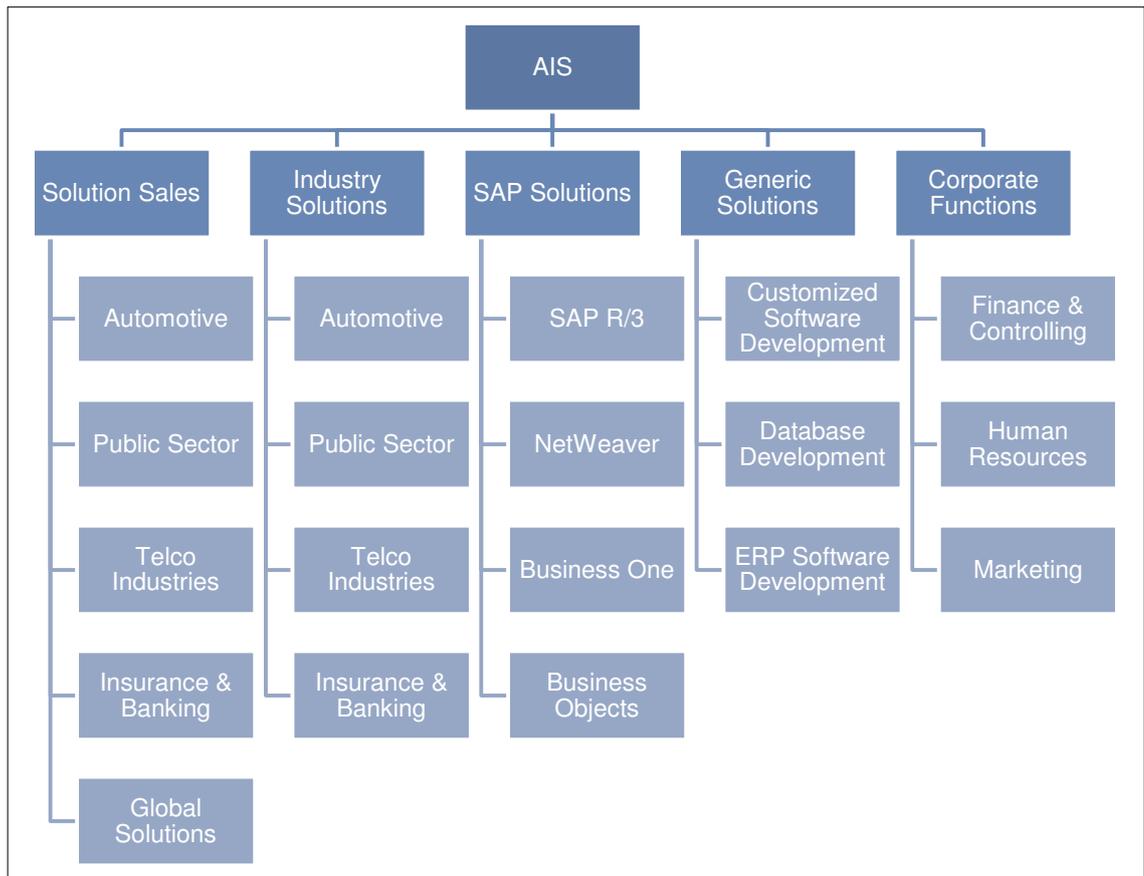


Figure 5-3: ACME Integration Solutions Organizational Structure

Comparing the service-offering portfolio with the organizational structure, it can be seen that there is no clear alignment between the two. A supply chain management solution for an automotive customer may require integration with a customer-relationship management software. Expertise for the first is located in the automotive unit, for the latter in the enterprise resource planning software development unit. In case customer-specific extensions are required, even a third department may come into the project.

Coming back to the strategic initiatives ACME Integration Solutions has chosen to implement, applying industrial methods such as systematic reuse is difficult due to an arbitrary context. The enterprise resource planning software department, for instance, can't specialize to the automotive industry as their services are also required by the telco industries – despite the fact that the characteristics of both industries in regard to Customer Relationship Management products are totally different. To be most effective, however, assets to be reused must be focussed to a particular domain.

### **5.1.2 Objective of strategic realignment**

ACME Integration Solutions' management understood the concerns and decided to reorganize its software development units. Reorganization is to be based on the implementation of industrialized software development in systems integration as presented in chapter 4. The key objectives are:

- Realign and focus service offering portfolio
- Proactively develop service offerings based on anticipated market demands
- Standardize service offerings to increase systematic reuse
- Improve efficiency and lower development cost

The first two objectives can be achieved with the implementation of software product lines. Different service offering elements will be allocated to a particular business domain as described by the organizational model for industrialized systems integration. This may also mean that in particular generic service offering elements may be present in more than one business domain. Subsequently, software product lines representing families of related products are being instantiated. The business domain analysis process furthermore ensures that marketable solutions are made available. The third objective is covered by the implementation of component-based development. The fourth objective, improve efficiency and lower development cost, is primarily available due to specialization and systematic reuse. However, further efficiency may be obtained by automating the development process with model-driven engineering.

To test the research questions E1 to E3, the implementation of the concepts developed in chapter 4 is presented in the following at the example of ACME Integration Solutions' automotive sector.

## **5.2 Automotive Business Domain Management**

Integration typically occurs within the boundaries of the respective industry. It can safely be assumed that a supply chain management system from the automotive sector will never be connected to an e-government solution from the public sector. Thus, similar to its service-offering portfolio, ACME Integration Solutions decided to structure systems integration into different business domains. In the future, these will be automotive, public & healthcare, telecommunications, and banking & insurance. To serve needs not covered by the business domains and their underlying software product lines, a customized solutions department will be established. This department will produce software conventionally from scratch. In the present example, the automotive business domain will be discussed in further detail. The principles are, however, applicable to the other business domains as well.

### **5.2.1 Business Domain Analysis and Portfolio Definition**

The first process of the business domain layer from the organizational model is business domain analysis. Therefore ACME Integration Solutions has conducted a market analysis project. The outcome describes the core processes of the automotive industry and the typical IT infrastructure required to support these. In the present example these are presented in the following figure:

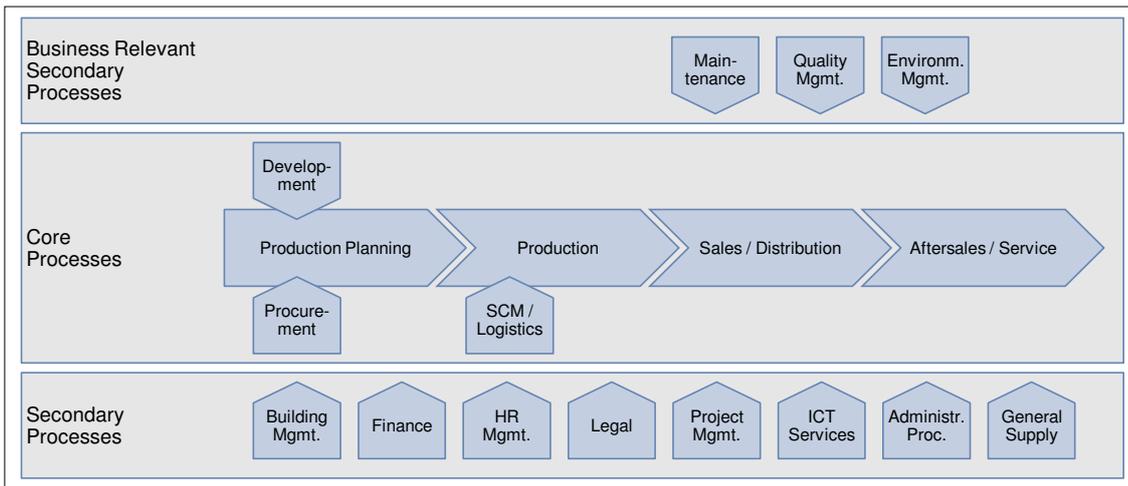


Figure 5-4: Core processes of the automotive industry (Ammer and Stolte, 2010, p. 6)

The project also identified current trends and requirements in information and communication technology in the automotive industry. From these trends, the economically most feasible ones were selected, such as integrating a product lifecycle management suite of an automotive manufacturer with the supply chain management system of its external supplier. On the other hand, it doesn't make sense to invest in a software product line for dealer management systems, as these usually are small and medium-sized enterprises which do not have sufficient funds to buy such products. Product lines for automotive product lifecycle management or supply chain management systems in turn will have a much larger market potential. The structure of ACME Integration Solutions' business domain will therefore appear as follows:

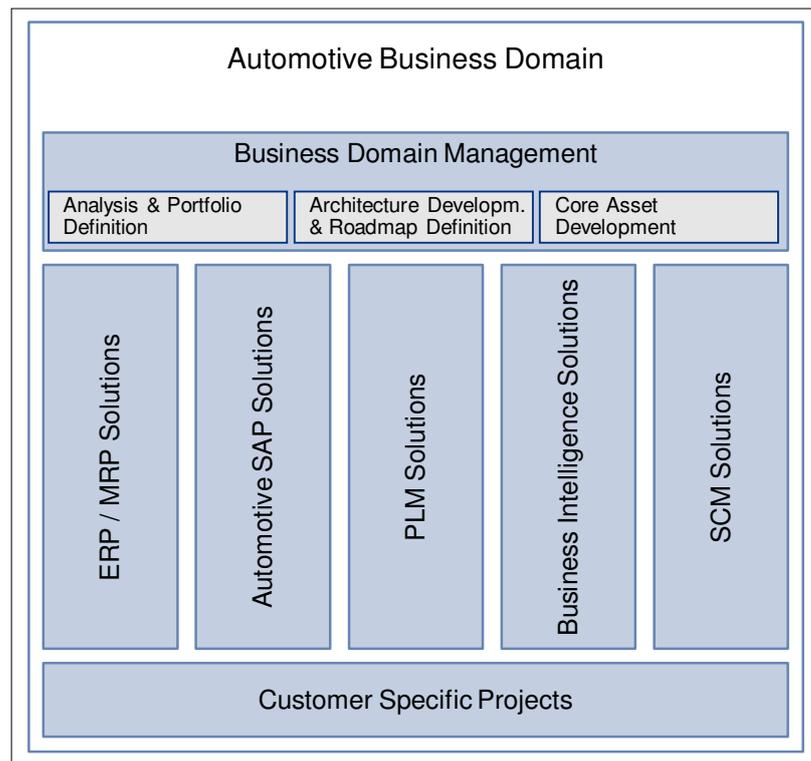


Figure 5-5: Automotive Business Domain of ACME Integration Solutions

Each vertical bar represents one software product line within the automotive business domain. An additional layer for customer specific projects occurring outside of the software product lines has been added below. This ensures that also non-standard requests can be handled. For all of the above software product lines as well as for each individually, joint features and generic functionality must now be defined. In the present example, one of each type is illustrated. It is important to note, however, that during the business domain processes, only those features or aspects are defined which are necessary for later integration of the products with those from neighbouring product lines.

- For all software product lines, the support of **EDIFACT** is mandatory. The Electronic Data Interchange for Administration, Commerce and Transport serves as a standard data interchange format in the automotive industry. It is further specified within an industry specific subset known as ODETTE. As this standard is officially recommended by the German Association of the Automotive Industry (Verband der Automobilindustrie e.V., 1991) and utilized among many different original equipment manufacturers and suppliers, it ensures

compatibility even with external systems to seamlessly exchange information and thus alleviates integration. EDIFACT/ODETTE messages are text based; a part of a typical message would appear as follows: 'PDN+12345:120814+1000' where PDN defines a previous delivery instruction, 12345 the order number, 120814 the date and 1000 the number of items sent.

- The second type is applicable only for supply chain management solutions. Each supply chain management system must adhere to the **SCOR-Model**. The Supply Chain Operations Reference model (Supply Chain Council, 2010) is a reference process model for supply chain management in manufacturing industries. It consists of three levels of process detail (see Supply Chain Council, 2010, p. 11): In Level 1 these are plan, source, make, deliver, and return, which are used to describe the scope and high-level configuration of a supply chain. Level 2 decomposes each of them to differentiate the strategy of the Level 1 processes, e.g. the make process is divided into make-to-stock, make-to-order, or engineer-to-order. On Level 3, steps to execute a Level 2 strategy are described. For make-to-order these would, for instance, be schedule product activities, issue product, produce and test, package, stage, dispose waste, release product. Layer 4 as the last layer describes the industry-specific activities to perform these steps.

Of course there are many more mandatory features, rules, and regulations for the complete business domain as well as individual software product lines. It is thus advisable to create a feature model for the business domain layer as well. For the present example, however, the above two shall be sufficient for further explanations.

In addition, a more precise definition of the product portfolio across all five software product lines is advisable. For the product portfolio of the supply chain management product line, the functional scope typical for integrated supply chain management suites is defined. It is based on the functional scope of typical packaged and customer specific applications so that the product line covers the majority of customer requirements in this area. These primarily include (Bretzke, 2006, p. 14):

- Strategic Network Planning
- Master Planning
- Demand Planning
- Material Requirements Planning
- Production Planning
- Scheduling
- Distribution Planning
- Transport Planning
- Demand Fulfilment

Later it will be up to the software product line to specify which products or technical implementations it will support and what the feature model will look like. Based on a variability model, a customer may then decide which functionality is required. Concluding the above, AIS began defining the business domain layer with a market analysis of the automotive industry. Current challenges were identified and product lines covering the most important market demands defined. In addition, the business domain layer has set a number of mandatory features and constraints each product line has to implement to ensure seamless integration of their respective products. Subsequently, the business domain architecture and roadmap definition process define how the products of the different product lines may interact with each other and which technologies must be supported.

### **5.2.2 Business Domain Architecture and Roadmap Definition**

The industry trends from the previous section have to be seen in the context of continuous cost-saving measures of the automotive industry. For the present example, this implies that custom-developed legacy software is replaced by packaged applications wherever possible in order to reduce software maintenance efforts (Naujoks, 2010, p. 28). This of course only happens if the overall return on investment is positive. On the other hand, due to “the concept of service-oriented architectures (SOA), using combined approaches is becoming increasingly important (e.g. integrating custom-specific software on SOA based standard platforms)” (Nau-

joks, 2010, p. 28). Inflexible but mission-critical legacy systems may not be replaced due to the risk of stopping production lines. ACME Integration Solutions has therefore decided to offer its integration solutions within a service-oriented architecture. The advantages of a service-oriented architecture are that abstraction levels are based on actual business processes, their workflows, and tasks (Cummins, 2009, p. 28). Orchestrating these in the right way helps to reuse standard functionality (e.g. message transfer or logical entities such as bills of materials) while providing customer-specific solutions. Such abstraction levels also fit very well together with the component-based approach for industrialized systems integration as described in section 4.2.3. Thus it is made mandatory by the business domain layer that all underlying software product lines are able to exchange information and provide functionality via a generic structure as presented in Figure 5-7. This so-called ‘automotive service bus’ shall be based on the web services architecture as defined by the World Wide Web Consortium. A service provider can publish functionality it provides to external applications with a directory service. In the present example, the supply chain management system would register an order-tracking service with the universal description, discovery, and integration service at the service broker. How this service may be used and what exactly it delivers is defined in a machine-readable document written in the web service description language. A potential client may search the service broker and download the web service description language document to directly invoke the service offered by the service provider. A system for manufacturing resource planning may, for instance, query the order tracking service provided by the supply chain management system to update its planning. Figure 5-6 illustrates the technical architecture of the automotive service bus, including some more advanced aspects like reliable messaging or security.

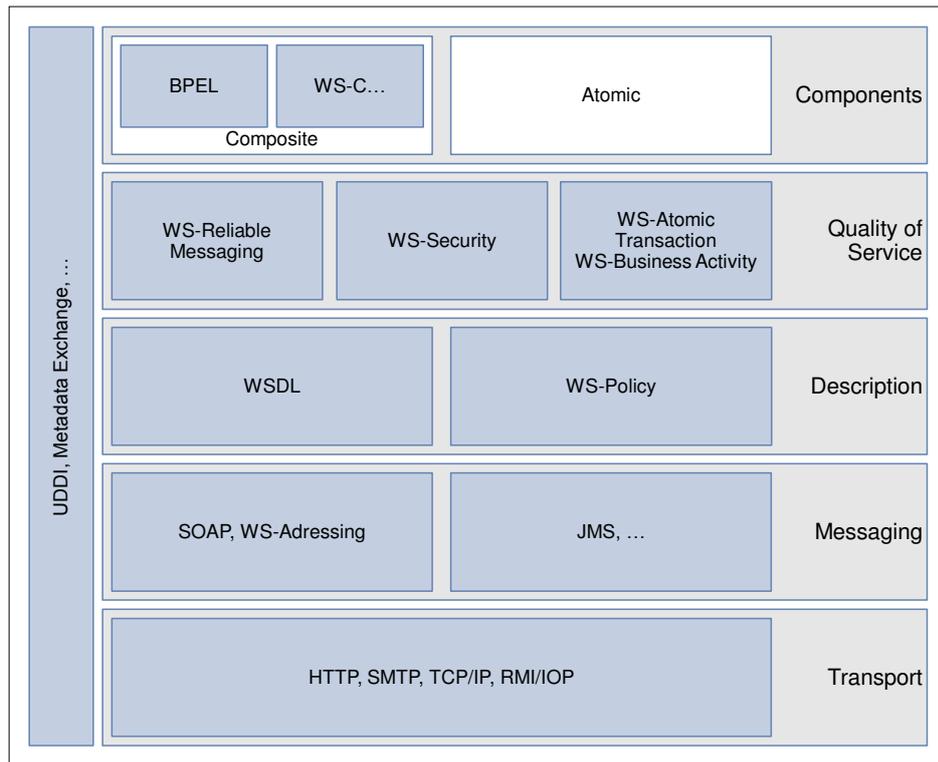


Figure 5-6: Automotive Service Bus Technical Architecture (Weerawarana, 2005, Section 3.1)

The transport, messaging, description, and quality of service layers are standardized and obligatory for integrative functions of any software product line within the automotive business domain. This means that adherence to the automotive service bus and its technical architecture is only necessary for functionality provided to or obtained from external applications. To provide the fundamental services to the software product lines, ACME Integration Solutions has decided to implement a commercially available enterprise service bus solution. Such systems usually provide various functionality, including the following (see (Auer et al., 2007, pp. 47, 48):

- Routing of service requests and conversion between different protocol versions
- Data transformation and mapping
- Communication infrastructure
- Monitoring and Management
- Enforcement of security policies
- Transaction management
- Service registry and meta data management

Based on the initial market research project, external studies on Enterprise Service Bus solutions (Vollmer, 2011) and already existing knowledge, ACME Integration Solutions decided to implement the IBM WebSphere Enterprise Service Bus. It provides all of the above features and may be extended with other WebSphere products at a later point in time. Of course in an actual implementation this decision would be based on a lot more criteria and include financial calculations. However, for the present example, this limited decisional base shall suffice.

In addition to the automotive service bus, the support of EDIFACT message transfer was defined as mandatory for all product lines. Transfer of such messages shall occur over X.400, a widely-used protocol among automotive manufacturers and suppliers. The overall structure of ACME Integration Solutions' functional architecture for the automotive business domain is depicted below.

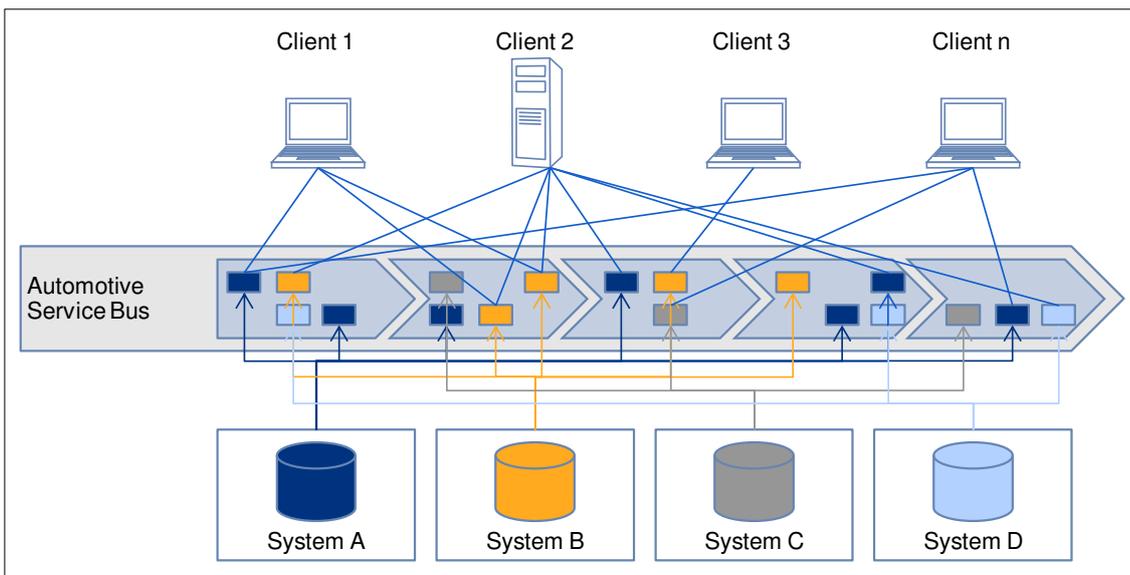


Figure 5-7: ACME Integration Solutions automotive functional architecture

In addition to the functional requirements, the industrial concepts defined in sections 4.2.3 and 4.3.2 are mandatory. This includes in particular adherence to the integration metamodel defined by Vogler (2006), as well as developing components in accordance with Herzum and Sims' Business Component Factory (2000). The former explicitly defines how an integrated system (which also includes those potentially being integrated with others at a later stage) must be subdivided and how the resulting entities may interact with each other. The latter defines how these

entities are being implemented as reusable components. Adherence to both by all software product lines ensures compatibility between the products and allows for simplified integration, especially if such becomes necessary at a later stage and was not anticipated earlier.

In a large business domain, it is also advisable to define a technical roadmap. It gives certainty to the planning of the product portfolio and allows a joint update to architectures and technologies used. For the present example it can be seen that there will be no technology changes until the end of 2013. After that, only the Simple Object Access Protocol, web service description language and the Java Message Service will remain unchanged. All other web service related technologies will be updated to the next version and the WebSphere installation will be upgraded to version 8. In addition, XML/EDIFACT, an XML based standard for EDIFACT message exchange, will be implemented and must be supported by all underlying software product lines. This is also the point at which a new major release of the software product lines becomes possible. Figure 5-8 illustrates the technology roadmap.

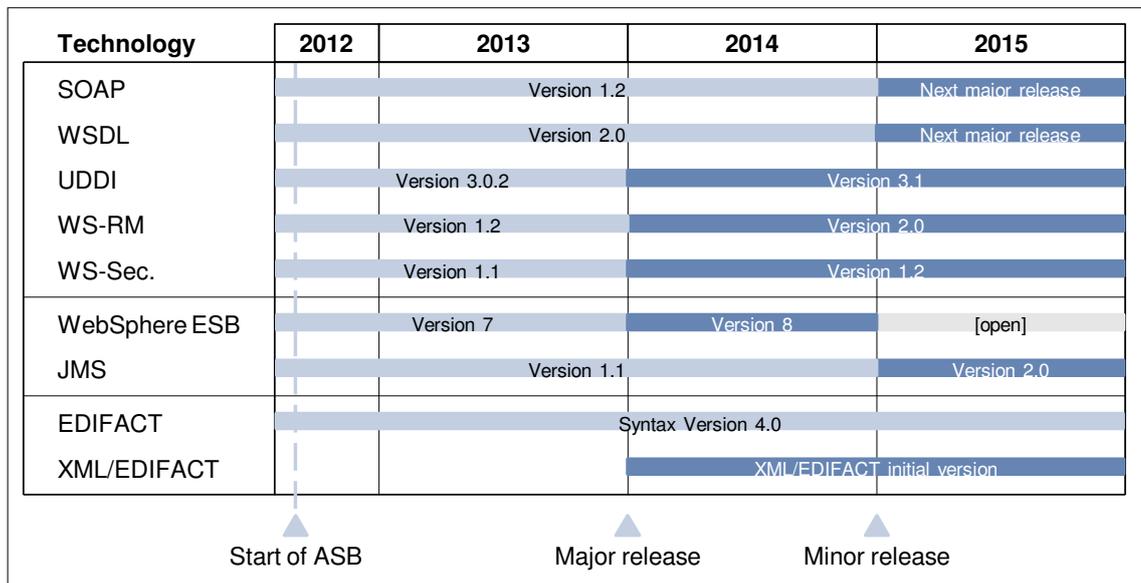


Figure 5-8: ACME Integration Solutions Automotive Service Bus technical roadmap

### 5.2.3 Core Asset Development

After defining the overall architecture and technologies to be used within the business domain, resulting core assets need to be developed. As stated in section 4.1.2.1, developing joint core

assets can significantly reduce set up cost for the underlying software product lines. These typically include a common feature model including descriptions for features available in all software product lines, state charts describing the features and their interaction with each other, common architecture and design principles (see above), component design documents based on the integration metamodel and business component approach, domain specific language artefacts and code generators, as well as common development processes and other tangible and non-tangible artefacts. Depending on the business domain architecture, some of these core assets are mandatory while others are optional. For the present example, three core assets shall be discussed in more detail. The first is the WebSphere Enterprise Service bus as an integration framework, the second is an exemplary definition of a business component for message transfer, and the third is the representation of a component in a domain-specific language along with the required code generator.

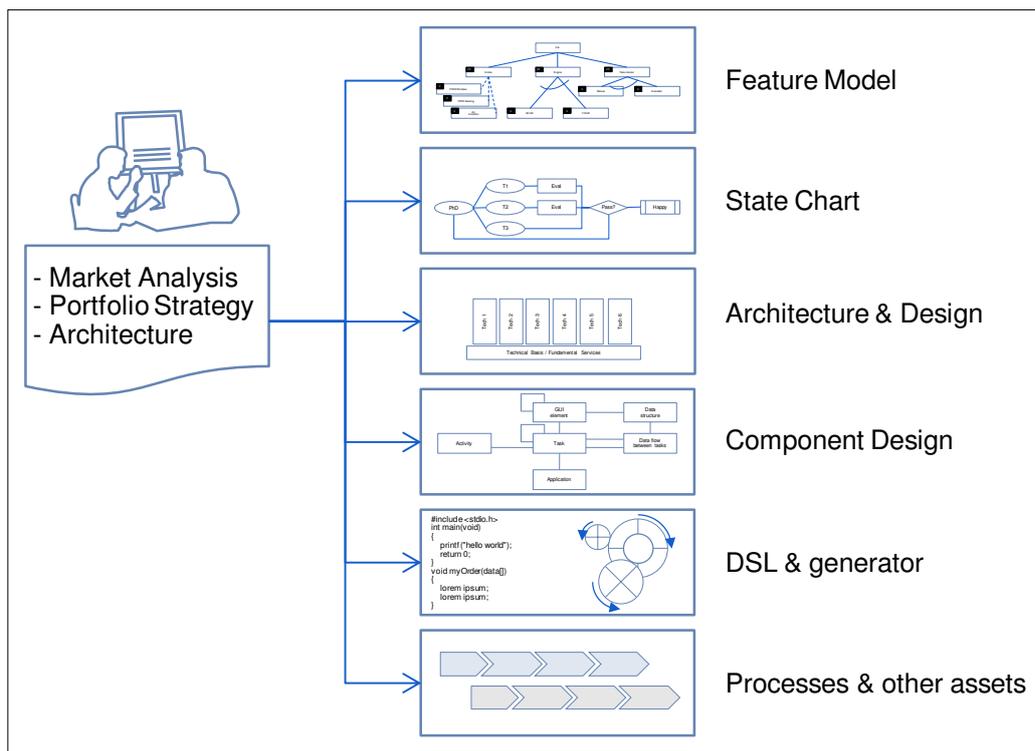


Figure 5-9: Typical core assets of a business domain (see Kang et al., 2002, p. 60)

### 5.2.3.1 Integration Framework and Infrastructure

The first core asset of the business domain layer is the IBM WebSphere Enterprise Service Bus. It serves as the basis for integration and is thus mandatory for all products of a given product line. The easiest way to deploy this core asset is to provide installation images of the software along with an answer file for unattended setup, which may look like the following.

```
#####
# (C) Copyright IBM Corporation 2012. All rights reserved. #
#####
-W silentInstallLicenseAcceptance.value="true"
-W esbDetectionPanel_InstallWizardBean.optionSelected="1"
# IBM WebSphere ESB, Installationspfad
-P esbProductBean.installLocation="C:\Program Files\WS-ESB"
# Custom install type
-W setuptypeInstallWizardBean.selectedSetupTypeId="Custom"
# Automatically profile the installation afterwards
-W summaryPanel_InstallWizardBean.launchPCAW="false"
-W pcawResponseFileLocationQueryAction_
InstallWizardBean.fileLocation=". \AutomotiveDomainEsbCfg.txt"
#####
```

To further configure the installation, a profile is necessary. Therein network information, user-names and passwords, or database details are specified. Variations between the different product lines are possible as long as compatibility is ensured. This may, for instance, be the case if one product line prefers an external database server on Oracle and another prefers an internal DB2 instance. Should products of the two product lines be integrated with each other, one of both Enterprise Service Bus instances must be deployed as an additional node of the other. A configuration file for server profiling may appear as follows.

```
#####
# (C) Copyright IBM Corporation 2012. All rights reserved. #
#####
-W profilenamepanelInstallWizardBean.profileName=
"profileStandAlone"
-W profilenamepanelInstallWizardBean.isDefault="false"
-P installLocation="C:\Program Files\IBM\WebSphere\ESB\
profiles\profileStandAlone"
-W nodehostnamepanelInstallWizardBean.hostName="ASB"
-W pctdefaultprofileportspanelInstallWizardBean.
WC_defaulthost="9080"
-W pctdefaultprofileportspanelInstallWizardBean.
WC_adminhost="9060"
-W pctdefaultprofileportspanelInstallWizardBean.
SOAP_CONNECTOR_ADDRESS="8880"
```

```
-W pctdefaultprofileportspanelInstallWizardBean.  
ORB_LISTENER_ADDRESS="9100"  
-W pctdefaultprofileportspanelInstallWizardBean.  
SIB_MQ_ENDPOINT_ADDRESS="5558"  
-W pctdefaultprofileportspanelInstallWizardBean.  
SIB_MQ_ENDPOINT_SECURE_ADDRESS="5578"  
[...]  
-W wbiCEIConfigInputPanelBeanId.database="MY_DB"  
-W wbiCEIDB2ConfigInfoBean.createDatabaseChoice="yes"  
-W wbiCEIDB2ConfigInfoBean.userId="UserID"  
-W wbiCEIDB2ConfigInfoBean.password="Password"  
-W wbiCEIDB2ConfigInfoBean.passwordConfirmation="Password"  
[...]  
-W profiletypepanelInstallWizardBean.selection="default"  
#####
```

Of course, the above configurations are far from complete. The options of the product are manifold and require thorough planning and testing before deploying it as a technical basis for any product. The example is, however, sufficient to illustrate how the business domain layer can take over these activities and reduce setup costs for the underlying product lines. The first core asset of the AIS automotive business domain is thus a standardized and automated installation package for the IBM WebSphere Enterprise Service Bus version 7.0. According to the technology roadmap, this core asset will not be changed or updated (except for bug fixes) until the end of 2013. In addition, it is conceivable to extend the Enterprise Service Bus installation with business domain-specific web services being deployed directly after installation. This may even include web services from neighbouring software product lines to prepare or establish an integration relationship. As the Enterprise Service Bus represents a joint architectural pattern, a web service functional in one software product line will also function in another.

### 5.2.3.2 Domain-Specific Components

For the present example, a messaging business component is envisioned that receives an EDIFACT message, transforms it to a standardized XML document, and transmits it across the automotive service bus (ASB, i.e. the WebSphere Enterprise Service Bus) and sends an acknowledgement message to the initial sender.

As presented above, an EDIFACT message is nothing more than plain text being transferred via a given medium, usually X.400. For this example, an X.400 listener component is required which receives messages from product line external or legacy systems. As such, a component

can't reasonably be divided any further and is used by many different product lines; it is implemented as a global distributed component (see section 4.2.2.1). On its own, it does not represent a complete business component but may be used to build one. Running on a central server listening to incoming messages, it is part of the enterprise resource domain as it does not interface with a user or a user's workspace. From a functional point of view it represents a utility business component.

The second component required is a process controller that manages the whole transaction. Depending on the data received, it decides what to do and which components to call next. If an EDIFACT message is received, the EDIFACT parser, XML serializer, ASB sender, and X.400 sender are invoked according to predefined business rules. The process controller is also implemented as a global distributed component and is part of the enterprise resource domain. Being responsible for the complete transaction, it represents a process business component.

Analyzing and parsing an EDIFACT string is done by an EDIFACT parser component. It separates the message's elements according to the respective standard and stores it as an EDIFACT message component. Both components are implemented as global distributed components which belong to the enterprise resource domain. According to the functional category, the first is a utility business component (parsing the incoming messages), while the second is an entity business component (storing the information). It represents a main business concept, i.e. an EDIFACT message, "on which business processes operate and provide services that support such processes and their business use [...]" (Herzum and Sims, 2000, p. 175).

Once the incoming message is parsed and stored in an entity business component, it is transformed into an XML document. This transformation is done by an XML serializer component which is again a global distributed component belonging to the enterprise resource domain and providing a utility function. As with the parser component above, the result of the serialization is stored in an entity business component representing an automotive service bus message. The message is then forwarded to the automotive service bus by the process controller.



```
    }
    @Id
    Public String getIdentity(){
        Return myIdent;
    }
    public addSegment(String identifier){
        elements.add(new Segment(identifier));
    }
    public addComposite(String sIdent, cName){
        //identify segment with the name sIdent and use its
        //addComposite method to add a new composite with
        //the name cName.
    }
    public addDataElement(String sIdent, cIdent, dIdent,
dValue){
        //identify segment sIdent, identify its composite with
        //the name cIdent and use its addDataElement method to
        //add a data element with name dIdent and value dValue.
    }
}
public class Segment{
    String myIdent;
    ArrayList composites = new ArrayList();
    public Segment(String identifier){
        myIdent = identifier;
    }
    public void addComposite(String identifier){
        composites.add(new Composite(identifier));
    }
    public void addDataElement(String cIdent, dIdent, dValue){
        //identify composite with the name cIdent and use its
        //addDataElement method to add a data element with
        //dIdent and dValue to it.
    }
}
public class Composite{
    String myIdent;
    ArrayList dataElements = new ArrayList();
    public Composite(String identifier){
        myIdent = identifier;
    }
    public void addDataElement(String ident, value){
        dataElements.add(new DataElement(ident, value));
    }
}
public class DataElement{
    String myIdent;
    String myValue;
    public DataElement(String ident, value){
        myIdent = ident;
        myValue = value;
    }
}
```

Further methods of the above example are omitted for brevity. The @Entity annotation marks the class as an entity bean, @Table defines the name of the database table the entity bean is

associated with, and `@Id` defines the primary key for each record (in this example `myIdent` of the `EdifactMessage` class).

The example business component for transforming incoming EDIFACT messages and sending them across the automotive service bus may now be implemented by various software product lines within the business domain. On the one hand it reduces the efforts required to implement such a component at all, on the other hand it ensures compatible translation between the EDIFACT system and the automotive service bus. With more core assets like this being provided by the business domain layer, economies of scope and scale increase.

### 5.2.3.3 Domain-Specific MDE Artefacts

To fully implement the industrialization concept presented in this work, automating the development process must also be possible. As introduced in section 2.4 (model-driven engineering) and 2.5 (prevailing issues and shortcomings) respectively, domain-specific languages, model transformation engines and code generators are not yet mature enough to be fully implemented. ACME Integration Solutions has therefore decided to follow an XML-based approach which allows configuring the features and components of an application and automatically generating a source code skeleton to be further refined where necessary. At this point, it is important to note that manual refinement must not be used to implement unforeseen variability mechanisms. These must be included in the official variability model and represented by the domain-specific language and its generator(s).

For the present example, the DSL representation of the previously introduced EDIFACT2ASB business component shall be presented. It is assumed that the component is fully functional and available as a jar-file, a format to compress, package and deliver all relevant artefacts of a JAVA EE software component. It usually includes an XML deployment descriptor, the java bean classes, remote and home interfaces to address the java bean, a primary key class to uniquely identify data records, as well as dependent classes and interfaces. With reference to Czarnecki and Eisenecker's generative programming approach (see sections 2.4.4 and 4.3.2), this component represents a part of the solution space by providing an answer to a business

problem. Working the way back, the next thing necessary is the configuration knowledge or grammar of the domain-specific language. What are the prerequisites, limitations, and characteristics to be considered when deploying this component, what are legal and illegal feature combinations? Using XML as a domain specific language implies XML schema definitions (XSD) to ensure the correctness of a business concern expressed in the problem space as well as XML stylesheet transformations (XSLT) to transform the description of the business problem (i.e. the XML document) into software. The following points present some of the constraints to be considered in the configuration knowledge:

- The deployment environment must provide a JAVA EE application server
- There must be only one EDIFACT2ASB component in an application
- The component must not be combined with the alternative EDIFACT2Web component

Of course, there will be a large number of additional constraints, dependencies, or limitations. For the present example, however, the above should sufficiently illustrate the concept. Figure 5-11 (see page 212) presents a simple implementation of the above requirements in the form of an XML schema definition file.

The specification of a given application must define the system environment it runs on within an attribute. The components being deployed may then be checked as to whether their requirements match the system environment or not. This happens in the `xs:assert` statements. The environment attribute of the application is compared to the `requiredEnvironment` attribute of the respective child elements, which in this case are `BusinessComponent`, `EDIFACT2ASB` and `EDIFACT2Web`. The application may furthermore contain an unlimited amount of business components and in addition to these either one `EDIFACT2ASB` or one `EDIFACT2Web` component, but not both or more than one. This limitation satisfies the second and third criteria. A simple XSD representation would appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xs:element name="Application">
```

```
<xs:complexType>
  <xs:sequence>
    <xs:element ref="BusinessComponent"
      maxOccurs="unbounded"/>
    <!-- begin second and third constraint -->
    <xs:choice>
      <xs:element ref="EDIFACT2ASB"/>
      <xs:element ref="EDIFACT2Web"/>
    </xs:choice>
    <!-- end second and third constraint -->
  </xs:sequence>
  <!-- definition of application environment -->
  <xs:attribute name="environment"/>
</xs:complexType>
<!-- begin first constraint -->
<xs:assert test="@environment eq
  root()/BusinessComponent/requiredEnvironment"/>
<xs:assert test="@environment eq
  root()/EDIFACT2ASB/requiredEnvironment"/>
<xs:assert test="@environment eq
  root()/EDIFACT2Web/requiredEnvironment"/>
<!-- end first constraint -->
</xs:element>

<xs:element name="BusinessComponent">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="BusinessLogic"
        maxOccurs="unbounded"/>
      <xs:element name="Interface"
        maxOccurs="unbounded"/>
    </xs:sequence>
    <!-- definition of component requirement -->
    <xs:attribute name="requiredEnvironment"/>
  </xs:complexType>
</xs:element>

<xs:element name="EDIFACT2ASB">
  <xs:complexType>
    <xs:all>
      <xs:element name="RX_Interface" type="xs:string"/>
      <xs:element name="TX_Interface" type="xs:string"/>
    </xs:all>
    <!-- definition of component requirement -->
    <xs:attribute name="requiredEnvironment"/>
  </xs:complexType>
</xs:element>

<xs:element name="EDIFACT2Web">
  <xs:complexType>
    <xs:all>
      <xs:element name="RX_Interface" type="xs:string"/>
      <xs:element name="TX_Interface" type="xs:string"/>
    </xs:all>
    <!-- definition of component requirement -->
    <xs:attribute name="requiredEnvironment"/>
  </xs:complexType>
</xs:element>
```

</xs:schema>

Normally the first criteria, ensuring the component is deployed to the correct environment, would not have been possible or have required a significant redesign of the overall schema, leading to high redundancy of information. Parts of above XSD only recently became available in the World Wide Web Consortium Schema definition version 1.1, released in April 2012 (World Wide Web Consortium, 2012). Among other aspects, version 1.1 now also allows the implementation of assertions and conditions. Both are indispensable when defining even a simple grammar as in this example. Thus for the more complex criteria three assertions were used and associated with the EDIFACT2ASB, EDIFACT2Web and other business components.

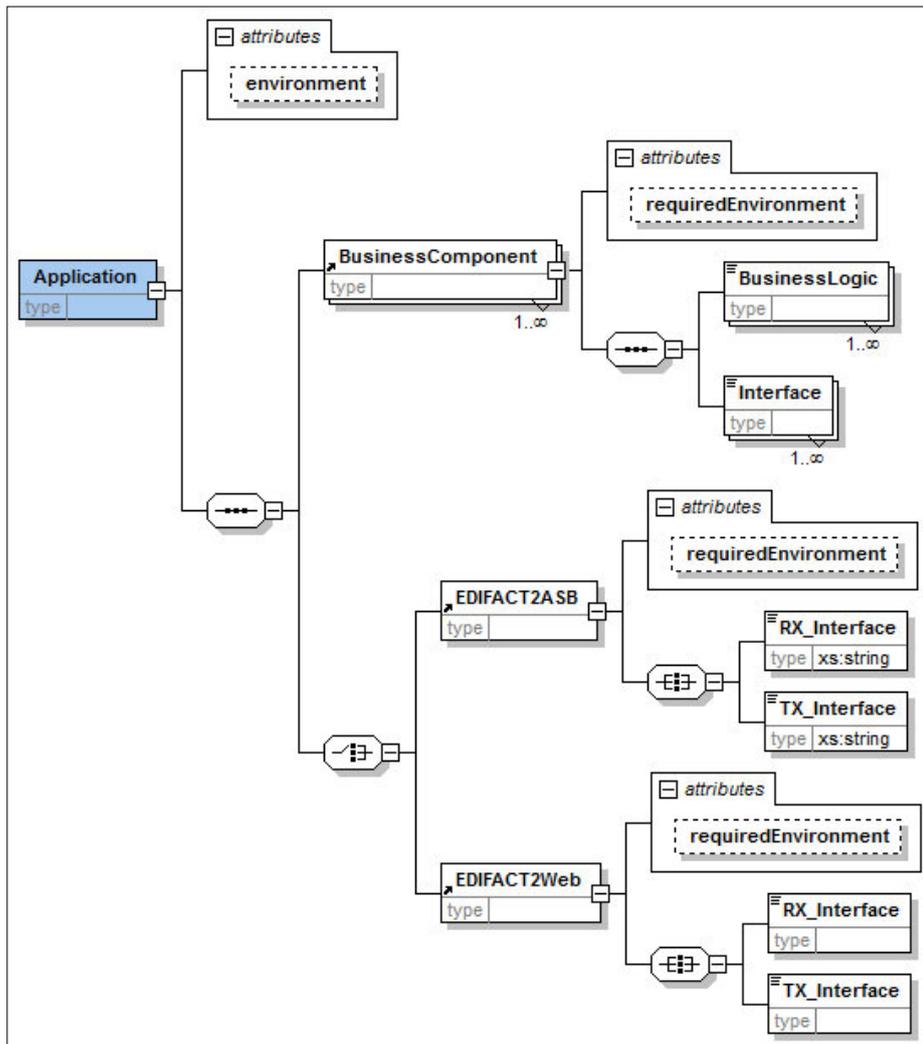


Figure 5-11: XML schema example<sup>4</sup>

The above definition defines how the different elements of an application may be deployed and which constraints have to be considered, i.e. the grammar of the domain specific language. According to this grammar, it is now possible to specify the business domain-specific parts of an application, which could appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Application environment="JAVA EE"
  xsi:noNamespaceSchemaLocation="BD_Requirements.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <BusinessComponent requiredEnvironment="JAVA EE">
    <BusinessLogic>component_a</BusinessLogic>
    <BusinessLogic>component_b</BusinessLogic>
    <BusinessLogic>component_c</BusinessLogic>
  </BusinessComponent>
</Application>
```

<sup>4</sup> Assertions not represented due to missing tool support for XSD 1.1 at the time of writing

```
<Interface>text</Interface>
<Interface>text</Interface>
</BusinessComponent>
<BusinessComponent requiredEnvironment="JAVA EE">
  <BusinessLogic>component_1</BusinessLogic>
  <BusinessLogic>component_2</BusinessLogic>
  <BusinessLogic>component_3</BusinessLogic>
  <Interface>databaseConnector</Interface>
</BusinessComponent>
<EDIFACT2ASB requiredEnvironment="JAVA EE">
  <RX_Interface>X400</RX_Interface>
  <TX_Interface>AutomotiveServiceBus</TX_Interface>
</EDIFACT2ASB>
</Application>
```

With the definition of the XML schema as the grammar and the XML file (partially) describing the application to be developed, a formal specification is available. A model transformation engine or code generator may now transform the model into a lower level one or directly generate executable artefacts from it. The latter does not necessarily imply source code; it may also mean generating deployment scripts or assembling parts of an application from pre-defined components such as jar-files. Such a generator is available with extensible stylesheet language transformations (XSLT), especially if the transformation is not too complex as it is the case when assembling an application from predefined components.

The initial objective of extensible stylesheets was to produce different result documents from an xml file and format them according to predefined rules (Vonhoegen, 2011, p. 243). A common example is the transformation of an xml source document containing a bill of materials, for instance, into an html file made available through a reporting website. Such a transformation always includes three parts: the xml document containing the information to be presented, an xslt stylesheet containing the transformation rules and an xslt processor applying the transformation rules to the source and creating the resulting document (Vonhoegen, 2011, p. 245). The resulting document can be any text document. The concept thus offers two possibilities for model-driven engineering: First, xml based models may easily be refined and transformed to a lower level xml model. Second, executable artefacts such as source code or deployment scripts may be derived from xml-based models.

For the present example, the transformation of the EDIFACT2ASB component is shown. According to the application model presented as an XML file above, the EDIFACT2ASB com-

ponent has an X.400 receiver interface and an automotive service bus sender interface. As these could also be anything else depending on a particular customer's requirements, they must be generated dynamically. The following XSLT file makes this possible. Blue parts are the actual Java source code being generated, while black parts indicate the XSLT rules necessary to transform the XML application model.

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0" xmlns:java="http://xml.apache.org/xslt/java"
  exclude-result-prefixes="java">
<xsl:output method = "text"/>
<xsl:template match="Application">
public class MainClass{
    public static void main(String[] args){
        System.out.println("&quot;Inside main method&quot;");
        <xsl:apply-templates select="BusinessComponent"/>
        EDIFACT2ASB myTransformer = new EDIFACT2ASB();
        myTransformer.send();
        myTransformer.receive();
    }
}
<xsl:apply-templates select="EDIFACT2ASB"/>
<xsl:apply-templates select="EDIFACT2Web"/>
</xsl:template>

<xsl:template match="BusinessComponent">
    System.out.println("&quot;Running <xsl:value-of
select="@componentName"/>&quot;");
</xsl:template>

<xsl:template match="EDIFACT2ASB">
public class EDIFACT2ASB {
    public EDIFACT2ASB() {
        //constructor goes here
    }
    <xsl:apply-templates select="RX_Interface"/>
    <xsl:apply-templates select="TX_Interface"/>
}
</xsl:template>

<xsl:template match="RX_Interface">
    public void receive_<xsl:value-of select="."/>() {
        System.out.println("&quot;Receiving via <xsl:value-of
select="."/>&quot;");
    }
</xsl:template>

<xsl:template match="TX_Interface">
    public void send_<xsl:value-of select="."/>() {
        System.out.println("&quot;Sending via <xsl:value-of
select="."/>&quot;");
    }
}
```

```
</xsl:template>  
</xsl:stylesheet>
```

The example does not contain anything about X400 or AutomotiveServiceBus being the sending and receiving interfaces of the EDIFACT2ASB component; neither does it contain information about the number or names of other business components. Together with the XML model of the application, however, the result appears as follows:

```
public class MainClass{  
    public static void main(String[] args){  
        System.out.println("Inside main method");  
        System.out.println("Running Business Component 1");  
        System.out.println("Running Business Component 2");  
  
        EDIFACT2ASB myTransformer = new EDIFACT2ASB();  
        myTransformer.send();  
        myTransformer.receive();  
    }  
}  
  
public class EDIFACT2ASB {  
    public EDIFACT2ASB() {  
        //constructor goes here  
    }  
    public void receive_X400() {  
        System.out.println("Receiving via X400");  
    }  
    public void send_AutomotiveServiceBus() {  
        System.out.println("Sending via AutomotiveServiceBus");  
    }  
}
```

By replacing the source code of the component definitions in the XSLT file, one can easily implement a generator for a large number of different components or applications. Adding additional attributes and elements allows further specializing components and expanding their potential of reuse. In addition, it is possible to write several XSLT files for the same XML application model. This may be useful if different output files are necessary, such as remote and home interfaces, or bean classes, for instance. Additional XSLT files may also be used to provide a component in another programming language such as Microsoft .Net instead of Java without changing the model. Other possibilities are setup scripts for the application server or infrastructural artefacts.

## 5.3 Software Product Line Engineering

In the previous section, the business domain layer of the ACME Integration Solutions automotive business domain was developed. It includes a product strategy and portfolio, a common architecture, reusable components, as well as domain-specific language elements and a code generator. Some of these assets were defined compulsory for the underlying software product lines, others are optional. Exploiting these assets reduces the initial set-up cost of the actual software product line. What remains to be done is explained in the following sections.

### 5.3.1 Product Line Requirements Engineering

During business-domain development, the functional scope of typical supply chain management solutions was identified. It is now up to the software product line layer to decide which functional parts of such a system are the most promising from an economic point of view. This decision is based on the efforts required to implement their core assets and the potential number of products to be produced in which these core assets can be reused. A complex but constantly demanded production planning module may thus provide a better return on investment than a comparably simple distribution planning requested by only a very small number of customers. Additionally, the maturity and stability of the products and their functionality is important (Linden, 2007, p. 33). A product line constantly adapting to market changes won't be very efficient. Based on several market studies conducted during earlier phases and experience in the field, ACME Integration Solutions decided to implement the following modules of a supply chain management solution within a software product line:

- Material requirements planning
- Network planning
- Strategic network planning
- Demand planning
- Master planning
- Production planning

The remaining functions, which are scheduling, distribution planning, transport planning and demand fulfilment, will be produced outside the product line upon specific customer request. The expected market demand does not justify their setup within a software product line.

For the above six functional modules, the software product line layer inherits the generic product and technology roadmap as well as functional and non functional assets. It may now further define the features of each within a detailed variability model. For the present example, the order creation function of the material requirements planning module was chosen. As presented in Figure 5-12, order creation may either occur via a World Wide Web Consortium-compatible web service or an EDIFACT message. The former is a mandatory feature and will be available in every product developed. The latter is optional and may be chosen by the customer. EDIFACT-based order creation will furthermore be subdivided into the conventional standard and a novel XML/EDIFACT standard (United Nations Economic Commission for Europe, 2006), gaining more and more acceptance in the industry. Web services and conventional EDIFACT support revert to core assets provided by the business domain layer. The support for XML/EDIFACT is an independent implementation developed by the supply chain management product line.

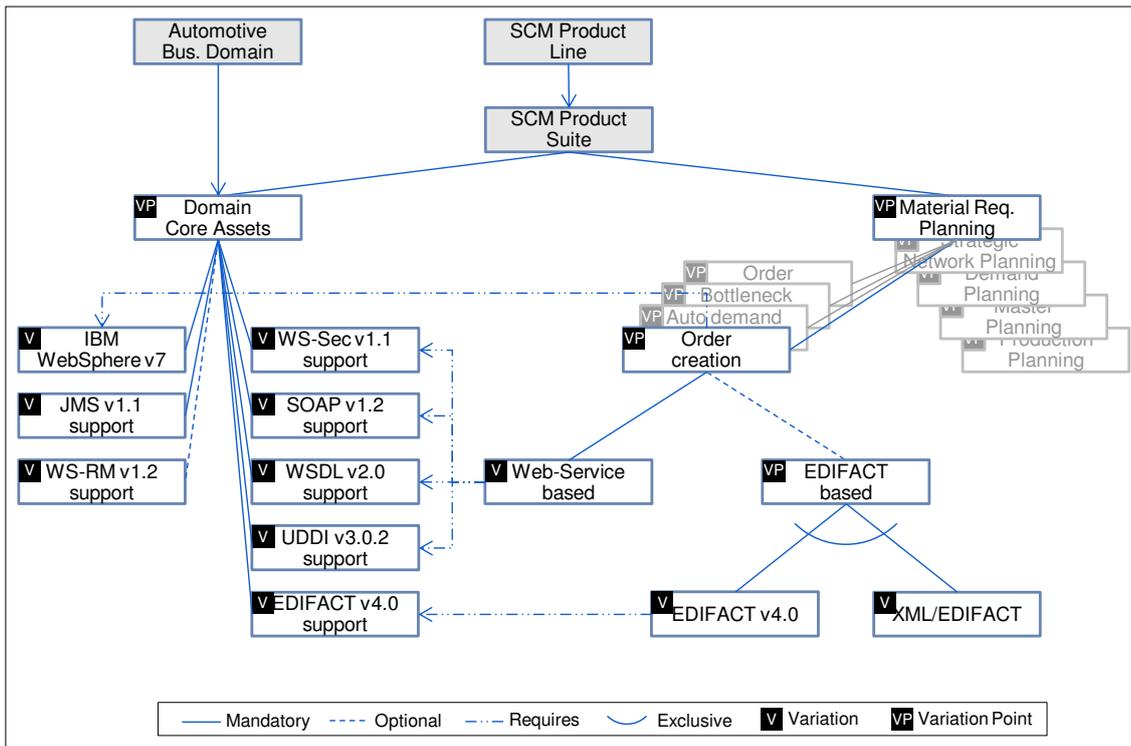


Figure 5-12: Partial variability model of the supply chain management product line

The variability model will now be further developed into a detailed feature model describing external (i.e. visible), internal (i.e. functional) and non-functional (i.e. quality or performance) features. It contains all the requirements for all current and prospective future applications (Linden, 2007, p. 49) as far as foreseeable. The feature model thereby orientates itself on the technical roadmap defining the technologies to be used in the business domain. Product line requirements engineering can be compared to the requirements engineering of traditional software development processes and results in a precise description of what needs to be done.

An actual implementation outside of this example will certainly define many more features, dependencies, and constraints. User interface mock ups would be built and precise definitions of data structures, performance, or reliability defined. Similar decisions have to be made for the remaining functions and functional modules. The subsequent processes architecture design and development and core asset development pick up these requirements and develop the product line's structure and building blocks from which the final products will be built.

### 5.3.2 Architecture Design and Development

Within the limitations of the automotive service bus set forth by the business domain layer, the architecture design and development process defines the internal structure of the products being built within the software product line. It takes the previously defined requirements as input and creates a reference architecture for the supply chain management platform (Linden, 2007, p. 51). The architecture is assumed to be stable and to merely change slightly over time. It therefore has to take the following requirements into account (see Clements et al., 2007, pp. 57 f.):

- The quality attributes for all features and variations as defined in the product line requirements engineering process.
- The systems' interactions and integration relationships with other systems, including the requirements implied by the business domain layer (e.g. EDIFACT support and Automotive Service Bus).
- The technical roadmap specified by the business domain layer, as well as internal roadmaps defining technologies to be used by the software product line exclusively (i.e. those relevant to supply chain management products only).
- Business goals of the software product line and the overall organization, in this case the ACME Integration Solutions' wish to propagate their Automotive Service Bus in the industry and benefit from potential follow-up business.
- The availability of components, i.e. using already-existing knowledge from within the organization, buying commercial ones, or deriving them from legacy systems.

To implement the actual architecture, three commonly-used paths are available (Clements et al., 2007, p. 60; Pohl et al., 2005, pp. 119 ff.). These are a top-down approach starting from the functional requirements and iteratively developing the architecture until a completely functioning system is available; an infrastructural approach focusing on those portions without which the required functionality cannot function; and a functional approach implementing the different features and variations in software components while being certain that the required infrastructure can be subsequently implemented without issues. In the present example, ACME Integra-

tion Solutions chose to follow an infrastructure-oriented approach (Clements et al., 2007, p. 60). It includes defining the operating system, communication protocols, middleware (in this case inherited from the business domain layer), component frameworks or system-specific support functionality. “This approach results in a system in which the application functionality can be implemented incrementally since even the smallest function will run” (Clements et al., 2007, p. 60). It thereby alleviates one of the major concerns of industrialized systems integration, i.e. high upfront investments from implementing the core assets before software production starts. In this case, once the infrastructure of the product line is in place, core assets may be developed on the fly during production of an actual application.

An architecture may be implemented following many different patterns (Conrad et al., 2006, pp. 228–229). Architectural patterns provide a building plan similar to templates, including a basic system structure, subsystems, behaviour, and relationships between the different elements involved (Linthicum, 2000, p. 109). Underneath architectural patterns, design patterns can be found, providing means to define components and subsystems independently from the implementation domain and technology (Linthicum, 2000, p. 109). In the present example, the automotive service bus was specified by the business domain layer. It will furthermore follow the integration metamodel, specifying the overall integration structure. The design pattern was incurred from the business-component approach which will be used for component-based development. For the present illustration, the component architecture of above EDIFACT v4.0 order creation feature is shown. It incorporates global distributed components from the business domain layer as well as local distributed and business components from the software product line layer.

According to the variability model of the software product line, an order may be created by a web service, an EDIFACT v4.0 message, and an XML/EDIFACT message. The product line designer has to decide how to implement these features while maintaining the variability model. Basically three techniques are available (see Linden, 2007, p. 40): The adaptation technique takes a single implementation with all features available and adjusts the interfaces accordingly, i.e. the component is configured to the specific requirements. In the replacement technique,

several implementations representing the desired feature combinations are available. During product development, one of these implementations is chosen or a new one developed. The extension technique builds on generic interfaces in the application architecture. In contrast to replacement, the interfaces do not specify the component's properties and interfaces may even be used for different components. In the present example, the adaptation technique was chosen. The order creation business component contains all possible features whereas the exact choice and thus the configuration of the component for the actual system depends on a particular customer's requirements.

Besides defining a component's design patterns and features, it is important to specify its technical and semantic interfaces. A technical interface includes functional signatures of methods and their parameters to be invoked as well as a definition of the required runtime environment (Linthicum, 2000, p. 61). What is also necessary is a description of a component's semantics, performance, quality, resource requirements, exceptions raised, or any other externally-observable behaviour (Szyperski et al., 2002, p. 54; Linthicum, 2000, p. 61). Although recently being researched (Heymans et al., 2011, pp. 170–172; Owe et al., 2007), a formal and broadly applicable specification of such contracts is not yet available. In the present example, it is therefore specified in a structured, human readable document included in the component specification.

The final architectural decision to be taken is the selection of a component model enabling component communication and coordination (Linthicum, 2000, p. 62). They range from simple inter process communication with procedure calls and shared memory to complex component models such as JavaBeans. In the experimental implementation at hand, the supply chain management product line is free to choose its own component model as long as it supports the constraints imposed by the business domain layer. Global distributed components, such as the EDI-FACT2ASB component introduced in the previous section, are provided in different technologies (see several XSL transformations in 5.2.3.3) and thus allow the choice of other component models for the product line. To illustrate extension technologies between the business domain

layer and the product line layer, the present example adheres to JAVA EE as implementation and component technology.

### **5.3.3 Core Asset Development**

Now that business domain and software product line layer with their respective artefacts are defined, the core assets required for actual product development must be implemented. Some core assets can be obtained from higher layers (e.g. a middleware connector), some must be extended (e.g. an entity component defining a purchase order), and others must be developed from scratch (e.g. an interface to a customer specific legacy system). For the present example, the order creation component is presented. It reuses several distributed components provided by the business domain layer and develops local ones to implement the new business component. The necessary changes include the implementation of the new components (i.e. their Java code, for instance), an XML model of the new business component, an XSL document representing the new grammar, as well as an XSLT-based code generator that extends the functionality of the one provided by the business domain layer.

#### **5.3.3.1 Product Line Specific Components**

The OrderCreation business component consists of six distributed components, of which two are obtained from the business domain layer. These are the X.400 interface and the EDIFACT Parser. In the following figure, these global distributed components are marked in light blue. The web service interface, process controller, purchase order and order management system interface are local distributed components (local distributed components) as they provide product line-specific services. A new order is received either via an external web service client or an EDIFACT message. The process controller receives the payload and creates a purchase order from it which is then sent to the order management system via the respective interface. The process controller also feeds back success or failure to the initial requester of the order. As with the global distributed components, the web service interface, purchase order and material requirement management system components are generic enough to be reused in other not directly

related functions of the product line. The component distribution of the present example illustrates how effective modularization allows maximising component reuse.

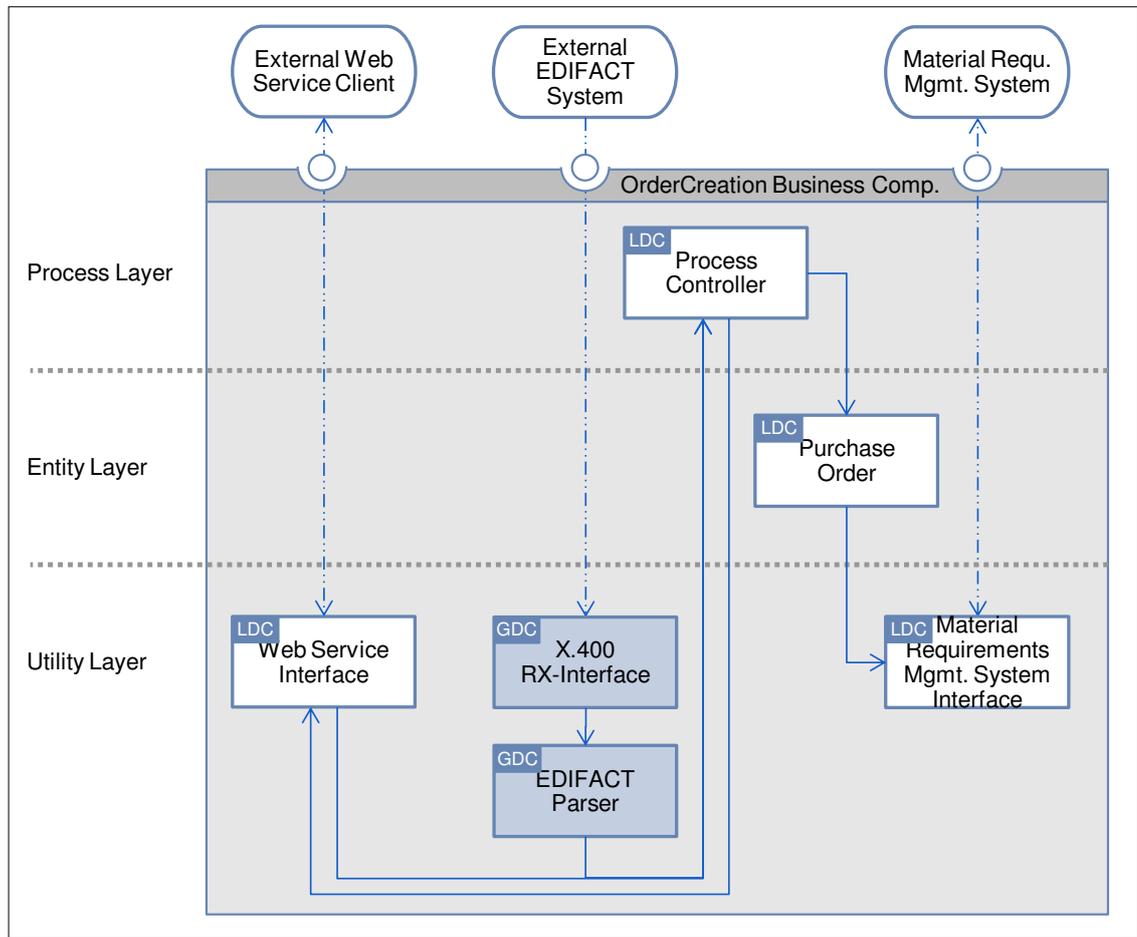


Figure 5-13: Example of the OrderCreation Business Component

Besides the actual implementation, it is important to provide a context-free specification of the interfaces, the component's pre and post conditions, as well as quality and performance characteristics. In the present example, the specification of the web service in the form of a web service description language file would appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns=" http://scm.asb.ais.com/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://scm.asb.ais.com/"
  name="CreateOrderService">

  <message name="order">
    <part name="productID" type="xsd:string"></part>
```

```
<part name="amount" type="xsd:int"></part>
</message>

<message name="orderResult">
  <part name="status" type="xsd:string"></part>
</message>

<portType name="CreateOrder">
  <operation name="order" parameterOrder="productID
    amount">
    <input message="tns:order"></input>
    <output message="tns:orderResult"></output>
  </operation>
</portType>
</definitions>
```

The component provides the CreateOrder interface which requires a productID as a string value and the amount required as an integer. Upon web service completion, a simple return string indicating success or failure is sent. A java implementation of the interface as a web service is also very simple and would appear as follows:

```
Package com.ais.asb.scm.OrderCreation;

import com.ais.asb.scm.ProcessController;
import javax.jws.WebService;
import javax.jws.soap.SoapBinding;
import javax.xml.ws.Endpoint;

@WebService
@SOAPBinding(style=Style.RPC)

public class CreateOrder {
  private ProcessController controller;

  public void CreateOrder() {
    //create a new process controller component
    //to handle incoming orders (illustrated)
    //and specify where to send them.
    controller = new ProcessController();
  };

  public string order(string orderID, int amount) {
    controller.createOrder(orderID, amount);
    while(controller.status() = "processing"){
      //implement a watchdog or something
      //different to comply with the contract
    }
    Return controller.status();
  }
}
```

```
public class Server {
    public static void main(String args[]){
        CreateOrder c_order = new CreateOrder();
        Endpoint site = Endpoint.publish
            ("http://localhost:1234/CreateOrder", c_order);
    }
}
```

Model-driven engineering techniques may now be used to adapt the interface's properties to the particular customer requirements, e.g. specifying the endpoint of the web service, which will be shown in the next section. Of course, several other characteristics of the component must be specified. Despite ongoing research in the field (see Feng et al., 2012; Sentilles, 2012; Tibermacine et al., 2011; Orbán and Kozma, 2012), no suitable approach in terms of simplicity and economic viability was found. Most of them implement their own proprietary language or toolset which is contradictory to a simple, lean and future-proof approach pursued in this work. The present example thus relies on predefined documents mandatory for all components which are to be interpreted by the software developers.

Similar specifications, although in a different form, must be defined for all other interfaces. At this point, the application architecture becomes essential. As described in the previous section, it is important to have an upfront technical and semantic specification of the interfaces, i.e. the data and the data format that will be transferred must be clear before component implementation. In the present example, this applies to the orderID, amount, and orderResult attributes. In the same way as above, the software product line now develops the remaining business components. The collection of these represents the basis from which application production starts to assemble customer-specific requirements.

In addition to reusable software components, the software product line layer also has to provide an appropriate infrastructure, i.e. a component framework. In the case at hand, the IBM WebSphere Enterprise Service Bus provided by the business domain layer is specified. It provides the required infrastructure for web services as well as enterprise java beans and thus all components of the present example. The exact version and potential upgrades can be obtained from the technology roadmap set forth by the business domain layer.

After specifying architecture, infrastructure and reusable components, a development process must be defined. As the software product line layer is concerned with the development of reusable components instead of complete systems, the rapid component development process is defined. It is a process “for thinking about, building, and testing an individual business component” (Herzum and Sims, 2000, p. 247). Building and testing a complete application consisting of several different business components representing a specific customer’s needs is done within the system architecture and assembly process. This process is conducted on the production layer and described in section 5.4. However, for an enterprise setting up industrialized systems integration, it may make sense to combine the two during the first few products into rapid system development to alleviate upfront investments to develop reusable software assets. “In this process, the business component system and its individual business components are, as far as possible, built concurrently” (Herzum and Sims, 2000, p. 252). Figure 5-14 illustrates the overall process.

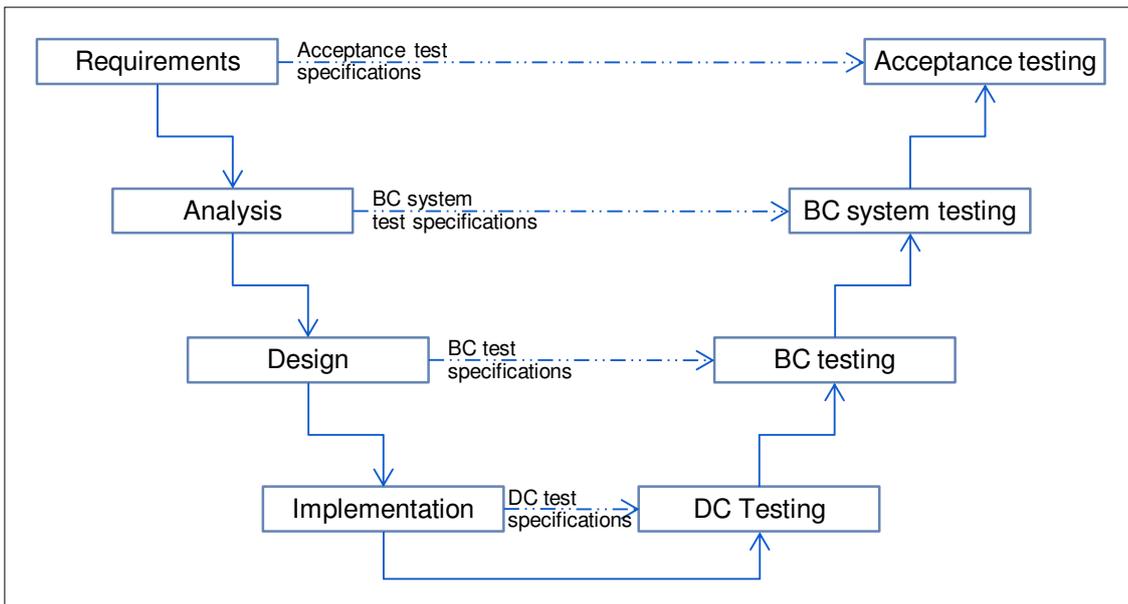


Figure 5-14: Rapid System Development process (Herzum and Sims, 2000, p. 255)

The software product line now has to define the constraints and deliverables of each process step. For requirements engineering, these would be a feature list, market requirements, an initial business process model of the system, a set of use cases, user interface prototypes, an initial business component model describing the information flows, and a test specification for

acceptance testing (Herzum and Sims, 2000, pp. 255–257). From the design phase, for instance, the software product line would require an external specification of the business components including a detailed description of their interfaces, the dependencies of the business components, their internal specifications and component models specifying the distributed components utilized in a business component (Herzum and Sims, 2000, pp. 264–268). Similar requirements must be defined for all other process steps as well.

### 5.3.3.2 Product Line Specific MDE Artefacts

Core asset development at the software product line layer can be subdivided into three types relevant to model-driven engineering. These are the creation of new components, the inclusion of already existing components in new ones and the extension and restriction of already-existing components. The following sections explain each type in further detail and present an actual example.

For those distributed and business components developed from scratch, such as the previously introduced order creation business component, the respective domain-specific language elements as well as their code generation artefacts must be created. As already known from the business domain layer, the grammar is represented by an XML schema definition and the code generator by an XSLT document. The schema definition itself differs only from a functional point of view and is therefore omitted at this place. New components will be provided with their schema definition (which defines how they can be integrated), one or more code generators (which translate the model into implementation languages, or create deployment scripts) and other required resources such as binary artefacts. If components provided by the business domain layer or new ones are to be included in the software product line, their schemas must be imported into the overall model of the application. Ambiguities can be avoided by using structured namespaces. For xsd files, this is done via the following statement:

```
<xs:import namespace="myBusinessComponent"  
  schemaLocation="myBusinessComponent.xsd"/>
```

For intra-namespace solutions `<xs:include.../>` may also be used. Including code generators from external components works in a similar way:

```
<xsl:include href="myBusinessComponentGenerator.xsl"/>
```

Above statement includes XSL templates at the exact position of the statement as if they were written precisely at this position. In this case, the first matching template for an element is applied. Subsequent matches will not be executed. An alternative to the `include` element is `<xsl:import.../>`, which must be positioned as a top-level element within the document. The precedence of the imported templates depends on the order of import with the original stylesheet document having the highest priority. As inclusions are actually replacements, they also precede imports.

Components may also extend existing ones, which often is the case for data entities. A purchase order entity of a shop floor system may, for instance, include item, order ID, amount and due date. For the supply chain management system, payment conditions and supplier rating are additionally required. It thus makes sense to specify the universal part of the component at the business domain layer, while the software product lines add their specific elements. The following example defines item name, order ID, amount, and price as the generic attributes available across all automotive product lines. In the shop floor product line, the due date is added and in the supply chain management product line payment conditions and supplier rating attributes are added. To extend a generic component, the extension element of XML schema definitions is used. It will be based on the following generic definition of a purchase order within the business domain layer.

```
<xs:complexType name="PurchaseOrderType">
  <xs:sequence>
    <xs:element name="itemName" type="xs:string"/>
    <xs:element name="orderID" type="xs:string"/>
    <xs:element name="amount" type="xs:int"/>
    <xs:element name="price" type="xs:decimal"/>
  </xs:sequence>
</xs:complexType>
```

For the shop floor system, the generic purchase order type is imported and then extended with the product line specific attributes.

```
<xsl:include href="BD_PurchaseOrderType.xsl"/>
<xs:complexType name="SF_PurchaseOrder">
  <xs:complexContent>
    <xs:extension base="PurchaseOrderType">
      <xs:sequence>
        <xs:element name="dueDate" type="xs:date"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

The same principle is applied within the supply chain management product line.

```
<xsl:include href="BD_PurchaseOrder.xsl"/>
<xs:complexType name="SCM_PurchaseOrder">
  <xs:complexContent>
    <xs:extension base="PurchaseOrderType">
      <xs:sequence>
        <xs:element name="payCond" type="xs:string"/>
        <xs:element name="rating" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

A valid XML document describing an supply chain management purchase order would appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<PurchaseOrder xsi:noNamespaceSchemaLocation=
  "SPL-PurchaseOrderExample.xsd" xmlns:xsi="http://www.w3.org/
  2001/XMLSchema-instance">
  <CreationDate>2012-08-03</CreationDate>
  <PODetails xsi:type="SCM_PurchaseOrder">
    <itemName>Engine</itemName>
    <orderID>ENG-ACK-01</orderID>
    <amount>1</amount>
    <price>1350.00</price>
    <payCond>myConditions</payCond>
    <rating>Excellent</rating>
  </PODetails>
</PurchaseOrder>
```

After extending the grammar of the domain specific language, i.e. the XML schema definitions, code generators must also be extended. The existing ones provided with the initial definition of the purchase order may no longer be used as they do not know the extended elements (see

below). For brevity reasons, only a very basic implementation of the abstract PurchaseOrder class is shown:

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0" xmlns:java="http://xml.apache.org/xslt/java"
  exclude-result-prefixes="java"
  xsi:noNamespaceSchemaLocation="SPL-PurchaseOrderExample.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<xsl:output method = "text"/>

<xsl:template match="PurchaseOrder">
abstract class PurchaseOrder {
  //defines business domain wide variables
  private String itemName, orderID;
  private int amount;
  private double price;
  //defines a business domain wide method
  public abstract String getPOType();
  //other functionality such as abstract getter
  //and setter methods would go here.
}
<xsl:apply-templates select="PODetails"/>
</xsl:template>
```

The above example will implement an abstract Java class containing the requirements to be inherited by the business domain layer. These are the features that allow for a seamless integration across different product lines. A product line actually implementing a purchase order has to provide its own definition of purchase order details which is executed by the above statement `<xsl:apply-templates select="PODetails"/>`. Each software product line now defines its own XSLT file with a `PODetails` template. For the Supply Chain Management Systems, its definition appears as follows:

```
<xsl:template match="PODetails
[@xsi:type='SCM_PurchaseOrder']">
public class SCM_PurchaseOrder extends PurchaseOrder {
  //extends business domain specification
  //with product line specific variables
  private String payCond;
  private String rating;
  //instantiates abstract method required
  //by the business domain layer
  public PurchaseOrder getPOType(){
    return "SCM_PurchaseOrder";
  }
  //other product line specific
  //functionality would go here.
}
```

```
</xsl:template>
```

The `PODetails` section may either be included in the initial file or imported/included to keep the sources separate between business domain and product line layer. Executing the code generator will lead to the actual java source file:

```
abstract class PurchaseOrder {
    //defines business domain wide variables
    private String itemName, orderID;
    private int amount;
    private double price;
    //defines a business domain wide method
    public abstract String getPOType();
    //other functionality such as abstract getter
    //and setter methods would go here.
}

public class SCM_PurchaseOrder extends PurchaseOrder {
    //extends business domain specification
    //with product line specific variables
    private String payCond;
    private String rating;
    //instantiates abstract method required
    //by the business domain layer
    public PurchaseOrder getPOType() {
        return "SCM_PurchaseOrder";
    }
    //other product line specific
    //functionality would go here.
}
```

Concluding the above, XML schema definitions can be used to specify and extend the grammar of a domain-specific language. The inclusion and extension of predefined language elements from the business domain layer ensure a semantic integrity of the software product lines among each other. A purchase order sent across the automotive service bus from a supply chain management system may be recognized by an order monitoring tool without the need of additional or adapted interfaces. It doesn't even need to know the specification of the supply chain management product line, as it can rely on the attributes defined by the business domain layer. Furthermore, application development may use predefined components, interfaces and code generators without the need of additional alignment. It is just important that all systems adhere to the definitions set forth by the business domain layer. XSL-based transformations allow for the specification of rules and regulations on how to transform a valid application model, i.e. an XML file, into source code or additional resources. It therefore contains several elements known

from programming languages, such as if-then-else constructs or loops (Vonhoegen, 2011, p. 248) which allow the implementation of simple code generators consisting of generators and cartridges (see section 2.4.3). The generator itself is the XSLT processor, while the cartridge is represented by the XSLT files.

For the example in point, it is assumed that the software product line for supply chain management systems has defined a domain-specific language describing all reusable software components (i.e. local distributed components and local business components), glue code or component infrastructure, additional resources such as database installation scripts, and appropriate code generators. From this listing it can be seen that the upfront investment must not be ignored during product line setup. However, as previously indicated, it can be alleviated by providing only the fundamental functionality and implement the actual features only as they are requested by a particular customer. One must, of course, resist allocating the complete development cost to the customer first asking for a feature and instead see them as an investment into the product line as a whole.

## 5.4 Product Development

Given that business domain and software product line layer have defined a domain-specific language and code generators able to at least represent a particular product's infrastructure, development may begin. The development process itself does not differ much from traditional software development (e.g. RUP or similar process models IBM, 2007, p. 3) and usually starts with a requirements specification and application analysis and design. It continues with application realisation and concludes with application testing (Linden, 2007, pp. 53–54; Pohl et al., 2005, pp. 31-34, 303 ff.) and deployment. The most significant difference with regard to software product lines lies in a two-staged approach, i.e. the process steps occur on both layers, although with a different focus.

### 5.4.1 Requirements Definition and Application Design

For the example at hand, it is assumed that a customer is asking for a material requirements management system to be integrated into his pre-existing supply chain landscape, primarily the shop floor system creating large numbers of purchase orders via a proprietary interface. He also states that in future projects some of the already existing systems may be replaced. An application requirements engineer discusses the customer's expectations based on the variability model of the software product line shown in Figure 5-12. The objective here is to align the customer's requirements to the variability model as far as possible to leverage economies of scope on both sides – customer and supplier. Subsequently, “the gap between what is available and what is required must be analysed, and a trade-off decision taken for each unsatisfied requirement” (Linden, 2007, p. 53). The customer either has the opportunity to adapt his requirements to match one of the product line's features or request an application specific extension at additional implementation cost. However, if the missing feature is assumed to be required by more than just one or two clients, “feedback may be given to the domain engineering lifecycle to achieve the solution of the next version of the platform satisfying this requirement” (Linden, 2007, p. 54). In the beginning of a software product line, the latter will very often be the case, as reusable assets are being developed as they are required in order to prevent high upfront investments. In the present example, the customer has specified several requirements shown in Figure 5-15 (simplified representation). The white boxes represent features defined and developed within the supply chain management software product line. They provide product-specific business functionality and are not to be reused within any other software product line. The blue boxes are features and functionality inherited from the business domain layer. These are available in all software product lines, as they represent generic business concepts and infrastructure suitable not only for supply chain management systems. The orange boxes represent customer-specific functionality which will be developed from scratch. These features do not exist in the supply chain management product line and are available for this particular product only. The customer will hence be charged for the full development cost.

Once the exact customer requirements are known, the particular application design may be developed. As the core asset base already specifies a generic architecture, the application architecture instantiates the former and, in accordance with the variability model, adapts it to the customer's needs. Should additional variation points or modes of variation be required, a decision on their accommodation in the product line's generic architecture must be made (Clements et al., 2007, p. 67). Developing outside of the product line architecture should always be the last resort. To achieve the most from systematic reuse, customer requirements should be matched with the product line's architecture and features, or, if economically feasible, the product line be enhanced accordingly (Clements et al., 2007, p. 114). For the present example, the automotive service bus technical architecture was adopted for the customer's application.

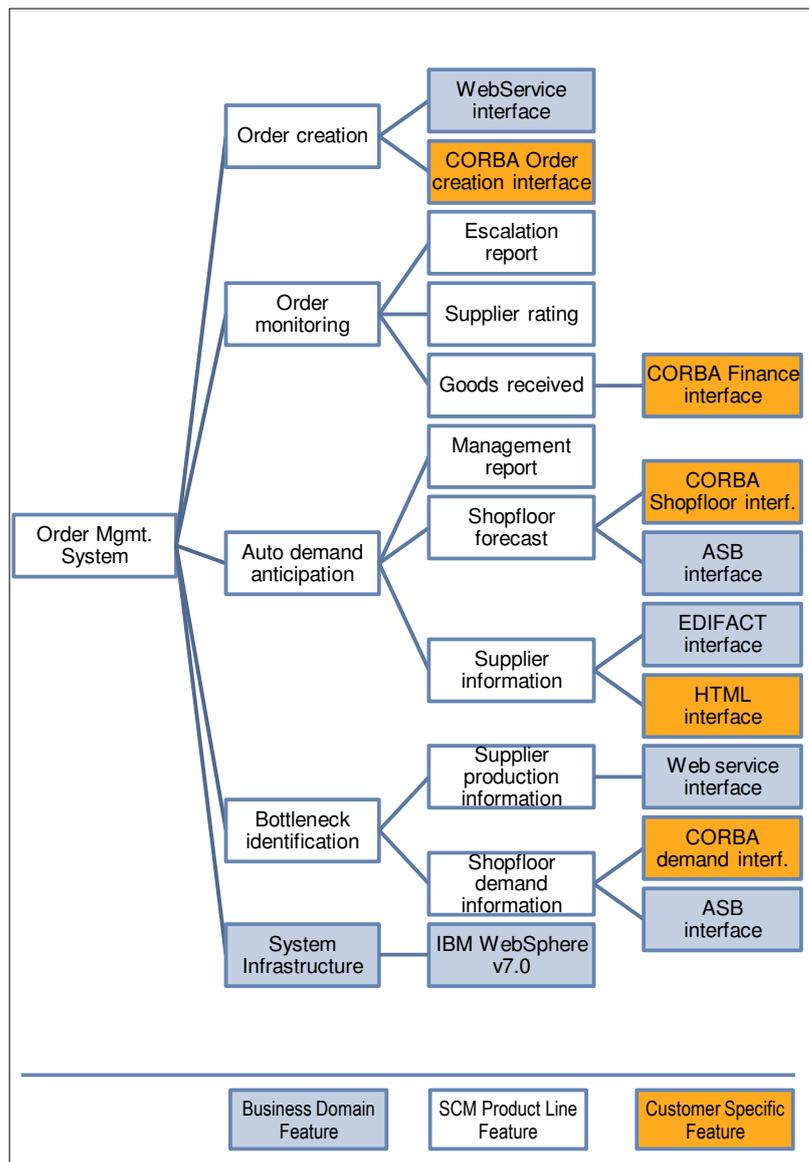


Figure 5-15: Simplified feature model of the example customer product

Besides the feature model and the application architecture, requirements definition and application design also result in stakeholder views, describing the system with regard to different stakeholder interests, several use cases describing user-to-system and system-to-system interaction, and change case models, identifying and capturing anticipated changes, such as component replacements or feature enhancements (Clements et al., 2007, pp. 114 f.). As these do not differ from development artefacts in conventional software development, their presentation is omitted here.

### 5.4.2 Application Realisation and Testing

After defining the application's features and structure, application realisation "takes the common assets of the product line" (Linden, 2007, p. 53) and uses them to implement the application according to a particular customer's requirements. Customer-specific components, marked orange in the present example, are implemented in a conventional way and do not need to adhere to product line guidelines. It may, however, be feasible to utilize already-existing distributed components as discussed in section 4.2.2.1.

Above-mentioned proprietary interface for order creation is assumed to be realized via a CORBA implementation. First of all, it specifies its interface with the interface description language, containing all technical details for communication. This interface definition is used by the external system to create a purchase order within the new system. Subsequently, the implementation of the order-creation feature is associated with an object request broker, which in turn obtains a naming reference from the naming service of the proprietary external system. After updating the client implementation (i.e. the proprietary system of the customer), new purchase orders may be created via the CORBA interface. After an initial data processing, the order details must be processed by the new order management system. Even if the proprietary interface is a customer-specific feature, it does make sense to utilize the purchase order local distributed component from the entity layer shown in Figure 5-13. Order details received via a remote method invocation from the external system are converted into the purchase order Enterprise Java Bean and may then be further processed by other parts of the order management system.

Features provided by the business domain layer, such as the web-service interface or application runtime environment, simply are included into the overall application model represented by several XML files as described in section 5.3.3.2. With regard to initial simplicity of the domain-specific language and the according code generators as well as the maturity of commercially available tools (see section 4.3.2.4 or Selic, 2008, p. 384), the application model in its initial stage will be more like a configuration template. Business domain-wide and software product line-specific features and reusable core assets are represented in the application model with their variability points instantiated. As discussed in the software product line engineering

section, component and feature descriptions are imported or included into the overall application model as are schema definitions to ensure their grammatical integrity. During this process, the XSLT documents are also prepared so that for each component or reusable asset, suitable code generation is available. In this example and given that the industrialization of systems integration is still in its infancy, directly executable code should not be expected or aimed for during product development. Model-driven systems integration should rather be seen as a configuration framework for reusable software components which still require manual refinement and completion – although at a much lower level than with conventional systems integration projects. The following code listings depict short extracts of the model itself, the grammar it is verified against, and the code generator for above order management system example. For reasons of brevity , details of a similar type are omitted and indicated by [...].

The application model was derived by the product development team in accordance with the customer’s requirements. It clearly distinguishes between business domain-specific, software product line-specific and external customer-specific features.

```
<?xml version="1.0" encoding="UTF-8"?>
<Application businessDomain="Automotive" productLine="SCM"
performanceClass="standard" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation
="XSDExample%20for%20production%20layer.xsd">
  <Product name="OrderManagementSystem" version="v1.0">
    <!--Modelling of System Infrastructure starts here-->
    <SystemInfrastructure architecture="v1.0"
environment="J2EE">
      <ApplicationServer>
        <Technology>IBM Websphere</Technology>
        <Version>v7.0</Version>
      </ApplicationServer>
      <Messaging>
        <Technology>WS-RM</Technology>
        <Version>v1.2</Version>
      </Messaging>
    </SystemInfrastructure>
    <!--Modelling of functionality starts here-->
    <ApplicationModules>
      <ApplicationModule type="SPL" id="OrderCreation">
        <Features>
          <Feature type="BD" id="WebServiceInterface"/>
          <Feature type="EXT" id="CorbaOrderCreation"/>
        </Features>
        [...]
      </ApplicationModule>
      <ApplicationModule type="SPL" id="OrderMonitoring">
```

```

    <Features>
      <Feature type="SPL" id="EscalationReport"/>
      <Feature type="SPL" id="SupplierRating"/>
      <FeatureModule type="SPL" id="GoodsReceived">
        <Feature type="SPL" id=
          "CorbaFinanceInterface"/>
      </FeatureModule>
    </Features>
    [...]
  </ApplicationModule>
</ApplicationModules>
</Product>
</Application>

```

The XML schema definition representing the grammar of the domain-specific language as specified by the software product line in accordance with the requirements inherited from the business domain layer:

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Application">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Product"/>
      </xs:sequence>
      [...]
    </xs:complexType>
  </xs:element>

  <xs:element name="ApplicationModules">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="ApplicationModule"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="FeatureModule">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="FeatureModule" minOccurs="0"/>
        <xs:element ref="Feature" minOccurs="0"/>
      </xs:sequence>
      [...]
    </xs:complexType>
  </xs:element>

  <xs:element name="Product">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element ref="SystemInfrastructure"/>
        <xs:element ref="ApplicationModules"/>
      </xs:sequence>
      [...]
    </xs:complexType>

```

```
</xs:element>

<xs:element name="ApplicationModule">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Features"/>
    </xs:sequence>
    <xs:attribute name="type" type="type"
      use="required"/>
    <xs:attribute name="id" type="xs:string"
      use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="Messaging">
  [...]
</xs:element>

<xs:element name="SystemInfrastructure">
  [...]
</xs:element>

<xs:element name="Features">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="FeatureModule"/>
      <xs:element ref="Feature"/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name="Feature">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="type" type="type"
          use="required"/>
        <xs:attribute name="id" type="xs:string"
          use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name="ApplicationServer">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Technology"/>
      <xs:element ref="Version"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Version" type="xs:string"/>
<xs:element name="Technology" type="xs:string"/>
<xs:simpleType name="type">
  <xs:restriction base="xs:string">
    <xs:enumeration value="BD"/>
  </xs:restriction>
</xs:simpleType>
```

```
        <xs:enumeration value="SPL"/>
        <xs:enumeration value="EXT"/>
    </xs:restriction>
</xs:simpleType>
</xs:schema>
```

For reasons of clarity, above example refrains from XSD inclusions and imports. Doing so to incorporate artefacts from the business domain layer into a particular software product line's grammar was already described in section 5.3.3.2.

Generating code out of the application model is again done with XML stylesheet transformations. Based on the DSL elements used in a particular model, the product development team assembles the required generation artefacts. These will then be used to pre-fabricate the customer's product based on the software product line. For the example at hand, the XSL document appears as follows. The blue lines mark the Java code that is actually being written to the source files. For reasons of clarity, not all feature templates are depicted, as they are equal in structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:fn="http://www.w3.org/2005/xpath-functions">
  <xsl:output method="text" version="1.0" encoding="UTF-8"/>
  <!--Helper variable to allow for dynamic template calls-->
  <xsl:variable name="vTemplate"
select="document('')/*/xsl:template"/>

  <xsl:template match="Application">
    <xsl:apply-templates select="child::node()"/>
  </xsl:template>
```

```

<xsl:template match="Product">package com.ais.abd.acm.oms;
  <xsl:apply-templates
    select="SystemInfrastructure/Messaging"/>

class OrderManagementSystem{
  public static void main(String args[]){
    //Order Management System primary logic,
    //calling various features
  }
<!--the following xsl code cycles through the model's
features and calls the respective templates to generate
each feature's source code-->
  <xsl:for-each select="ApplicationModules">
    <xsl:for-each select="child:*">
      <xsl:for-each select="Features">
        <xsl:for-each select="Feature">
<!--dynamic template call for direct features-->
          <xsl:apply-templates select="$vTemplate[@name
= concat(current()/@type, '.', current()/@id)]"/>
        </xsl:for-each>
        <xsl:for-each select="FeatureModule">
          <xsl:for-each select="child:*">
<!--dynamic template call for module included features-->
            <xsl:apply-templates
select="$vTemplate[@name = concat(current()/@type, '.',
current()/@id)]"/>
          </xsl:for-each>
        </xsl:for-each>
      </xsl:for-each>
    </xsl:for-each>
  }
</xsl:template>
<!--templates for each feature which may also be imported-->
<xsl:template match="Feature" name="BD.WebServiceInterf">
private void webServiceInterface(String rx, String tx) {
  //Code for WebServiceInterface Feature
}</xsl:template>

<xsl:template match="Feature" name="BD.ASBInterface">
private void asbInterface(String rx, String tx) {
  //Code for ASBInterface Feature
}</xsl:template>

<xsl:template match="Feature" name="SPL.EscalationReport">
private Report escalationReport(DataSet myInput) {
  //Code for EscalationReport Feature
  return myReport;
}</xsl:template>

<xsl:template match="Feature" name="SPL.ManagementReport">
private Report managementReport(DataSet myInput) {
  //Code for ManagementReport Feature
  return myReport;
}</xsl:template>

```

```
<xsl:template match="Feature" name="EXT.HTMLInterface">
private void ext_HTMLInterface(String rx, String tx) {
    //Code for external HTMLInterface Feature
}</xsl:template>

[...]
</xsl:stylesheet>
```

During the actual code generation, the XSLT processor reads the XML model of the customer-specific order management system and applies the XSLT stylesheet to it. This means that the processor walks through the model and for each element it recognizes, it executes a specific template. As each feature is represented by such a template containing the feature's actual code, the model is transformed into compilable source files. For the example at hand, the generated code appears as follows (condensed for ease of reading):

```
package com.ais.abd.acm.oms;
import com.ais.abd.WS-RM.*;
//Skeleton code generated for each product of the SPL
class OrderManagementSystem{
    public static void main(String args[]){
        //Order Management System primary logic,
        //calling various features
    }
    //Code generated by the respective templates
    private void webServiceInterface(String rx, String tx) {
        //Code for WebServiceInterface Feature
    }
    private Report escalationReport(DataSet myInput) {
        //Code for EscalationReport Feature
        return myReport;
    }
    private Report managementReport(DataSet myInput) {
        //Code for ManagementReport Feature
        return myReport;
    }
    private void asbInterface(String rx, String tx) {
        //Code for ASBInterface Feature
    }
    private void ext_HTMLInterface(String rx, String tx) {
        //Code for external HTMLInterface Feature
    }
    [...]
}
```

The present section has shown how the automation of software development may be accomplished with previously existing tools and technologies. Based on the analysis of a particular customer's requirements, a feature model of the order management system maintained by the supply chain management systems software product line was derived. It orientates itself at the

architecture and requirements inherited from the business domain and product line layer. To make sure that the model adheres to the product line, it is validated by the XML schema document containing the grammar of the domain specific language. The model is then transformed into compilable source code by the XSLT processor. It walks through the model's elements and generates the source code for each feature.

## 5.5 Conclusions from the Exemplary Implementation and Implications on the Overall Model

Exemplarily implementing the concepts developed throughout the course of research as presented in chapter 4 has proven to be helpful in identifying shortcomings of the initial approach. Considering the IT Design Research cycle introduced as the methodological approach in section 1.4, the exemplary implementation represents the first part of the justification phase of the present research. The evaluation phase was completed in the initial chapters by identifying the actual state of science, i.e. current application of industrial key concepts in the field of software development, and current issues regarding the characteristics of the industrialisation of software development in systems integration. The practical problem of theorizing the solutions was solved with the development of a new concept for industrialized systems integration. The justifications of the three individual solutions in terms of their validity were achieved by discussing the results with subject matter experts from the industry and presenting each individual concept to the scientific community. Shortcomings identified during the exemplary implementation were immediately resolved and the concepts updated accordingly. The work at hand thus contains the revised version of the research based on the consolidated findings from the exemplary implementation and the expert interviews presented in chapter 6. The approach's initial weaknesses identified during the exemplary implementation and their respective solutions primarily affect the automation of software development. They can be subsumed as follows:

- **Depth of software development automation:** During the course of research and especially the case study, it was found that fully flexible and automated component-based development was hardly possible. With the XML-based model-driven engineering approach

presented in this work, one would need to provide glue code for almost every imaginable combination of components. As customer requirements and market trends are hardly foreseeable in such detail, providing such glue code is not economically feasible. Furthermore, conventional language structures such as conditional branching and loops would also be required. In fact, a Turing complete implementation of a domain-specific language would be inevitable. Although XSLT and XQuery (a query language closely related to XPath) have been proven to be suitable for such, complexity becomes extremely high, as they were not explicitly designed for programming purposes (see Kepser, 2004). In the present rather immature situation of model-driven engineering (Selic, 2008, p. 386), it doesn't seem advisable to pursue a fully automated approach.

To avoid inadequate complexity in a field where a highly efficient implementation of industrial concepts is necessary, a configuration centric approach was chosen early in the development stage. The domain-specific language models distributed components and business components on both the business domain and the software product line layer. They form the core assets to be reused during product development. However, instead of allowing an entirely free assembly of reusable elements, the software product line provides a product skeleton in which distinct features can be selected and deselected according to the feature model. This removes the necessity for more complex language elements and reduces potential mistakes during actual product development. Customer-specific features not provided by the software product line can be implemented manually within empty function or class stubs generated by the code generator.

- **Suitability of XML schema definitions for grammar specification:** What was further experienced is the limited capability of XML schema definitions to properly define inheritance of components. While monotonic inheritance is indeed possible (although with limitations on certain attributes), inheriting attributes and elements from more than one parent or polymorphism is not (Wang and Liu, 2003, p. 402). Nonetheless, it is possible to restrict attributes and elements to appear in a derived element, but it is not possible to use restrictions and extensions at the same time. For instance, a generic purchase order containing part number, quantity and price may be extended with a due date, but the price attribute

may not be removed concurrently. Workarounds are only possible with third-party extensions as provided by Schematron, for instance, a schema-validation language based on XSLT. Unfortunately, such extensions are not directly supported by conventional XML tool suites and may lead to a technology lock-in if such extensions don't advance with the underlying XML standards.

For this reason and the ease of implementation, the configuration-based approach with its predefined application structure was once more confirmed. Although not intended, it is still possible to extend predefined elements (e.g. business components defined by the business domain layer) on the software product line and production layer. On the other hand, attributes not required may still be available due to missing restrictions.

- **Suitability of XSLT v1.0 for code generation:** Using XML stylesheet transformations as a replacement code generation technique, several shortcomings in model traversal could be identified. With the given features of XSLT v1.0, it was not always possible to express code generation concerns and call the respective replacement templates. As the model of an application is not known in advance, the generator must provide the flexibility to dynamically call templates based on the actual model it is parsing. In the initial approach developed, it was necessary to know all possible features in advance and check if one of the elements matches them. Implementing a new feature during the advancement of the software product line was cumbersome, as the code generator had to be extended at various places in order to recognize the new feature.

For this reason, it became necessary to utilize XSLT v2.0 which offers more possibilities. One of these possibilities is the introduction of customized functions that can be called from throughout the whole document. Such a function could, for instance, be used to walk through the model several times and each time create different parts of the source code. Each feature's import statements of a Java source file could, for instance, be generated before the actual method implementations. Another possibility only available since XSLT v2.0 is the ability to build expressions to replace qualified names, which makes dynamic template invocation possible. However, invoking templates based on more than one attribute made it necessary to combine several attributes into one string identifier to uniquely

identify templates. What came also apparent is that debugging XSLT code generators is rather difficult, especially when multiple files are involved. This is especially the case when the code generator is assembled from templates provided by different feature development teams.

- **Multi-level generation approach required:** XSLT transformations are only capable of creating one output file at a time. As more complex applications consist of more than one source file, it became necessary to run multiple generation steps in order to create all artefacts. During the experimental implementation, it became apparent that most of these additional files would be scripts executed at a later stage. These scripts contain set-up instructions for infrastructure services, copy binary artefacts such as pictures or database files, or compilation instructions for more complex resources and eventually the application's source code. Another possibility of multi-stage code generation is to generate additional schema definitions validating the specifications of the business domain and product line based on the features actually used in an application. The same applies for code generators. In a first step, the code generator could analyze the application model and assemble the actual code generator to create the application's source code. This allows enforcing the adherence to the requirements inherited from the business domain and software product line layer and thus ensuring the ease of integration with products from other software product lines in the domain.

The exemplary implementation of the developed concepts has shown their ability to advance the industrialization of systems integration, although not always as extensive as in other fields of software development. This applies in particular to model-driven engineering which has shown to be far more complex than today's tools and technologies can handle. Particularly here were most deficiencies of the developed concepts identified. Most of them were resolved "on the fly", as they immediately became apparent in the integrated development environment and the chapters discussing the developed concepts were adapted accordingly. Chapter 5 thus achieves the fourth objective of this thesis by testing the proposed method of industrialised systems integration in an exemplary implementation based on a typical real world scenario.

However, what was not possible to evaluate under laboratory conditions was the organizational aspects, i.e. the three-layered approach to industrialized systems integration. The distribution of resources in the business domain layer, software product line layer, and production layer is hard to validate as it involves a huge real-life study subject willing to change its organizational structure. Furthermore, the distribution of complex system functionality according to the business component approach is hard to judge without sufficient experience in the development of such large-scale systems on a regular basis. To overcome these shortcomings in validation of the present research, the following chapter describes interviews conducted with industry experts from different multinational enterprises active in the field of systems integration.

## 6 Feasibility – A Practitioner Discussion

The approach developed during the research phase of this thesis not only affects technological aspects like component frameworks and code generation tools but also management related ones. In economically performing enterprises, it is hardly conceivable that new technologies can be implemented without sufficient management support. Their pure existence does not automatically lead to a wide distribution. A typical example is the still limited utilization of component-based development due to, among others, cultural issues in software development departments (Selic, 2008, p. 388). Management support is required to encourage software developers to adopt new technologies and use them in their day-to-day projects. Furthermore, the organizational framework must also be adapted to enable and facilitate their use.

In chapter 4, both aspects are discussed. The three-tiered approach towards software product lines in systems integration sets forth the organizational requirements, while component-based and model-driven systems integration define the technologies required. The latter two have been validated in laboratory research, i.e. an exemplary implementation presented in chapter 5. In an environment where researchers are interested in a successful implementation of previously-developed concepts, only their technical ability can be verified. What cannot be assured is their capability in a real-world context where market influences, social aspects and personal opinions of individuals become relevant. Unfortunately, validating a new concept in a real life environment is often not possible. For the present work, this would mean reorganizing a complete business division of a large systems integrator involving hundreds of employees and putting company success and customer satisfaction at risk. To overcome this limitation, expert interviews with experienced managers from different international systems integrators representing the target audience of the present research as defined in section 1.3 were conducted. Focus Groups as an alternative to expert interviews and means of qualitative research would be possible as well. However, as the participants of such focus groups would have to be senior managers from different companies throughout the country, organizing such a focus group seems hardly possible. Originating from empirical social research, expert interviews allow for the obtaining of knowledge on a research subject, in this case a large system integrator, that lies with the experts

due to their experience in a particular field (Gläser and Laudel, 2010, p. 11). Based on the interviews with subject matter experts, it becomes possible to identify further shortcomings of the concepts developed when applied in a real world example.

Science must always aim for objective and reproducible results, achieved with a methodological approach. Some of these methods, especially qualitative ones such as expert interviews, bear the risk of being influenced by the conducting researcher. It is thus highly recommended to precisely design the research and not to deviate from this design during execution. Section 6.1 thus describes the details of this qualitative approach, including expert criteria and selection, preliminary information provided to the experts, interview guidelines and interview methodology and evaluation.

## **6.1 Interview Design**

Based on the preliminary theoretical considerations and the results from the exemplary implementation, the expert interviews conducted for the present research can be defined as focused interviews, i.e. interviews with a stimulus as a reference (Helfferich, 2011, pp. 36, 45; Lamnek, 2005, pp. 368ff.). To keep the introduction to the topic short and ensure a common understanding among the experts, participants were provided with a summary of the concepts developed in chapter 4, annotated with the findings from the case studies in chapter 5. The document can be found in appendix A.4. In addition, this methodology allows for the obtaining of expert knowledge on other aspects not yet considered during the course of research (Trinczek, 2002, p. 211).

### **6.1.1 Interview guideline development**

Each empirical study is based on one or more research questions to close specific knowledge gaps (Gläser and Laudel, 2010, p. 62). An interview guideline may serve as a reference during the actual interview and ensures that all research questions are covered. It furthermore helps to make the interview understandable and conclusive for other researchers (Trinczek, 2002, p. 209; Gläser and Laudel, 2010, p. 63). To ensure unbiased answers, the operationalization of the research and thus the guideline questions must not influence the expert in his opinion. This

would, for instance, be the case with leading questions or closed questions with predefined answers. Ideal questions are easy to understand, neutral, posed in everyday speech and encourage the interview partner to narrate (Gläser and Laudel, 2010, p. 145). Hypothetical questions and those asking for personal opinions must be defined carefully and correspondingly considered in the subsequent evaluation of the interview (Gläser and Laudel, 2010, p. 145). The following research goals were defined for the interview in order to achieve the fourth objective of this research, i.e. testing the proposal's validity with the help of subject matter experts:

1. Practical foundation of the concepts developed throughout the underlying research and technically verified during the experimental implementation.
  - a. Is the newly developed organizational model for industrialized systems integration a viable option to implement software product lines in a large scale systems integrator as defined in section 1.3?
  - b. Does the suggested approach to component-based development in systems integration represent a feasible alternative to current software development processes and does its alignment with the integration metamodel provide adequate benefits for the integration of products developed based on this approach?
  - c. Is the limited approach to model-driven engineering a possible start to automated software development and does it provide a positive return on investment?
  - d. Are there any other aspects to consider when implementing industrial concepts on a large scale?
  - e. Does the overall concept provide enough economic benefits to bear the entrepreneurial risk of upfront investments and significant organizational changes?
2. Willingness of the developing organization and customers to perform a paradigm shift towards product lines and standardized products.
  - a. Are customers willing to forfeit specific features and functionality in favour of reduced cost and increased quality?

- b. Considering the self-concept of software developers, is it possible to divide current development departments into software product line engineering and application development?

The questions were derived by identifying those aspects of the developed approaches which could not be tested in a laboratory environment. In contrast to a standardized questionnaire, an interview guideline merely defines a framework, i.e. it leaves the interviewer a certain degree of freedom in his or her decision which question to ask and in what form (Gläser and Laudel, 2010, p. 142). As its name suggests, the document is just a guideline how to obtain the information of the expert being interviewed. However, the interview questions should adhere to the following principles (Gläser and Laudel, 2010, p. 145):

- Questions must be asked in clear and easy to understand everyday speech.
- Hypothetical questions and questions aiming for the interview partner's opinion are only meaningful if the personal situation of the interview partner is known with regards to the research subject.
- Simulatory questions are feasible for circumstances hard to explain (e.g. informal rules).
- Questions aiming for facts should be posed in an open way and stimulate to narrate.
- Stimuli to narrate are more important than asking for detail.
- Questions must be verified upon their independence and freedom of prejudices.

The resulting questions were operationalized in an interview guideline which is subdivided in four sections, each containing open questions about the topic. The sections were organized in favour of a smooth conversation, i.e. the interview started with a general introduction to the topic and then followed the industrialization concepts as described in chapter 4. After talking about each of them in further detail, the interview was concluded with the expert's assessment of the cultural changes required and the economic feasibility of the overall concept. The goal of the questions was to stimulate the interview partners to present their knowledge of software industrialisation in the context of systems integration. The interview guideline can be found in appendix A.5.

### 6.1.2 Expert selection

After defining the research objectives and operationalizing them in an interview guideline, the experts to be interviewed were selected. According to literature, an expert is a person with access to exclusive knowledge about the research subject (Meuser and Nagel, 2005, p. 72). In the context of an ongoing research, the criteria an expert has to fulfil often arise from the research subject. However, as the selection of the experts has a significant influence on the results, this decision must be made understandable and conclusive (Gläser and Laudel, 2010, pp. 95 f.). For the research at hand, the requirements were derived from the position of the thesis described in section 1.3. and can be summarized as follows:

- The person must be an employee of an enterprise providing customers from different business segments with software solutions to be integrated with other software systems or software solutions enabling the integration of such systems with each other. The total number of software developers employed at the enterprise must exceed 1000.
- The person must either be an executive manager for a complete business segment or explicitly responsible for the definition and introduction of software development processes and tools in such a segment.
- The person must have at least two years of practical experience in the industrialization of software development.
- Interview partners working as consultants in the field of software engineering processes and industrialization for systems integrators are also acceptable.

The experts interviewed were found during regular meetings of the “Software Development Processes and Tools” working group of the German BitKom in which the author participates. BitKom is the leading professional association of the German information and communication technology sector. In the referred working group, representatives from different companies active in software development-related business participate and discuss the implementation of new software development processes and tools in practice. The work often results in guidelines

for practitioners, such as a guideline to industrial software development (Achert et al., 2010) to which the author contributed.

### **6.1.3 Interview process**

The interview process follows a generic and idealized interview approach suggested by Mayring (2002, p.71), consisting of five steps. It starts with a problem analysis, which in this case was done during development of new industrialization concepts for systems integration and is presented in chapter 4. The approach continues with the construction of an interview guideline as described above. Part of the guideline is the preparative material which was sent to the participants in advance and contains an explanation of the overall concept and represents the stimulus of the focused interview. Both can be found in appendix A.4 and A.5, respectively. Mayring then suggests a pre-test phase of the questionnaire and an interviewer training. For the present research, this step has been omitted for the following reasons: It is correct that questionnaires, especially standardized ones and those used for quantitative research, must be thoroughly tested in advance. However, as a focused interview with an interview guideline should be as close as possible to natural language, the questions to encourage the participant communicating his knowledge can be in any order as long as their original meaning is ensured (Helfferich, 2011, p. 36; Gläser and Laudel, 2010, p. 42). This means that the exact wording is not essential and questions may be changed, added, and excluded, based on the participant's narration. Furthermore, the interview guideline is not supposed to standardize an interview (Gläser and Laudel, 2010, p. 150) but to ensure that the required expert knowledge is collected. Mayring continues with the execution of the interview, including the guideline, explorative, and ad-hoc questions. The interviews conducted during the course of research are depicted in

Table 6-1. The approach ends with a recording of the interview for later reference and evaluation. These are available upon request from the author.

Table 6-1: Experts interviewed

Company	Job title	Business domain	Development experience	Industrialization experience
SAP	Head of Custom Development Execution Americas and EMEA	Various	20 years	10 years
CSC	Manager Software Development - Public Sector	Public	17 years	5 years
SQS	Head of Software Engineering Research	Various	15 years	15 years

## 6.2 Interview evaluation

Assessing the results of the expert interviews requires a standardized and accepted methodology. In the empirical social research, two major strategies are known. According to Gläser and Laudel (2010, p. 26), the first strategy quantitatively searches for causal connections between different variables by collecting data in a standardized way and applying statistical methods to them. The second strategy is to conduct a qualitative content analysis to identify cause and effect mechanisms of different variables (Gläser and Laudel, 2010, p. 26). The identification of these mechanisms results from the detailed analysis of one or more cases (Gläser and Laudel, 2010, p. 26). Due to the application of statistical methods, the first strategy implies that a sufficient number of samples are available and that all of them can be represented in a standardized way. For the underlying research, neither was the case. For most statistical methods, such as the chi-square test, a minimum of 50 or more samples is suggested (Backhaus, 1996, p. 425). This was not possible, as there are hardly enough enterprises which fulfil the criteria in Europe. Furthermore, as the subject of the research is rather new, the questions could not be operationalized in a standardized questionnaire. In such a standardized questionnaire, it would not be possible to revise the evaluation criteria *ex post* in case of new insights from the interviews. Both are not an

issue in qualitative content analyses. However, in order to be comprehensible, the content analysis has to be based on a structured approach and must document each of the steps in detail. According to Mayring (2002, p. 114) and Gläser and Laudel (2010, p. 203), the approach can be summarized as follows:

- **Theoretical considerations:** Formulates the research question, theoretically analyses the problem and defines research variables. The research questions for the expert interviews were already developed within the interview guideline; the theoretical analysis is represented by the overall approach to be tested in chapter 4. The theoretical considerations result in a number of variables to be analysed during data extraction.
- **Preparation of extraction:** Defines search strategies and patterns as well as categories to summarize and organize the data of the variables identified during theoretical considerations. The step also includes the identification of indicators for each variable.
- **Extraction:** Interprets and extracts information from the interview material and stores it in a database. In case of ambiguities, extraction rules are specified and previously unknown characteristics added where necessary to the categories, variables, and indicators .
- **Data Processing:** Sorts the extracted information-based temporal or factual aspects, summarizes synonymous statements and removes mistakes from the extraction.
- **Data Analysis:** Analyses the processed information to derive causal connections between variables and presents them in the context of the research question.

For each of the above steps, the following sections specify how the qualitative content analysis was prepared to accommodate traceability of the subsequent interpretations.

### **6.2.1 Categorization and Definition of Variables and Indicators**

Qualitative content analysis is a method to generate a new information basis from the source material containing only that information necessary to answer the research questions (Gläser and Laudel, 2010, p. 200). By extracting and summarizing the experts' statements, large amounts of raw data are condensed to a manageable minimum. This is done with the help of search patterns based on the theoretical considerations made before, in the case at hand the

introductory paper and the interview guideline. The extracted information is then allocated to different categories based on the indicators developed for each. To remain comprehensible, the categories and their indicators must be defined and the position of the extracted material in the original data specified. The categories of the present analysis follow those of the interview guideline and are defined as follows:

Table 6-2: Extraction categories for qualitative content analysis

Category	Variables	Indicators (expert statements)
Paradigm shift	V1 Customer readiness	- Price sensitivity, limitation of functionality, trade offs, coverage of features
	V2 Supplier readiness	- Market anticipation, willingness for organizational changes
	V3 Employee readiness	- Developer self-concept, engineering-like methodology
Product Lines in Systems Integration	V4 Business Domain Layer	- Consolidate similar functionality, portfolio definition, architecture & roadmap, central core assets
	V5 Software Product Line Layer	- Product line requirements, product architecture & design, product core assets
	V6 Production Layer	- Application assembly, reuse of core assets
Component Based Systems Integration	V7 Business Component Factory	- Architecture, modularization, component granularity, ability to reuse
	V8 Integration Metamodel	- Necessity to standardize, compatibility between products, integration efforts

Model Driven Systems Integration	V9 MDE Maturity V10 XML based Alternative	- Experience, tools, implementation effort, maturity - Feasibility, implementation effort
Economic feasibility	V11 Economic feasibility	- Return on invest, investment size, cost reduction, quality, customer satisfaction
Other aspects	V12 Unconsidered aspects	- <i>[various]</i>

It should be noted that the variables and indicators have been changed based on new arguments raised by the experts during the interviews. As knowledge extraction from the raw data occurred after all interviews had been completed, the changes had no effect on the results.

### 6.2.2 Knowledge Extraction and Transcription

Extracting knowledge from the interview text is based on a personal interpretation of the scientist (Gläser and Laudel, 2010, p. 218). Based on this interpretation, statements are allocated to the categories and variables from above and the information summarized to create a basis for further analysis. As the interpretations are a result of subjective considerations, expert statements may be allocated to more than one category and variable; there is no right or wrong (Gläser and Laudel, 2010, p. 218). The decision for one or the other must, however, be reasonable and reproducible for other scientists. To ensure consistency in repeatedly occurring decisions, a list of extraction rules is imperative (Gläser and Laudel, 2010, p. 218). For the present survey, these have been defined for each variable and contain a cited example of an expert statement as shown in Table 6-3. In case of ambiguous statements, additional extraction rules were added.

Table 6-3: Extraction rules for qualitative content analysis<sup>5</sup>

Variable	Example	Extraction rules
V1 Customer readiness	“Yes, the discussion [about functional trade offs] takes place over and over again, and then it very much depends on the power dynamics at the customer’s enterprise: is the functional or the IT department more powerful?”	Only extract statements about customer readiness, not about the interaction between customer and supplier.
V2 Supplier readiness	“That’s hard to answer. Also the management layers are driven differently. On the lower, on the team layers, it’s probably more difficult to enforce than on the strategic layers.”	Extract statements about supplier readiness. Statements involving the interaction between customer and supplier may be extracted as well.
V3 Employee readiness	“Well, you definitely need to have an incentivization factor with cares for the long term objectives of the product line.”	Only extract statements about the employees themselves, not about organizational structures (these must be added to V4 and V5).
V4 Business Domain Layer	“Uhm, it, well, it very much depends on the organization [the customer] for which we are doing this [the business domain layer].”	Statements describing the interaction with lower layers must be allocated to V4. Statements regarding upfront efforts and economic feasibility must be allocated to V12.

<sup>5</sup> Exemplary verbal quotes have been translated by the author. The context necessary to understand the quote has been added in brackets and italicized.

V5 SPL Layer	“That [the utilization of business domain core assets] then must be accompanied by strong incentives but also certain regulations.”	Statements regarding up-front efforts and economic feasibility must be allocated to V12.
V6 Production Layer	(no example available as experts did not explicitly mention this variable)	Statements regarding the assembly of applications and systematic reuse of executable components must be allocated to V7.
V7 Business Component Factory	“Absolutely. Also in customer-specific individual software, there is potential [for reusable components].”	Statements regarding up-front efforts and economic feasibility must be allocated to V12.
V8 Integration Metamodel	“Uhm, I think that [adhering to an integration metamodel] only becomes relevant if we push into bigger scales.”	Statements regarding up-front efforts and economic feasibility must be allocated to V12.
V9 MDE Maturity	“[...] and the automated generation process also occurs in a very limited way.”	
V10 XML based Alternative	“But that’s [the XML based alternative] definitely something that could be tested in a pilot project.”	Statements regarding up-front efforts and economic feasibility must be allocated to V12.
V11 Economic feasibility	“I’d rather see this [the overall approach] for long running cases [product lines].”	
V12 Unconsidered aspects	“[...] but basically every step I do in the direction of industrialization alleviates [offshore] outsourcing.”	

After defining the extraction rules, all interview texts are analyzed and significant statements allocated to the variables. Instead of a full citation, the knowledge extraction process uses an interpretation to reduce linguistic complexity (Gläser and Laudel, 2010, p. 218). To ensure

traceability from the original material, the extracted and interpreted information contains a reference to the respective recording and a time code at which the original statement may be found. The following table shows some exemplary knowledge extractions. The full list may be found in appendix A.5.

Table 6-4: Exemplary knowledge extractions

Variable	Expert statement	Ref.
V02-2	Industrialization must be driven by higher level management; team leaders or department managers may not have sufficient influence to implement a paradigm shift.	E1-12:42
V10-1	A currently more feasible way for automation in systems integration is highly customizable applications which include all possible features but may be adapted to a particular customer's needs based on XML models.	E2-40:53
V11-1	The suggested methodology is a valid and sound approach, although not the only one.	E3-43:21

## 6.3 Presentation and Interpretation of Results

Despite the interpretation of the experts' statements, those of the researcher must be separated from the experts'. This allows the clear identification of the differences between the conducted theoretical research and the empirical validation. The following sections therefore first summarize the 112 expert statements obtained from the interviews conducted. Subsequently, the consequences and their influence on the presented approach are depicted.

### 6.3.1 Category 1: Readiness for a Paradigm Shift

The overall readiness for industrialized software development in systems integration has been subdivided in customer readiness, supplier readiness, and employee readiness.

- Customer readiness: All experts stated that generally customers are willing to accept functional trade-offs in favour of cost benefits as possible in software product lines. However, this may not always be possible. Two of the experts claimed that trade offs depend on a

good IT governance on the side of the customer. As functional departments become stronger, they may favour a broader functionality and thus not adhere to a company-wide architecture aimed at lower costs favoured by the IT department. In addition, there may be regulatory necessities preventing trade-offs such as general administrative provisions in the public sector. Here such decisions are not at the discretion of the customer. The experts were divided about the question as to which business processes should be standardized and thus covered in a software product line. One suggested concentrating on high-value services instead of commodities and mass products within a product line. This opinion conflicts with another expert statement indicating that, although generally willing, customers are reluctant to adapt their core business processes to predefined IT systems from a software product line. According to this expert, core business processes will remain individual while supporting processes (e.g. IT, human resources, finance and controlling) may benefit from software product lines. Further, he stated that software product lines will concentrate on functional layers not visible to the end customer. In summary, it can be said that the experts agree upon the general willingness of customers to accept trade-offs and limitations in functionality in favour of lower costs. They disagree on the scope of the software product lines, i.e. should such product lines cover supporting or core business processes.

- Supplier readiness: All experts agreed that a detailed market anticipation with resulting up-front investments in software product lines is not possible for the time being. Only very few highly regulated industries promise sufficient potential for reuse justifying up-front investments. Projects should rather be analyzed for reusability after their completion. If reusable artefacts are found, investments may be made to develop them into reusable components or products. One expert furthermore mentioned that market analysis is already often being conducted to a certain level identifying a particular stack of technologies a customer group concentrates on. Besides pursuing a bottom-up approach, organizational support was mentioned as being very important, i.e. an internal governance driven by higher level management is mandatory to implementing a paradigm shift. It was furthermore stated that the willingness for organizational changes and investments depends on the management key performance indicators and personal objectives. A cost-driven manager is more likely to

implement industrialization concepts than a utilization-driven manager. Thus an incentive rewarding long term cost reduction must be implemented for the responsible managers.

- Employee readiness: The necessary changes in the development culture were identified as a major challenge by all experts. Developers may suffer from the “not-invented-here” syndrome and may prefer redeveloping functionality instead of reusing already-existing implementations. This may be alleviated by training developers in systematic reuse and obtaining their acceptance for the concept. Furthermore, convincing employees of the advantages of industrialization is not enough. Management by objectives and incentives, such as awards, reuse measurement, and budget control, are inevitable. Care must be taken that the objectives are not too strict and thus encourage developers to optimize their reuse goals instead of developing more efficiently. One expert also stated that from his experience, developers already develop components and frameworks for further reuse voluntarily. During the course of the interview, it was, however, mentioned that this usually applies to small grained artefacts on source code level.

With regard to the approach presented in this thesis, the experts stated that, from their perspective, suppliers, customers, and employees are ready for a paradigm shift towards industrial software development in systems integration. What should additionally be considered are objectives and incentives for managers and developers to foster the adoption of industrial methodologies. In contrast to common literature on software product lines suggesting a detailed up-front market analysis (Pohl et al., 2005, p. 164; Linden, 2007, p. 49; Clements et al., 2007, p. 284), the experts suggested pursuing an opportunistic bottom-up approach to keep up-front investments and thus financial risks low. This is generally in line with the approach presented in this thesis which aims to reduce the implementation efforts of industrial key concepts. To better represent the expert’s opinion in the approach, further emphasis should be placed on harvesting already existing assets (see section 4.1.2) and implementing other industrialization concepts step-by-step by selecting a suitable MDE approach as described in section 4.3.1.3.

What could not be conclusively answered by the experts was the question concerning which area systems integration should be industrialized. While one suggests high value services

instead of commodities and mass products, others suggest the opposite by concentrating on supporting business processes or functional layers not visible to the customer. As it must be assumed that core business processes are more complex and individual than supporting ones, concentration on the latter during the first implementation of industrial key concepts in systems integration is suggested. Once first business domains and product lines are successful, more complicated and profitable ones may be implemented.

### **6.3.2 Category 2: Product Lines in Systems Integration**

The second category is concerned with the feasibility of the organizational model for software product lines in systems integration. Based on its structure, it has been subdivided into the business domain layer, the software product line layer, and the production layer.

- **Business Domain Layer:** In general, the experts agreed with the organizational model for industrialized systems integration. Bundling identical activities and functionality from product lines of the same domain is inevitable, as integration primarily occurs within one business domain. However, they also pointed out that such a structure depends on the context in which it is applied. The feasibility of the business domain layer depends on the number of product lines, the number of products being developed, and customer segments, for instance. The larger the volume, the more beneficial the approach, which should always follow a case-based reasoning. According to one of the experts, development for smaller customer segments may also occur outside the product line in functional units specialized in particular technologies. With regard to the scope of reuse, two experts suggested concentrating on industry wide standards. Here customers themselves often define standards and specifications which, implemented as reusable software components, can be seen as useful core assets. Other alternatives may be found in middleware technologies such as an enterprise service bus. With regard to the type of reusable assets, the experts also mentioned processes, tools, and architectures as very important and a good way to achieve efficiency gains. In addition, one of the experts mentioned proper governance as a prerequisite to a

multi-layered organizational structure, i.e. the correct allocation of core assets must be ensured.

- **Software Product Line Layer:** As with the business domain layer, the benefits of the organizational model at the software product line layer depend highly on the context and cannot be generalized. If there are enough broadly reusable components, then the effort at the product line layer can be reduced as suggested by the model. This applies in particular to processes, tools, and common architectures. Care must be taken in stipulating the use of central assets. Deviations from the business domain layer must be fought against with incentives and the necessity to justify a decision. However, a too-strict governance may lead to inefficiencies.
- **Production Layer:** The production layer was touched only briefly during the expert interviews. This was due to the fact that it does not differ from development in conventional software product lines. Furthermore, expert statements about application assembly and the reuse of core assets were allocated to variable V7 (business component factory) which will be discussed in section 6.3.3.

The discussion of the organizational model for software product lines in systems integration during the expert interviews has confirmed its general feasibility. However, as with customer and supplier readiness for a paradigm change, the concept's implementation is dependent on a market segment large enough, sufficiently standardized business entities on the customer side, as well as sufficient governance on the supplier side. Considering the position of the approach developed in this thesis, i.e. a large systems integrator active in several business domains as discussed in section 1.3, suitable market segments are assumed to be found. As a result of the expert interviews, it is furthermore suggested that concentration be directed to a very mature and highly standardized market. Here the implementing organization can benefit from already-existing standards and norms by implementing them in their core assets and product architectures. In any case, the implementation of the organizational model should follow a (business) case-based reasoning.

### 6.3.3 Category 3: Component-Based Systems Integration

The third interview category is concerned with the suitability of component-based development for systems integration. Besides the overall suitability of reusable components supporting self-contained business concepts or processes, the implementation of an integration metamodel was also discussed.

- **Business Component Factory:** The experts all stated that utilizing reusable software components is a reasonable way to reduce development efforts and sometimes even requested by customers. This also applies to customer specific developments as well as for generic functionality not visible in the end product. Regarding the functional scope of reusable components, the experts were of different opinions. On the one hand, it was stated that components should not only concentrate on standardized interfaces but also process-related functionality such as micro-processes or even selected business-critical tasks. It was also assumed that business components will increase in size, reflecting complete business concepts. On the other hand, a lack of standards on the organizational layer of the customers was mentioned and thus the disbelief in business components representing customer core processes. If the customers do not have their own processes standardized, only supporting ones such as human resources, procurement, and finance and controlling should be depicted in systematically reusable assets.
- **Integration Metamodel:** As with component based development in general, the experts also agreed to the implementation of an integration metamodel. Adhering to such does make sense and should be pursued in order to obtain a common understanding and standardize the internal architecture of the systems being developed. As with all technologies, the economic feasibility depends on the number of products being developed. Scepticism was expressed regarding the specificity of the metamodel. It was mentioned that it should be further detailed according to the customer segment attended. Furthermore, following an industry-wide integration standard requested by customers was regarded even more beneficial, although such are not yet existent.

The expert interviews confirmed that component-based systems integration is a necessary and feasible step towards industrialized systems integration. However, the ideal functional scope of a reusable component remains unclear. To accommodate the different expert opinions, the business component factory adapted to the characteristics of systems integration in section 4.2.3 implements different levels of component granularity and distribution domains. The former allows starting with small-grained distributed components such as interfaces, middleware, and data operations, and subsequently advances to business components and business component systems representing more complex business concepts and processes. As already suggested in 4.2.3, system integrators pursuing the industrialization approach should start small and, once successful, advance to more complex components.

The integration metamodel was assumed beneficial as well, although its specificity must be taken care of. To do so, the architecture & roadmap definition process at the business domain layer (see section 4.1.2.1) and the architecture design and development process of the product line layer (see section 4.1.2.2) must describe how their deliverables map to the integration metamodel. By adding further detail to the model, it becomes less abstract and allows one to actually see how and by which technical means integration takes place. Without this allocation of functionality, the experts' scepticism may become real and the advantages of a common understanding cannot be leveraged.

### **6.3.4 Category 4: Model-Driven Systems Integration**

Sections 2.5.2 and 4.3.2 already indicated that model-driven engineering in its current form is insufficient for a broad adoption in software development. To further validate this finding, Category 4 initially asked for the experts' experience with model-driven engineering. The interview continued with a discussion about the XML-based modelling alternative developed within this thesis.

- **Model-Driven Engineering:** All experts have experience in the application of model-driven engineering and all of them do not see the concept as a viable option for software development as of today. In addition to small experiments, several tools were tested to implement

model-driven development but were not regarded beneficial. Furthermore, the concept depends highly on specific vendors which software developing companies do not want to become dependent on. For the time being, it is used at the most to create UML-based models, whereas automated code generation is only of very low significance. Upfront investments are too high and the initial productivity of the developers is very low. Besides similar experiences due to insufficient tool support, the experts also identified a too-high heterogeneity, not only in systems integration, as an obstacle. It may be an option in very small and extremely standardized niche areas such as medical or avionic systems, but for the majority of business processes and underlying systems it is not yet mature enough. Although the experts acknowledged the potential of this technology, they do not envision its breakthrough in the near future.

- XML-based Alternative: Representing a product line's variability model as an XML model from which the final product is derived was seen as a possible alternative by all experts. Its feasibility, though, depends on several aspects. First of all, up-front investments, the scalability, and the number of customers have to be considered. It was furthermore stated that such an approach only works if the models used are fully specified and do not leave any room for interpretation. With regard to the functional scope, especially generating the actual product from the XML model, one expert suggested implementing the complete application upfront and using the XML models as a runtime configuration file for a customer-specific installation. New features may then be activated upon customer request and payment. The advantage of this approach was identified as a reduced implementation effort and lower maintenance due to multiple different systems. Another concern brought up was the risk that customers may not understand the models presented, as they are not experienced in model-driven engineering.

The situation of model-driven engineering described in this thesis was confirmed by the expert interviews. The biggest drawback is the lack of tools and a too high heterogeneity of systems to be considered. It is thus assumed to be reasonable to postpone model-driven engineering as a method for automation in software development for the time being.

The XML-based feature modelling approach developed as an alternative was acknowledged as an intermediate solution by the experts. As with the previous two key concepts of industrialization, it depends on a critical mass to break even. Furthermore, feature modelling requires a complete and context-free specification of the model, i.e. a feature must either be fully automated or fully manual. The critical mass depends on the market segment the systems integrator is active in. Considering similar expert statements about up-front investments in categories 2 and 3, it must therefore be suggested that industrialization be started in a preferably large and mature business segment. Ensuring complete and context-free models can be achieved with the modelling structure summarized in section 4.3.2.4 and illustrated in Figure 5-15 of the experimental implementation. Here, business domain and product line features are separated from customer-specific features to be implemented manually. In case a business domain or software product line feature does not fully represent a potential customer's needs, it has to be developed as a customer-specific one without any generation. In this case, the model would just contain an empty feature template. Another suggestion made by one of the experts was to completely develop a product suite and configure it at installation or runtime, depending on the customer's needs. This has, however, several disadvantages. First of all, everything would have to be anticipated and implemented in advance. This is in contradiction with the objective of this thesis to reduce upfront investments as much as possible in order to ensure a positive return on investment. It would furthermore not be possible to include customer-specific extensions to the overall product. The suggestion therefore remains unconsidered. Another concern was mentioned with regard to customer involvement, especially if they are not familiar with model-driven engineering. In this case it should be noted that modelling in general involves several refinements with the addition of technical details until the final model is available (Stahl and Bettin, 2007, p. 195; Petrasch and Meimberg, 2006, p. 125; Beltran et al., 2007, p. 70). The customer would only see the higher layers of the model without any technical details.

### 6.3.5 Category 5: Economic Feasibility of the Approach

Variable 11 evaluates the economic feasibility of the approach based on respective expert statements. The objective was to find out if the approach developed in this thesis provides sufficient financial benefits to justify its implementation.

- **Economic feasibility:** The experts interviewed agreed with the overall economic feasibility of the approach developed in this thesis. Industrialization was seen as a definite must which may already be observed in the software industry. From a conclusive point of view, model-driven engineering was identified as the most difficult and cost intensive step, component-based development as an absolute must, and software product lines as one option for specialization, although alternatives exist. The latter was confirmed by another expert mentioning that the suggested and customized combination of methodologies is a valid and sound approach, but not the only conceivable one. Nevertheless, it was confirmed that especially software product lines and component-based development are beneficial with regard to cost reduction and customer satisfaction. One of the experts stated that a typical investment for a software product line includes 3000 to 4000 person days and breaks even after serving about 20 customers. This correlates with other expert statements demanding a sufficient lifespan of the product line and its reusable assets. Here a long-lasting product line with a high strategic value for the customer should be chosen. Such may, for example, be found in the aviation industry with product cycles of 20 years or more, where investments in industrialization methodologies are definitely worth the effort. For an opportunistic approach, however, industrialization will never break even. At any rate, investments must be evaluated by a detailed business case.

Although generally confirmed during the expert interviews, a clear statement about the expected investments and returns thereof was only made in one. The remaining experts just expressed their concerns about the importance of a long-lasting product line in which industrialization efforts eventually break even. These experiences are contradictory to common literature indicating a positive return on investment after only 3 to 5 clients served (Leitner and Kreiner, 2010, p. 4; Pech et al., 2009, p. 293; Pohl et al., 2005, p. 10). It must be assumed that the reason

therefore lies in the nature of systems integration with its high heterogeneity resulting in a lower share of reuse per product. This unclear situation leaves room for further research in which a detailed business case for a pilot software product line in systems integration should be developed. Besides, the expert interviews have shown that it is highly recommended to start in a mature and slowly changing market.

### **6.3.6 Category 6: Additional Aspects to be Considered**

One of the advantages of qualitative expert interviews is the ability to identify other aspects not yet considered in one's own work, which is also referred to as explorative research (Gläser and Laudel, 2010, p. 86). This was also the case with the expert interviews conducted in this thesis. Besides extracting knowledge based on the theoretical model and predefined variables, additional aspects worth looking into have been discovered and may lead to further research:

- The experts were consistently of the opinion that integration metamodels and domain-specific assets should not come only from systems integrating software suppliers. It rather seems that significant benefit can be expected from industry-wide domain models developed by the industry itself. Based on these, it would be possible to develop domain-specific languages, product lines, component standards, and integration architectures. Such industry-wide standards are assumed to be beneficial on the customer side as well.
- One expert also mentioned that systems integration projects are not necessarily developed from scratch. Today, many of them rely on enterprise service bus systems or other integration middleware providing large amounts of functionality to be reused.
- Another statement was that industrialization should not be limited only to software product lines, component-based development and model-driven engineering. Other assets like libraries, frameworks, platforms, tools, and processes must also be considered and may be alternatives to the previous. Furthermore, according to the experts, the statement that component-based development and model-driven engineering cannot be successfully implemented without specialization is not assumed to be true.

- Once software industrialization is mature enough, it is also seen as a methodology to improve the efficiency of offshore outsourcing.
- It should be evaluated how the presented approach relates to agile software development techniques.
- It should be evaluated how component-based development matches with web services in a service-oriented architecture.

With regard to the suggested industry-wide domain models, the author agrees with the interviewed experts. Having such models would simplify upfront design and architecting efforts tremendously. The reason behind this is that everyone participating in this field would use the same model of logical concepts and thus ensuring a minimum degree of compatibility. However, as such are not available, creating simplified versions covering the necessities of software development cannot be neglected.

The expert opinion concerning systems integration projects not entirely being developed from scratch is true at first sight due to reusable functionality from middleware or service busses. However, this also applies to operating systems and databases. Integration technologies as of today should rather be seen as part of the infrastructure without any business-specific functionality. The latter still need to be developed from the start in the majority of projects.

Considering libraries, frameworks, platforms, tools, and processes is indeed necessary in an industrialized software development approach. As discussed in sections 4.1.2.1 and 4.1.2.2, these are actually included and constitute a very important share of a software product line's reusable core assets. The expert's statement about the ability to implement component-based development and model-driven engineering without specialization cannot be supported. Certainly, technically it is indeed possible, but implementing such concepts without limiting the scope and thus increasing their reuse potential seems economically disputable.

The benefits of industrialization for offshore outsourcing as mentioned by one of the experts can only be confirmed. Based on personal experience in real-world projects and various literature on the topic (e.g. Salger 2010 Salger et al., 2010 and Sako 2009 Sako, 2009), major challenges of

offshore outsourcing are a lack of standardization and thus insufficient modularization of the software to be developed in self-contained deliverables. It is therefore believed that the industrialization of software development in general (i.e. not only in the field of systems integration) will help in meeting these challenges.

The question about the applicability of agile development depends on the maturity of the implementing organization. Contrary to popular belief, agile development is not “chaotic”, but requires a certain amount of organizational structure, skilled software developers, mature processes, technical infrastructure, and sufficient project management (Chow and Cao, 2008, p. 963). It is therefore suggested that the industrial key concepts be first implemented and then, once mature enough, agile development methods introduced.

The last statement about additional aspects to be considered is concerned with the matching between component based development and web services. As presented in section 2.3.1, a software component is defined as a “unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties” (Szyperski et al., 2002, p. 27). This also applies to web services in a service-oriented architecture. Components do not necessarily need to be implemented in a common component framework such as .Net or J2EE.

## **6.4 Summary of the Practitioner Discussion**

Chapter 6 concludes the justification phase of the design research cycle by discussing the approach developed in the underlying research with subject matter experts from different enterprises active in the field. To this end, the experts were provided with a summary of the overall approach before the interviews. The interviews were then based on the contents of this paper and structured with an interview guideline to ensure consistency among the participants. The raw data available as audio recordings was subsequently condensed to significant expert statements with the help of predefined variables. These were then summarized in section 6.3 and their effects on the presented approach analyzed.

In conclusion, it can be said that the experts confirmed the presented approach as a valid and sound methodology to implement industrial key concepts in the field of systems integration, albeit not the only one conceivable. They especially mentioned software product lines and component-based development as important prerequisites to increase efficiency in software development. For software product lines, the organizational model from section 4.1 was also confirmed, although all experts expressed their concerns regarding the necessity of a minimum size to break even. The alignment of reusable components with an integration metamodel was regarded valuable as well, yet not as much as the other concepts presented. The experts pointed out that it would be more important to agree on processes, architectures, guidelines, and tools to ensure a common understanding and efficient development. Here the integration metamodel may serve as a good starting point. For model-driven engineering, the expert opinion was similar to the one expressed in this thesis. The methodology is not yet mature enough to be implemented in an industrial setting. Alternatively, a feature model configuration-based approach may be pursued but should be considered as a final step after successfully adopting software product lines and component-based development. Due to the scope of the interviews, i.e. obtaining expert knowledge from the management level for those aspects not able to be validated in an experimental implementation, only small changes to the overall technical solution became necessary. For economic aspects, however, the interviews led to a new perception with regards to the importance of the three key concepts of software industrialization, as well as the model's overall applicability:

- While software product lines and component based development were stated essential for industrialized systems integration, conventional model driven engineering was put into question. This confirms the results from the experimental implementation in which a feature model configuration-based approach was developed as an alternative. The experts appreciated this as an interim solution, however, only as a final step after successfully adopting software product lines and component-based development. The focus on industrializing software development in systems integration should thus be put on adapting the organizational structure and systematically reusing product and production artefacts.

- The overall applicability of the developed model should be limited to larger development departments serving mature and slowly changing industries. Here software engineering departments have sufficient time to plan, implement and fine tune the instruments of industrialization. Based on these experiences, industrial methods may be advanced to the needs of faster changing industries. Also a minimum size to break even was identified mandatory by the experts. For the present approach this break even point was assumed to be reached after serving about 20 customers. This is far more than the three to four usually mentioned in conventional software product line literature (Leitner and Kreiner, 2010, p. 4; Pech et al., 2009, p. 293; Pohl et al., 2005, p. 10).

The present chapter concludes the research this thesis is based upon by providing the second part of the justification phase of the design research cycle. After theoretically developing an approach for the industrialization of systems integration, the first justification cycle was accomplished with the exemplary implementation of the solution. As not all aspects could be tested in a laboratory environment, the present chapter carried out an additional validation by conducting interviews with subject matter experts. The approach was then updated with the shortcomings identified along the way. It thereby achieves the fourth and objective (being the last remaining one) of the underlying research to interview subject matter experts in the field of industrialisation and systems integration to obtain their opinion on the viability of the research.

## 7 Conclusion, Outlook, and Further Research

Industrialization still represents a highly complex topic for software developing companies and requires substantial changes in technology and culture and organizational alignment. Applying specialization, standardization and automation as the three industrial key concepts to software development seems unavoidable in order to increase efficiency. Especially in an economic environment where labour-intensive tasks are being outsourced to offshore service providers, industrialized software development appears as one of very few ways to keep up with ever increasing competition. Several efforts have been made to apply these principles to software development which are represented by software product lines (specialization), component-based development (standardization) and model-driven engineering (automation).

During the course of research, it became apparent that industrialized software development is still in its infancy. This is due to a number of reasons: First and foremost, specialization as the most important principle of industrialization was the last to be invented. It first emerged in 1995 and was further developed by the Carnegie Mellon Software Engineering Institute (Clements et al., 2007, p. xix). Significant research on the topic only began with the first software product line conference in August 2000. History shows that in every other business segment, specialization was the first industrial concept to appear (see Encyclopaedia Britannica, 2005a, 2005b, 2005c). Only within a specific context does the development of purpose-built tools and machinery make sense. In software development, this was not always the case, as can be learned from the possible failure of computer-aided software engineering (CASE) in the late 1980s (Selic, 2008, p. 382) or the still-pending revolution of reusing previously-developed components. In an arbitrary context it is impossible to standardize and automate. The second issue is of a cultural nature. Working in a software product line splits development in two major areas: developing core assets from scratch according to market requirements, and assembling customer applications from already-existing ones. Developers responsible for the former will no longer have the immediate satisfaction of seeing whether or not their work successfully translates into executable programs. Software development for core assets will be much more like a true engineering profession with extensive research, architecting and planning before writing the first code

statement; trial and error approaches are no longer possible (Selic, 2008, p. 388). Developers working on the latter, the assembly of customer-specific applications, may feel displeased as they will no longer be involved in the creative process of creating something. Rather, they will be reusing other people's work. In an environment where the majority of problems can be solved by reusing already existing solutions, the self-concept of a developer may become restricted. Besides a necessary change in the management of software developers, the customers will also have to change their views on individual software development. In an industrialized environment with a focus on mass customization it will no longer be possible to implement each and every feature request except at high additional cost. The customers' requirements will be mapped to the feature and variability model of the product line, which means that they may have to abandon a specific requirement in favour of lower cost and higher quality due to standardization. If they insist on that feature, they will have to pay an additional development fee. The third important reason for the poor adoption of industrialization and in particular model-driven engineering is the lack of established standards and tool support. Graphic models and precompiled software components can hardly be handled by traditional text-centric development environments software engineers are used to these days. This involves not only editors or compilers but also supporting tools such as a version management system. Compared to these tools, current graphic ones for model-driven engineering may even reduce productivity (Selic, 2008, p. 9) as they are ambiguous and do not provide the usability and customizability developers are used to. In addition, given that model-driven engineering aims to describe a complete system, it must be assumed that real-world models will become incomprehensibly complex and large. It is not yet known how such models can be modularized for concurrent development or even be mapped to a versioning system (Selic, 2008, p. 10).

As can be seen from above and throughout the present work, there are a number of reasons why software industrialization has not yet found a broad acceptance in practice. However, the concepts are there and most issues have been resolved by academia, especially for software product lines and component-based development which are slowly finding their way into practice. Yet businesses act with discretion due to uncertain investments. For model-driven engineering, the

situation is different. Major issues prevail and research is still in progress. For the time being, a full-scale implementation in an industrial setting cannot be recommended. Yet the present research has identified some MDE related aspects already possible today which help to reduce development efforts in an industrialized setting.

Apart from conventional development of commodity products such as operating systems and office suites, there are many other areas where software engineering plays a significant role and thus industrialization may be of interest. One of these areas is systems integration which was the focus of the underlying research. In today's fast changing world, IT faces continuous challenges in quickly adapting to new requirements and business processes. The number of IT systems and technologies required to implement these is increasing significantly in their complexity. The result is high heterogeneity in both technology and data distribution. This situation inevitably leads to systems integration efforts in order to provide new business functionality and data access (Fischer, 1999, p. 86; Leser and Naumann, 2007, p. 3). Here, at the very latest, software industrialization in its current form has reached its limits. Current concepts and implementation approaches of software product lines, component-based development, and model-driven engineering are not able to cope with a fast changing and highly heterogeneous environment. The characteristics of systems integration can be subsumed as follows:

- Unawareness of the IT landscape leading to unforeseeable consequences of integration decisions (Vogler, 2006, pp. 21,25; Puschmann and Alt, 2001, p. 1).
- Suboptimal degree of integration resulting from ad-hoc peer-to-peer connections in order to depict a specific business process on the existing IT landscape (Linthicum, 2000, p. 8).
- Unknown integration relationships on a technical level due to insufficient documentation and architecture, especially from urgent and badly planned projects (Vogler, 2006, p. 27).
- Lack of methodological approach in integration projects leading to an increased complexity of integration projects (Gorton et al., 2003, p. 1; Vogler, 2006, p. 28).
- Neither standardized integration platforms nor architectures are available in enterprises; autonomous development often results in different approaches to similar problems (Conrad et al., 2006, p. 14).

- High technical heterogeneity resulting from different hardware platforms, operating systems, development technologies, and enterprise resource planning systems (Gorton et al., 2003, p. 1; Linthicum, 2000, p. 8).
- Legacy applications must be integrated which often can't be replaced. Their development usually did not anticipate any systems integration (Vogler, 2006, p. 29; Themistocleous et al., 2001, p. 7).
- Redundancy of data due to different information silos which evolved in previously independent IT systems (Lui et al., 2011, p. 2; Linthicum, 2000, p. 11).

After identifying the characteristics of systems integration in general, their influence on the existing industrialization concepts represented by software product lines, component-based development and model-driven engineering was researched. It was found that the special situation of systems integration is a major drawback for the application of industrialized software development. The multiplicity of different technologies, caused by technological heterogeneity, redundant data sources and inflexible legacy systems complicates the definition of distinct software product lines and hence the development of purpose-built product and production artefacts. Subsequently their consequences on the application of industrial key principles to software engineering were examined. Based on these findings, the remaining research questions were confirmed or revised. The following sections anticipate the results of the first question and explain how the subsequent ones were derived.

In a software product line covering customer relationship management software, products may have to be integrated with a finance and logistics system. Including support for any potential integration relationship undermines the idea of specialization. A too-strict definition, in turn, would force development to occur outside of the product line. Furthermore, as the IT environment is different for each customer, systems integration projects yield one-only software products not suitable for any other customer. Benefits from economies of scale therefore cannot be exploited. The initial set-up cost for software product lines are contraindicative, as the return of investment cannot be ensured.

For component-based development in systems integration, difficulties in assembling products from reusable software components could not be identified throughout the research. However, CBD relies on systematic reuse and thus a specialized and preferably stable environment in which developing reusable artefacts is reasonable. In present organizational structures and software development approaches, it is not possible to establish such an environment, especially not in systems integration with constantly changing environments across different customers. The lack of system integration standards and architectures (Vogler, 2006, p. 146; Gorton et al., 2003, p. 1) is a further drawback to systematic reuse.

With regard to model-driven engineering, the efforts required to design and implement a domain-specific language as well as transformation engines and code generators is a major obstacle. Their implementation only makes sense if they are reused on a constant basis. Also, domain-specific languages require a delimited scope to be most powerful, a requirement that cannot be met in current integration approaches due to custom-built applications and frequently changing IT environments. In addition to that, model-driven engineering in its current form is largely immature (Selic, 2008, p. 16; Freeman and Webb, 2004, p. 199) and is assumed to require several years to break through (Kamp, 2012, p. 4).

The aim of the research was therefore defined as to investigate whether techniques of software industrialisation can be applied to software systems integration within economic and technical constraints and, if feasible, to propose a means for doing so.

Based on the characteristics of business informatics sciences and the objective of bridging the gap between academia and practice, IT Design Research was chosen as a suitable framework for the underlying explorations (March et al., 1995). These were conducted in three nested cycles, where the results of each were fed back into the overall justification phase (see section 1.4). During the first cycle, the research questions were identified and further elaborated upon. The outcome was then taken for the development of a theoretical concept for industrialized systems integration. The second cycle was conducted with an experimental implementation. The previ-

ously-developed theoretic concepts were practically applied in a laboratory environment and tested as to their technical feasibility. Identified shortcomings were immediately analysed and the concepts adapted accordingly. The third cycle was conducted with the help of expert interviews. Inherent in the nature of the research, it was not possible to completely simulate the concept in a laboratory environment. Conducting a real-world example in turn would impose too much risk to the research subject, as the implementation of the concept requires significant re-organization of a software development department. To verify a yet unconfirmed approach, the risk of economic failure with unforeseeable consequences for the enterprise, its employees, and customers seems inappropriate.

The following section presents the answers to the above research questions and explains the novelty for the field in further detail. Section 7.2 describes the limitations of the research in terms of applicability and validity. The chapter concludes with a discussion of possible further research for software industrialization.

## **7.1 Contributions to the field of business informatics**

The present work was carried out with the objective of advancing the industrialization of software development in systems integration. After researching the characteristics of this specific field, shortcomings of existing concepts for software industrialization were identified. They have been summarized and represent the first findings of this work. It allowed the refinement and adaptation of existing industrialization concepts in their current form to the needs of systems integration. These adaptations occurred in three areas: software product lines, component-based development, and model-driven engineering. For each of them, the key findings are summarized in the following.

### **7.1.1 An Organizational Model for Industrialized Systems Integration**

In section 4.1, a novel approach to implement software product lines as the first and most important principle of industrialization was developed. To do so, first a common strategy for system integration providers and two target scenarios reflecting software product lines therein was

defined. Based on these scenarios, the primary processes of product line engineering and product development were composed according to their objects of work and degree of required interaction between the development teams. Out of these, two organizational structures suitable for the implementation of software product lines were derived. In order to reflect the characteristics of systems integration, they were embedded in a three-layered approach consisting of a business domain layer, a product line layer, and a production layer. The primary advantage of this distribution lies in the consolidation of similar tasks for all software product lines within a given business segment. By moving them to an abstract level, their range of application can be broadened and only product line specific core assets need to be instantiated and enhanced where applicable. Projected on the characteristics of systems integration, an integration of products across different product lines becomes possible due to joint architectures and core assets within a given business segment. The issue of high heterogeneity will at least partially be resolved due to a joint technology roadmap and compatible core assets. It has, however, no effect on heterogeneity introduced by legacy or third party systems. The return of investment can be more easily achieved as large parts of software product line engineering have been consolidated to a higher layer. This reduces the effort and thus allows for breaking even after fewer products have been developed.

When changing the organizational structure of software product lines, of course their processes must also be adapted to the new construct. For the newly created business-domain layer, the processes were defined as follows:

- Business Domain Analysis explores the typical IT landscape of the business domain in scope and identifies areas of expertise required to develop and provide the products and services under consideration.
- Portfolio Definition evaluates the information from the domain model and develops a product portfolio for the particular business segment including typical applications and problem solutions.

- Architecture & Roadmap Definition develops integration architecture and basic product line specifications as well as a component framework applicable for all product lines. A technology furthermore ensures compatibility across the different product lines.
- Core Asset Development develops reusable assets applicable to all or many software product lines within the business segment and includes, for instance, software components, processes, and tools.

The second layer, the product line layer, is concerned with the implementation of a specific product line. This occurs multiple times within a given business domain and is based on the product families potential customers expect. The product line layer's processes are:

- Product Line Requirements Engineering further elaborates the requirements for each product based on the generic product and technology roadmap inherited from the business domain layer. Based on these, it specifies the requirements for the respective product line.
- Architecture Design & Development transforms the scope defined in requirements engineering into a technical architecture for the product line and its products. It describes the functional parts, defines relationships and interfaces, and establishes rules for their implementation. It thereby follows the architectural requirements from the business domain layer.
- Core Asset Development designs and implements the reusable artefacts required by the product line. These may either be defined by the business domain layer or the previous architecture design & development process.

The production layer reflects the actual development of a product within a software product line. In its processes it does not differ from conventional software product line development. In case a customer requires a specific feature not available from the software product line, the production layer may implement it at an additional fee.

The novelty can be summarized in the development of a new three-tiered structure for software product lines in systems integration including the processes required to successfully set it up. Furthermore, an organizational structure was derived for all three layers based on strategic scenarios of a typical large systems integrator.

### 7.1.2 Component-Based Systems Integration

As to the maturity and well-researched foundation of component-based development, it was decided to analyze existing concepts concerning their suitability for systems integration. After considering several approaches available in literature (see section 4.2.1.2), it was decided to pursue an adaptation of Herzum and Sims' business component approach (2000). Their work depicts a methodology to design, implement, test and maintain large-scale distributed systems built from reusable software artefacts. The major advantage with regard to systems integration lies in their focus on business processes being distributed over various independent systems. While technically mature, it would still be negatively affected by the characteristics of systems integration. Foremost to mention is its lack of specialization which would result in gracious generality and thus less effective tools and reusable components. Although it does divide business functionality into different technical levels of granularity, at the same time the approach does not provide a semantic classification of components, which is necessary for a common understanding of integration relationships. It should, however, be noted that Herzum and Sims' work has a more generic focus and should not be criticised for the shortcomings in the present context.

The business component approach was therefore aligned with the previously developed organizational model for software product lines in systems integration (see section 4.1) as well as the integration metamodel developed by Vogler (see section 3.2 and Vogler, 2006). The former allowed for the consolidation of recurring tasks of CBD into the business domain layer of the organizational model and thus the reduction of the overall efforts required for implementation of the second industrial key principle, i.e. standardization and systematic reuse. It was furthermore shown how the different dimensions of the business component factory approach must be aligned with the organizational structure of software product lines in systems integration. This includes considerations about responsibilities for project management processes, technical architectures, and core asset development. Additionally, the integration metamodel allowed for the defining of how the entities of the metamodel can be represented within the structure of the business component approach (e.g. utility, process, and entity components) and how different

entities can be distributed in a large-scale system (i.e. user workspace domain, and the enterprise resource domain). Concluding the above, the five constituent dimensions of Herzum and Sims' methodology were adopted as follows:

- The architectural viewpoint dimension was allocated to the organizational model. The business domain layer is now responsible for defining the project management architecture, the business domain architecture and the common parts of the technical architecture. In the software product line layer, each product line is responsible for defining the specific parts of the technical architecture, the application architecture, as well as the functional architecture of the products to be built.
- The component granularity dimension was also allocated to the organizational model. However, as granularity levels may occur at both the business domain and the software product line layer, two new granularity levels were introduced: Global distributed components and global business components are being provided by the business domain layer and contain domain-wide functionality or business concepts to be reused in the underlying product lines. Local distributed and business components contain functionality and business concepts required in the particular product line they are being developed in.
- The development process dimension was allocated to the organizational model as well. The approach's rapid component development process will now be executed on the business domain as well as the product line layer to produce distributed and business components as described in the previous paragraph. System architecture and assembly as well as federation architecture and assembly (i.e. composing a new large scale system from different other systems) is the responsibility of the product development layer and reflects the actual implementation of a customer specific system according to the product line's architecture and standards.
- The distribution domain dimension was aligned with the integration metamodel. Its technical nature describes the distribution of components across the system landscape, for example, if a particular component is to be deployed on a central server or resource, or within the client application on an end user's desktop PC. This allocation was done for all elements of

Vogler's integration metamodel and thus provides a common understanding about the structure of an integrated application.

- Functional categories as the last dimension of the business component approach were also aligned with the integration metamodel. It maps each of Vogler's metamodel entities to a process, entity, utility, and auxiliary component category which allow for the separation of functional concerns in a structured way.

The novelty can be summarized in the adaptation of an existing component-based development approach to the particular needs of systems integration. The allocation of the business component approach with the organizational model defines the method for distributing the different responsibilities described in the first three dimensions. The remaining two were aligned with Vogler's integration metamodel. Thereby it becomes apparent how functionality is to be divided into different component categories and where these components have to be deployed in the system environment.

### **7.1.3 Model-Driven Systems Integration**

Whereas for specialization and standardization as the first two industrial principles feasible solutions could be found, for automation this was not fully possible. During the course of research it became apparent that model-driven engineering in its present state is largely immature. This view is backed by various literature and experience reports such as Kamp (2012), Selic (2008) and Shirtz et al. (2007, p. 181), Staron (2006, pp. 68–69), MacDonald (2005, pp. 18–193). The biggest shortcomings identified are a lack of continuous tool support and cultural issues among developers used to conventional software development. A domain-specific language represented by a graphic notation inevitably takes up large amounts of space in an integrated development environment when a complex system is to be modelled. It is yet unknown how graphic models can be represented in version control systems and how changes to the same model can be merged into a joint branch. Another issue is the modularization of a model into separate independent parts to allow concurrent development. Current integrated development environments for MDE may furthermore reduce productivity (Selic, 2008, p. 9), as their user

interfaces are not adapted to the characteristics of the domain-specific language to be used and thus cannot provide auto completion or type checking as known from conventional programming languages. Besides the usability issues, there is no such thing as a common modelling standard vendors adhere to. It “is rarely possible to effectively exchange models from equivalent tools from different vendors” (Selic, 2008, p. 10) or to exchange models between corresponding tools, such as security or performance analysis tools (Selic, 2008, p. 10). As long as there is no “de-facto standard” or clear market leader, tool vendors will try to bind customers to their own (even incomplete) tool suite and thus do not promote interoperability. From the model-driven engineering approaches examined in section 2.4.4, none was able to overcome above characteristics. Without major improvements in standardization and tool capabilities, an advancement of model-driven engineering beyond academia seems not to be feasible. In complex and heterogeneous environments as found in systems integration, this is even more obstructive.

Instead of implementing immature concepts, it was decided to change an existing one in a way that above issues no longer have any affect and that the changes made are future proof and do not bring an implementing enterprise into a tool or technology lock-in. This was done by altering Czarnecki and Eisenecker’s Generative Programming approach (Czarnecki, 2005a; Czarnecki and Eisenecker, 2000). It was chosen in favour of others as in large parts it depends on the concept of specialization found in software product lines and introduced as the first industrial key principle in the present thesis. Their approach is based on a generative domain model which describes the problems to be solved with domain-specific concepts, feature models and customer requirements, the configuration knowledge which specifies how to combine different features, component dependencies, and construction and optimization rules. Applying the configuration knowledge to the problem space results in the solution space containing elementary artefacts with maximum combinability and minimum redundancy. The solution space is then used to assemble customer-specific requirements. To align the approach with the implementation of the first two industrialization concepts, process activities not yet covered were added to the organizational model and component-based systems integration as follows:

- The business domain layer as well as the software product line layer was enhanced with DSL design activities. In the former, the overall structure and domain-wide syntax and semantics are defined, such as the entity “car” for an automotive business domain. The latter covers product line-specific syntax and semantics such as “purchase order” for an order management system produced in a particular software product line. The distribution is illustrated in Figure 4-16 on page 170. In addition, the software product line layer was enhanced with some more formal domain requirements engineering methods such as Domain Analysis and Reuse Environment, Feature Oriented Domain Analysis, and Organization Domain Modelling. It now also covers the implementation of code generators and transformation engines.
- With regard to the business component approach, it was found that both generative programming as well as the business component approach benefit from each other, the former especially from extensive componentization (described in the component granularity domain of the business component approach) and thus systematic reuse on a larger scale than originally suggested. The same applies to architectural viewpoints, distribution tiers and functional categories. The business component approach benefits from a more advanced development process provided by generative programming, especially in conjunction with feature modelling processes such as Feature Oriented Domain Analysis or FeatuRSEB (Czarnecki and Eisenecker, 2000, pp. 69 ff.).

To model customer specific applications and implement the solution space, Czarnecki and Eisenecker suggest various concepts for domain-specific languages and code generators. However, these appear too fine-grained and labour intensive to be applied in the field of systems integration. Furthermore, for the shortcomings of MDE introduced above, an XML-based approach was developed:

- An XML schema definition is used to specify the grammar of a domain-specific language. By cascading different schema definitions, a software product line may also inherit the grammatical elements defined in the business domain layer.

- Once the grammar is defined, customer specific application models may be created based on the specifications of the product line such as architecture and feature model. The model is then validated against the schema definition of the product line.
- With the help of XML stylesheet documents, an XSLT parser translates the XML model into source files, setup and deployment scripts or other artefacts needed. By using several passes and generators, different generation artefacts or intermediate models may be derived.

The advantage of the XML-based approach is that there are a large number of tools available, models can be easily serialized and modularized and put in a version management system. XML is furthermore supplier independent and a widely-accepted standard. The approach, however, could not alleviate the shortcomings of insufficient user interfaces and graphic representation.

The contribution to the field can be summarized in the findings that model-driven engineering in its current form is not suitable for industrialized software development, let alone in a complex field like systems integration. During the underlying research, an alternative approach utilizing well-known and broadly available technologies has been developed, although this approach does not fully support model-driven engineering as known from different scientific literature. These concepts are not yet mature enough to be deployed on an industrial scale.

## **7.2 Research Limitations**

The research described in the present thesis was conducted within the boundaries imposed by the study subject. According to section 1.3, the focus was put on large enterprises developing complex software solutions to be integrated with each other and developing software solutions to allow the integration of new or already existing IT systems into a customer-specific IT landscape. This type of software development is also known as systems integration (Conrad et al., 2006, p. 11; Leser and Naumann, 2007, pp. 3–5). Enterprises providing such systems integration solutions are usually organized in a vertical structure with each unit serving a group of similar customers (Pierre Audoin Consultants, 2009) and employing up to several hundred software developers. The general applicability of the elaborated results is therefore limited as follows:

- The organizational model for industrialized systems integration developed herein can only be recommended for large enterprises with several hundred software developers serving similar customers in different business segments. The implementation of such a structure does not make sense if the business domain layer can't generate sufficient economies of scale to benefit the underlying software product lines. For example, with only two product lines, it is questionable if a superordinate business domain layer is still economically feasible. The approach also does not work for systems integrators organized in technology departments, e.g. consisting of a Java, C++ and SAP department. Here synergies of a similar functional scope cannot be leveraged and thus purpose-built tools would be less powerful than if organized according to business segments.
- For each of the developed concepts, it is important to have a medium- to long-term strategy and only implement it in mature markets. Although reduced due to economies of scale, product lines, systematic reuse, and automation still require up-front investments. If the strategy is being changed too fast and too often, the concepts will never result in breaking even.

Furthermore, initial knowledge acquisition about systems integrators and their particular problems in practice was largely obtained from Germany-based enterprises and extensive literature review. Although the concepts developed are not language-dependent, there may be some limitations with regard to culture and the ability to cooperate with service providers from other countries.

- The cultural concerns described in section 2.1 regarding the paradigm shift for customers and software developers may be of higher or lower significance in other cultures. In countries with a strong hierarchical thinking, culture may be less of a problem than in those where employees are largely independent and organize their work by themselves.
- In Germany, offshore outsourcing is somewhat limited by the language barrier, especially in companies well-established in the German market. This makes local software development more expensive and thus the benefits to be obtained from industrialization are more signifi-

cant. It can therefore not be ruled out that in English-speaking markets the benefits of industrialized systems integration are smaller than those of outsourcing to lower wage countries.

It must furthermore be assumed that only very few experiences from the industry find their way into literature such as experience reports or performance analyses. This fact somehow limits the validity of the conclusions drawn from literature. To overcome these issues it is suggested to extend the initial problem definition phase to a large number of enterprises by methodologically obtaining particularities of systems integration and experiences with software industrialisation.

The validation of the research described is based on an experimental implementation to test the technical feasibility of the results and several expert interviews to confirm those aspects not viable in a laboratory setting, i.e. the applicability of the concepts in large-scale enterprises. This is especially important as the organizational model for industrialized systems integration involves a complete reorganization of at least one subdivision being responsible for a particular business segment. Such a change usually involves a large number of employees, negotiations with labour unions and worker's councils and puts the economic success and customer satisfaction of the enterprise at risk. Due to these threats, it would be negligent to validate a still unproven concept in a real-world setting. In this situation, interviews with experts from several different enterprises similar to those described in the position of this thesis are the only feasible way for validation. The experts interviewed were either executive managers for a complete business segment or explicitly responsible for the definition and introduction of software development processes and tools in such a segment. Although carefully selected, the validity is limited by the number and professional experience of the interview partners. Increasing the research's overall validity should therefore be achieved by extending the expert interviews to a broader range of research subjects. Subsequently, the implementation of one or more real case studies could help prove the research a viable option in practice. However, it must be noted that this approach would take several years and significant efforts to complete. Other enterprises willing to implement the concepts developed herein therefore need to carefully analyze their own organizational, economic, and customer structure for potential characteristics preventing a successful implementation.

With regard to the research questions, the first three could be answered by analyzing the characteristics of systems integration and developing alternative approaches to the implementation of software product lines and component-based development. What was not completely possible was the implementation of model-driven engineering in systems integration. This was due to various reasons, foremost the immaturity of currently-known concepts and a lack of tool support. Automation as the last key principle of industrialization, therefore, could not be fully achieved. The alternative presented is closer to defining an instance of a predefined variability model than real-model-driven engineering as defined in scientific literature. As research in this particular field progresses, it may become possible to model completely new applications from a domain-specific language. To maximize future suitability, XML was chosen as the foundation for what is possible today. Enterprises implementing the suggested modelling approach should be aware that for the time being it cannot be assured as to what model-driven engineering will look like in the future.

### **7.3 Possible Further Research for Software Industrialization**

During the course of research, several other ideas, not always directly related to the research questions, evolved. What's more, the work conducted is not all embracing, nor does it cover every little detail to be considered when implementing industrialized systems integration. From an economic point of view, the following questions are of further interest and need to be researched in subsequent work.

- As of today, there is no medium- to long-term evaluation of industrialization concepts in software development available. The only thing researchers can rely on are industry reports on the implementation of individual concepts (e.g. Catal 2009 Catal, 2009, Linden 2007 Linden, 2007, pp. 121–265, Clements et al. 2007 Clements et al., 2007, pp. 349 ff., 417 ff., 443 ff., 485 ff., or Pohl et al. Pohl et al., 2005, pp. 413–433). Even in these, no detailed financial information about their economic viability is available. Here it would be interesting to conduct an industry project implementing all three industrial key concepts and evaluating them on a medium-term basis over three to five years. The results could be used to create a

universal business case template on which the decision as to whether to industrialize or not could be based.

- Further validation of the technical concepts in a comprehensive experimental implementation across several projects would be of interest. While technically possible, it would be interesting to analyze different large-scale customer projects in further detail to see where the processes developed during this research can be improved. This would allow a comprehensive fine tuning of the existing course of action and thus increase economic feasibility. However, such fine-grained evaluations only make sense if the process models are widely accepted and utilised first.
- The evaluation of the concepts presented in the present thesis against other specific areas of software development such as embedded or real time system development would also be of interest. As here, similar characteristics as in systems integration are assumed; a leaner and more efficient path towards industrialization may be helpful to these fields as well.

Besides economic considerations, the combination of industrial concepts with other software development techniques would also be of further interest. This applies especially to software development outsourcing in low wage countries (i.e. near- or offshore outsourcing), as well as the suitability of modern development techniques such as agile development.

- It is an interesting fact that despite missing standardization, software suppliers have split up their value chain by outsourcing development activities to low wage countries in Asia or eastern Europe. In other industries, this principle can normally only be seen after successful industrialization (Sako, 2006, p. 510; Mikkola, 2003, p. 440). Standardization and systematic reuse seem to be the most feasible way to allow modularizing engineering and production activities and subcontracting them to external suppliers not under direct control. This lack of modularization and standardization is regularly reported as one of the key issues in offshore outsourcing of software development activities (e.g. Salger 2010 Salger et al., 2010 and Sako 2009 Sako, 2009). It would be very interesting to see if industrializing prior to outsourcing would alleviate the issues frequently reported from globally distributed development.

- More recent approaches to software development include agile development techniques. They are aimed at streamlining existing development processes and concentrating on the technical problems to be solved. According to the often-cited agile manifesto (see Highsmith and Cockburn 2001 Highsmith and Cockburn, 2001; Beck et al., 2001), individuals and interactions, working software, customer collaboration and responding to change are valued more highly than processes and tools, comprehensive documentation, contract negotiation and following a plan. This rather “free” approach seems contradictory to a very structured and organized approach of software development. It would thus be very interesting to see if agile development concepts can at all be applied to an industrialized setting and if yes, to which extent and in which processes.

As model-driven engineering is not yet mature enough to be deployed at an industrial scale, further research is necessary once more advanced concepts and especially tool suites supporting the modelling, transformation and generation processes are available. It is assumed that besides specialization and systematic reuse, model-driven engineering will lead to an additional significant improvement in software development efficiency.

## **7.4 Coverage of Research Objectives**

Subsuming the above it can be concluded that the aim and objectives of the research have been reached as defined in section 1.3.

The literature review in chapters 2 and 3 have answered the first objective, i.e. identifying the characteristics of software industrialisation and systems integration and their individual prerequisites. They also presented the impact of systems integration on existing industrialisation concepts.

Chapter 4 has analysed the existing methods in further detail to identify if and how they can be used to overcome the shortcomings previously identified. Subsequently, viable means of applying industrialisation to systems integration were developed.

Chapter 5 has tested the proposed methods by examining how they could be applied in an example based on a typical real world scenario.

Chapter 6 has further validated the concepts by interviewing experts in the field of industrialisation and systems integration to gauge their opinion on the viability of the proposal.

Throughout the research project, the proposals were also tested by publishing articles on some of the ideas in peer reviewed journals and conference proceedings.

The overall aim of the research is thus considered being achieved:

To investigate whether techniques of software industrialisation can be applied to software systems integration within economic and technical constraints and, if feasible, to propose a means for doing so.

## Appendices

### A.1 List of Publications

Title of Paper	Name & location	Type	Date	Publication
Software Industrialization in Systems Integration	SEIN 2008, Plymouth / United Kingdom	Conference Paper	11.2008	Proceedings SEIN 2008; Wrexham, Wales 2008; ISBN 978-1-84102-196-6
Industrializing Software Development in Systems Integration	IST 2009 Ulyanovsk / Russia	Conference Paper	09.2009	Interactive Systems and Technologies, Vol. 3, p. 66-74; ed.: Sosnin, P.; Ulyanovsk 2009
Software Industrialization in Systems Integration (fully revised version)	ICCESSE 2009, Singapore / Singapore	Conference Paper	08.2009	WASET Vol. 56, p. 343-350; Singapore 2008; ISSN: 2070-3724
Industrielle Softwareentwicklung – Leitfaden und Orientierungshilfe		Code of practice	03.2010	Industrielle Softwareentwicklung; BitKom e.V., Berlin, Germany
An Organizational Approach to Industrialized Systems Integration	INC 2010, Heidelberg / Germany	Conference Paper	07.2010	Proceedings of the 8th INC 2010; Heidelberg, Germany; ISBN: 978-1-84102-259-8
Component Based Development in Systems Integration	Informatik 2011, Berlin / Germany	Conference Paper	10.2011	Lecture Notes in Informatics – Proceedings of the Informatik 2011; Bonn, Germany; ISBN 978-88579-286-4
Model Driven Engineering in Systems Integration	INC 2012, Port Elizabeth, South Africa	Conference Paper	07.2012	Proceedings of the Ninth International Network Conference: INC 2012; Plymouth; ISBN: 978-1841023151

## A.2 Integration Metamodel Entities

The following table describes the entities of the integration metamodel presented in section 3.2.

It was taken from Vogler (2006, pp. 90–101) and translated and summarized by the author.

Meta entity type	Description
Activity	<p>Activities are self contained units of execution in the workflow which are managed with the help of process integration. They combine tasks which the user coherently executes in a common functional and temporal context. Activities may be executed autonomously or in interaction with the user.</p> <p>For activities requiring user interaction, the controlling component creates messages to users with execution rights. Once the users have completed all tasks of an activity, the activity is completed. The controlling component defines the order and responsibilities of each task and activity or directly invokes automatically executable ones.</p> <p>With regard to reuse, activities may be used as building blocks in one or more workflows. As such they are isolated and not associated with a predefined workflow.</p>
Activity in workflow	<p>A workflow uses a predefined choice of activities in a certain order. A position in an organizational unit executes an activity within a workflow. Execution authorization as well as the definition of the execution order is not related to activities themselves but on activities in a workflow.</p>
Application	<p>Information processing within an enterprise is done by applications. An application is responsible for input, processing, and output of business relevant information. It can consist of one or more programs and data collections.</p> <p>An Application may provide several interfaces for integration which may be implemented as application or data interfaces.</p>
Program	<p>A program implements computerised operations of a certain area which usually process similar data and is responsible for data storage.</p> <p>Programs and their data collections constitute applications.</p> <p>To cover the functionality of a whole business process, often several programs are involved. From an integration point of view a program may provide templates, execute logic transactions, or consist of service programs.</p>

Application interface	An application interface offers methods which can be accessed from outside the application. Each application may implement more than one application interface.
Execution condition	Process integration defines the control of activities and based on control data. Execution conditions define the conditions under which an activity in a workflow may be executed and consist of valid control data values.
Execution authorization	The execution authorization defines which position in an organizational unit may execute which activity in a workflow. These authorizations only apply to independent workflows and do not cover heterogeneous business applications.
Data request	A process requires a number of data structures (entity types). The meta entity type data request contains these structures. Furthermore it describes which data collection may deliver the required data structure or if an additional demand must be created.
Data element	The data element describes a part of a data structure which cannot be further subdivided from a logical or feasibility point of view.
Data flow between tasks	Interfaces ensure the data flow between tasks which represent the connecting points between tasks of a given activity (e.g. data exchange between two applications on a user's desktop). The data flow is defined by a data structure specifying the data flow between the tasks. Besides primary key information, a data flow may also contain detailed business data.
Data collection	Data collections are parts of applications and may provide data collection interfaces. They serve as a permanent storage for data, which occurs under the requirements of the application being responsible for this data. A data structure defines the formal structure of the data collection. Compared to relational data bases, a data structure represents the data model (table definitions) and the data collection the actual data itself.
Data collection interface	Data collections may offer interfaces over which their data can be accessed from the outside. They contain a formal structure defined by data structures.
Data structure	A data structure defines the formal structure of data. It is a structured combination of data elements which is defined by the relationships between data components. Data components are data structures or data elements themselves. A data structure in the relational model represents an entity type.
Data transfer	A data transfer describes the propagation of data between two interfaces

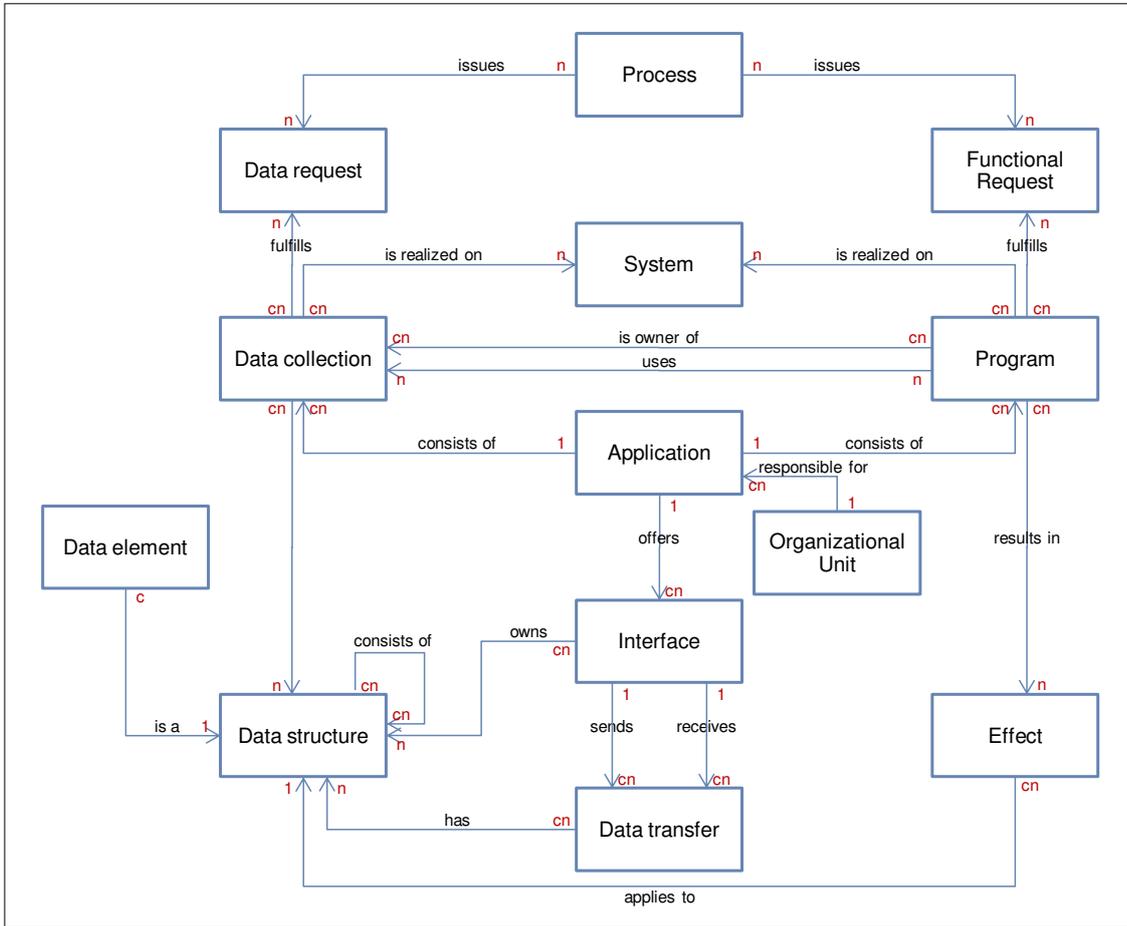
	of applications. This propagation is based on predefined rules to comply with data requests. The data transfer may occur between data collection interfaces, application interfaces, or both. The responsibility of the data transfer is the assurance of the data propagation and consistency of the information system, such as redundancy checks between systems.
Dialogue element	Dialogue elements ensure communication between the information system and the end user. An interactive task is realized with dialogue elements which are provided either by the called application or must be developed independently. The user navigates in a predefined order through the elements.
Effect	Applications access data structures and create effects. Effects describe the use of the data, distinguishing between append, read, modify or delete.
Functional request	A process defines multiple functional requests which the information system supports with applications. If the functional requests cannot be served, a demand for new applications is generated.
Method	The functionality provided by applications is realized as methods which lead to effects on data.
Middleware	A middleware is a software layer between the business applications and the system software (e.g. operating system, data bases, etc.). Based on standardized interfaces, they provide transparent communication for distributed systems and thus form the infrastructure for application integration in a distributed and heterogeneous IT landscape.
Organizational unit at site	An organizational unit located at a particular site.
Organizational unit	An organizational unit is a self contained, independent part of the organizational structure of an enterprise. They represent all types of organizational units, such as departments, sections, groups, offices, etc. In case an enterprise contains more than one site, an organizational unit may exist on one or more sites as well.
Process	<p>A process summarizes a number of activities to be completed in a predefined order, supported by information technology applications. Its value proposition are services for other processes inside and outside of the own organization. It is enhanced by process managers, process circles or process boards and measured by key performance indicators.</p> <p>An enterprise concentrates on those processes essential for their economic success. These core processes are defined with a clear delineation of busi-</p>

	ness segments, product portfolio, or organizational structure.
Interface	An interface describes data which the application provides or receives from the outside, how it is being accessed, and how data can be exchanged. It may also contain a number of methods to be called in the same manner. An interface of an application is either a program or a data interface.
Site	Sites are physical, geographic locations. A site contains organizational units and systems. The site information extends an organizational unit.
Position	A position is the entirety of the expectations against an owner of a position. Within this metamodel it contains all rights, privileges and responsibilities of the position owner and his relationship to the enterprise. A position is thus a number of activities which the owner of the position may or must execute. Positions are embedded in organizational units and allocated to specific sites.
Control data for activity	<p>The propagation of control data between activity and controlling component is based on interfaces which define connection points between activities. The controlling components define order and execution of activities with the help of execution conditions.</p> <p>A data structure defines the structure of the data which an activity sends and receives and thus describes input and output interfaces of the activities. The controlling component saves (e.g. for protocol reasons) the required control data itself, independent from their possible ability in other business applications. To avoid inconsistencies, the control data ideally only contain primary keys such as customer id and document number.</p>
System	A system is a platform (computer, operating system) on which programs, applications, workflows, data collections or middleware are being realized.
Task	<p>Tasks are elementary steps of an activity to be completed and are primarily based on their atomicity. Tasks are either manual or IT supported and describe an activity in further detail. The required degree of detail is achieved once a responsible organizational unit can complete a given activity based on the tasks provided to them.</p> <p>Tasks can be executed automatically in the background or with user interaction. Interactive tasks are realized with a user interface, i.e. they integrate business applications into the activity. For the user they are identified by forms, documents, or screens.</p> <p>The procession order defines under which temporal and logic dependen-</p>

	<p>cies the tasks of an activity are to be completed. Their control is the subject matter of desktop integration.</p>
Workflow	<p>A process is implemented on information systems with the help of workflows. They consist of a number of activities and are hierarchically structurable. Their flow control is based on controlling components such as a workflow management system. Workflows are realized on systems which may be geographically distributed.</p> <p>The separation of a process into workflows orientates itself on services and thus customer requirements, as well as implementation aspects of the information system.</p> <p>Workflows delineate those units in a process which, together with desktop and information system integration and the information system, constitute a self contained and data flow oriented workflow. A process administrator takes over the responsibility of a workflow. His responsibility especially contains maintenance and operation of the technical infrastructure.</p>
State	<p>A state corresponds to a time span during the runtime of a workflow. The controlling data attributes define the state of a workflow whereas a combination of attributes is combined to a state. The identified states represent the phases of a workflow which can be tracked and monitored.</p> <p>A workflow knows multiple states. During its runtime it is in exactly one state. In a state the user may execute one or more activities which each invoke one state transition. In addition the execution conditions define if an activity is existent at all.</p>
State transition	<p>The state transitions define in which timely and logic dependency the activities are to be executed within the workflows. They combine the single activities in a workflow to a sequence and control flow.</p> <p>Activities invoke state transitions. In one state, a user may execute one or more activities which each invoke another state transition. An activity leads a workflow from an initial state into a target state whereas both can be equal.</p>

### A.3 Integration Metamodel – Information System Viewpoint

The following figure contains the information system viewpoint of Vogler’s integration meta-model (Vogler, 2006, p. 90).



## **A.4 Expert Interview Preparation Paper**

The following pages contain the preparation paper which was sent to the experts in advance to the interview. For presentation reasons the paper was reformatted.

# The Industrialization of Systems Integration

Matthias Minich  
School of Computing and Mathematics  
University of Plymouth  
Plymouth, United Kingdom  
matthias.minich@plymouth.ac.uk

Bettina Harriehausen-Muehlbauer & Christoph Wentzel  
Department of Computer Science  
University of Applied Sciences Darmstadt  
Darmstadt, Germany  
b.harriehausen@c.wentzel@fbi.h-da.de

**Abstract**—Software development in systems integration projects is still reliant on craftsmanship of skilled workers due to high heterogeneity and constantly changing environments. It is questionable if the concepts of software industrialization in their current form can efficiently be applied to the field. High upfront investments and a potentially limited degree of reuse compromise the return of investment for software product lines, component based development, and model driven engineering. The present paper analyses these challenges imposed by systems integration and suggests an alternative and light weight approach for the implementation of aforementioned concepts.

## I. SOFTWARE INDUSTRIALIZATION

Several efforts have been taken to apply the industrial key principles of specialization, standardization, and automation to the field of software engineering. The first is represented by Software Product Lines (SPL), standardization is available with Component Based Development (CBD), and Automation can be achieved with Model Driven Engineering (MDE). A Software Product Line is “a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” [4]. It exploits economies of scope rather than scale by reusing as many product and production artifacts as possible. Component Based Development is an approach to exchange and systematically reuse standardized software artifacts. “A software component is a unit of composition with contractually specified interfaces and context dependencies only” [27]. Components can be independently utilized and composed to applications and usually represent a complete business concept. Using visual or textual models as a human readable but formal description of software, Model Driven Engineering as the third industrial principle aims to raise the level of abstraction in order to fill the gap between the semantic problem solution and its technical implementation [9].

Unfortunately the most important concept for software industrialization, i.e. specialization, was invented last. Significant research on the topic was started with the first Software Product Line Conference in August 2000 only. Adoption in the industry, however, is rather small due to implementation cost, availability of tools, and significant changes in the development culture [16; 26; 4]. Without limiting the scope of one’s production, subsequent industrial concepts like CBD and MDE cannot successfully be implemented. As of the resulting gracious generality, a large scale adoption of CBD and MDE in their initial occurrence possibly failed [10; 14; 26]. For reusable components especially the lack of standardization, insufficient component descriptions, and an unknown intellectual property rights situation prevented a wide spread, inter-company adoption and the formation of component market places [28]. Modeling software and automatically generating code primarily suffers from usability problems and tool availability [26; 15]. A lack of interoperability between different vendors, and scalability problems of large and very large models are the foremost to be mentioned [26; 15]. Considering the previous shortcomings, it may be assumed that the industrialization of the software sector is largely immature. The majority of products are still developed manually without any significant specialization, standardization, or automation in place. Adapting these concepts to the field of systems integration (SI) seems even more difficult. Compared to conventional software development of commodity applications, systems integration comes with certain characteristics requiring special consideration.

## II. SYSTEMS INTEGRATION AND ITS CHARACTERISTICS

Systems integration deals with the steps required to move an IT system from a given degree of integration to a higher one by merging distinct entities into a cohesive whole or integrating them into

existing systems [25; 7]. It can be further divided into information integration and application integration [3; 20]. Information integration covers the integration of data sources, while application integration deals with the combination of software systems to support specific business processes. The present work concentrates on the latter with a focus on software development.

Considering previous papers from Hasselbring [12], Vogler [30], and Conrad [3] on different layers of systems integration, the following three dimensions can be defined and will be taken as a reference in the following sections:

- **Business process:** defines the core business processes, organizational objectives, and structure of an enterprise. They define which functionality and data is required and how IT systems semantically interact.
- **Workflow:** subdivides a process into activities and maps them to IT systems. It defines data sources and functionality required from a technical point of view, as well as their internal and end user interaction.
- **Technology:** defines which applications may access which data or functionality, how this is done, and how data management (e.g. redundancy) takes place.

Situations from which SI efforts arise are manifold and reach from new business models over mergers and acquisitions to phasing out legacy systems. Each of these leads to reengineering business processes, integrating new data sources, or developing new applications [30]. When doing so, the characteristics of SI must be considered as well.

For the business process domain, cooperating with other companies for instance, requires a frequent exchange of information, such as bills of material, production line data, or financial transactions [1; 30; 22]. The same applies to offering new services to customers, requiring new processes and information systems [1; 18]. Other common reasons are mergers and acquisitions between enterprises. Problematic here is that existing integration relationships are often unknown, thus a change preventing process definition is not possible [30]. This unawareness of the IT landscape may lead to unforeseen consequences of integration decisions [30] and thus additional efforts.

In the workflow dimension, making a change to an already implemented business process usually requires changes to the underlying IT systems. Should the change affect other than process exclusive IT systems, changes to other processes cannot be ruled out [30; 22]. This in turn leads to a loop back to the business process domain. Other drivers of application integration may result from regulatory necessities, such as data preservation laws for telecommunications providers. Subdividing a business process into workflows and depicting them ad hoc on different IT systems may lead to a suboptimal degree of integration [18]. In such an environment the integration relationships may be unknown due to insufficient documentation and architecture [30], especially in case of urgent and badly planned projects. The root cause here lies in a missing methodological approach. Although they have been defined in literature during the last years, they are not yet known or adopted in the industry [11]. Heterogeneity caused by the previous problems prevents the implementation of holistic integration platforms or architectures within enterprises. It is reinforced by the fact that autonomous system development always results in different approaches to related problems [3]. Such autonomous system development in addition results in stovepipe applications [18] or information silos [17].

Integration drivers on the technology dimension result from higher dimensional requirements or are technology driven. Legacy systems may for instance run out of service and thus need to be replaced [3; 30; 18], or other integration decisions may require a change in the technical integration architecture [8]. The reasons for the latter can often be found in a historically grown landscape which lacks a clear and future-proof architecture. From a technical point of view, heterogeneity is the major issue in systems integration. It results from different hardware platforms, operating systems, or enterprise resource planning systems which evolved over time [11; 18]. Implementation efforts disproportionately rise with the number of systems unless common integration architecture is used. Another issue is the integration of legacy applications [30]. These were often designed as standalone solutions with no integration in mind. Obsolete data management, interfaces, or a lack of documentation or maintenance make their integration extremely difficult [29]. The final challenge lies in the redundancy of data. In integrated environments it becomes difficult to define which data resides where, how it is accessed and how redundancy is managed. Information may easily become outdated and inconsistent, leading to serious issues in business process execution [17; 18].

Recapitulating above characteristics, it can safely be assumed that systems integration projects differ from traditional commodity software development. Successfully implementing processes or updated requirements can only be achieved with a clear strategy and flexible software development approach. In addition, enterprises usually have autonomously grown systems and information silos, which result in a highly complex and heterogeneous IT landscape with redundant and inconsistent data sources. In this case

software development has to care for all of the above without adding new complexity.

Today systems integration solutions are still implemented from scratch by utilizing traditional software development methods. These however were designed with monolithic systems in mind, as integration was not of interest at the time of their development. The result may be an ‘integrated monolithic system’ with highly complex dependencies. Moreover, these development models do not incorporate the basic principles of industrialization and thus may not leverage potential improvements in cost, efficiency and quality as initially stated.

As introduced before, SPL, CBD, and MDE represent specialization, standardization, and automation for software development. The concepts are well understood and first literature is available on combining them in factory like development environments, as for example in Greenfield and Short’s book on Software Factories [10] or Czarnecki and Eisenecker’s Generative Programming [5; 2]. However, software development in the context of systems integration has to challenge a multiplicity of technologies, inflexible legacy systems, once only technology combinations and a very high complexity. It seems disputable whether the existing concepts for industrialized software development will ever break even in such environments.

### III. THE INTEGRATION METAMODEL

To analyze current industrialization concepts and adapt them to the specific needs of systems integration, a precise description of objects, entities, and their relationships is necessary. Such can be found in Vogler’s integration metamodel [30]. The model has four different viewpoints, which are process integration, desktop integration, systems integration, and information system. With exception to information system, they can be mapped to the business process, workflow, and technology domain. The information system as an entity itself is omitted as it does not have an effect on the integration effort. Fig. 1 shows an overview of the model, while the following sections describe the viewpoints in more detail.

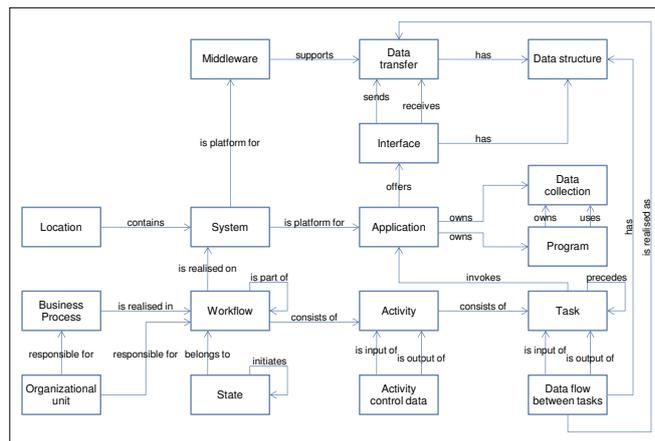


Fig. 1. Overview of the Integration Metamodel [30]

The process integration view (q.v. [30]) describes the business process as its central element. It is implemented as one or more workflows and realized on one or more IT systems on a particular site. The workflow itself is represented as a finite state machine, moving from one state into one or more subsequent states. These state transitions initiate an activity consisting of one or more tasks and data sources available from enterprise applications. Execution of activities and underlying tasks depends on execution conditions and authorizations provided by the responsible organizational unit. Vogler also defines the organizational structure in more details which is necessary for the overall integrity of the model, but can be neglected in the context of the present work.

The desktop integration view (q.v. [30]) describes how an activity bundles one or more tasks which may invoke applications for user interaction. An application in the metamodel context is defined as a collection of one or more programs used to provide data access and complex business functionality through clearly defined interfaces. Programs in turn are responsible for data storage and management and provide functional building blocks for more complex business applications [30]. Based on a data structure, tasks may also send and receive information to or from other tasks. The task itself can be realized with one or more user interface elements, but may also be executed without any user interaction.

The systems integration viewpoint describes integration relationships with other applications or data sources which are not visible from higher viewpoints. This is due to logical entities not necessarily being mapped 1:1 to information systems, i.e. one IT system may represent several process or desktop entities. Changes to the previous two will therefore also cause changes to the systems integration viewpoint, which

is one of the reasons why enterprise application integration projects are far from trivial. To establish these relationships, data transfer between different applications is required. It is based on interfaces which may either be program or data collection interfaces (not depicted in Fig. 1). A program interface provides a variety of business methods or functions belonging to one particular program. The program in turn is part of the underlying application responsible for more complex data access and business functionality. A data collection interface belongs to a data collection, which is defined by one or more data structures. This data collection belongs to a particular program which utilizes this and (or) other data collections to fulfill its tasks. A middleware ensures the technical data transfer between the different enterprise applications and their program and data collection interfaces.

The metamodel will be used to align component based development in section V. Before doing so, specialization as the first industrial key principle must be implemented.

#### IV. SOFTWARE PRODUCT LINES IN SI

As discussed before, it seems disputable whether SPLs can be implemented in SI as well. The major reasons can be subsumed as (1) integration across SPL boundaries is difficult without broadening the scope too much, (2) multiplicity of technologies and high heterogeneity lead to little software reuse, and (3) an uncertain return of invest as the minimum number of products to break even cannot be ensured.

It is assumed that an integration of different IT systems mostly occurs within the boundaries of a particular industry. An automotive supplier for instance will hardly need integration with an e-government solution. Yet he may require integrating his SAP accounting system with a logistics application of one of his suppliers. This assumption is backed by organizational structures of major systems integrators, which are based on a vertical structure [24]. Any integration architecture should therefore at least support the typical systems of the respective industry. However, implementing such architecture within a single SPL would broaden its scope far beyond being efficient and thus being feasible for industrialization. This especially applies to reusable core assets, which would be too generic to provide any benefit. Leaving too much out would force development occur outside of the product line.

To overcome these issues, a three layered organizational model has been developed. It essentially adds a layer of abstraction on top of conventional software product lines. The contents of each have been defined as follows:

The Business Domain Layer is a new super ordinate layer that spans over a complete division or business segment within a system integrator's organizational structure. It identifies the major requirements of the business domain in scope, the required product lines and their products and conceptually defines fundamental core assets, technologies, and systems typically used therein. The development of abstract system landscape and integration architecture ensures the interoperability of different systems and product lines within the business domain. Reusable assets needed in several underlying product lines are provided as well and reduce subsequent efforts. The business domain layer consists of the four core processes Business Domain Analysis, Portfolio Definition, Architecture & Roadmap Definition, and Core Asset Development.

The Product Line Layer consists of several software product lines identified in the Portfolio Definition process of the Business Domain Layer. The Engineering processes of these software product lines differ only marginally. The most obvious variance to conventional software product line engineering is the lack of the Business Domain Analysis process, and a reduced Domain Requirements Engineering process. These functions are now incorporated in the Business Domain Layer and provide their findings to the subsequent product lines. All other processes remain the same but must adhere to the specifications and utilize the provided core assets from the business domain layer. Product lines may additionally benefit from joint core assets provided by the business domain layer.

The Production Layer contains the actual implementation of a product within a software product line. It does not differ from the conventional concept of software product lines.

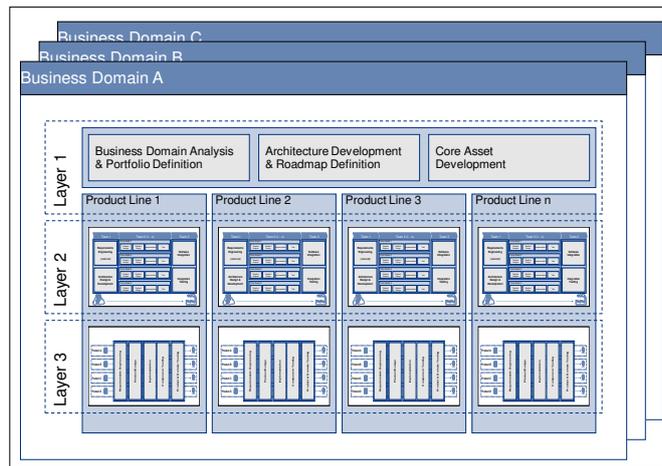


Fig. 2. Three-layered organizational model to Software Product Lines in SI

The work objects of the previous processes can be combined into a product line template, which will be instantiated by a particular software product line. This rather abstract layer for industrialized software development is expected to have a positive effect on the previously mentioned major issues of industrialized systems integration. The first concern, integrating products from different product lines, may be solved by the Portfolio Definition and Architecture & Roadmap Definition processes. The abstract architecture applicable to all software product lines ensures compatibility of products within a given business domain. The second one, multiplicity of technologies, can be alleviated by a joint technology roadmap. It will limit the number of utilized technologies within the software product lines and thus reduce their heterogeneity. This does not reduce the heterogeneity introduced by legacy systems or third party applications. The latter may however be improved by joint interface components across multiple product lines. The third concern, ensuring the return of investment, can also be attenuated. Software product line engineering may instantiate the predefined skeleton and has a greatly reduced effort in the processes Business Domain Analysis, Business Domain Architecture, Architecture Design & Development, and Core Asset Development. Due to reduced efforts and thus cost, the breakeven point of a SPL may be reached earlier. Although this approach may not be as efficient as traditional software product lines, the author assumes that it still helps to advance their economical feasibility in SI and that it will have a positive effect on the overall product quality.

As implementing the business domain layer is a singular and novel undertaking, work is decomposed based on work objects. Thereby the processes Business Domain Analysis and Portfolio Definition are combined due to a presumably close interaction. Architecture Development and Roadmap Definition, as well as Core Asset Development remain separate as they only rely on their predecessor's outcomes but do not significantly influence them. Based on the nature of the core assets to be developed, it is also conceivable to break it down into different teams. These teams may then be responsible for particular assets throughout their lifetime and also take over their maintenance.

The resulting structure of the three layered approach is depicted in figure 2 (a more detailed view is attached to the end of this document). It should be noted that the Product Line layer does no longer contain the Business Domain Analysis Process from Software Product Line Engineering and also reduces the responsibilities of the Domain Requirements Engineering Process. These functions are now incorporated in the Business Domain Layer and provide their findings to the subsequent product lines. All other processes remain the same but must adhere to the specifications and utilize the provided core assets from the business domain layer. The internalization and thus organizational structure of the remaining product line engineering processes remains the same.

## V. COMPONENT BASED DEVELOPMENT IN SI

Similar to the above, the ability to successfully implement CBD is limited. In SI projects, a great variety of combinations from different technologies, business processes, or regulatory requirements are possible. Considering the fact that most system integrators are active in multiple industries with multiple customers, chances that one project is similar to another are extremely small. The encapsulation of business and application logic into standardized and reusable units of composition thus seems problematic. To overcome these limitations, upfront investments must be reduced and the ability of reuse increased. Different implementation approaches are available and range from rather technical to business level concepts. One of the latter is Herzum and Sims' Business Component Factory [13]. It is a methodology to model, analyze, design, construct, validate, deploy, customize, and maintain large scale

distributed systems. The constituent parts of the approach can be easily mapped to the organizational model from above, leading to reduced efforts due to economies of scale and scope. A mapping with the integration metamodel furthermore helps to reduce heterogeneity and alleviate integration efforts for products from different product lines. The approach primarily describes architectural viewpoints, component granularity, development processes, distribution tiers, and functional layers as core features of their model. They will be mapped in the following.

A. Architectural Viewpoints

Four architectural viewpoints define the execution environment, development patterns and standards, functional design and scope, as well as organizational decisions, including tools and guidelines for component based development. The project management architecture covers architectural and organizational decisions, tools, and guidelines. Such assets are also part of software product lines and considered core assets in software product line development [4]. They will be implemented within the business domain layer, which ensures the interoperability of different systems and product lines within the business domain.

The second viewpoint is concerned with the technical architecture. It defines the fundamental infrastructure required to implement and operate a component based application. With reference to software product line engineering, very similar activities are found that can be summarized under the term architecture design & development [23; 4; 19]. In the organizational model for industrialized SI, the business domain layer defines mandatory technologies, architectures, and systems. These artifacts will then be further refined within the actual product line. Such a joint technical architecture ensures the interoperability of different systems and product lines, reduces technical heterogeneity, and helps to achieve a positive return on investment by consolidating architectural efforts. The technical architecture of the business component factory is therefore aligned with the business domain and SPL engineering layer of the organizational model for industrialized SI.

The application architecture as the third viewpoint covers architectural decisions, patterns, guidelines, and standards required to build a component based system. Such can be architectural principles (e.g. noncircularity) and styles (e.g. type-based vs. instance-based), collaboration patterns for transactions, and system wide error handling mechanisms [13]. With regard to the organizational model for industrialized SI, a single application architecture is too specific for a whole business domain due to the variety of many different product lines and products. The application architecture is therefore aligned with the software product line layer.

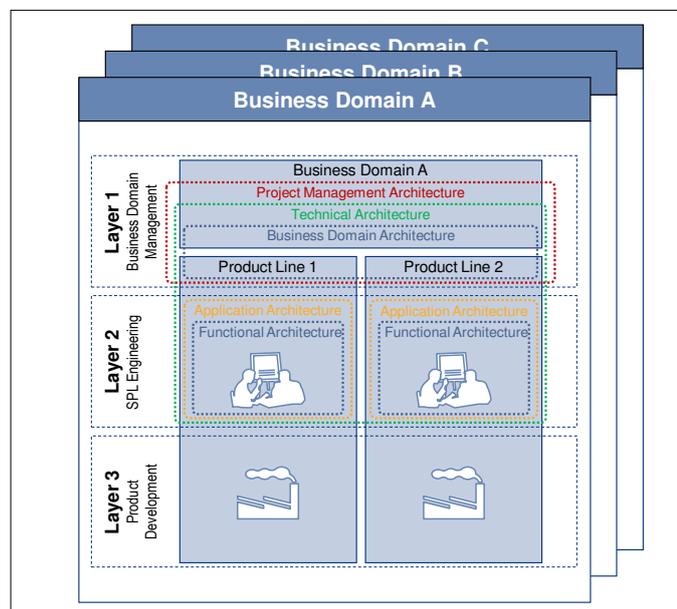


Fig. 3. Architectural Viewpoint Alignment

The functional architecture as the most detailed viewpoint covers the functional aspects, including specification and implementation according to the customer’s needs [13]. It consists of two key processes: component-based business modeling and component-based design. The former can be further broken down into business modeling and functional modeling. The business modeler produces a model of the problem space. The functional architect aims to support the production of a software application. In the organizational model, business modeling represents the ‘business domain analysis & portfolio definition’ process within the business domain layer. Functional modeling in turn, fits into the ‘architecture design &

development' process within the SPL engineering layer, as does the second core process of the functional architecture, i.e. component-based design. It is therefore suggested to separate business modeling from the rest of the functional architecture and align it with the business domain layer of the organizational model. Functional modeling and component based design should be aligned with the product line engineering layer to define the functional architecture of the software product line.

**B. Component Granularity**

The business component approach differentiates five levels of granularity. The present work omits the smallest (a language class) and largest (a federation of systems) one as the former is not independently deployable and the latter is too large to be reused in an SI context. The first feasible granularity is the distributed component. They may for instance be implemented as EJB, CORBA, or DCOM and represent simple functionality such as database connectivity or an input dialogue. Several distributed components form a business component defining an autonomous business concept (entity or process), such as a credit verification process. A set of business components collaborating together and delivering a business solution is a business component system representing the third level of granularity.

Allocating the above based on the entity types of the integration metamodel does not make sense. A distributed or business component may be used in any entity type. Component granularity is therefore aligned with the organizational model which allows clear responsibilities for each component, as demanded by the approach itself [13]. Additionally, definition and refinement between the different layers of the organizational model become possible, i.e. one of the key benefits of software product lines in SI. The levels of granularity are thus extended to differentiate between local and global components. Global components are developed and maintained on the Business Domain Layer. They provide reusable functionality for all or some of the underlying software product lines. Good examples for global business components are logic entities of a certain business domain, such as invoice, order, bill of materials, or product. In addition, global distributed components may provide standardized interfaces between different product lines. Making some of the global components mandatory ensures compatibility between products from different product lines. In turn, local business and distributed components are developed and maintained on the software product line level. They are either developed individually, or partially inherited from the business domain. In both cases, they represent a business concept or functionality which is unique for the respective product line. Among other non tangible assets, an appropriate choice of local and global distributed and business components represent the reusable core assets of a software product line. Figure 4 illustrates.

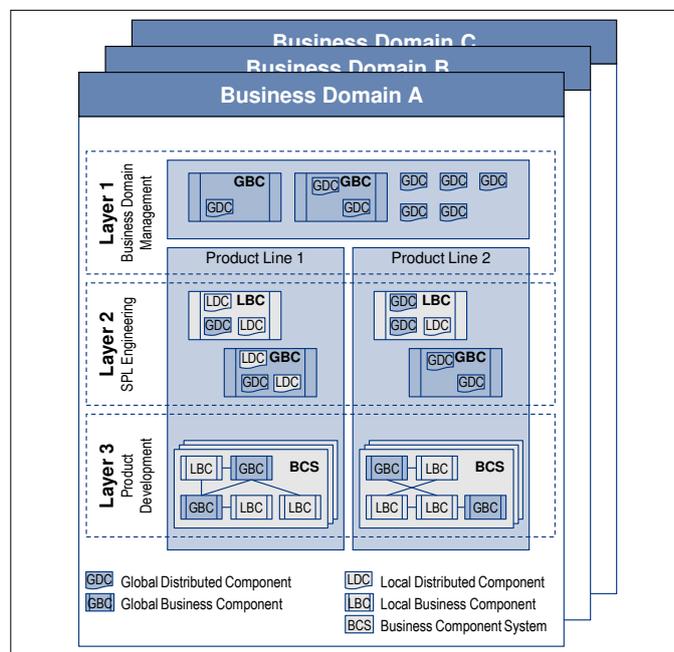


Fig. 4. Business Component Alignment

**C. Development Process**

The three-part development process of the approach defines the chronological sequence in which the activities of the other four dimensions are carried out. It consists of rapid component development, covering definition, building, and testing of individual business components. Due to the separation between global and local components, the process is allocated to both, the business domain as well as the

SPL engineering layer of the organizational model. System architecture and assembly represents the second manufacturing process and covers “architecting, assembling, and testing a system using business components” [13]. It selects, adapts, and deploys already existing distributed and business components based on a predefined architecture according to customer specific requirements. It is the actual manufacturing process in its original sense and therefore aligned with the production layer of the organizational model. Federation architecture and assembly is the most advanced activity in business component based development. It designs, assembles, and tests a federation of system level components, i.e. complete business component systems according to customer specific needs. This last part of the development process is therefore aligned with the production layer as well, although far from being trivial and depending on several years of experience [13].

#### D. Distribution Domains

Herzum and Sims differentiate four different distribution tiers, which are user, workspace, enterprise, and resource. They are grouped into the user workspace, and the enterprise resource distribution domain. This is because local and enterprise wide functionality is usually separated from each other and treated differently in large-scale systems. The user workspace domain is responsible to “support a single human being’s view of system facilities through some user interaction/interface technology” [13]. The enterprise-resource domain in turn, implements “a set of computing facilities within which state changes to important (probably concurrently shared) resources can reliably be made” [13]. As this architectural viewpoint is more focused on the logical structure of a component based system and how to scope and distribute functionality, aligning it with the organizational model would not be appropriate. Here Vogler’s integration metamodel is most suitable, which can be taken as a foundation when planning and designing an integrated system [30].

The metamodel defines similar layers or tiers, which are desktop, process, and systems integration. Desktop integration takes distinct tasks as its reference point. It is responsible to present them to the user, interact with him, exchange data with other tasks, and provide an interface to enterprise applications and resources where required [30]. This exactly describes, in other words, the responsibilities of the user workspace domain, which is also responsible to interact with the user and provide local business functionality. However, there may be circumstances in which the metamodel entities workflow, activity, control data, and state also find their way into the user workspace domain. This is the case if an end user application independently represents a complete business process or workflow, including local process integration activities.

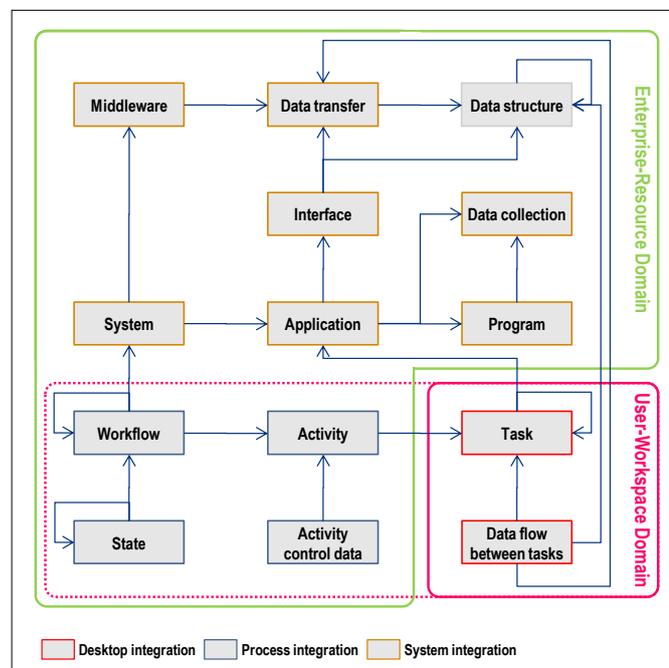


Fig. 5. Distribution Domain Alignment

The process integration layer of the integration metamodel is concerned with process control by defining one or more workflows, which are hierarchically structurable and consist of different activities and transactions [30]. Such processes usually involve multiple other enterprise resources and end users. The system integration tier is concerned with interfaces and data transfers between different applications and resources. It may also use a middleware, for instance, to support such activities [30]. With reference

to the business component model, the enterprise tier implements “enterprise-level business rules, validation and interaction between enterprise components, and it also manages the business aspects of data integrity” [13]. These activities reflect those of the process integration tier of the integration metamodel. The resource tier “manages the physical access to shared resources” [13] and shields the business logic from technical aspects. They represent the activities of the system integration tier. Figure 5 illustrates.

E. Functional Categories

As with distribution domains, functional categories describe a concept realized separately for each product line and product. They are “concerned with the functional aspects of the system, including the actual specification and implementation of a system that satisfies the functional requirements” [13]. The business component factory defines three functional categories for business components, which are utility business components, entity business components, and process business components.

The first defines supporting concepts available to other components to be used in a variety of systems. Examples are a middleware system, print services, or a currency converter. As such they are aligned with parts of the systems integration layer of the integration metamodel. As the model does not differentiate between technical and functional aspects, mapping with system, middleware, and data transfer entities is suggested, although a wide variety of other utility business components is conceivable (but not represented in the metamodel). Entity business components represent logical concepts and entities within a system and may contain persistent data. Such concepts are usually specific to a business domain [13] (e.g. the automotive industry) or a product line (e.g. a product line for shop floor solutions). Examples are a work order or bill of materials.

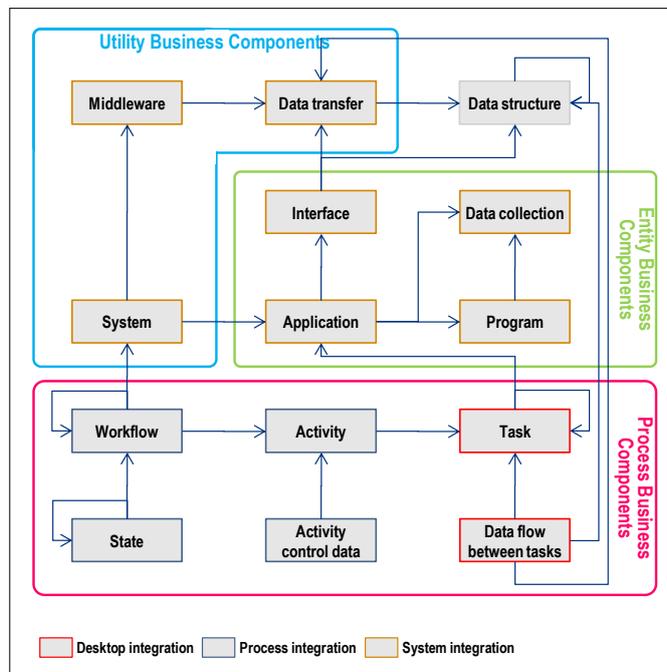


Fig. 6. Functional Category Alignment

With regards to the integration metamodel, a precise point of demarcation cannot be found. It is therefore assumed that applications and programs (together with their data collections and interfaces) can be seen as system level components as described in section B. For new systems, however, these would be replaced by according entity business components, which rely on utility components for data transfer and are being utilized by process business components. “Process business components define a business process as a workflow with different activities and tasks to be performed, and also control these to ensure process completion [13]. Such processes are very customer specific and can hardly be reused. Examples are invoice processing or order management. Process business components are thus aligned with the metamodel entities of the process and desktop integration layer. Both together exactly represent the responsibilities of the process business component category by defining workflows, activities, and tasks together with their respective control data and data flows.

VI. MODEL DRIVEN ENGINEERING IN SI

The final step towards fully industrialized systems integration is the implementation of automated software development, represented by MDE. Different implementation approaches exist or are being discussed in literature. The most prominent ones are Model Driven Architecture (MDA), which is an approach from the Object Management Group based on a separation of functional and technical concerns [21]. It uses UML as its modeling language to specify different levels of abstraction between the business need and the actual implementation. Code generators and model transformation engines allow transforming models into more specific ones and eventually generating source code. Based on the work of Czarnecki and Eisenecker [5], Generative Programming (GP) aims at automating development within a software product line. It defines a problem space expressed by a Domain Specific Language and a solution space consisting of implementation-oriented abstractions instantiated to implement the specifications expressed in the problem space [5]. The mapping between both contains the configuration knowledge such as illegal feature combinations, default settings, or construction rules. These rules are implemented within a generator returning the solution space, which may either be an intermediate model or executable program code. Software Factories is an approach similar to GP which uses SPLs and CBD along with a highly customized IDE. It is based on schemes which define the products to be developed from different viewpoints. A scheme in turn is part of a template which can be loaded into an IDE and provides wizards, patterns, frameworks, templates, domain specific languages, or editors. DSLs furthermore allow (semi-) automatic model to model transformations and code generation.

In contrast to MDA, GP has a domain oriented focus which is usually found in SPLs. Additionally it allows creating DSL, generator, and other assets as necessary during regular software development. This reduces high upfront investments and leads to exactly tailored artifacts. In contrast to GP and MDA, Software Factories are currently based on proprietary IDEs and modeling frameworks from Microsoft. Furthermore, most of the infrastructure needs to be in place before software development may start, leading to high upfront investments. For its flexibility and domain oriented focus, GP was chosen for further advancement of industrialized SI.

#### A. Generative Programming and Software Product Lines

Software Product Lines and Component Based Development already cover large parts of the GP processes. In the following the remaining adjustments to incorporate GP in the organizational model are described.

The Business Domain Layer of the organizational model was developed to align domain wide functionality and utilize economies of scope originating from similar concepts and core assets among different product lines of a given domain. It therefore contains the processes domain analysis & portfolio definition, architecture development & roadmap definition, and core asset development. As to Generative Programming, these processes already cover the GP processes (q.v. figure 7) 1 and 2, such as development of a domain or feature model. Furthermore, the activities of GP processes 3 and 4 are already enclosed in Architecture Development & Roadmap Definition, and Core Asset Development. However, as the Business Domain Layer only features concepts suitable for more than one product line, it must differentiate between global (business domain wide) and local (product line specific) aspects of GP. This means that there will for instance be DSL design activities in both, the Business Domain and the Product Line Layer. In the former, the overall structure and domain wide syntax and semantics are defined, whereas the latter covers product line specific syntax and semantics, such as bill of materials for a shop floor system produced in a particular software product line. The distribution is illustrated in Figure 7.

The Software Product Line Layer consists of several software product lines identified in business domain analysis and portfolio definition processes of the business domain layer. The most obvious variance to a conventional software product line is the lack of the business domain analysis process, and a simplified domain requirements engineering process. These functions are now incorporated in the business domain layer and provide their findings to the subsequent product lines. All other processes remain the same but must adhere to the specifications and utilize the provided core assets from the business domain layer.

As to Generative Programming, we can find all but the first development process within the Software Product Line Layer. However, due to the separation of domain wide and product line specific concerns, the GP processes 2 to 4 only handle product line related concerns. A systems integrator's feature model for the automotive industry may for instance define the entity car with several features, such as model, engine, transmission, color, price, owner, and so on. These features exist in all products of the underlying product lines. A product line for shop floor systems may however extend this feature model by adding features like electronic control unit (ECU) type, brake type, or parts list. As this has no implication on the functionality of the car itself or the customer, these features are not necessary to be known in other product lines. A financial system does not need to know what type of ECU is built into a car, but it does need to know the price and the owner of the car. This same principle applies to Common Architecture &

Component Definition and Domain Specific Language Design. GP processes 5 to 8 are carried out in the software product lines only.

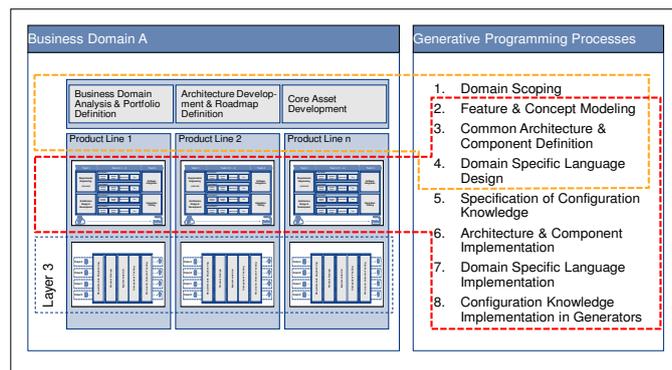


Fig. 7. GP Process allocation

### B. Generative Programming and Business Component Factory

Given that development of components occurs with GP, the following sections will show how the respective processes fit together with the dimensions of the business component factory.

**Architectural Viewpoints:** GP does not include aspects of the project management architecture (PMA). These are however included in the organizational model within the Business Domain Layer, whose organizational decisions, guidelines, and tools influence the development in GP. The remaining three, rather technical viewpoints, are concerned with the execution infrastructure and programming frameworks (Technical Architecture), development patterns, guidelines, and programming standards (Application Architecture), as well as the functional aspects of a system including its implementation (Functional Architecture). Generative Programming in turn only offers the generic process common architecture & component definition. It is therefore suggested to replace the respective GP process with the actual implementation of the much more detailed architectural viewpoints from the Business Component Model. For GP, this replacement offers a more comprehensive view on different aspects of the architecture, while for CBD it ensures coverage of more component related artifacts, such as the component infrastructure or execution environment.

**Component Granularity:** GP does not explicitly refer to well defined components as EJBs or Corba components. Also it doesn't conceptually concentrate on business processes and therefore does not know reasonable levels of granularity. It rather concentrates on technologies and means to develop reusable artifacts of variable sizes, depending on the intended usage. The Business Component model in turn follows a more distinctive approach. Its five levels of granularity and thus discrete partitioning of the problem is more beneficial in an environment with systematic reuse. For each layer of recursion, a developer has to define scope, characteristics, packaging, and deployment [13]. In an environment where components are to be reused as much as possible, it seems more beneficial to define these layers of recursion on a common basis. A middleware messaging adaptor for a specific ERP system will most likely exist as a distributed component as introduced above. A developer can rely on this concept and build his application accordingly. It is therefore suggested to introduce discrete recursion to the Generative Programming approach if it is to be used with CBD and systematic reuse in mind.

**Development Process:** The Business Component Model encompasses a set of manufacturing processes called rapid system development (RSD). It is following the well known V-Model, whereas requirements to implementation denote the left, and component, system, and acceptance testing the right side of the V [13]. RSD allows subsequently engineering reusable artifacts and building the respective end product. The advantage is that reusable artifacts evolve on the fly. The disadvantage is that, beginning with customer specific requirements, one may easily miss important variation points or even take architectural decisions which may conflict with the overall scope of the product line. GP in turn focuses much more on domain engineering activities and the technical implementation of reusable artifacts, rather than development of the end product. It puts explicit focus on feature modeling processes such as FODA or FeatuRSEB [2], as all GP artifacts rely on a detailed domain model. As research in the field has progressed, we also considered PLUSS (Product Line Use Case Modeling for Systems and Software engineering) [6] being a viable alternative for precise domain modeling. Following GP it is thus suggested to define a precise domain model before implementation. This seems especially important if DSLs and generators are to be built, although they will be rather simple in the beginning. It is therefore suggested to enhance the Requirements, Analysis, and Design activities of Herzum and Sims' rapid system development process with one of the above feature modeling methods.

Distribution Tiers: In their model, Herzum and Sims separate between user, workspace, enterprise, and resource tier, defining which activities are being handled by which components. Such detailed differentiation of reusable components and their internal structure is not provided by the Generative Programming approach. Being more generic, GP leaves such decisions to the target architecture of the product line, which is in turn depending on the overall feature model [2]. With regard to the Business Component model, feature model and architecture will already be available and are furthermore influenced by the conceptual structure of business components. In combination with GP, no issues can be conceived when implementing the four distribution tiers by means of Generative Programming.

Functional Categories: The final dimension defines utility, entity, process, and auxiliary business components [13] and describes the functional scope of business components. As with the distribution tier above, Generative Programming does not know any functional categories. However, a detailed feature model in connection with component granularity, distribution tiers, and functional categories, will provide a structured and standardized approach to generative development of business components. As such it is assumed that systematic reuse is more likely to be achieved than with a structure that is flexible from component to component or across software product lines.

With regards to domain specific languages and code generators, GP offers various alternatives. They all have in common that they are rather complex as they allow almost any level of detail to model software. Given that MDE and especially its tool support and standardization are still considered immature [26], modeling on such a level of detail in a field as systems integration seems pointless. As an alternative we suggest to lift the modeling approach up one level and concentrate on the assembly of distributed and business components in a predefined architecture. The approach can be compared with an extensive configuration framework for a software product line's products. Based on a particular customer's requirements the variability model of the product line is instantiated and the respective components are automatically being assembled. Where this is not possible due to deviating requirements, conventional software development from scratch (at the respective cost) is being applied. This concept removes large amounts of complexity from model driven engineering and allows for easier implementation of the domain specific languages and code generators.

During the present research XML and its corresponding languages were identified as viable alternative to proprietary DSL tools and techniques. With XML it is easily possible to model an application based on customer requirements and in accordance with the product line's architecture and asset base. To ensure that only valid elements of the domain specific language are being used and that the application adheres to the overall architecture, XML Schema Definitions (XSD) are employed. They define the 'grammar', i.e. what is allowed and what's not, of the domain specific language. With these two concepts it is possible to create formal and context free models of applications ready to be processed by code generators. Code generation in turn is achieved with XML Stylesheet Transformations (XSLT). An XSLT processor parses an XML document and based on the contents of the XSLT document it replaces model elements with text modules. An element describing a certain feature for instance, will be replaced by the respective source code. The approach will also handle attributes, data types, and other aspects being defined in the model. A multi staged generation process will furthermore ensure the creation of various different artifacts such as compilation, deployment, and installation scripts.

## VII. CONCLUSION AND FURTHER RESEARCH

As explained in section II, systems integration comes with certain characteristics requiring a highly efficient and cost effective way of implementing the industrial key concepts of specialization, standardization, and automation. The low number of similar products in SI seems contradictory to Software Product Lines, Component Based Development, and Model Driven Engineering. Current implementations of these concepts require high upfront investments and only work for products being very similar to each other. To overcome these limitations, SPLs, CBD, and MDE have been analyzed for their shortcomings with respect to systems integration. For each of these concepts, alternative or adaptations to existing approaches for implementation have been developed. It was furthermore ensured that the adjusted implementation approaches are consistent with each other.

The novel and comprehensive approach to industrialized systems integration consists of the following three parts: The organizational model for software product lines, reflecting specialization as the first industrial key principle. Subsequently, the alignment of Herzum and Sims' Business Component Model with Vogler's Integration Meta Model describing how to divide a system into a set of reusable artifacts. Generative Programming has been identified as a potential way towards automation as the final industrial key principle. The family of XML languages serves as an alternative to today's proprietary and immature domain specific language and code generation concepts.

Continuing the present research it would be interesting if industrialization enhances an organization's

ability to outsource software development to near- and offshore countries. A look at other industries shows that these usually industrialized their production before outsourcing. It is assumed that due to higher standardization and process maturity, offshore outsourcing will be much more efficient. In addition to outsourcing, other software development techniques, such as agile development, should be researched upon their applicability to industrialized software development. It seems questionable if such an approach goes well with well structured industrial development.

#### VIII. REFERENCES

- [1] Bahli, B.; Ji, F.: An assessment of facilitators and inhibitors for the adoption of enterprise application integration technology: An empirical study. In *Business Process Management Journal*, 2007, 13; P. 108–120.
- [2] Czarnecki, K.; Eisenecker, U.: *Generative programming. Methods, tools, and applications*. Addison Wesley, Boston, 2000.
- [3] Conrad, S.; Hasselbring, W.; Koschel, A.: *Enterprise Application Integration. Grundlagen Konzepte Entwurfsmuster Praxisbeispiele*. Elsevier Spektrum Akad. Verl., München, Heidelberg, 2006.
- [4] Clements, P.; Northrop, L.: *Software product lines. Practices and patterns*. Addison-Wesley, Boston, 2007.
- [5] Czarnecki, K.: Overview of Generative Software Development. In (Banâtre, J.-P.; Fradet, P.; Giavitto, J.-L.; Michel, O. Ed.): *Unconventional Programming Paradigms*. Springer Berlin / Heidelberg, 2005; P. 97-97.
- [6] Eriksson, M.; Börstler, J.; Borg, K.: Software Product Line Modeling Made Practical. In *Communications of the ACM*, 2006, 49; P. 49–53.
- [7] Fischer, J.: *Informationswirtschaft Anwendungsmanagement*. Oldenbourg, München, Wien, 1999.
- [8] Frank, U.: Standardisierungsvorhaben zur Unterstützung des elektronischen Handels: Überblick über Anwendungsnahe Ansätze. In *Wirtschaftsinformatik*, 2001, 43; P. 283–293.
- [9] Frankel, D.: *Model driven architecture. Applying MDA to enterprise computing*. Wiley, New York, 2003.
- [10] Greenfield, J.; Short, K.; Cook, S.: *Software factories. Assembling applications with patterns, models, frameworks, and tools*. Wiley, Indianapolis, Ind., 2004.
- [11] Gorton, I.; Thurman, D.; Thomson, J.: Next generation application integration: challenges and new approaches: Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003. IEEE Comput. Soc, 2003; P. 576–581.
- [12] Hasselbring, W.: Information System Integration. In *Communications of the ACM*, 2000, 43; P. 32–38.
- [13] Herzum, P.; Sims, O.: *Business component factory. A comprehensive overview of component-based development for the enterprise*. John Wiley, New York, 2000.
- [14] Hahn, H.; Turowski, K.: Drivers and inhibitors to the development of a software component industry. In (Crnkovic, I. Ed.): *Proceedings of the 20th IEEE Instrumentation Technology Conference EURMIC-03*. IEEE, 2003; P. 128–135.
- [15] Hutchinson, J. et al.: Empirical assessment of MDE in industry. In (ACM Ed.): *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. ACM Press, 2011; P. 471–480.
- [16] Janßen, R.: Die Psychologie des Entwicklers. In *Informatik-Spektrum*, 2005, 28; P. 284–286.
- [17] Lui, M. et al.: *Pro Spring integration*. Apress, New York, NY, 2011.
- [18] Linthicum, D. S.: *Enterprise application integration*. Addison-Wesley, Reading, Mass, 2000.
- [19] Linden, F.: *Software product lines in action. The best industrial practice in product line engineering*. Springer, Berlin, Heidelberg, New York, 2007.
- [20] Leser, U.; Naumann, F.: *Informationsintegration. Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*. dpunkt-Verl., Heidelberg, 2007.
- [21] Object Management Group: *MDA Guide Version 1.0.1*, 2003.
- [22] Puschmann, T.; Alt, R.: Enterprise Application Integration - The Case of the Robert Bosch Group. In (IEEE Computer Society Ed.): *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, Washington DC, 2001; P. 1–10.
- [23] Pohl, K.; Böckle, G.; Linden, F.: *Software product line engineering. Foundations, principles, and techniques ; with 10 tables*. Springer, Berlin, 2005.
- [24] *Software and IT Services Industry (SITSI) Report*, Paris, 2009.
- [25] Riehm, R.: *Integration von heterogenen Applikationen*. Dissertation, St. Gallen, 1997.
- [26] Selic, B.: Personal reflections on automation, programming culture, and model-based software engineering. In *Automated Software Engineering*, 2008, 15; P. 379-391.

- 
- [27] Szyperski, C.; Gruntz, D.; Murer, S.: Component software. Beyond object-oriented programming. ACM Press; Addison-Wesley, New York, London ;, Boston, 2002.
  - [28] Santos, R. P. dos; Werner, C. M. L.: Revisiting the concept of components in software engineering from a software ecosystem perspective. In (Cuesta, C. E. Ed.): Proceedings of the Fourth European Conference on Software Architecture. ACM Press, New York, 2010; P. 135–142.
  - [29] Themistocleous, M. et al.: ERP problems and application integration issues: an empirical survey: Proceedings of the 34th Annual Hawaii International Conference on System Sciences. IEEE Comput. Soc, 2001; P. 10.
  - [30] Vogler, P.: Prozess- und Systemintegration. Evolutionäre Weiterentwicklung bestehender Informationssysteme mit Hilfe von enterprise application integration. Dt. Univ.-Verl., Wiesbaden, 2006.

## A.5 Expert Interview Guideline

The following guideline has been used to structure and guide the expert interviews described in chapter 6. For easier reference it has been translated to English, although the German version had been used during the interviews.

Interviewleitfaden	Interview Guideline
<p>Ziel des Interviews ist die Evaluierung des zuvor theoretisch ausgearbeiteten und in einer Fallstudie getesteten Konzepts zur Industrialisierung der Systemintegration. Hierbei geht es um jene Aspekte die im Rahmen eines Laborexperiments nicht nachgestellt werden können. Diese umfassen insbesondere organisatorische Veränderungen eines bestehenden Unternehmens, kulturelle Vorbehalte verschiedener Interessensgruppen und den Umgang damit, sowie die ökonomischen Erfolgsaussichten einer solchen Unternehmung. Das Interview orientiert sich am dem Interviewpartner zuvor zur Verfügung gestellten Aufsatz „Software Industrialization in Systems Integration“.</p> <p><b>Begriffsabgrenzung</b></p> <p>Zur Sicherstellung eines gemeinsamen Verständnisses müssen vor Beginn des Interviews folgende Begriffe definiert werden:</p> <ul style="list-style-type: none"> <li>• <b>Industrialisierung:</b> Beschreibt die Einführung von Methoden die der Spezialisierung, Standardisierung, und Automatisierung von Produktionsprozessen dienen. Sie wird als notwendiger Schritt für ökonomisches Wachstum, technologischen Fortschritt und gesellschaftlichen Wohlstand verstanden.</li> <li>• <b>Systemintegration:</b> Beschreibt die Entwicklung von Geschäftsanwendungen oder Tools die kundenspezifische Geschäftsprozesse abbilden und hierzu verschiedene interne oder externe Informationssysteme nutzen. Dies umfasst auch die Entwicklung von Anwendungen die eine solche Zusammenarbeit ermöglichen (z.B. Middlewares).</li> <li>• <b>Software Product Lines:</b> Beschreibt die Spezialisierung einer organisatorischen Einheit auf die Entwicklung von Softwareprodukten mit gemeinsamer Architektur und untereinander wiederverwendbaren Artefakten. Kundenanforderungen müssen innerhalb dieser Architektur ab-</li> </ul>	<p>Objective of the interview is the evaluation of the previously theoretically developed and case-tested concept for the industrialization of systems integration. It relates to those aspects not possible to be tested in a laboratory experiment. They especially include organizational changes of an existing enterprise, cultural scepticisms of different stakeholder groups and the handling of these scepticisms, as well as the potential for economic success of such an undertaking. The interview orientates itself on the previously provided paper „Software Industrialization in Systems Integration“.</p> <p><b>Delineation of terms</b></p> <p>To ensure a common understanding, the following terms must be delineated in advance to the interview:</p> <ul style="list-style-type: none"> <li>• <b>Industrialization:</b> Describes the introduction of methods serving specialization, standardization and automation of production processes. They are seen as necessary step for economic growth, technological advantage and societal wealth.</li> <li>• <b>Systems Integration:</b> Describes the development of business applications or tools implementing customer specific business processes, utilizing different internal or external information systems. This also includes the development of applications enabling such a collaboration (e.g. middlewares)</li> <li>• <b>Software Product Lines:</b> Describe the specialization of an organizational unit to the development of software products with a joint architecture and among each other reusable artefacts. Customer requirements must be depicted within this architecture or developed externally.</li> </ul>

<p>gebildet oder extern entwickelt werden.</p> <ul style="list-style-type: none"> <li>• <b>Component Based Development:</b> Beschreibt die Entwicklung von Anwendungssystemen mit Hilfe wiederverwendbarer Komponenten. Eine Komponente ist ein eigenständig (ggf. innerhalb einer Laufzeitumgebung) verteil- und ausführbares Softwareartefakt welches über klar definierte Schnittstellen eine bestimmte Funktionalität erbringt.</li> <li>• <b>Model Driven Engineering:</b> Beschreibt die Modellierung von Anwendungssystemen in einer für ein bestimmtes Gebiet entwickelten Sprache auf einem zu Beginn hohen Abstraktionslevel. Das resultierende Modell wird anschließend automatisch oder halbautomatisch in detailliertere Modelle transformiert bevor letztendlich ausführbarer Programmcode generiert wird.</li> </ul> <p><b>Interviewfragen</b></p> <p><b>Einstiegsfragen</b></p> <ol style="list-style-type: none"> <li>1. Beschreiben Sie Ihre Rolle im Unternehmen, wie kann ich mir ein typisches Projekt Ihrer Abteilung vorstellen?</li> <li>2. Wie stark ist Ihre Abteilung im Bereich Systemintegration involviert, haben Sie täglich damit zu tun?</li> <li>3. Wie würden Sie die typischen Probleme der Systemintegration beschreiben, wo stoßen Entwickler auf Herausforderungen?</li> </ol> <p><b>Kategorie 1: Paradigmenwechsel bei Anbieter und Kunde</b></p> <ol style="list-style-type: none"> <li>4. Halten Sie es für möglich eine Produktlinie an der zukünftig erwarteten Marktentwicklung auszurichten und auch vorab darin zu investieren?</li> <li>5. Glauben Sie das Kunden bereit sind auf bestimmte Funktionalität zu verzichten um noch innerhalb der Produktlinie zu bleiben und damit Kostenvorteile zu erreichen?</li> <li>6. Wie sehen Sie den organisatorischen Umbau der mit einer Industrialisierung der Softwareentwicklung einhergeht, insbesondere was die Arbeitsweise der Programmierer angeht?</li> </ol> <p><b>Kategorie 2: Ein Organisationsmodell für Software Product Lines in der SI</b></p> <ol style="list-style-type: none"> <li>7. Wie beurteilen Sie den neu entwickelten Business Domain Layer im Hinblick auf die Konsolidierung gleichartiger Tätigkei-</li> </ol>	<ul style="list-style-type: none"> <li>• <b>Component Based Development:</b> Describes the development of application systems with the help of reusable components. A component is an independently (where applicable in a runtime environment) distributable and executable software artefact which provides clearly defined interfaces for specified functionality.</li> <li>• <b>Model Driven Engineering:</b> Describes the modelling of application systems in a language developed for a particular business area on an initially high level of abstraction. Subsequently, the resulting model will be automatically or semi-automatically transformed into more detailed models until eventually executable code is generated.</li> </ul> <p><b>Interview questions</b></p> <p><b>Introductory questions.</b></p> <ol style="list-style-type: none"> <li>1. Please explain your role in your company, how can I envision a typical project in your department?</li> <li>2. How strongly involved is your department in the field of systems integration, do you have something to do with it on a daily basis?</li> <li>3. How would you describe the typical issues of systems integration, which challenges do the developers face?</li> </ol> <p><b>Category 1: A paradigm shift at supplier and customer</b></p> <ol style="list-style-type: none"> <li>4. Do you think it is possible to construct a product line according to future market developments and also invest in it upfront?</li> <li>5. Do you believe that customers are willing to waive certain functionality in favour of staying within the product line and thus achieving cost advantages?</li> <li>6. How do you think of the organizational changes coming with an industrialization of software development, especially considering the principles of work of software developers?</li> </ol> <p><b>Category 2: An organizational model for software product lines in SI</b></p> <ol style="list-style-type: none"> <li>7. How do you assess the newly developed business domain layer with regards to the consolidation of similar tasks of software</li> </ol>
---	---

<p>ten von Software Product Lines?</p> <p>8. Wie beurteilen Sie den angepassten Software Product Line Layer im Hinblick auf den verbleibenden Aufwand?</p> <p>9. Wie beurteilen Sie die Möglichkeit Anwendungen zur Systemintegration im Rahmen des vorgestellten Modells zu entwickeln?</p> <p><b>Kategorie 3: Component Based Systems Integration</b></p> <p>10. Halten Sie es für sinnvoll mehrfach benötigte Funktionalität in Form von wiederverwendbaren Komponenten abzubilden?</p> <p>11. Halten Sie eine System- und Komponentenarchitektur gemäß einem für alle Produktlinien verbindlichen Metamodell der Integration für sinnvoll?</p> <p><b>Kategorie 4: Model Driven Systems Integration</b></p> <p>12. Wie beurteilen Sie den Stand modellgetriebener Softwareentwicklung zum heutigen Zeitpunkt?</p> <p>13. Halten Sie es für sinnvoll die Merkmalsausprägungen eines Produkts innerhalb einer Produktlinie mit Hilfe einer einfachen domänenspezifischen Sprache zu beschreiben und anschließend automatisiert Artefakte zu generieren?</p> <p>14. Erwarten Sie durch diesen Ansatz Effizienzvorteile?</p> <p><b>Kategorie 5: Ökonomische Betrachtung der Industrialisierung in der SI</b></p> <p>15. Wie beurteilen Sie das Kosten-Nutzen-Verhältnis des vorgestellten Konzepts?</p> <p>16. Gibt es einzelne Aspekte durch die dieses Verhältnis positiv beeinflusst werden könnte? (z.B. Herauslassen des einen oder anderen Ansatzes)</p> <p><b>Ausstiegsfragen</b></p> <p>17. Gibt es Ihrer Ansicht nach Punkte die im vorgestellten Konzept nicht oder nur unzureichend berücksichtigt wurden?</p> <p> </p> <p>Ich danke Ihnen für Ihre Zeit und die Bereitschaft Ihr Wissen zum Thema Softwareindustrialisierung in der Systemintegration zur Verfügung zu stellen.</p>	<p>product lines?</p> <p>8. How do you assess the adapted software product line layer with regard to the remaining effort?</p> <p>9. How do you assess the possibility to develop applications for systems integration within the presented model?</p> <p><b>Category 3: Component Based Systems Integration</b></p> <p>10. Do you think it is reasonable implementing repeatedly used functionality in reusable components?</p> <p>11. Do you think a system and component architecture according to a meta model obligatory for all product lines is reasonable?</p> <p><b>Category 4: Model Driven Systems Integration</b></p> <p>12. How do you assess the current state of model driven software engineering as of today?</p> <p>13. Do you think it is reasonable describing features and variabilities of a product within a product line with the help of a simple domain specific language and subsequently automatically generating artefacts?</p> <p>14. Do you expect efficiency gains from this approach?</p> <p><b>Category 5: Economic consideration of industrialization in systems integration</b></p> <p>15. How do you assess the cost-benefit ratio of the presented concept?</p> <p>16. Are there any aspects from which this ratio may be positively influenced? (e.g. by omitting one or the other aspect)</p> <p><b>Exit questions</b></p> <p>17. According to your point of view, are there any aspects which the presented concept did not or only insufficiently cover?</p> <p> </p> <p>Thank you for your time and the willingness to share your knowledge about software industrialization in systems integration.</p>
--	---

## A.6 Expert Interview Transcription

The following table contains the transcribed and interpreted expert statements. Column 1 indicates the variable a statement was allocated to and the indicator by which it was found. V02-1 for instance identifies a supplier readiness statement identified with the price sensitivity indicator. The reference column indicates the interview itself and the time code at which the expert statement can be found. E1 stands for (Minich, 2012a), E2 for (Minich, 2012b), and E3 for (Minich, 2012c). Further details may be found in the references section.

Variable	Expert statement	Reference
V01-1	Try to concentrate on product lines which provide high-value services for customers. No commodities, no mass products.	E1-07:07
V01-2	Generally customers are willing to accept functional trade-offs in favour of cost benefits.	E1-07:54
V01-2	In recent years, structural changes in customer organizations could be observed. Functional departments become more important and may insist on individual software instead of following an enterprise wide architecture.	E1-07:56
V01-2	Functional trade-offs are possible, although they depend on the governal strength of a central IT department.	E1-08:57
V01-2	Governance on customer side is a big issue. Functional departments have other interests than group wide departments. The latter may favour reduced cost from a joint integration architecture while the functional departments favour a broader functionality.	E2-08:15
V01-2	Customers are willing to reduce functionality in favour of lower prices. However, this is only possible where such functionality is at the discretion of the customer (may be difficult in the public sector due to laws and regulations, for instance)	E2-14:34
V01-2	The willingness of the customer to accept a limited functionality depends on the context. A customer active in the public sector may need to adhere to administrative directives which may not be changed easily. Others are more independent are thus willing to accept such limitations.	E2-29:47

<b>Variable</b>	<b>Expert statement</b>	<b>Reference</b>
V01-3	The willingness of customers for trade-offs is assumed to be rather low for business core processes.	E3-10:45
V01-3	The customer rather pays a higher price for specifically implemented features than adapting his business processes to the IT systems.	E3-12:44
V01-4	It is assumed that software product lines will concentrate on functional layers not visible to the end customer.	E3-13:12
V01-4	There will be no software product lines containing core business processes or business specific know-how of customers.	E3-13:43
V01-4	Core business processes will remain individual while supporting processes (e.g. IT, HR, or F/C) may benefit from software product lines.	E3-14:50 E3-14:39
V02-1	Projects should be analyzed for reusability in detail after their completion. If reusable artefacts are found, investments are made to develop them into components or small products.	E1-04:58
V02-1	Only very few industries promise sufficient potential for reuse in which systematic reuse in software product lines makes sense. In other cases an opportunistic approach is pursued	E1-05:42
V02-1	Market anticipation is already being conducted, also without software product lines.	E2-15:32
V02-1	Many strategic customers already follow a certain stack of technologies on which the supplier concentrates his services.	E2-16:02
V02-1	It is assumed that product lines evolve over time in a bottom up approach. Commonalities are identified and potential for reuse is exploited once it arises.	E3-07:56 E3-09:20
V02-1	Product lines will not be based on a top-down market anticipation with proactive implementation of reusable components.	E3-08:23 E3-09:54
V02-2	Willingness for organizational changes and investments depends on the management KPIs. A cost driven manager is more likely to implement industrial concepts than a utilization driven manager.	E1-12:07
V02-2	An incentive rewarding long term cost reduction must be implemented for the responsible managers.	E1-12:23
V02-2	Industrialization must be driven by higher level management, team leaders or department managers may not have sufficient influence to implement a paradigm shift.	E1-12:42

<b>Variable</b>	<b>Expert statement</b>	<b>Reference</b>
V02-2	Organizational support for industrial concepts is very important. By just describing the concepts they will not be used in the daily work.	E2-45:26
V02-2	An internal governance is absolutely necessary.	E2-46:03
V03-1	Cultural changes for software developers are generally a challenge but can be alleviated with management by objectives and incentives, such as awards, reuse measurement, and budget used to implement certain functionality.	E1-10:20
V03-1	The problem of a cultural change can be solved but care must be taken not to be too strict and thus encourage developers to optimize their personal reuse goals instead of implementing most efficiently	E1-11:38
V03-1	Developers voluntarily and proactively develop components and frameworks to be reused in future projects.	E2-18:42
V03-1	Developers are at risk to suffer from the “not-invented-here syndrome”. However, the resulting additional efforts from re-inventing functionality must be avoided to remain competitive in Germany.	E3-16:58
V03-1	Developers must be trained in systematic reuse and must obtain an acceptance of the concept.	E3-17:24
V03-1	The “not-invented-here syndrome” is a major risk in industrialization and may prevent its success.	E3-17:36
V03-1	The “not-invented-here syndrome” is not fully controllable. People will always try to include individual aspects, although not feasible.	E3-17:51
V03-1	One must be aware of the human urge for individualism and be able to control or handle it in order to successfully implement software industrialization.	E3-18:58
V03-2	Convincing developers from the advantages of industrialization is not enough, incentives are inevitable.	E1-12:27
V04-1	Bundling identical activities and functionality from product lines of the same domain is inevitable.	E1-13:39 E1-14:50
V04-1	Ensuring the correct allocation of core assets to the different layers is important and must be ensured by proper governance.	E1-14:10
V04-1	The feasibility of the business domain layer depends on the number of core assets equal to all software product lines.	E1-37:16
V04-1	Consolidating similar functionality of separate product lines depends on the customer segment in which the product lines are applied.	E2-22:00

<b>Variable</b>	<b>Expert statement</b>	<b>Reference</b>
V04-1	For smaller customer segments development occurs out of any product line and is consolidated in functional units specialized on particular technologies.	E2-23:12
V04-1	Besides functionality defined in industry wide standards heterogeneity between different customers is too large to benefit from systematic reuse.	E2-27:35
V04-1	Despite the assumptions made, integration across business domain boundaries may indeed take place. An automotive supplier is for instance integrated with a governmental employment office to report open positions or tax data.	E2-49:11
V04-1	The focus of integration efforts in the field of systems integration lies within one business domain.	E2-50:39
V04-1	The feasibility of the business domain layer highly depends on the specific context in which it is applied. It cannot be seen as a generic model for systems integration but has to follow a case-based reasoning.	E3-23:42
V04-1	The feasibility of the business domain layer depends on the number of product lines, the number of products being developed, and their size, for instance.	E3-23:54
V04-2	An alternative may be found in middleware technologies such as a service bus architecture.	E2-28:07
V04-4	Besides executable software components, core assets on the business domain layer should also include processes and procedures and architectures.	E1-38:28
V04-4	Efficiency gains are easier to achieve from joint development processes and architectures than from executable code.	E1-39:43
V04-4	In very large industry segments the customers themselves often define standards and specifications which, provided as reusable software components, can be seen as useful core assets. For these a central development and distribution does make sense.	E2-25:56
V05-1	As with the business domain layer, the benefits at the software product line layer highly depend on the context and cannot be generalized.	E3-24:24
V05-2	Deviations from the business domain layer must be worked against with incentives and the necessity to justify a decision.	E1-15:50

<b>Variable</b>	<b>Expert statement</b>	<b>Reference</b>
V05-3	The final decision whether or not using a central component should be left to the product architect, i.e. the software product line engineer.	E1-15:22
V05-3	Care must be taken in stipulating the use of central assets for software product lines. Too strict governance may lead to inefficiencies.	E1-15:36
V07-1	Reusable components most likely concentrate on very few business critical functionality.	E2-17:21
V07-2	In customer specific development it definitely also makes sense to modularize functionality in reusable components.	E1-18:44
V07-3	Developing supplier internal components to be reused in customer projects is a feasible and proven approach.	E2-19:25
V07-3	Reusable components should not only cover standardized interfaces but also process related functionality, i.e. micro-processes.	E2-28:23
V07-4	Reuse of existing business functionality is definitely possible and requested by customers.	E2-32:31
V07-4	Reusable components increase in their size and complexity and may often be found in web services.	E2-33:01
V07-4	Implementing functionality within reusable components is possible for supporting business processes only.	E3-27:46
V07-4	The reason for insufficient industrialization of core processes lies in the lack of standards on an organizational layer. This in turn results from insufficient process descriptions.	E3-28:49
V08-1	Adhering to an integration metamodel in industrialized software development depends on the number of products to be developed.	E1-19:11
V08-1	The suitability of an integration metamodel depends on the already existing customer infrastructure.	E2-35:09
V08-1	A standardized toolbox including an integration metamodel does make sense but must be tailored for each individual customer project.	E2-35:31
V08-1	Generally speaking, an integration metamodel does make sense and should be pursued.	E2-35:37
V08-1	On a smaller level, not visible to the customer, a metamodel definitely makes sense as it allows us to more efficiently provide our services at a higher quality to the customer.	E2-36:13
V08-1	The integration metamodel should more be seen as an ontology ensuring that all parties have the same understanding of systems integration.	E3-34:15

<b>Variable</b>	<b>Expert statement</b>	<b>Reference</b>
V08-1	The integration metamodel risks to be too generic to provide a benefit to implementing organizations.	E3-35:17
V08-2	If the integration metamodel is seen as a joint ontology on which architectures and systems are based on, then it does indeed make sense.	E3-36:36
V08-3	It would be more efficient to have an industry wide integration standard that customers request their suppliers to adhere to.	E1-19:29
V09-1	Only small experiments with model driven engineering were made, it is not yet a big topic.	E1-20:13
V09-1	Automated code generation is of very low significance only.	E1-22:23
V09-1	Model driven development for the enterprise the expert is employed at is currently limited to creating standardized UML models about the software to be developed.	E1-22:06
V09-1	MDE is assumed to be “dead” in its current state.	E2-37:55
V09-1	Model driven development approaches are not assumed to break through in the next at least five years as the world to be modelled is way too complex.	E2-38:52
V09-1	Another major reason for the possible fail of MDE is the possibility of very high heterogeneity in any field, not only systems integration.	E2-39:01
V09-1	MDE may be an option in very small and limited niche areas, but not in a broader context.	E2-39:26
V09-2	Several tools were tested to implement model driven development, but not regarded beneficial.	E1-20:41 E1-22:43
V09-2	For the time being model driven engineering highly depends on tool vendors. However, enterprises do not want to become dependent on a particular vendor.	E3-38:14
V09-2	Model driven engineering in a highly formalized and standardized context (e.g. medical, avionic) does make sense. In other areas such as finance, mobile computing, and similar, it is not yet mature enough.	E3-38:55
V09-3	Upfront investments are very high and the initial productivity of the developers is very low.	E1-20:23
V09-3	Model driven engineering definitely comes with significant efficiency improvements.	E3-41:40
V09-4	It is assumed that model driven engineering will increase during the next few years.	E3-37:34

<b>Variable</b>	<b>Expert statement</b>	<b>Reference</b>
V09-4	A by the authorities highly regulated business domain is more likely to benefit from model driven engineering than a very individual one.	E3-39:32
V10-1	An xml based approach to configure an application of a product line according to its variability model is feasible.	E1-22:59
V10-1	The feasibility however depends on the upfront investments, the scalability of the approach, and the number of customers to be served.	E1-23:07
V10-1	The developed concept seems feasible enough to be implemented in a real-world pilot project.	E1-23:15
V10-1	A currently more feasible way for automation in systems integration are highly customizable applications which include all possible features but may be adapted to a particular customer's needs based on XML models.	E2-40:53
V10-1	Instead of defining generators for each and every single feature it seems more viable to develop a fully functional system and configure it at runtime.	E2-42:27
V10-1	Utilizing a model based configuration approach as an alternative for model driven engineering depends on the skill level of the developers.	E3-40:35
V10-1	Discussing models with customers may be difficult if he is not experienced in model driven engineering. Care must be taken to present application requirements in a customer understandable form.	E3-40:39
V10-1	The suggested approach may be an option if the models used are fully specified and leave only little room for interpretation.	E3-41:03
V10-2	The advantage of this concept is that the customer may alter the functionality of his system without the need to hire the external supplier. He is then charged for the activation of additional features.	E2-43:13
V11-1	The overall feasibility of the developed approach depends on the expected lifespan of the product line.	E1-24:05
V11-1	In a slow changing business domain such as the aviation industry with product cycles around 20 years, investing in an industrialization approach is definitely worth the effort.	E1-24:21
V11-1	For an opportunistic approach industrialization efforts will probably never break even.	E1-25:58

<b>Variable</b>	<b>Expert statement</b>	<b>Reference</b>
V11-1	Industrialization should be focussed on long lasting product lines with a high strategic value for the customers.	E1-25:28
V11-1	Each invest in a software product line must be evaluated in a detailed business case prior to its implementation with a time horizon of 3 years.	E1-06:15
V11-1	The presented concepts are indeed important for a positive return on invest but not sufficient.	E2-44:28
V11-1	The presented approach with software product lines, component based development, and model driven engineering as means of industrialization is correct, but not necessarily the only one possible.	E3-26:18
V11-1	From an overall point of view model driven engineering is definately the most difficult and cost intensive step, component based development is a must, and software product lines may be used for specialization, although several other possibilities exist.	E3-42:58
V11-1	The suggested methodology is a valid and sound approach, although not the only one.	E3-43:21
V11-1	Industrialization is definately a must and can be observed in the software industry already.	E3-43:32
V11-2	A typical invest of 3000-4000 person days is expected to break even with at least 20 customers served.	E1-07:22
V11-2	A typical invest for a software product line in the case at hand includes 3000 to 4000 person days.	E1-06:44
V11-3	Generally it can be said that especially software product lines and component based development are beneficial with regards to cost reduction and customer satisfaction.	E1-30:33
V12-0	A significant benefit is expected from industry wide domain models. These are not necessarily to be developed by software suppliers but by the industry itself. Based on these it would be possible to develop domain specific languages, product lines, component standards, and integration architectures.	E1-26:31
V12-0	Industrialization is expected to improve the efficiency of offshore outsourcing if it is mature enough.	E1-32:43
V12-0	The industrialization concept could also have positive effects if implemented on the customer side.	E1-34:44

---

<b>Variable</b>	<b>Expert statement</b>	<b>Reference</b>
V12-0	It should be evaluated how industrialization works with agile development.	E1-35:23
V12-0	How does component based systems integration relate to service oriented architecture, i.e. the asynchronous connection of services?	E1-35:43
V12-0	Besides the presented industrialization concepts, suitable tools and processes are inevitable.	E2-44:40
V12-0	The statement that CBD and MDE cannot be successfully implemented without specialization is not assumed to be true.	E2-47:31
V12-0	The statement that systems integration projects are still developed from scratch should be reconsidered. Today many integration projects rely on enterprise service bus systems or other integration middleware providing large amounts of functionality to be reused.	E2-48:17
V12-0	Industrialization in software development is not necessarily limited to software product lines, component based development, and model driven engineering. Other aspects like libraries, frameworks, and platforms must also be considered and may be alternatives to the above.	E3-25:21

## References

**A** Achert, W., Becker, T., Biskup, H., Hellebrand, D., Herczeg, J., Krause, S., Kuhmann, M., Maar, F., Marschall, F., Minich, M., Simon, F. and Ziegler, S. (2010), *Industrielle Softwareentwicklung: Leitfaden und Orientierungshilfe*, Berlin.

Adamek, J. and Hnetynka, P. (2008), “Perspectives in component-based software engineering”, in Crnković, I. and Nawrocki, J. (Eds.), *Proceedings of the 2008 international workshop on Software Engineering in east and south europe - SEESE '08*, ACM Press, p. 35.

Adler, R., Schaefer, I., Trapp, M. and Poetzsch-Heffter, A. (2010), “Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems”, *ACM Transactions on Embedded Computing Systems*, Vol. 10 No. 2, pp. 1–39.

Afonso, M., Vogel, R. and Teixeira, J. (2006), “From Code Centric to Model Centric Software Engineering: Practical case study of MDD infusion in a Systems Integration Company”, in Machado, R.J. (Ed.), *Joint Meeting of the Fourth Workshop on Model-Based Development of Computer-Based Systems and the Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software: Proceedings, MBD/MOMPES 2006, 30 March, 2006, Potsdam, Germany*, IEEE Computer Society Press, Los Alamitos, Calif, pp. 125–134.

Ammer, C. and Stolte, P. (2010), *White Paper - Services for Automotive: In welchen automobilen Primärprozessen ist der Nutzen für eine Serviceorientierung am höchsten?*, T-Systems, Frankfurt am Main.

Amrani, M., Lucio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Le Traon, Y. and Cordy, J.R. (2012), “A Tridimensional Approach for Studying the Formal Verification of Model Transformations”, in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE Computer Society Press, pp. 921–928.

Andresen, A. (2004), *Komponentenbasierte Softwareentwicklung mit MDA, UML 2 und XML, 2., neu bearb. Aufl.*, Hanser, München.

Armbrust, M., Stoica, I., Zaharia, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D. and Rabkin, A. (2010), “A view of cloud com-

puting”, *Communications of the ACM*, Vol. 53 No. 4, pp. 50–58.

Auer, K., Schmid, F. and Strauch, S. (2007), *Marktstudie - SOA und Web Services Produkte*, Universität Stuttgart, Stuttgart.

**B** Baaken, T. and Launen, M. (1993), *Software-Marketing*, Vahlen, München.

Babar, M.A., Ihme, T. and Pikkarainen, M. (2009), “An industrial case of exploiting product line architectures in agile software development”, in Muthig, D. and McGregor, J. (Eds.), *Proceedings of the 13th International Software Product Line Conference*, Carnegie Mellon University, Pittsburgh, PA, USA, pp. 171-179.

Backhaus, K. (1996), *Multivariate Analysemethoden: Eine anwendungsorientierte Einführung ; mit 205 Tabellen*, 8th ed., Springer-Verl., Berlin [u.a.].

Bahli, B. and Ji, F. (2007), “An assessment of facilitators and inhibitors for the adoption of enterprise application integration technology: An empirical study”, *Business Process Management Journal*, Vol. 13 No. 1, pp. 108–120.

Balzert, H. (1996), *Software-Entwicklung*, Spektrum Akad. Verl., Heidelberg.

Balzert, H. (2008), *Lehrbuch der Softwaretechnik: Softwaremanagement*, 2. Aufl., Spektrum Akad. Verl., Heidelberg.

Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T. and DeBaud, J.-M. (1999), “PuLSE”, in Jazayeri, M., Mili, A. and Mittermeir, R. (Eds.), *Proceedings of the 1999 symposium on Software reusability - SSR '99*, ACM Press, pp. 122–131.

Beck, K., Beedle, M., Bennekum, A.v., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D. (2001), “The Agile Manifesto”, available at: <http://www.agilemanifesto.org/> (accessed 13 August 2012).

Becker, H. (Ed.) (2010), *Darwins Gesetz in der Automobilindustrie*, Springer-Verl., Berlin, Heidelberg.

Beltran, J.F., Holzer, B., Kamann, T., Kloss, M., Mork, S., Niehues, B., Pietrek, G., Thoms, K. and Trompeter, J. (2007), *Modellgetriebene Softwareentwicklung: MDA*

und MDS in der Praxis, Entwickler.press, Frankfurt Main.

Berger, S., Grossmann, G., Stumptner, M. and Schrefl, M. (2010), "Metamodel-Based Information Integration at Industrial Scale", in Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O.M., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Petriu, D.C., Rouquette, N. and Haugen, Ø. (Eds.), *Lecture Notes in Computer Science*, Springer-Verl., Berlin, Heidelberg, pp. 153–167.

Bertolino, A., Cooper, K.M., Koziol, A. and Reussner, R. (2011), "Towards a generic quality optimisation framework for component-based system models", in Crnkovic, I. and Stafford, J.A. (Eds.), *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering - CBSE '11*, ACM Press, p. 103.

Betz, C.T. (2007), *Architecture and patterns for IT service management, resource planning, and governance: Making shoes for the cobbler's children*, Elsevier/Morgan Kaufmann, Amsterdam, Boston.

Blau, B. and Hildenbrand, T. (2011), "Product Line Engineering in Large-Scale Lean and Agile Software Product Development Environments. Towards a Hybrid Approach to Decentral Control and Managed Reuse", in Sterrit, R. (Ed.), *2011 Sixth International Conference on Availability, Reliability and Security*, IEEE Computer Society Press, pp. 404–408.

Bliemel, F.W. and Fassott, G. (2006), "Produktmanagement", in *Wirtschafts-Lexikon: das Wissen der Betriebswirtschaftslehre*, Schäffer-Poeschel, Stuttgart, pp. 4723–4733.

Bosch, J. (2001), "Software Product Lines: Organizational Alternatives", in IEEE Computer Society (Ed.), *Proceedings of the 23rd International Conference on Software Engineering: 12 - 19 May 2001, Toronto*, IEEE Computer Society Press, Los Alamitos, pp. 91–100.

Bosch, J. (2009), "From software product lines to software ecosystems", in Carnegie Mellon University (Ed.), *Proceedings of the 13th International Software Product Line Conference*, Carnegie Mellon University, Pittsburgh, PA, USA, pp. 111-119.

Breitbart, Y., Garcia-Molina, H. and Silberschatz, A. (2010), "Overview of multidatabase transaction management", in Ng, J. and Couturier, C. (Eds.), *CASCON First Dec-*

---

ade High Impact Papers on - CASCON '10, ACM Press, pp. 93–126.

Bretzke, W.-R. (2006), *IT-Systeme im Supply Chain Management*, Bretzke, Krefeld, available at: [http://www.bretzke-online.de/downloads3/IT-Systeme\\_im\\_SCM.pdf](http://www.bretzke-online.de/downloads3/IT-Systeme_im_SCM.pdf) (accessed 07 February 2012).

Brown, A., Conallen, J. and Tropeano, D. (2005), “Models, Modeling, and Model-Driven Architecture (MDA)”, in Beydeda, S., Book, M. and Gruhn, V. (Eds.), *Model-Driven Software Development*, Springer-11645 /Dig. Serial], Springer-Verl., Berlin, Heidelberg, pp. 1–18.

Brummermann, H., Keunecke, M. and Schmid, K. (2012), “Formalizing distributed evolution of variability in information system ecosystems”, in Eisenecker, U.W., Apel, S. and Gnesi, S. (Eds.), *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '12*, ACM Press, pp. 11–19.

Butschek, F. (2006), *Industrialisierung: Ursachen, Verlauf, Konsequenzen*, UTB, Vol. 8338, Böhlau, Wien, Köln, Weimar.

**C** CAFÉ Consortium (2012), “The FAMILIES Project”, available at: <http://www.esi.es/Families/index.html> (accessed 10 May 2012).

Carr, N. (2003), “IT Doesn't Matter”, *Harvard Business Review*, Vol. 81 No. 5, pp. 41–49.

Catal, C. (2009), “Barriers to the Adoption of Software Product Line Engineering”, *ACM SIGSOFT Software Engineering Notes*, Vol. 34 No. 6, pp. 1–4.

Checkland, P. and Holwell, S. (1998), “Action Research: Its Nature and Validity”, *Systemic Practice and Action Research*, Vol. 11 No. 1, pp. 9–21.

Chen, L., Babar, M.A. and Nour, A. (2009), “Variability management in software product lines: a systematic review”, in CMU (Ed.), *Proceedings of the 13th International Software Product Line Conference*, Carnegie Mellon University, Pittsburgh, PA, USA, pp. 81- 90.

Chow, T. and Cao, D.-B. (2008), “A survey study of critical success factors in agile software projects”, *Journal of Systems and Software*, Vol. 81 No. 6, pp. 961–971.

- Clark, C. (1940), 'The conditions of economic progress', Macmillan, London
- Clements, P. and Northrop, L. (2007), *Software product lines: Practices and patterns*, [Reprint], Addison-Wesley, Boston.
- Conchúir, E.Ó., Ågerfalk, P.J., Olsson, H.H. and Fitzgerald, B. (2009), "Global software development", *Communications of the ACM*, Vol. 52 No. 8, p. 127.
- Conejero, J.M. and Hernández, J. (2008), "Analysis of crosscutting features in software product lines", in Pinto, M. and Chitchyan, R. (Eds.), *Proceedings of the 13th international workshop on Software architectures and mobility - EA '08*, ACM Press, p. 3.
- Conrad, S., Hasselbring, W. and Koschel, A. (2006), *Enterprise Application Integration: Grundlagen Konzepte Entwurfsmuster Praxisbeispiele*, 1. Aufl., Elsevier Spektrum Akad. Verl., München, Heidelberg.
- Cook, S. (2007), *Domain-specific development with Visual Studio DSL tools*, Addison-Wesley, Upper Saddle River, NJ.
- Councill, B. and Heineman, G.T. (2001), "Definition of a Software Component and Its Elements", in Heineman, G.T. and Councill, W.T. (Eds.), *Component-based software engineering: Putting the pieces together*, Addison-Wesley, Boston, pp. 5–19.
- Crnkovic, I. and Larsson, M. (2000), "A case study: Demands on Component-based Development", in Ghezzi, C., Jazayeri, M. and Wolf, A.L. (Eds.), *Proceedings of the 22nd international conference on Software engineering - ICSE '00*, ACM Press, pp. 23–31.
- Crnkovic, I. and Larsson, M. (2002), "Challenges of component-based development", *Journal of Systems and Software*, Vol. 61 No. 3, pp. 201–212.
- Crnkovic, I., Hnich, B., Jonsson, T. and Kiziltan, Z. (2002), "Specification, implementation, and deployment of components", *Communications of the ACM*, Vol. 45 No. 10.
- Crnkovic, I., Stafford, J.A. and Szyperski, C. (2011), "Software Components beyond Programming: From Routines to Services", *IEEE Software*, Vol. 28 No. 3, pp. 22–26.
- Cubo, J. and Pimentel, E. (2011), "DAMASCo: A Framework for the Automatic Com-

position of Component-Based and Service-Oriented Architectures”, in Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O.M., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Crnkovic, I., Gruhn, V. and Book, M. (Eds.), *Lecture Notes in Computer Science*, Springer-Verl., Berlin, Heidelberg, pp. 388–404.

Cummins, F. (2009), *Building the agile enterprise: With SOA, BPM and MBM*, MK/OMG Press/Elsevier, Amsterdam, Boston.

Cusumano, M. (2008), “The Changing Software Business: Moving from Products to Services”, *Computer: innovative technology for computer professionals*, Vol. 41 No. 1, pp. 20–27.

Czarnecki, K. (2005a), “Overview of Generative Software Development”, in Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O.M., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Banâtre, J.-P., Fradet, P., Giavitto, J.-L. and Michel, O. (Eds.), *Lecture Notes in Computer Science*, Springer-Verl., Berlin, Heidelberg, pp. 326–341.

Czarnecki, K. (2005b), “Overview of Generative Software Development”, in Banâtre, J.-P., Fradet, P., Giavitto, J.-L. and Michel, O. (Eds.), *Unconventional Programming Paradigms*, *Lecture Notes in Computer Science*, Vol. 3566, Springer-Verl., pp. 97-97.

Czarnecki, K. and Eisenecker, U. (2000), *Generative programming: Methods, tools, and applications*, Addison-Wesley, Boston.

Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K. and Wąsowski, A. (2012), “Cool features and tough decisions”, in Eisenecker, U.W., Apel, S. and Gnesi, S. (Eds.), *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '12*, ACM Press, pp. 173–182.

**D** Davis, S.M. (1987), *Future perfect*, Addison-Wesley, Reading, Mass.

DeMarco, T. and Lister, T. (2003), *Bärentango: Mit Risikomanagement Projekte zum Erfolg führen*, Hanser, München.

Derungs, M. (1997), “Workflowsysteme zur Prozessumsetzung”, *Dissertation*, Univer-

sität St. Gallen, St. Gallen, 1997.

Dillon, T., Wu, C. and Chang, E. (2010), “Cloud Computing: Issues and Challenges”, in IEEE Computer Society (Ed.), 2010 24th IEEE International Conference on Advanced Information Networking and Applications, IEEE Computer Society Press, pp. 27–33.

Doe, J. (2012), AIS Market Model 2012, Anonymized Source. Details are available from the author upon request.

Dumslaff, U. and Lempp, P. (2012), Studie IT Trends 2012: Business-IT Alignment sichert die Zukunft, Düsseldorf.

**E**clipse Foundation (2012), “Graphical Modeling Framework Documentation”, available at: [http://wiki.eclipse.org/GMF\\_Documentation](http://wiki.eclipse.org/GMF_Documentation) (accessed 16 December 2012).

Ehlers, J. and Hasselbring, W. (2011), “A Self-adaptive Monitoring Framework for Component-Based Software Systems”, in Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O.M., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Crnkovic, I., Gruhn, V. and Book, M. (Eds.), Lecture Notes in Computer Science, Springer-Verl., Berlin, Heidelberg, pp. 278–286.

Eisenhardt, K.M. (1989), “Building Theories from Case Study Research”, *The Academy of Management Review*, Vol. 14 No. 4, pp. 532–550.

Encyclopaedia Britannica (2005a), “Industrial Engineering and Production Management”, in Encyclopaedia Britannica (Ed.), *The new Encyclopaedia Britannica: In 32 volumes*, Vol. 21, 15. ed., Encyclopaedia Britannica, Chicago, London, New Delhi, Paris, Seoul, Sydney, Taipei, Tokyo, pp. 266–295.

Encyclopaedia Britannica (2005b), “Industrial Revolution”, in Encyclopaedia Britannica (Ed.), *The new Encyclopaedia Britannica: In 32 volumes*, Vol. 6, 15. ed., Encyclopaedia Britannica, Chicago, London, New Delhi, Paris, Seoul, Sydney, Taipei, Tokyo, pp. 304–305.

Encyclopaedia Britannica (2005c), “Industrialisierung”, in F. A. Brockhaus (Ed.), *Brockhaus Enzyklopedie: In 30 Bänden*, 21st ed., Brockhaus, Leipzig, pp. 253–254.

Encyclopaedia Britannica (2005d), “Modernization and Industrialization”, in Encyclopaedia Britannica (Ed.), *The new Encyclopaedia Britannica: In 32 volumes, Vol. 24*, 15. ed., Encyclopaedia Britannica, Chicago, London, New Delhi, Paris, Seoul, Sydney, Taipei, Tokyo, pp. 280–291.

Encyclopaedia Britannica (2005e), “Standardization”, in Encyclopaedia Britannica (Ed.), *The new Encyclopaedia Britannica: In 32 volumes, Vol. 11*, 15. ed., Encyclopaedia Britannica, Chicago, London, New Delhi, Paris, Seoul, Sydney, Taipei, Tokyo, p. 209.

Eriksson, M., Börstler, J. and Borg, K. (op. 2005), “The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations”, in Obbink, H. and Pohl, K. (Eds.), *Software product lines: 9th international conference, SPLC 2005, Rennes, France, September 26-29, 2005 proceedings*, Springer-Verl., Berlin, New York, pp. 33–44.

Eriksson, M., Börstler, J. and Borg, K. (2006), “Software Product Line Modeling Made Practical”, *Communications of the ACM*, Vol. 49 No. 12, pp. 49–53.

Estublier, J., Dieng, I.A. and Leveque, T. (2010), “Software product line evolution”, in Rubin, J., Botterweck, G., Mezini, M., Maman, I. and Lero, A.P. (Eds.), *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering - PLEASE '10*, ACM Press, pp. 32–39.

ExpertOn Group (2010), *Industry Report Automotive*, ExpertOn Group, Ismaning.

**F** F. A. Brockhaus (2005), “Standardisierung”, in F. A. Brockhaus (Ed.), *Brockhaus Enzyklopedie: In 30 Bänden, Vol. 26*, 21st ed., Brockhaus, Leipzig, pp. 152–153.

Feng, J., Zhan, D., Nie, L. and Xu, X. (2012), “A feature-oriented approach to platform-specific modelling of coarse-grained components”, *International Journal of Computer Applications in Technology*, Vol. 44 No. 1, p. 46.

Figueiredo, R., Dinda, P. and Fortes, J. (2005), “Guest Editors’ Introduction: Resource Virtualization Renaissance”, *Computer: innovative technology for computer professionals*, Vol. 38, pp. 28–31.

Fischer, J. (1999), *Informationswirtschaft Anwendungsmanagement*, Oldenbourg,

München, Wien.

Fisher, A. G. (1935), *'The clash of progress and security'*, Macmillan, London

Floyd, C., Budde, R. and Zullighofen, H. (1992), *Software development and reality construction*, Springer-Verl., Berlin; New York.

Fourastié, J. (1954), *Die grosse Hoffnung des 20. Jahrhunderts*, Bund Verl.

Fowler, M. and Rice, D. (2008), *Patterns of enterprise application architecture*, 14. print., Addison-Wesley, Boston, Mass.

France, R. and Rumpe, B. (2007), "Model-driven Development of Complex Software - A Research Roadmap", in Briand, L.C. and Wolf, A.L. (Eds.), *Future of software engineering, 2007: FOSE '07* ; 23 - 25 May 2007, Minneapolis, Minnesota ; [at] ICSE 2007, [29th International Conference on Software Engineering], IEEE Computer Society Press, Los Alamitos, Calif.

Frank, U. (2001), "Standardisierungsvorhaben zur Unterstützung des elektronischen Handels: Überblick über Anwendungsnahe Ansätze", *Wirtschaftsinformatik*, Vol. 43 No. 3, pp. 283–293.

Frank, U., Klein, S., Krcmar, H. and Teubner, A. (1998), "Aktionsforschung in der Wirtschaftsinformatik: Einsatzpotentiale und Einsatzprobleme", in Schütte, R., Siedentopf, J. and Zelewski, S. (Eds.), *Wirtschaftsinformatik und Wissenschaftstheorie. Grundpositionen und Theoriekerne: Arbeitsberichte des Instituts für Produktion und Industrielles Informationsmanagement. Nr. 4*, Universität Essen, pp. 71–90.

Frankel, D. (2003), *Model driven architecture: Applying MDA to enterprise computing*, Wiley, New York.

Freeman, R. and Webb, P. (2004), "<CTRL>+<ALT>+<TOOL PARADIGM SHIFT>?", in Vlissides, J. and Schmidt, D. (Eds.), *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '04*, ACM Press, pp. 198–199.

Frese, E. (1998), *Grundlagen der Organisation: Konzept, Prinzipien, Strukturen*, 7., überarb. Aufl., Gabler, Wiesbaden.

Frese, E. and Noetel, W. (1992), *Kundenorientierung in der Auftragsabwicklung: Strategie, Organisation und Informationstechnologie*, VDI-Verl., Düsseldorf.

Fritz, W., Förster, F., Wiedmann, K.-P. and Raffée, H. (1988), “Unternehmensziele und strategische Unternehmensführung. Neuere Resultate der empirischen Zielforschung und ihre Bedeutung für das strategische Management und die Managementlehre”, *Die Betriebswirtschaft*, Vol. 48 No. 5, pp. 567–586.

**G** Galster, M. and Avgeriou, P. (2011), “The notion of variability in software architecture”, in Heymans, P., Czarnecki, K. and Eisenecker, U. (Eds.), *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '11*, ACM Press, pp. 59–67.

Gassner, C. (1996), “Konzeptionelle Integration heterogener Transaktionssysteme”, Dissertation, Institut für Wirtschaftsinformatik, Universität St. Gallen, St. Gallen, 1996.

Geller, T. (2012), “Cloud-based HPC”, *Communications of the ACM*, Vol. 55 No. 3, p. 21.

Ghanam, Y., Maurer, F., Abrahamsson, P. and Cooper, K. (2009), “A report on the XP workshop on agile product line engineering”, *ACM SIGSOFT Software Engineering Notes*, Vol. 34 No. 5, p. 25.

Gläser, J. and Laudel, G. (2010), *Experteninterviews und qualitative Inhaltsanalyse: Als Instrumente rekonstruierender Untersuchungen*, 4th ed., Verl. für Sozialwissenschaften, Wiesbaden.

Gonsalves, C. (2011), “Standardized IT Services: As IBM Goes, So Goes the World”, available at: <http://channelnomics.com/2011/03/24/ibm-standardized-services/> (accessed 5 October 2012).

Gorton, I., Thurman, D. and Thomson, J. (2003), “Next generation application integration: challenges and new approaches”, in *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, IEEE Computer Society Press, pp. 576–581.

Greenfield, J., Short, K. and Cook, S. (2004), *Software factories: Assembling applica-*

tions with patterns, models, frameworks, and tools, Wiley, Indianapolis, Ind.

Grochla, E. (1995), *Grundlagen der organisatorischen Gestaltung*, Sammlung Poeschel, P 100, Nachdr., Schäffer-Poeschel, Stuttgart.

Groher, I., Schwanninger, C. and Voelter, M. (2008), “An integrated aspect-oriented model-driven software product line tool suite”, in Schäfer, W., Dwyer, M.B. and Gruhn, V. (Eds.), *Companion of the 13th international conference on Software engineering - ICSE Companion '08*, ACM Press, p. 939.

Grossmann, G., Schrefl, M. and Stumptner, M. (2008), “Modelling inter-process dependencies with high-level business process modelling languages”, in *Proceedings of the fifth Asia-Pacific conference on Conceptual Modelling - Volume 79*, Australian Computer Society, Inc, Darlinghurst, Australia, Australia, pp. 89-102.

**H** Hahn, H. and Turowski, K. (2003), “Drivers and inhibitors to the development of a software component industry”, in Crnkovic, I. (Ed.), *Proceedings of the 20th IEEE Instrumentation Technology Conference EURMIC-03*, IEEE Computer Society Press, pp. 128–135.

Haines, G., Carney, D. and Foreman, J. (1997), *Component Based Software Development / COTS Integration*, Software Technology Review, Pittsburgh, PA, USA.

Halevy, A., Ashish, N., Bitton, D., Carey, M., Draper, D., Pollock, J., Rosenthal, A. and Sikka, V. (2005), “Enterprise information integration: successes, challenges and controversies”, in Widom, J. (Ed.), *SIGMOD 2005: Proceedings of the ACM SIGMOD International Conference on Management of Data ; Baltimore, Maryland, June 14 - 16, 2005*, ACM Press, New York, NY, pp. 778–787.

Halevy, A., Ives, Z., Suciu, D. and Tatarinov, I. (2003), “Schema mediation in peer data management systems”, in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, IEEE Computer Society Press, pp. 505–516.

Halevy, A., Rajaraman, A. and Ordille, J. (2006), “Data integration: the teenage years”, in IEEE Computer Society (Ed.), *Proceedings of the 32nd international conference on Very large data bases, VLDB Endowment*, pp. 9-16.

Halevy, A.Y. (2003), “Data Integration: A Status Report”, in Weikum, G., Schöning,

- 
- H. and Rahm, E. (Eds.), BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz, Ges. für Informatik, Leipzig, pp. 24–29.
- Halmans, G. and Pohl, K. (2001), “Considering Product Family Assets when Defining Customer Requirements”, in Schmid, K. and Geppert, B. (Eds.), Proceedings of the PLEES'01: International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing, Fraunhofer IESE, Kaiserslautern, pp. 37–42.
- Halmans, G. and Pohl, K. (2003), “Communicating the variability of a software-product family to customers”, Software and Systems Modeling, Vol. 2 No. 1, pp. 15–36.
- Hansen, H.R. and Neumann, G. (2001), Grundlagen betrieblicher Informationsverarbeitung, Wirtschaftsinformatik, Vol. 1, 8th ed., revised, Lucius & Lucius, Stuttgart.
- Hasselbring, W. (2000), “Information System Integration”, Communications of the ACM, Vol. 43 No. 6, pp. 32–38.
- Helfferrich, C. (2011), Die Qualität qualitativer Daten: Manual für die Durchführung qualitativer Interviews, 4th ed., Verl. für Sozialwissenschaften, Wiesbaden.
- Henning, M. (2008), “The rise and fall of CORBA”, Communications of the ACM, Vol. 51 No. 8, p. 52.
- Herzum, P. and Sims, O. (2000), Business component factory: A comprehensive overview of component-based development for the enterprise, John Wiley, New York.
- Heymans, P., Czarnecki, K., Eisenecker, U.W., Nguyen, T., Colman, A., Talib, M.A. and Han, J. (2011), “Managing service variability”, in Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '11, ACM Press, pp. 165–173.
- Highsmith, J. and Cockburn, A. (2001), “Agile software development: the business of innovation”, Computer: innovative technology for computer professionals, Vol. 34 No. 9, pp. 120–127.
- Hubaux, A., Heymans, P. and Unphon, H. (2008), “Separating variability concerns in a product line re-engineering project”, in Whittle, J. and Mussbacher, G. (Eds.), Proceedings of the 2008 AOSD workshop on Early aspects - EA '08, ACM Press, pp. 1–8.

Hutchinson, J., Whittle, J., Rouncefield, M. and Kristoffersen, S. (2011), “Empirical assessment of MDE in industry”, in ACM (Ed.), *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, ACM Press, pp. 471–480.

**I** IBM (2007), *IBM Rational Unified Process*, Sommers, USA.

International Organization for Standardization (2012), *Software and Systems Engineering - Reference model for product line engineering and management (Working Draft 2.0)* ISO/IEC CD 26550, 2nd ed., International Organization for Standardization, available at: [http://www.iso.org/iso/iso\\_catalogue/catalogue\\_tc/catalogue\\_detail.htm?csnumber=43075](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43075) (accessed 30 May 2012).

Istoan, P., Biri, N. and Klein, J. (2011), “Issues in model-driven behavioural product derivation”, in Heymans, P., Czarnecki, K. and Eisenecker, U.W. (Eds.), *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, ACM Press, New York, NY, USA, pp. 69-78.

**J** Jacobson, I. (1998), *Object-oriented software engineering: A use case driven approach*, Reprint, Addison-Wesley, Harlow.

Janßen, R. (2005), “Die Psychologie des Entwicklers”, *Informatik-Spektrum*, Vol. 28 No. 4, pp. 284–286.

Jatain, A. and Goel, S. (2009), “Comparison of Domain Analysis Methods in Software Reuse”, *International Journal of Information Technology and Knowledge Management*, Vol. 2 No. 2, pp. 347–352.

Johnston, W.J., Leach, M.P. and Liu, A.H. (1999), “Theory Testing Using Case Studies in Business-to-Business Research”, *Industrial Marketing Management*, Vol. 28 No. 3, pp. 201–213.

**K** Kaib, M. (2004), *Enterprise Application Integration: Grundlagen, Integrationsprodukte, Anwendungsbeispiele*, 1st ed., Dt. Univ.-Verl., Wiesbaden.

Käkölä, T. (2010), “Standards Initiatives for Software Product Line Engineering and Management within the International Organization for Standardization”, in IEEE Computer Society (Ed.), *2010 43rd Hawaii International Conference on System*

---

Sciences, IEEE Computer Society Press, pp. 1–10.

Käkölä, T. and Dueñas, J.C. (op. 2006), *Software product lines*, Springer-Verl., Berlin, New York.

Kamp, P.-H. (2012), “My Compiler Does Not Understand Me”, *Queue - Networks*, Vol. 10 No. 5, pp. 1–5.

Kang, K.C., Lee, J. and Donohoe, P. (2002), “Feature-oriented product line engineering”, *IEEE Software*, Vol. 19 No. 4, pp. 58–65.

Kanstren, T., Piel, E. and Gross, H.-G. (2012), “Using Reverse-Engineered Test-Based Models to Generate More Tests: Where is the Sense in That?”, in *2012 Ninth International Conference on Information Technology - New Generations*, IEEE Computer Society Press, pp. 247–252.

Kepser, S. (2004), “A Simple Proof of the Turing-Completeness of XSLT and XQuery”.

Kharlamov, E. and Nutt, W. (2008), “Incompleteness in information integration”, *Proceedings of the VLDB Endowment*, Vol. 1 No. 2, pp. 1652- 1658.

Klimmer, M. (2007), *Unternehmensorganisation eine kompakte und praxisnahe Einführung ; [Lehrbuch]*, nwb, Herne.

Kloppmann, M., Leymann, F. and Roller, D. (2000), “Enterprise Application Integration mit Workflow Management”, *HMD - Praxis der Wirtschaftsinformatik*, Vol. 213, pp. 23–30.

**L** Lamnek, S. (2005), *Qualitative Sozialforschung: Lehrbuch*, 4th ed., Beltz, Weinheim, Basel.

Lang, C. (2004), *Organisation der Software-Entwicklung: Probleme Konzepte Lösungen*, 1. Aufl., Dt. Univ.-Verl., Wiesbaden.

Lau, K.-K., Safie, L., Stepan, P. and Tran, C. (2011), “A component model that is both control-driven and data-driven”, in Crnkovic, I., Stafford, J.A., Bertolino, A. and Cooper, K.M. (Eds.), *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering - CBSE '11*, ACM Press, p. 41.

- Lee, J. and Muthig, D. (2006), “Feature-oriented variability management in product line engineering”, *Communications of the ACM*, Vol. 49 No. 12, p. 55.
- Leitao, A.M. (2009), “The next 700 programming libraries”, in ACM (Ed.), *Proceedings of the 2007 International Lisp Conference on - ILC '07*, ACM Press, pp. 1–14.
- Leitner, A. and Kreiner, C. (2010), “Managing ERP configuration variants”, in ACM (Ed.), *Proceedings of the 2010 Workshop on Knowledge-Oriented Product Line Engineering - KOPLE '10*, ACM Press, pp. 1–6.
- Lence, R., Fuentes, L. and Pinto, M. (2011), “Quality attributes and variability in AO-ADL software architectures”, in Hasselbring, W. and Gruhn, V. (Eds.), *Proceedings of the 5th European Conference on Software Architecture - ECSA '11*, ACM Press, p. 1.
- Leser, U. and Naumann, F. (2007), *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*, 1. Aufl., dpunkt-Verl., Heidelberg.
- Lewin, K. (1963), *Feldtheorie in den Sozialwissenschaften*, Huber, Bern; Stuttgart.
- Liebhart, D., Schmutz, G., Lattmann, M., Heinisch, M., Könings, M., Kölliker, M., Pakull, P. and Welkenbach, P. (2008), *Integration architecture blueprint: Leitfaden zur Konstruktion von Integrationslösungen*, Hanser, München.
- Linden, F. (2007), *Software product lines in action: The best industrial practice in product line engineering*, Springer-Verl., Berlin, Heidelberg, New York.
- Lingyun, F., Guang, S. and Jianli, C. (2010), “An Approach for Component-Based Software Development”, in IEEE Computer Society (Ed.), *2010 International Forum on Information Technology and Applications*, IEEE Computer Society Press, pp. 22–25.
- Linthicum, D.S. (2000), *Enterprise application integration*, Addison-Wesley, Reading, Mass.
- Longo, J. (2001), “The ABCs of Enterprise Application Integration”, *EAI Journal*, Vol. 0 May, pp. 56–58.
- Lorenz, D.H. and Rosenan, B. (2011), “Cedalion: a language for language oriented

---

programming”, ACM SIGPLAN Notices, Vol. 46 No. 10, p. 733.

Lui, M., Gray, M., Chan, A. and Long, J. (2011), Pro Spring integration, Apress, New York, NY.

Lyons, B. (2001), “Universal Turing Machine in XSLT”, available at: <http://www.unidex.com/turing/utm.htm> (accessed 31 July 2012).

**M** MacDonald, A., Russell, D. and Atchison, B. (2005), “Model-Driven Development within a Legacy System: An Industry Experience Report”, in 2005 Australian Software Engineering Conference, IEEE Computer Society Press, pp. 14–22.

Magalhaes, A.P., David, J.M.N., Maciel, R.S.P. and Araujo da Silva, F. (2011), “Modden: An Integrated Approach for Model Driven Development and Software Product Line Processes”, in IEEE Computer Society (Ed.), 2011 Fifth Brazilian Symposium on Software Components, Architectures and Reuse, IEEE Computer Society Press, pp. 21–30.

March, S. T., Smith, G. F. (1995), “Design and natural science research on information technology” in Decision Support Systems, Vol. 15 No. 4, pp. 251–266.

Mayring, P. (2002), Einführung in die qualitative Sozialforschung: Eine Anleitung zu qualitativem Denken, 5th ed., Beltz, Weinheim.

McGregor, J. (2010), “A method for analyzing software product line ecosystems”, in Gorton, I., Babar, M.A. and Cuesta, C.E. (Eds.), Proceedings of the Fourth European Conference on Software Architecture Companion Volume - ECSA '10, ACM Press, p. 73.

McIlroy, D. (1969), “Mass Produced Software Components”, in Naur, P. and Randell, B. (Eds.), Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Bussels, pp. 138–156.

Meng, S. and Barbosa, L.S. (2010), “Towards the introduction of QoS information in a component model”, in Shin, S.Y., Ossowski, S. and Schumacher, M. (Eds.), Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10, ACM Press, p. 2045.

MetaCase (2012), “MetaEdit+ Modeler”, available at: <http://www.metacase.com/mep/>

(accessed 12 June 2012).

Meuser, M. and Nagel, U. (2005), "ExpertInneninterviews - vielfach erprobt, wenig bedacht. Ein Beitrag zur qualitativen Methodendiskussion", in Bogner, A., Littig, B. and Menz, W. (Eds.), *Das Experteninterview: Theorie, Methode, Anwendung*, 2nd ed., Verl. für Sozialwissenschaften, Wiesbaden, pp. 71–93.

Mikkola, J.H. (2003), "Modularity, component outsourcing, and inter-firm learning", *R and D Management*, Vol. 33 No. 4, pp. 439–454.

Minich, M. (2012a), *Expert Interview about Software Industrialization in Systems Integration*, E1, Darmstadt.

Minich, M. (2012b), *Expert Interview about Software Industrialization in Systems Integration*, E2, Duesseldorf.

Minich, M. (2012c), *Expert Interview about Software Industrialization in Systems Integration*, E3, Cologne.

Minich, M., Harriehausen-Mühlbauer, B. and Wentzel, C. (2008), "Software Industrialization in Systems Integration", in Bleimann, U.G. (Ed.), *Proceedings of the Fourth Collaborative Research Symposium on Security, e-learning, Internet and networking*, Glyndwr University, Wrexham, 6-7 November 2008, Centre for Information Security and Network Research, Plymouth, pp. 91–109.

Minich, M., Harriehausen-Mühlbauer, B. and Wentzel, C. (2009a), "Industrializing Software Development in Systems Integration", in Sosnin, P. (Ed.), *Interactive Systems and Technologies*, Ulyanovsk, Russia, 02.-04.09.2009, Ulyanovsk University, Ulyanovsk, pp. 24–32.

Minich, M., Harriehausen-Mühlbauer, B. and Wentzel, C. (2009b), "Software Industrialization in Systems Integration", *WASET*, Vol. 3 No. 32, pp. 343–350.

Minich, M., Harriehausen-Mühlbauer, B. and Wentzel, C. (2010), "An Organizational Approach for Industrialized Systems Integration", *Proceedings of the Eighth International Network Conference: INC 2010*, University of Plymouth, Plymouth, pp. 399-470.

Minich, M., Harriehausen-Mühlbauer, B. and Wentzel, C. (2011), "Component Based

Development in Systems Integration”, in *GI Lecture Notes in Informatics 2011: Informatik schafft Communities*, Ges. für Informatik, Bonn, p. 470.

Minich, M., Harriehausen-Mühlbauer, B. and Wentzel, C. (2012), “Model Driven Engineering in Systems Integration”, in Botha, R., Dowland, P.S. and Furnell, S.M. (Eds.), *Proceedings of the Ninth International Network Conference: INC 2012*, University of Plymouth, Plymouth, pp. 173–178.

Mohan, K., Ramesh, B. and Sugumaran, V. (2010), “Integrating Software Product Line Engineering and Agile Development”, *IEEE Software*, Vol. 27 No. 3.

Moreno-Rivera, J.M. and Navarro, E. (2011), “Evaluation of SPL Approaches for WebGIS Development: SIGTel, a Case Study”, in *2011 44th Hawaii International Conference on System Sciences*, IEEE Computer Society Press, pp. 1–10.

**N** Naujoks, S. (2010), *Germany 2010 - Automotive Industry: Snapshot*, SITSI Verticals, Paris.

Naumann, F., Freytag, J.-C. and Leser, U. (2004), “Completeness of integrated information sources”, *Information Systems*, Vol. 29 No. 7, pp. 583–615.

Nierstrasz, O.M. and Tsichritzis, D. (1995), *Object-oriented software composition*, Prentice-Hall, London.

Nöbauer, M., Seyff, N., Dhungana, D. and Stoiber, R. (2012), “Managing variability of ERP ecosystems”, in Eisenecker, U.W., Apel, S. and Gnesi, S. (Eds.), *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '12*, ACM Press, pp. 21–26.

**O** Object Management Group (2003), *MDA Guide Version 1.0.1*, accessed on 13.08.2012.

Object Management Group (2011), *Query View Transformation No. 1.1*, Object Management Group, available at: <http://www.omg.org/spec/QVT/1.1/PDF/>, accessed on 13.08.2012.

Orbán, G. and Kozma, L. (2012), “Defining contracts with different tools in software development”, *Annales Univ. Sci. Budapest*, Vol. 36 No. 1, pp. 323–339.

Orfali, R., Harkey, D. and Edwards, J. (1996), *The essential distributed objects survival guide*, Wiley, New York, NY.

Owe, O., Schneider, G. and Steffen, M. (2007), “Components, objects, and contracts”, in *Proceedings of the 2007 conference on Specification and verification of component-based systems 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering - SAVCBS '07*, ACM Press, pp. 95–98.

Özsu, M.T. and Valduriez, P. (2011), *Principles of distributed database systems*, 3rd ed., Springer-Verl., New York.

**P** Papazoglou, M.P. and Heuvel, W.-J.v.d. (2007), “Service oriented architectures: approaches, technologies and research issues”, *The VLDB Journal*, Vol. 16 No. 3, pp. 389–415.

Parra, P., Polo, O.R., Knoblauch, M., Garcia, I. and Sanchez, S. (2011), “MICOBS”, in Crnkovic, I., Stafford, J.A., Bertolino, A. and Cooper, K.M. (Eds.), *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering - CBSE '11*, ACM Press, p. 1.

Pech, D., Knodel, J., Carbon, R., Schitter, C. and Hein, D. (2009), “Variability management in small development organizations: experiences and lessons learned from a case study”, in Carnegie Mellon University (Ed.), *Proceedings of the 13th International Software Product Line Conference*, Carnegie Mellon University, Pittsburgh, PA, USA, pp. 285-294.

Petrasch, R. and Meimberg, O. (2006), *Model Driven Architecture: Eine praxisorientierte Einführung in die MDA*, 1. Aufl., dpunkt-Verl., Heidelberg.

Pham, H., Qusay, M., Alexander, F. and Alireza, S. (2007), “Applying Model Driven Development to Pervasive Systems Engineering”, in *2007 First International Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments: (SEPCASE 2007)* ; Minneapolis, Minnesota, 20 - 26 May 2007 ; [29th International Conference on Software Engineering, ICSE 2007], IEEE Computer Society Press, Piscataway, NJ.

Pierre Audoin Consultants (2009), *Software and IT Services Industry (SITSI) Report*,

Paris.

Piller, F.T. (2006), *Mass customization: Ein wettbewerbsstrategisches Konzept im Informationszeitalter*, 4th ed., Dt. Univ.-Verl., Wiesbaden.

Pohl, K., Böckle, G. and Linden, F. (2005), *Software product line engineering: Foundations, principles, and techniques ; with 10 tables*, Springer-Verl., Berlin.

Potts, C. (1993), "Software-Engineering Research Revisited", *IEEE Software*, Vol. 10 No. 5, pp. 19–28.

Puschmann, T. and Alt, R. (2001), "Enterprise Application Integration - The Case of the Robert Bosch Group", in *IEEE Computer Society (Ed.), Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, IEEE Computer Society Press, Washington DC, pp. 1–10.

**R** Rapos, E.J. and Dingel, J. (2012), "Incremental Test Case Generation for UML-RT Models Using Symbolic Execution", in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE Computer Society Press, pp. 962–963.

Rashid, A. (2012), "The AMPLE Project", available at: <http://www.ample-project.net/> (accessed 10 May 2012).

Rashid, A., Royer, J.-C. and Rummler, A. (2011), "Software product line engineering challenges - Introduction", in *Rashid, A., Royer, J.-C. and Rummler, A. (Eds.), Aspect-oriented, model-driven software product lines: The ample way*, Cambridge University Press, Cambridge ;, New York.

Ridene, Y. and Barbier, F. (2011), "A model-driven approach for automating mobile applications testing", in *Gruhn, V. and Hasselbring, W. (Eds.), Proceedings of the 5th European Conference on Software Architecture - ECSA '11*, ACM Press, p. 1.

Riehm, R. (1997), "Integration von heterogenen Applikationen", *Dissertation*, Universität St. Gallen, St. Gallen, 1997.

Robra, C. (2007), *Modellierung komponentenbasierter Software-Architekturen: Grundlagen, Konzepte und Methoden*, VDM Müller, Saarbrücken.

Roy, N., Dubey, A., Gokhale, A. and Dowdy, L. (2011), “A Capacity Planning Process for Performance Assurance of Component-based Distributed Systems”, in Kounev, S. and Cortellessa, V. (Eds.), *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering - ICPE '11*, ACM Press, p. 259.

Rudametkin, W., Touseau, L., Donsez, D. and Exertier, F. (2010), “A Framework for Managing Dynamic Service-Oriented Component Architectures”, in IEEE Computer Society (Ed.), *2010 IEEE Asia-Pacific Services Computing Conference*, IEEE Computer Society Press, pp. 43–50.

Ruh, W.R., Brown, W.J. and Maginnis, F.X. (2001), *Enterprise application integration at work: How to successfully plan for EAI*, Wiley, New York.

Ruz, C., Baude, F. and Sauvan, B. (2010), “Component-based generic approach for reconfigurable management of component-based SOA applications”, in Karastoyanova, D. and Kazhamiakin, R. (Eds.), *Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond - MONA '10*, ACM Press, pp. 25–32.

Ryssel, U., Ploennigs, J. and Kabitzsch, K. (2011), “Extraction of feature models from formal contexts”, in Schaefer, I., John, I. and Schmid, K. (Eds.), *Proceedings of the 15th International Software Product Line Conference on - SPLC '11*, ACM Press, p. 1.

**S**

Sabraoui, A., Ennouaary, A., Khriiss, I. and Koutbi, M.E. (2012), “An MDA-Based Approach for WS Composition Using UML Scenarios”, in *2012 Ninth International Conference on Information Technology - New Generations*, IEEE Computer Society Press, pp. 306–313.

Sako, M. (2006), “Outsourcing and Offshoring: Implications for Productivity of Business Services”, *Oxford Review of Economic Policy*, Vol. 22 No. 4, pp. 499–512.

Sako, M. (2009), “Technology Strategy and Management Globalization of knowledge-intensive Professional Services”, *Communications of the ACM*, Vol. 52 No. 7, pp. 31–33.

Salger, F., Engels, G. and Hofmann, A. (2010), “Assessments in global software development”, in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, ACM Press, p. 29.

Sametinger, J. (1997), *Software engineering with reusable components: With 26 tables*, Springer-Verl., Berlin.

Santos, R.P. dos and Werner, C.M.L. (2010), "Revisiting the concept of components in software engineering from a software ecosystem perspective", in Cuesta, C.E. (Ed.), *Proceedings of the Fourth European Conference on Software Architecture*, ACM Press, New York, pp. 135–142.

Schmidt, D. (2006), "Model-Driven Engineering", *Computer: innovative technology for computer professionals*, Vol. 39 No. 2, pp. 25–31.

Schryen, G. and Bastian, M. (2001), *Komponentenorientierte Softwareentwicklung in Softwareunternehmen: Konzeption eines Vorgehensmodells zur Einführung und Etablierung*, 1. Aufl., Dt. Univ.-Verl., Wiesbaden.

Selic, B. (2008), "Personal reflections on automation, programming culture, and model-based software engineering", *Automated Software Engineering*, Vol. 15 3-4, pp. 379-391.

Sentilles, S. (2012), "Managing extra-functional properties in component-based development of embedded systems", *School of Innovation, Design and Engineering*, Mälardalen University, Västerås, Sweden, 2012.

She, S., Lotufo, R., Berger, T., Wąsowski, A. and Czarnecki, K. (2011), "Reverse engineering feature models", in Taylor, R.N., Gall, H. and Medvidović, N. (Eds.), *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, ACM Press, p. 461.

Shirtz, D., Kazakov, M. and Shaham-Gafni, Y. (2007), "Adopting Model Driven Development in a Large Financial Organization", in Akehurst, D.H., Vogel, R. and Paige, R.F. (Eds.), *Lecture Notes in Computer Science*, Springer-Verl., Berlin, Heidelberg, pp. 172–183.

Simonyi, C. (2012), "Web presence of Intentional Software Corp.", available at: [www.intentsoft.com](http://www.intentsoft.com) (accessed 15 June 2012).

Simonyi, C., Christerson, M. and Clifford, S. (2006), "Intentional software", *ACM SIGPLAN Notices*, Vol. 41 No. 10, p. 451.

- Singh, Y. and Sood, M. (2009), "Model Driven Architecture: A Perspective", in IEEE Computer Society (Ed.), Proceedings of the 2009 IEEE International Advance Computing Conference, IEEE Computer Society Press, Patiala, pp. 1644–1652.
- Smojver, K., Belani, H. and Car, Z. (2009), "Building a hybrid process model for a complex software system integration", Proceedings on the 10th International Conference on Telecommunications, IEEE Computer Society Press, Zagreb, pp. 147-153.
- Software Engineering Institute (2012a), "A Framework for Software Product Line Practice, Version 5.0", available at: [http://www.sei.cmu.edu/productlines/frame\\_report/index.html](http://www.sei.cmu.edu/productlines/frame_report/index.html) (accessed 10 May 2012).
- Software Engineering Institute (2012b), "Software Product Line Hall of Fame", available at: <http://www.splc.net/fame.html> (accessed 10 May 2012).
- Soomro, T.R. and Awan, A.H. (2012), "Challenges and Future of Enterprise Application Integration", International Journal of Computer Applications, Vol. 42 No. 7, pp. 42–46.
- Sparling, M. (2000), "Lessons Learned. Through Six Years of Component Based Development", Communications of the ACM, Vol. 43 No. 10, pp. 47–53.
- Stahl, T. and Bettin, J. (2007), Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management, 2., aktualisierte und erw. Aufl., dpunkt-Verl., Heidelberg.
- Staines, M. (2011), "Sourcing: Keep it Simple and Standard", available at: <http://www.cioupdate.com/budgets/article.php/3934391/Sourcing-Keep-it-Simple-and-Standard.htm> (accessed 10 May 2012).
- Stake, R. E. (1995), The Art of Case Study Research, Sage Publications, Thousand Oaks.
- Staron, M. (2006), "Adopting Model Driven Software Development in Industry – A Case Study at Two Companies", in Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O.M., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Whittle, J., Harel, D. and Reggio, G. (Eds.), Lecture Notes in Computer Science, Springer-

---

Verl., Berlin, Heidelberg, pp. 57–72.

Supply Chain Council (2010), Supply Chain Operations Reference (SCOR) Model: Overview - Version 10.0, Cypress, TX, USA.

Szyperski, C., Gruntz, D. and Murer, S. (2002), Component software: Beyond object-oriented programming, 2nd ed., ACM Press; Addison-Wesley, New York, London, Boston.

**T** Tanriverdi, H. and Uysal, V.B. (2011), “Cross-Business Information Technology Integration and Acquirer Value Creation in Corporate Mergers and Acquisitions”, *Information Systems Research*, Vol. 22 No. 4, pp. 703–720.

Taubner, D. (2005), “Software-Industrialisierung”, *Informatik-Spektrum*, Vol. 28 No. 4, pp. 292–296.

Teppola, S., Parviainen, P. and Takalo, J. (2009), “Challenges in Deployment of Model Driven Development”, in 2009 Fourth International Conference on Software Engineering Advances, IEEE Computer Society Press, pp. 15–20.

The Eclipse Foundation (2012), “Graphical Modeling Framework Documentation”, available at: [http://wiki.eclipse.org/GMF\\_Documentation\\_Index](http://wiki.eclipse.org/GMF_Documentation_Index).

Themistocleous, M., Irani, Z., O’Keefe, R.M. and Paul, R. (2001), “ERP problems and application integration issues: an empirical survey”, in Proceedings of the 34th Annual Hawaii International Conference on System Sciences, IEEE Computer Society Press, p. 10.

Tibermacine, C., Sadou, S., Dony, C. and Fabresse, L. (2011), “Component-based specification of software architecture constraints”, in Crnkovic, I., Stafford, J.A., Bertolino, A. and Cooper, K.M. (Eds.), Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering - CBSE '11, ACM Press, p. 31.

Tizzei, L.P. and Rubira, C.M.F. (2011), “Aspect-Connectors to Support the Evolution of Component-Based Product Line Architectures: A Comparative Study”, in Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O.M., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Crnkovic, I., Gruhn, V. and Book, M. (Eds.), Lecture

Notes in Computer Science, Springer-Verl., Berlin, Heidelberg, pp. 59–66.

Töpfer, A. (1985), “Umwelt- und Benutzerfreundlichkeit von Produkten als strategische Unternehmensziele”, *Marketing ZFP*, Vol. 7 No. 4, pp. 241–251.

Torres, M., Masiero, P., Kulesza, U., Sousa, M., Batista, T., Teixeira, L., Borba, P., Cirilo, E., Lucena, C. and Braga, R. (2010), “Assessment of product derivation tools in the evolution of software product lines”, in Apel, S., Batory, D., Czarnecki, K., Heidenreich, F., Kästner, C. and Nierstrasz, O.M. (Eds.), *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development - FOSD '10*, ACM Press, pp. 10–17.

Tracz, W. (2001), “COTS Myths and Other Lessons Learned in Component-Based Software Development”, in Heineman, G.T. and Council, W.T. (Eds.), *Component-based software engineering: Putting the pieces together*, Addison-Wesley, Boston, pp. 99–111.

Trinczek, R. (2002), “Wie befrage ich Manager? Methodische und methodologische Aspekte des Experteninterviews als qualitativer Methode empirischer Sozialforschung”, in Bogner, A. (Ed.), *Das Experteninterview: Theorie, Methode, Anwendung*, Leske und Budrich, Opladen.

TU Berlin (2012), “The Tiger Project”, available at: <http://user.cs.tu-berlin.de/~tigerprj/> (accessed 12 June 2012).

**U** United Nations Economic Commission for Europe (2006), “Core Components Technical Specification Version 2.2”, available at:

[http://www.unece.org/cefact/forum\\_grps/tmg/CCTS-PublicReview.pdf](http://www.unece.org/cefact/forum_grps/tmg/CCTS-PublicReview.pdf).

United Nations Economic Commission for Europe (2010), “United Nations Directories for Electronic Data Interchange for Administration, Commerce and Transport”, available at: <http://www.unece.org/trade/untdid/directories.htm> (accessed 14 August 2012).

**V** Verband der Automobilindustrie e.V. (1991), *Datenfernübertragung von ODETTE-Nachrichten: EDIFACT - Nutzerdatenrahmen, VDA-Empfehlung*, Frankfurt am Main.

Voelter, M. and Groher, I. (2007), “Product Line Implementation using Aspect-Oriented and Model-Driven Software Development”, in Kang, K.C. (Ed.), *11th Inter-*

national Software Product Line Conference (SPLC 2007), IEEE Computer Society Press, pp. 233–242.

Vogel, O., Arnold, I., Chughtai, A., Ihler, E., Kehrer, T., Mehlig, U. and Zdun, U. (2009), *Software-Architektur: Grundlagen - Konzepte - Praxis*, Springer-11774 [Digital], 2. Auflage., Spektrum Akad. Verl., Heidelberg.

Vogler, P. (2006), *Prozess- und Systemintegration: Evolutionäre Weiterentwicklung bestehender Informationssysteme mit Hilfe von enterprise application integration*, 1. Aufl., Dt. Univ.-Verl., Wiesbaden.

Volkswagen AG (2010), “Volkswagen Produktion bereitet sich auf Modularen Querbaukasten vor”, available at: [http://www.volkswagenag.com/content/vwcorp/info\\_center/de/news/2010/10/MQB.html](http://www.volkswagenag.com/content/vwcorp/info_center/de/news/2010/10/MQB.html) (accessed 28 May 2012).

Vollmer, K. (2011), *The Forrester Wave: Enterprise Service Bus, Q2 2011*, Cambridge, New York.

Vonhoegen, H. (2011), *Einstieg in XML: Grundlagen, Praxis, Referenz*, 6th ed., Galileo Press, Bonn.

**W** Wallnau, K.C., Hissam, S. and Seacord, R.C. (2002), *Building systems from commercial components*, Addison-Wesley, Boston.

Walter, S.M., Böhmman, T. and Kremer, H. (2007), “Industrialisierung der IT - Grundlagen, Merkmale und Ausprägungen eines Trends”, *HMD - Praxis der Wirtschaftsinformatik*, Vol. 256, pp. 6–16.

Wang, G. and Liu, M. (2003), “Extending XML Schema with Nonmonotonic Inheritance”, in Goos, G., Hartmanis, J., Leeuwen, J., Jeusfeld, M.A. and Pastor, Ó. (Eds.), *Lecture Notes in Computer Science*, Springer-Verl., Berlin, Heidelberg, pp. 402–407.

Wang, L., Laszewski, G., Younge, A., He, X., Kunze, M., Tao, J. and Fu, C. (2010), “Cloud Computing: a Perspective Study”, *New Generation Computing*, Vol. 28 No. 2, pp. 137–146.

Weerawarana, S. (2005), *Web services platform architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and more*, Prentice-Hall,

Upper Saddle River, NJ.

Weiss, J. (2002), *Industrialisation and globalisation: Theory and evidence from developing countries*, Routledge, London.

Wieringa, R. (2009), “Design science as nested problem solving”, in *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology - DESRIST '09*, ACM Press, n. p.

Wöhe, G. and Döring, U. (2008), *Einführung in die allgemeine Betriebswirtschaftslehre*, 23., vollst. neu bearb. Aufl., Vahlen, München.

World Wide Web Consortium (2012), “Two XML Schema Specifications are Recommendations”, available at: <http://www.w3.org/News/2012#entry-9412>.

Wu, Y., Peng, X. and Zhao, W. (2011), “Architecture Evolution in Software Product Line: An Industrial Case Study”, in Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O.M., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G. and Schmid, K. (Eds.), *Lecture Notes in Computer Science*, Springer-Verl., Berlin, Heidelberg, pp. 135–150.

**Z** Zhang, H., Liu, J., Zheng, L. and Wang, J. (2012), “Modeling of Web Service Development Process Based on MDA and Procedure Blueprint”, in *Proceedings of the 2012 IEEE/ACIS 11th International Conference on Computer and Information Science*, IEEE Computer Society Press, Los Alamitos, Calif, pp. 422–427.

Zhu, J. and Zhang, L. (2006), “A Sandwich Model for Business Integration in BOA (Business Oriented Architecture)”, in *2006 IEEE Asia-Pacific Conference on Services Computing (APSCC'06)*, IEEE Computer Society Press, pp. 305–310.

Zwintzsch, O. (2005), *Software-Komponenten im Überblick: Einführung, Klassifizierung & Vergleich von JavaBeans, EJB, COM+, .Net, CORBA, UML 2*, W3L-Verl., Herdecke u.a.

The following pages include the material  
published throughout the course of research.

# Software Industrialization in Systems Integration

Matthias Minich

University of Plymouth, Plymouth, United Kingdom

matthias.minich@plymouth.ac.uk

## Abstract

Today's economy is in a permanent change, causing merger and acquisitions and co operations between enterprises (Vogler, 2004). Consequential process adaptations and realignments usually result in systems integration and software development projects. Processes and procedures to execute such projects are still reliant on craftsmanship of highly skilled workers (Greenfield, 2004b). A generally accepted, industrialized production, characterized by high efficiency and quality, seems inevitable.

In spite of this, current concepts of software industrialization are aimed at traditional software engineering and do not consider the particularities of systems integration. From the author's point of view it distinguishes itself from traditional software development in various points. The present work, and the subsequent research, will therefore focus on the implementation of industrialization concepts in the area of systems integration. The present paper briefly describes the idea of software industrialization, depicts current concepts from science, discusses the particularities of systems integration and suggests further areas of research. The objective of the suggested research should bring the area of systems integration closer to an industrialized production, allowing a higher efficiency, quality and return on investment.

## Keywords

Software Industrialization, Systems Integration, Software Product Lines, Software Factories, Model Driven Engineering, Component Based Development.

## 1. Industrialization

Industrialization can be defined as the spreading of standardized and highly productive methods in production of goods and services in all economic areas ("Brockhaus-Enzyklopädie", 1989, Butschek, 2007). The principle of industrialization is seen as a necessary step for economic growth, technological advances and increasing wealth. Only industrial production methods allow to produce a multiplicity of goods in a sufficient amount and quality ("Brockhaus-Enzyklopädie", 1989).

From a production point of view, omitting prerequisites such as the availability of resources and commodities or communication and transportation technologies, the key concepts of industrialization can be outlined as follows:

- Standardization
- Specialization
- Systematic Reuse
- Automation

These key concepts are often implemented in an “[...] organization of work known as the factory system, which entailed increased division of labour and specialization of function” (“Encyclopedia Britannica”, 1991). As of today, the above principles can be found in almost all industries at different levels of penetration. Standardization and specialization advance the level of automation as e.g. in the electronics industry, whereas creative tasks (which cannot be standardized), such as product design, are still performed by highly skilled workers.

## **2. Current concepts of industrialized software development**

Software development is “[...] slow and expensive, and yields products containing serious defects that cause problems of usability, reliability, performance and security” (Greenfield and Short, 2004). It can be assumed that most of a program’s functionality has already been developed in previous projects. If a consistent level of reuse and automation can be achieved, significant improvements in efficiency and quality can be made, which come along with noteworthy cost savings. In the following software development concepts are depicted which show signs of industrialization, as discussed in the previous chapter.

### **2.1. Model Driven Engineering**

Model Driven Engineering aims to raise the level of abstraction of software engineering to fill the gap between the problem solution to be implemented and the actual technology utilized to do so. Once a suitable level of abstraction is found, the description of the solution has to be refined by adding previously omitted details until an executable implementation is available. The distance between the description and technical implementation characterizes what is commonly referred to as the abstraction gap.

Raising the level of abstraction has been researched on in the 1980s already with the upcoming of CASE-Tools. They encouraged development methods based on graphical representations of software with e.g. state machines, structure diagrams or data-flow diagrams (Schmidt, 2006) to generate source code. The graphical representations however were too generic to precisely describe the intended solution and did poorly map to the underlying technologies. The result was highly complex source code which had to be altered by hand. The corresponding models were out of date very soon as the CASE tools could hardly depict manual changes to the code.

To overcome previously described difficulties, Model Driven Engineering (MDE) combines two important approaches: Domain Specific Languages (DSLs, also referred to as “Domain Specific Modelling Languages”) and Transformation Engines and Generators (Schmidt, 2006), as described in the following.

### 2.1.1. Domain Specific Language

A Domain Specific Language (DSL) models concepts found in a specific domain, such as financial online services, e-commerce applications, CRM systems, or anything else clearly delimited. The characteristics of a specific domain are represented by metamodels, precisely specifying semantics and constraints associated with this particular domain (Schmidt, 2006).

*“A modelling language is a visual type system for specifying model-based programs. It raises the level of abstraction, bringing the implementation closer to the vocabulary understood by subject matter experts, domain experts, engineers and end-users.”*(Greenfield and Short, 2004)

One of the most successful examples of a Domain Specific Language can be found in WYSIWYG-Editors for graphical user interfaces. While in the beginning GUIs could only be built by highly skilled developers, today’s wizards and code generators allow almost everyone to develop powerful user interfaces. What made this possible was the definition of a highly specialized, domain specific language, implemented in GUI design tools. Their elements (buttons, panes, text fields, etc.) can be combined based on clearly specified rules (e.g. Buttons can only appear within panes or windows etc.). Other well known examples are Event Driven Process Chains or the Entity Relationship Model (Beltran et al., 2007). With DSLs it should for instance be possible to assemble an online shopping system with credit approval, product catalogue and payment system without having to worry about the particular implementation and interaction of the components. To sum it up, DSLs have several important advantages (Beltran et al., 2007):

- Specifications can be described faster and more precise with DSLs
- Change requests can be captured precisely and unambiguously with DSLs
- Specifications are context free and leave no room for interpretations
- Code generators can be built for a specific domain and are thus more powerful and easier to handle as e.g. former CASE tools
- Transforming a model to source code by a generator is less error-prone than manual implementation for each product

### 2.1.2. Transformation Engines and Generators

Once a software system in a defined problem space has been specified with the help of the appropriate DSL(s), the thereby created set of models can be transformed to either intermediate models, or directly into source code. The former can be useful if the abstraction gap between a problem domain and the technical implementation capabilities is too large, e.g. if a specified system is supposed to run on different platforms - intermediate models would then take care of the particular requirements of these platforms. To generate subsequent artefacts out of models, transformation engines or code generators need to be provided together with meta-models of the source and target model, as well as a set of mapping rules between them. Whereas the meta-models are already available by the definition of the Domain Specific Languages, the transformation rules must be expressed within a transformation language (Pham et al., 2007).

In their book “Software Factories”, Greenfield and Short address Model Driven Engineering as one of the key innovations for software industrialization. They follow the differentiation of transformations as depicted by Czarnecki in (Czarnecki, 1999), which can be vertical, horizontal or oblique. Vertical transformations refine an existing model to a lower level, more concrete model or directly to source code. An example of vertical transformation is the transformation of a model describing a business process to a more detailed one, as for example the distribution over different web-services (Greenfield and Short, 2003). Horizontal transformations in contrast may either be refactoring or delocalizing transformations. “Refactoring transformations reorganize a specification to improve its design without changing its meaning”, whereas “delocalized transformations can be used to optimize an implementation or to compose parts of an implementation that are specified independently” (Greenfield and Short, 2003). The former may for example adapt a model to a given architecture of a product line, whereas the latter may weave a security framework into the existing model.

## 2.2. Component Based Development

The idea of separating software into delimited parts out of which applications can be stitched together as needed, is probably as old as software development itself. It first appeared in literature at the NATO Software Engineering Conference where M.D. McIlroy suggested that we need a software component sub industry, “available in families arranged according to precision, robustness, generality and time span performance” (“Software Engineering”, 1968). In his book about component software (Szyperski, 1998), Szyperski defines a component as follows:

*“A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties”.*

A component requires a defined environment and interacts with this environment via defined interfaces, without revealing the actual implementation of the functionality it provides. Ideally, components are language neutral and neither platform constraint, nor application bound. Based on Brown (Brown and Wallnau, 1996) and adapted by Haines and Foreman (Haines et al., 1997), component based development can be subdivided into four major steps:

During the first step (component qualification) existing components are discovered and evaluated against their potential to be deployed in another context. The result of the qualification defines whether certain functionality can be integrated from existing artefacts or must be manually developed. Component qualification may include functional and non-functional requirements such as algorithms or interfaces and quality or performance.

If required, suitable components can be adapted in the next step. Adaptations could be wrappers for underlying platforms or the integration of certain aspects as e.g. security concepts. Components can be categorized into white-box, grey-box and black-box ones (Haines et al., 1997). The former allow significant changes to the component at the cost of compatibility and replace ability. Adaptations to the latter have very little negative side effects, but may not allow the required flexibility. Grey-Box

components do not allow changes to their source code but provide extension languages or APIs (Haines et al., 1997) to adapt them to specific requirements.

In a third step the previously qualified and adapted components are assembled to a new application. This assembly is usually built on frameworks which provide the implementation base for the components. „It is therefore very important that there exist a context in which [...] [components] can be used“ (Crnkovic et al., 2002). Frameworks furthermore overlap with patterns, which „[...] define a recurring solution to a recurring problem“ (Crnkovic et al., 2002).

The final step focuses on maintenance and enhancement. Components are replaced with their improved or debugged versions or with totally new ones, combining the functionality of multiple already existing ones (Haines et al., 1997).

### 2.2.1. Current implementations and frameworks

The IT landscape provides several implementations of component based development, which are primarily concerned with the technical mechanisms of enabling components to communicate with each other. The most prominent representatives are CORBA, COM/DCOM, Web Services and EJBs:

- **CORBA:** The Common Object Request Broker Architecture (CORBA) defines a standardized model for inter-component communication and defines specific operations which describe the collaboration of distributed systems. The central concept of CORBA is the Object Request Broker (ORB), through which different components communicate with each other ("Lexikon der Kommunikations- und Informationstechnik", 2001). It takes requests, locates the required component and forwards the request transparently. Interoperability between languages is ensured by an Interface Definitions Language to describe the external boundaries of a component in a standardized way ("Computer und Informationstechnologie", 2005), and language specific ORB implementations.
- **COM/DCOM:** The Distributed Component Object Model is an architecture developed by Microsoft for the communication between components, based on a Windows operating platform. It uses proxies, providing interfaces and stub code by abstract methods and memory pointers ("Computer und Informationstechnologie", 2005). They can be seen as a virtual substitute, forwarding requests to the actual component. Communication within a single computer system occurs directly through shared memory, for distributed systems it occurs via Remote Procedure Calls (distributed COM or DCOM). Interfaces of components are described with the Microsoft Interface Definition Language (MIDL). To allow interoperability between different programming languages, any data is converted into a normalized format. However, support for proxies and component discovery is primarily available for windows platforms. ("Computer und Informationstechnologie", 2005).
- **Web-Services / SOA:** One of the most recent approaches is depicted by the Service Oriented Architecture (SOA). It aims at the provision of business

functionality as clearly delimited services in order to avoid the “[...] duplication of code [...] for enabling similar business functions across multiple business processes, spanning one or more lines of businesses” (Dan et al., 2008). Ideally, services are delimited, available in a network, have published interfaces, are platform independent, and registered in a repository. The most prominent implementation, Web Services, is based on three major concepts: Universal Description Discovery and Integration (UDDI) for component registration and indexing, Web Service Description Language (WSDL) for a precise, XML-based description of the supported functionality, methods and parameters, and the Simple Object Access Protocol (SOAP) for XML-based communication between service consumer and provider, encapsulated in common internet protocols such as HTTP for example.

- **EJBs:** Enterprise Java Beans is a component oriented framework for distributed information systems. It is based on the J2EE library and allows the invocation of remote methods via Web Services, IIOP (Internet Inter ORB Protocol, based on CORBA), Native Java, or JMS (Java Message Service). While Java and JMS require a Java implementation on both sides of the connection, Web Services and IIOP allow platform independent communication between different components. However, the EJB concept does not offer any transformation of data.

While MDE or CBD were successfully introduced in smaller areas, different technologies, platform dependencies or high initial investments anticipated the successful integration of already existing components into new applications across-the-board. Apart from Web Services, CORBA can be seen as the dominant model for a corporate wide system landscape as it is platform and language independent and an open standard, already in its third generation (Lewandowski, 1998, Schryen, 2001). Despite some experimental adaptations, COM/DCOM relies on concepts of the Microsoft Windows platform, which prevent it from being adopted by major business software providers which usually offer their products on different platforms (e.g. UNIX). The EJB concept may be platform independent, but relies on Java implementations on both sides. From a market perspective, only CORBA and COM/DCOM have enough momentum, supplier support and a large enough feature set to serve as long term technologies (Lewandowski, 1998).

Web Services in contrary focus more on the distributed and software-as-service aspect and offer their services platform and language independent over networks. The author therefore expects CORBA to become the major concept for component oriented and distributed, but company wide system landscapes. Web services may find their focus in specific services offered by external providers over the internet, such as credit approval by financial institutions for example.

### 2.3. Software Product Lines and Software Factories

Greenfield and Short suggest the term “economies of scope” to describe the basic principle of software industrialization. “Economies of scope arise when multiple similar but distinct designs and prototypes are produced collectively, rather than in-

dividually” (Greenfield and Short, 2004). Economies of scale in contrast arise when several copies of exactly the same product are created. As this can be done very easily with software, economies of scale do not offer any advantages. Economies of scope can be compared to a car manufacturer for example. Besides model specific body parts, car makers mostly assemble their cars from standardized components like engine blocks, gearboxes or electronic control units. Depending on the customer’s wishes, specific components are selected and assembled to a complete car. Most of the components may also be used in another model.

The concept of software product lines requires to separate product development from product line development. The former produces the actual software product, while the latter produces all the required assets to support the product development process. The concept furthermore groups closely related products to a product family. This has the advantage that the assets of a product line are more specific and powerful to a problem than generic concepts could be. “A software product line systematically captures knowledge of how to produce the assets, such as components, processes and tools, and then applies those assets to produce the family members” (Greenfield and Short, 2004).

In their book “Software Factories”, Greenfield and Short take the software product line approach one step further by introducing the concept of software factories.

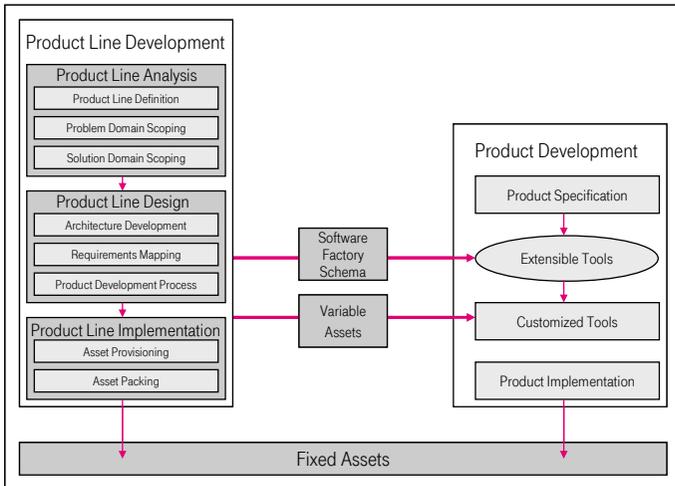


Figure 1: A Software Factory

It proposes a way to “categorize and summarize development artefacts, such as XML documents, models, configuration files, build scripts, source code files, [...] in an orderly way [...]” to define relationships and dependencies among them (Greenfield and Short, 2004). Given a certain product, the software factory identifies the required artefacts and assets of the respective product line in order to develop the product. It does so by defining software factory schemas which exactly define which assets like micro-processes, frameworks, architectures and tools or domain specific languages, are to be used to produce a family member, as illustrated in the previous figure.

To implement the idea of software product lines and software factories, Greenfield and Short demand the further development of four critical innovations (Greenfield and Short, 2004):

- **Systematic Reuse:** Technologies like CORBA, J2EE or COM/DCOM offer the basic principles required for reuse. However, the main problem with such technologies is the lack of a specific context. Components are too generic to cover all possible implementation scenarios in arbitrary contexts (Greenfield and Short, 2004). Components developed in a specific context may be reused more easily in a similar context. A component used for payment verification can be much more powerful if it is only used within the context of web based applications and not within mainframes as well.
- **Development by Assembly:** This critical innovation subsumes five prerequisites required to support development by assembly. Platform independent protocols to avoid interoperability problems between components. Self description (or contracts), including assumptions, dependencies and behaviour, allow for proper selection and validation of assemblies. To be able to customize, a deferred encapsulation of existing components is necessary, which “[...] reduces architectural mismatch by waving adaptations into published components” (Greenfield and Short, 2004). To reduce the risk of architectural mismatch, architecture driven development must be implemented by imposing assumptions and constraining design decisions. Similar to web-services, the fifth prerequisite suggests the assembly of components by orchestration. The latter can be seen as an automated combination and functional management of independent components.
- **Model Driven Development:** Further automation of software development requires formal specifications in a way humans and machines can understand. Thus MDD uses formalized models to precisely capture developer intent. The models can then be used to either refine or transform the requirements to a more detailed layer or to generate code artefacts out of them. Formalized models can be expressed in a Domain Specific Language, which is designed for an explicit purpose such as a software product family. “A well-defined DSL is a powerful implementation language, providing much greater rigor than a general purpose modelling language like UML” (Greenfield and Short, 2004). Additional improvements can be achieved if the abstractions of the model are used to generate a framework which guides the developer in completing the application.
- **Process Frameworks:** As with components, many process frameworks are too abstract and thus require rethinking about how to apply a process to a specific task. More specialized processes centre on the development of assets of a product within a software product line. Further gains in productivity can be achieved by integrating these processes into development environments, guiding and constraining the developers in their work. (Greenfield and Short, 2004)

As can be seen from the previously mentioned four critical innovations, the concept of software product lines, and software factories respectively, combines prevailing concepts like Model Driven Development (which can be seen as a part of Model Driven Engineering), Component Based Development and Software Reuse technologies into a holistic approach of software industrialization.

### 3. Particularities of Systems Integration

The field of systems integration (SI) comes with several particularities, distinguishing it from the domain of conventional software development. Systems integration has to challenge a multiplicity of technologies, once only technology combinations and a very high complexity of to be integrated systems. According to Vogler in (Vogler, 2004), potential problems can furthermore be categorized as follows:

Table 1: Integration problems and problem areas

<i>Problem Area</i>	<i>Problems</i>
Know-How	Lack of knowledge about potential solutions Unknown consequences of integration decisions
Management	Suboptimal degree of integration Unknown integration relationships High time pressure within the integration project No methodical approach Unknown complexity of the project Lack of standards
Information systems	Heterogeneity of systems to be integrated Lack of flexibility in legacy systems Data redundancy within different systems

#### 3.1. Know-How related

The problem area related to know how issues, embraces the lack of knowledge about potential solutions for a given problem. The multiplicity of different systems and technologies make it difficult for system engineers to select the optimal implementation. It is for example very unlikely that an expert for Siebel CRM Systems will also be an expert for SAP. Besides the technical implementation, it is also necessary to consider the pivotal business process during integration (Vogler, 2004). Furthermore, companies may not be aware of solutions and products available on the market and may not be able to develop integrated concepts for their IT landscape.

Another know-how related problem is the uncertainty of consequences if a system in a highly integrated environment is altered. This becomes especially evident as systems integration often occurs on a per project basis, implementing merely the prevailing requirements without aiming at a company wide integration concept. This may lead to  $n*(n-1)$  relationships between different systems and thus requires a very careful consideration of affected systems before conducting a change.

### **3.2. Management related**

One problem is the suboptimal degree of integration. According to (Vogler, 2004), two extremes can be found: isolated applications or highly integrated ones with peer-to-peer characteristics. The former is usually specialized in a particular task, not providing any interfaces to link it to other systems. While the former is hard to integrate, the latter is tightly interwoven with the IT landscape. Only very few enterprises consistently use a common architecture like a messaging middleware for example (Longo, 2001, Vogler, 2004).

Unknown integration relationships impose a problem on ad-hoc changes to information systems. Short and simple workarounds to quickly fix a problem may not be documented and thus remain unconsidered for potential changes. This lack of transparency prevents completeness and consistency checks for interfaces (Vogler, 2004).

High time pressure within the integration project may lead to the omittance of documentation and testing. Unfortunately both are crucial in an integrated environment as other systems rely on the interface descriptions and a credible service provisioning. However, a trade-off must be found between the efforts put into documentation and the benefits it generates.

To solve complex problems in software engineering, methodologies are being used (Heinrich et al., 2004) such as the Rational Unified Process or V-Model XT. While it is performed for years now, still no generally accepted methodology or approach for systems integration has been found (Vogler, 2004, Engel, 2006). This shortcoming is assumed to origin from the fact that integration is often seen as a purely technical problem which has to be resolved after completion of the underlying systems (Gassner, 1996).

Unknown consequences of changes to the IT landscape, unknown integration relationships or highly interweaved systems lead to a very high complexity which may remain unidentified, thus leading to increased cost and time to complete.

Similar to the previously depicted missing methodologies, systems integration also lacks generally accepted standards. This shortcoming is caused by the heterogeneity of applications (Vogler, 2004) and the fact that a prospective integration is unforeseeable during the development of applications. However, recent work in the field of Enterprise Application Integration (EAI) has developed fist concepts and frameworks, usually based on interapplication middlewares or Service Oriented Architectures, as for example in (Lee et al., 2003, Gorton and Liu, 2004, Sutherland and van den Heuvel, 2002, Strüver, 2006).

### **3.3. Information Systems related**

One of the core issues or particularities of systems integration is the heterogeneity of to be integrated systems (Longo, 2001, Stickel, 2001, Riem, 1997). Differences can not only be found on a technical layer (programming languages, operating systems) but also on a logical and conceptual layer (system architecture, frameworks, data structures) (Vogler, 2004). Both layers have a major influence on an adequate inte-

gration and thus need to be considered when implementing industrialization concepts in the field of systems integration. This heterogeneity anticipates the formation of standards as e.g. company wide integration architectures, which in turn leads to a discontinuity of media (media disruption). Furthermore, heterogeneity is reinforced by the fact that integrated systems are usually connected on a peer-to-peer basis with each other, leading to  $n*(n-1)$  relationships. As of the high costs of enterprise information systems, applications are usually not replaced frequently. “[...] SI aims at building applications that are adaptable to business and technology changes while retaining legacy applications and legacy technology as long as possible” (Hasselbring, 2000). This disadvantage further complicates systems integration due to insufficient reusability, outdated data management and user interfaces, monolithic constructions or inadequate maintainability (Vogler, 2004).

If different applications are merged into an integrated system, data and even functional redundancy may occur. Unless one data storage is a definite master or synchronization takes place, each transaction has to ensure that it works with the most actual data to prevent data inconsistency. In addition to the syntactical consistency of data, their semantics must also be ensured across different applications.

## **4. Industrialization in Systems Integration**

The focus of the intended research is aimed at the application of industrial production principles in the specific domain of systems integration. As described in chapter 3, systems integration differs in certain areas from the development of traditional software products. Especially the heterogeneity of products is one of the major differences (Longo, 2001, Stickel, 2001, Riem, 1997), which in turn leads to the question whether currently discussed industrialization concepts are suitable.

### **4.1. Software Factories in regard of SI particularities**

The present section will briefly discuss the SI particularities (q.v. Table 1 and chapter 3) in context of industrialized software development. It thereby centres around the idea of Software Factories, as the underlying concept comprises Component Based and Model Driven Development (as part of Model Driven Engineering), together with Software Product Lines, code generation and systematic reuse (Greenfield and Short, 2004). From the author’s point of view, Software Factories is currently the most advanced and comprehensive concept of software industrialization. Software Factories concentrate on the following four critical innovations to be introduced.

#### **4.1.1. Systematic reuse**

Greenfield & Short (Greenfield and Short, 2004) suggest to partition software engineering efforts into clearly delimited product lines. In doing so, design and development occur in a particular context, sharing common features and solving common problems conjointly. Product families may either be tailored around complete products or a series of related components. They concentrate on reusable implementation artefacts, as well as frameworks, processes and tools.

*“Program families enable a more systematic approach to reuse, by letting us identify and differentiate between features that remain more or less con-*

stant over multiple *products and those that vary*” (Greenfield and Short, 2004).

With reference to the particularities of systems integration, the multiplicity of different technologies, caused by high heterogeneity, inflexible legacy systems and different data sources, seems to be a major drawback to the definition of distinguished product lines. In a product line covering Customer Relationship Management (CRM) systems for example, products may be highly integrated with third party logistics and finance systems. Including all eventualities by supporting attached systems undermines the advantages of a delimited context, while excluding them will force development to occur outside the industrialized concepts. An additional drawback is the de-facto development of one-off solutions. Barely any development operates in the same environment or is interconnected with the same type of systems. The initial set-up cost for software product lines may therefore be contraindicative as the return of investment cannot be ensured.

#### 4.1.2. Development by Assembly

The second critical innovation is the logical consequence of systematic reuse. According to Greenfield & Short (Greenfield and Short, 2004), development by assembly itself has certain requirements which must be met: Platform independent protocols (e.g. XML), self-description of components (formalized and enhanced meta-data within components), deferred encapsulation (allowing to interweave new aspects), assembly by orchestration (machine controlled interaction and management of components), and architecture driven development (to promote the availability of well-matched components) (Greenfield and Short, 2004). The latter is seen to be most critical for development by assembly.

With regard to systems integration, the author does not see any major difficulties to technically apply development by assembly. However, the assembly approach relies on systematic reuse and thus on a methodical approach in a clearly delimited context, which may not be easy to define. This context also has an influence on the availability of predefined software architectures, as well as the number of reusable components. Furthermore, systems integration standards as e.g. service oriented architectures are not common until now (Lee et al., 2003, Gorton and Liu, 2004, Sutherland and van den Heuvel, 2002, Strüver, 2006). The most important challenge to be met is the definition of software architectures and standards in which development by assembly may occur.

#### 4.1.3. Model Driven Development

Model Driven Development, and in a greater sense Model Driven Engineering, raises the level of abstraction to alleviate increasing complexity and expressing domain concepts more efficiently (Schmidt, 2006) and context free. It consists of domain specific modelling languages, along with transformation engines and generators. The former allow a powerful description of the intended products of a product line, whereas the latter provide model transformation to a lower, more specific layer or eventually the generation of source code.

With regard to systems integration, the efforts required to define a domain specific language could become an obstacle, especially if applied to very small product lines. Furthermore, to automate the development process by generating source code or transforming models to a lower level, transformation engines and code generators have to be implemented. As directly related to domain specific languages, they are also product line specific. The integration aspect itself may be an additional challenge. Domain specific languages, models and architectures have to be compatible between the product lines whose products are to be integrated with each other.

#### 4.1.4. Process Frameworks

“The key to process maturity is preserving agility while scaling up to high complexity created by project size, geographical distribution, or the passage of time” (Greenfield, 2004a). While process frameworks like RUP, XP or Waterfall XT are widely available, Greenfield & Short (Greenfield and Short, 2004) demand an extensive customization of development processes to balance cumbersome formalism and agility. Depending on the selected product line features, the process framework can be further customized to support the development of the actual software. In a subsequent step, the process definitions may be incorporated into development tools, providing active guidance to the developer without hindering agility.

Yet again it comes back to clearly delineating a specific context, which is currently not given in the domain of systems integration. As with model driven development, process frameworks also need to be compatible to each other between different product lines in order to simplify integration. The incorporation of process definitions and process imposed restrictions or boundaries into development tools is a requirement which can hardly be solved by software development companies. The author assumes this to be subsequently solved by tool suppliers as software industrialization advances and becomes more accepted as a new development paradigm. It is therefore beyond the scope of the intended research.

## 4.2. Areas requiring further research

As can be seen in the previous section, existing concepts of software industrialization may not necessarily suit the particularities found in the field of systems integration. Thus further research is required to either adapt or enhance existing concepts, while considering how to align organizational structures to support the application of industrial production paradigms. Out of the previous sections, certain key questions arise, which will be briefly discussed in the following.

### 4.2.1. Organizational aspects

Organizational aspects focus on the surrounding conditions of industrialization in SI. They should be carefully considered before performing a paradigm shift throughout the organization.

1. How can we define areas of specialization in systems integration, considering the multiplicity of different technologies and their rare combinations within integration products?

The definition of narrow and clearly delimited problem domains seems inevitable for an industrialized production. With regard to systems integration, how can we carve out the combination of business domain knowledge with a multiplicity of different technologies? What is a reasonable organizational structure and how can we ensure that integration requirements can still be mapped onto the new organizational structure?

2. How can we measure the degree and success of software industrialization in systems integration?

In large organizations, efficient steering mechanisms are required. Conventional software engineering provides measures like function points per time unit for productivity or defects per function point for quality. But what are reliable measures to manage and monitor an industrialized production? Can we develop something like an Industrialization Maturity Model, similar to CMMI for example?

#### 4.2.2. Technological aspects

The technological aspects focus more on the actual implementation of critical innovations and key concepts within the context of systems integration.

3. Can we apply essential innovations of software industrialization to delimited problem domains within systems integration?

Given that an expedient classification of activities into e.g. product lines or services has taken place, can we still apply the essential innovations such as systematic reuse, development by assembly, model driven development and specialized process frameworks?

4. Are these essential innovations suitable for all application domains within systems integration, such as SAP, Siebel or PeopleSoft?

Many projects in the field of systems integration include development work for more sophisticated IT systems such as SAP for example. As these systems are often customized by using graphical development tools, how do concepts like component oriented or model driven development / engineering fit?

5. Which preconditions must be met to automate e.g. model transformation and code generation?

The probably most ambitious objective of an industrialized software development is the automated creation of artefacts such as model transformations to more detailed models or source code generation. Playing into organizational aspects as well, how high is the effort to implement such a concept? Do we need separate tools for each problem domain or can we reuse their foundation?

#### 4.2.3. Integrative aspects

The following research questions are closely related to technological aspects, as they discuss the interoperability of product domains between organizations.

6. How can we ensure compatibility between domain specific tools and assets of different problem domains?

Assumed key question 3 has successfully been answered and the critical innovations are implemented in clearly delimited problem domains, how can we ensure that we still can combine a multiplicity of technologies in an integration product? Are Domain Specific Languages compatible to each others or can we fit components of problem domain A into the framework of problem domain B? Systems usually need to be planned and designed in a holistic approach (at least on a coarse level).

7. Can industrialized systems integration be aligned along broadly accepted standards in the field of systems integration?

As discussed in chapter 3, systems integration lacks standards and methodical approaches and thus suffers from high heterogeneity. Is it reasonable to align the industrialization concept on broadly accepted standards (if available) in order to alleviate such problems in the future?

## 5. Summarization and Outlook

Systematic reuse of existing software artefacts hardly takes place and the majority of goods is still produced from scratch. With increasing complexity and size of today's IT systems, a generally accepted and industrialized production principle becomes necessary.

Promising approaches, notably Model Driven Development, Component Based Development, and Software Product Lines, are currently being developed and implemented, as described in chapter 2. The proposal of Software Factories by Greenfield & Short (Greenfield and Short, 2004) combines new and already existing concepts into a holistic approach of software industrialization. However, as software engineering takes place in a wide variety of application domains, we cannot be sure whether the available industrialization models can be applied to every one of them. One of these domains is systems integration in which IT-Systems are adapted and interconnected to support new business processes or business requirements. To better understand the particularities of this field, chapter 3 depicts its substantial differences. Consequently, chapter 4 discusses the suitability of existing concepts with reference to systems integration and identifies the following difficulties and shortcomings:

- A high heterogeneity in the projects of a systems integrator prevents the traditional implementation of software product lines, unless they are exceptionally narrow.
- Diverse technologies in a product family prevent building up technical expertise. Dedicated (technical) development teams per product line don't seem to be viable.
- The implementation efforts for setting up and maintaining the previously described "critical innovations" in small software product lines may consume potential savings.
- Organizational aspects and requirements of software industrialization in systems integration are yet unknown, especially with respect to the previously described difficulties and shortcomings.
- The lack of standardized frameworks and architectures in the field of systems integration may prevent an industrialized collaboration between enterprises, e.g. to form a software supply chain.

Section 4.2 subsequently identifies further areas of research and categorizes them into organizational, technological and integrative aspects. The first category is concerned with the future organizational structure of a systems integration organization, in respect of clearly delimited problem domains. Technological aspects cover the actual implementation of technical concepts, their suitability for particular areas, and preconditions for an increased level of automation. The final category, integrated aspects, deals with the compatibility of industrialized development methods across problem domains.

The present paper outlines particularities and potential challenges of industrialized systems integration, as well as further areas of research to get there. In order to pursue a structured approach and as some research topics depend on the answers of others, the author suggests the following redefined order and consequential structure of the research project, based on the key questions (KQ) in section 4.2:

- **KQ 1:** Elaboration of an organizational structure for industrialized systems integration with reference to the specialization in a heterogeneous environment, as well as (anticipating KQ 4) the application of critical innovations as depicted in section 4.
- **KQ 3:** Evaluation of the applicability of essential innovations as e.g. component oriented or model driven development to delimited problem domains of systems integration. This question should also bear cost and return on investment in mind.
- **KQ 6:** Analyse the interoperability of assets derived from different problem domains in order to support the fundamental concept of systems integration.
- **KQ 2:** Once the most fundamental concepts and questions are in place and answered, develop measures and metrics representing the degree and success of industrialization.
- **KQ 5:** With regard to the size of potential problem domains, identify the preconditions and efforts incurred with automated model transformation and code generation (if applicable).
- **KQ 4:** Identify problem domains with more sophisticated products and development tools such as SAP or Siebel and evaluate the applicability of software industrialization concepts in these particular areas.
- **KQ 7:** Provide an outlook on the interoperability of industrialized system integrators with regard to generic standards and frameworks in the field of systems integration.

The above depicted further research on major problems of industrialized systems integration will be conducted in close collaboration with representatives of the industry to obtain first hand experiences and validate the results of the latest research in practice. The obtained results of the particular problems will be presented within scientific papers and conference contributions, whereas the concluding dissertation will draw a holistic picture of industrialized systems integration and demonstrates methods and techniques to get there.

## 6. References

- Beltran, J. C. F., Holzer, B., Kamann, T., Kloss, M., Mork, S. A., Niehues, B. & Thoms, K. (2007) Modellgetriebene Softwareentwicklung, Frankfurt, entwickler.press.
- Brockhaus-Enzyklopädie. (1989) Brockhaus-Enzyklopädie. 19 ed. Mannheim, F.A. Brockhaus.
- Brown, A. W. & Wallnau, K. C. (1996) Component-Based Software Engineering: Selected Papers from the Software Engineering Institute, Los Alamitos, IEEE Computer Society Press.
- Butschek, F. (2007) Industrialisierung, Ulm, Ebner & Spiegel.
- Computer und Informationstechnologie. (2005) IN GREULICH, W. (Ed. Der Brockhaus. Leipzig, Mannheim, F.A. Brockhaus GmbH.
- Crnkovic, I., Hnich, B., Jonsson, T. & Kiziltan, Z. (2002) Specification, Implementation, and Deployment of Components. Communications of the ACM, 45, 6.
- Czarnecki, K. (1999) Generative Programming - principles and techniques of software engineering based on automated configuration and fragment-based component models. Illmenau, Technische Universität Illmenau.
- Dan, A., Johnson, R. D. & Carrato, T. (2008) SOA Service Reuse by Design. International Conference on Software Engineering Leipzig, ACM.
- Encyclopedia Britannica. (1991) Encyclopedia Britannica. 15 ed. Chicago, Encyclopedia britannica inc.
- Engel, T. (2006) Ein Beitrag zur unternehmensübergreifenden Integration von Informationssystemen. Institut für Rechneranwendung in Planung und Konstruktion. Karlsruhe, Universität Karlsruhe.
- Gassner, C. (1996) Konzeptionelle Integration heterogener Transaktionssysteme. St. Gallen, Universität St. Gallen.
- Gorton, I. & Liu, A. (2004) Architectures and Technologies for Enterprise Application Integration. International Conference on Software Engineering. Edinburgh, IEEE.
- Greenfield, J. (2004a) Problems and Innovations - Building Distributed Applications. Microsoft Architect Journal.
- Greenfield, J. (2004b) Scaling Up Software Development. Microsoft Architect Journal.
- Greenfield, J. & Short, K. (2003) Software Factories - Assembling Applications with Patterns, Models, Frameworks and Tools. International Conference on Object-Oriented Programming, Systems, Languages, and Applications. Anaheim, USA, ACM.
- Greenfield, J. & Short, K. (2004) Software Factories - Assembling Applications with Patterns, Models, Frameworks, and Tools, Indianapolis, John Wiley & Sons.
- Haines, G., Carney, D. & Foreman, J. (1997) Component Based Software Development / COTS Integration. Pittsburg, Carnegie Mellon University.
- Hasselbring, W. (2000) Information System Integration. Communications of the ACM, 43, 7.
- Heinrich, L., Heinzl, A. & Roithmayr, F. (2004) Wirtschaftsinformatiklexikon. 7 ed. München, Wien, Oldenbourg.
- Lee, J., Siau, K. & Hong, S. (2003) Enterprise Integration with ERP and EAI. Communications of the ACM, 46, 7.

- Lewandowski, S. M. (1998) Frameworks for Component-Based Client/Server Computing. *ACM Computing Surveys*, 30, 25.
- Lexikon der Kommunikations- und Informationstechnik. (2001) IN KLUßMANN, N. (Ed. *Lexikon der Kommunikations- und Informationstechnik*. 3 ed. Heidelberg, Hüthig Verlag.
- Longo, J. (2001) The ABCs of Enterprise Application Integration. *EAI Journal*, 3.
- Pham, H. N., Mahmoud, Q. H., Ferworn, A. & Sadeghian, A. (2007) Applying Model-Driven Development to Pervasive System Engineering. *International Conference on Software Engineering*. Minneapolis, IEEE Computer Society.
- Riem, R. (1997) *Integration von heterogenen Applikationen*. St. Gallen, Universität St. Gallen.
- Schmidt, D. C. (2006) Model Driven Engineering. *IEEE Computer*, 39, 7.
- Schryen, G. (2001) *Komponentenorientierte Softwareentwicklung in Softwareunternehmen: Konzeption eines Vorgehensmodells zur Einführung und Etablierung*, Wiesbaden, Deutscher Universitäts-Verlag.
- Software Engineering. (1968) IN NAUR, P. & RANDELL, B. (Eds.) *NATO Software Engineering Conference*. Garmisch Partenkirchen, NATO Science Committee.
- Stickel, E. (2001) *Informationsmanagement*, München, Wien, Oldenbourg.
- Strüver, S.-C. (2006) *Standardisiertes EAI-Vorgehen am Beispiel des Investment Bankings*, Berlin, GITO.
- Sutherland, J. & Van Den Heuvel, J. (2002) Enterprise Application Integration and Complex Adaptive Systems. *Communications of the ACM*, 45, 6.
- Szyperski, C. (1998) *Component Software: Beyond Object-Oriented Programming*, Massachusetts, Addison-Wesley.
- Vogler, P. (2004) *Prozess- und Systemintegration - Evolutionäre Weiterentwicklung bestehender Informationssysteme mit Hilfe von Enterprise Application Integration*, Wiesbaden, Deutscher Universitäts-Verlag.

# Industrializing Software Development in Systems Integration

Matthias Minich, M. Sc.	Prof. Dr. B. Harriehausen-Mühlbauer	Prof. Dr. C. Wentzel
Univ. of Appl. Sciences Darmstadt, Germany University of Plymouth, United Kingdom matthias.minich@plymouth.ac.uk	Univ. of Appl. Sciences Darmstadt, Germany b.harriehausen@fbi.h-da.de	Univ. of Appl. Sciences Darmstadt, Germany c.wentzel@fbi.h-da.de

**Abstract.** Today's economy is driven by permanent change, causing restructuring and merger & acquisitions between enterprises. Consequential process adaptations and realignments often result in systems integration and software development projects, which are still reliant on craftsmanship of highly skilled workers. As of its distinctive characteristics, it seems questionable if recent concepts of software industrialization can be applied to the field of systems integration as well. The present paper briefly depicts these concepts, discusses the particularities of systems integration and points out the difficulties to combine both. The current state of science is assessed and potential gaps are identified. Based hereon the paper suggests further areas of research, describes the intended methodology and defines the primary research questions.

## Software Industrialization

From a generic point of view, the term industrialization is defined as the dissemination of industries within an economy, in proportion to agriculture, handicraft and small trade. Relating to the production of goods and services, it is defined as the implementation of standardized and highly productive methods in order to increase efficiency and reduce cost [Enc91, S. 304–305].

Software development however is “[...] slow and expensive, and yields products containing serious defects that cause problems of usability, reliability, performance and security” [GSC04, S. XV]. Compared to other industries it lacks the industrial key concepts of specialization, standardization & systematic reuse and automation. In the software industry, standardization would allow common structures within applications and thus the interchangeability of artefacts (e.g. source code, templates, development resources) between different products. Furthermore, the value and effectiveness of such artefacts can be enhanced by narrowing down their scope and specialising in clearly delimited product families. Subsequently, software components and development resources could systematically be reused for different members of a product family. In a final step, domain specific design languages and dedicated code generators can be developed, which would allow for a certain degree of automation.

The objectives of every software development project can be categorized into quality, quantity, time and cost [Bal08, S. 196; Sne87, S. 42]. Harry M. Sneed depicted their interaction as the Devil's Square [Sne87, S. 42], in which the four factors are in an antagonistic relationship. As the available productivity of the performing organization is limited and cannot satisfy all needs, tradeoffs have to be made. For example, doing more work in a higher quality will result in higher cost and a longer development time. However, by applying industrial methods and thus increasing productivity, we can increase quality and product complexity, and at the same time reduce cost and production time.

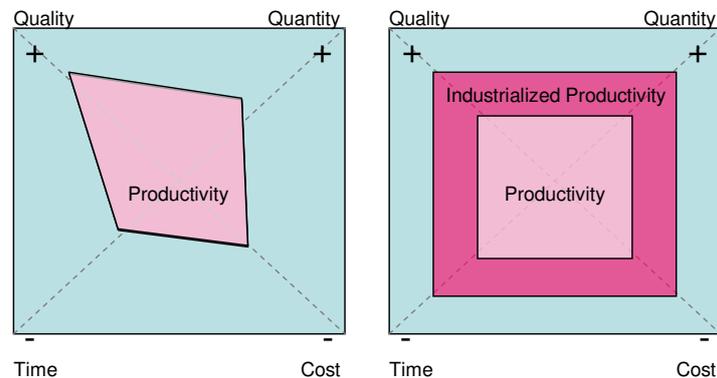


Fig. 1. Sneed's Devil's Square: Tradeoff vs. Industrialized Productivity

Compared to conventional industries, software engineering needs suitable mechanisms to support the industrial key concepts of standardization, specialization & systematic reuse, and automation. However, recent developments in the area show that the previously mentioned key principles of industrialization can be achieved.

- **Software Product Lines:** It seems to be very difficult or even impossible to determine how mechanisms for reuse or automation should be implemented in an arbitrary context. Systematic reuse must be planned for and cannot occur coincidentally. A Software Product Line (SPL) therefore spans a clearly delimited frame around a family of software products, sharing “[...] a common, managed set of features satisfying the specific needs of a particular segment or mission” [CN07, S. 5]. The concept requires to separate product development from product line development. The former produces the actual software product, while the latter produces the required assets to support and automate the development process. By concentrating on a clearly delimited scope, production assets can be much more powerful as, for instance, reusable components or frameworks and architectures. However, specialization alone would not allow for any diversification as required by different customers. Software Product Lines therefore identify recurring functionality and points of variation to define a common framework or architecture in which customer specific requirements can be considered. This concept is also known as mass customization in other industries.

- **Component Based Development:** One of the first ideas of using component based development as an industrial principle came up in October 1986 on the NATO conference on software engineering, where M.D. McIlroy suggested to develop applications by assembling previously produced components, as most of a software's functionality has already been developed or will be required in many other applications as well [McI69]. In his book about component software [Szy99, S. 27 ff.], Szyperski defines a component as follows:

*“A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”*

By using current component standards, it is possible to encapsulate business logic within reusable software building blocks. The context in which this occurs can be set by Software Product Lines. They define a delimited scope to employ reusable components for the development of new product line members. Current CBD standards define the requirements such a component has to fulfil from a syntactic and semantic point of view. They furthermore define interface specifications, component allocation and component interaction across different programming languages and platforms. The underlying architecture can also be provided by these technologies in a way that it becomes possible to completely implement the commonalities of a certain product line, while allowing to “plug in” customer specific requirements [GSC04, S. 426–427]. Component Based Development therefore very well reflects the key concept of standardization & systematic reuse. The four most widely adopted component technologies today are Sun's Java 2 Enterprise Edition (J2EE), the Corba Component Model (CCM) by the Object Management Group, Microsoft's Distributed Component Object Model (DCOM), as well as their .NET Framework.

- **Model Driven Engineering:** Model Driven Engineering depicts the third key concept of industrialization. It aims to raise the level of abstraction in order to fill the gap between the problem solution and the technical implementation, bringing the latter closer to a vocabulary understood by subject matter experts [GSC04, S. 142]. Omitted details are subsequently added until detailed models or even executable code is available. Of course this process is by far not trivial: The extensive degree of freedom and context sensitivity becomes an issue if the model is to be interpreted by a code generator. To overcome this problem, MDE combines Domain Specific Languages and Transformation Engines & Generators [Sch06, S. 27]. Both are uniquely designed for a particular application domain, reducing the degree of freedom and possible contexts by providing a clearly specialized vocabulary and grammar. Once a system has been defined with an appropriate DSL, the resulting set of models can be transformed into either intermediate models or directly generated into source code. Well known examples of domain specific languages are, for instance, editors for graphical user interfaces or event driven process chains.

## Systems Integration

In today's business world, IT faces high demands in quickly adopting to new requirements. As legacy systems often do not offer the flexibility to do so, new systems are implemented which need to interact with the existing IT landscape. This situation inevitably leads to systems integration efforts, joining the different subsystems into a cohesive whole, in order to alleviate functionality or data access via a common interface [Fis99, S. 86; LN07, S. 3].

Systems Integration deals with the steps required to move an IT system from a given degree of integration to a higher one, by merging distinct entities into a cohesive whole or integrating them into already existing systems [Rie97, S. 10; Fis99, S. 86]. This process of integration can be further divided into data integration and application integration [CHK06, S. 11; LN07, S. 3–5]. Data integration concentrates on the integration of different data sources, for instance, data consolidation or data warehousing. Application integration in turn covers the combination of different software systems supporting business processes. It is also referred to as Enterprise Application Integration (EAI) with the following interpretations [Vog06, S. 52]:

- EAI as an integration middleware solution
- EAI as a high level integration on a semantic or processual level
- EAI as an integration framework architecture
- EAI as an approach to implement business requirements, including strategic and process related considerations, utilizing different integration techniques

The present research focuses on Enterprise Application Integration and adheres to the fourth definition of the term, i.e. EAI as an integration approach from a strategic, process and technology oriented perspective. It does so because each layer may have severe influence on its neighbouring ones and thus may not be considered in isolation.

### Dimensions of integration

Within the previously adopted definition of the term EAI, literature usually defines several layers or dimensions of integration. They start from a strategic and business process point of view, through process partitioning for different systems, to the actual data and functionality management on an implementation level. Vogler for example defines process, desktop and systems as the three subdomains of integration [Vog06, S. 53–57]. Hasselbring offers a similar classification in [Has00] by defining a business, application and technology layer. Another example can be found in Fischer [Fis99, S. 90], who separates the term into a business, organizational, functional and technical dimension.

Taking the previous definitions into consideration, the present paper defines the following three dimensions of integration:

- **Business Process:** On the business process dimension, the organizational objectives, structure and core business processes of an enterprise are

characterized. They define which business functionality and information is required and how the involved IT systems must interact from a semantic point of view. Integration decisions on this dimension are usually driven by mergers & acquisitions, collaboration agreements, or realignment of company objectives, and depict major changes in an organization's business model.

- **Workflow:** The workflow dimension subdivides a business process into distinct activities and maps these to the different IT systems. It defines the data sources and functionality required from the available IT systems from a technical point of view, as well as the interaction among each other and with the end users. On the strategy dimension these data sources and functionalities map to the semantic steps of the business process. Integration decisions on this dimension may inherit from higher or lower dimensions, or are driven by process adaptations due to regulatory influences or improvement activities, for instance.
- **Technology:** Information and communication infrastructure of an integrated systems landscape is implemented at the technology domain. It defines which applications may access which data or functionality, how this is done, and how data management (e.g. redundancy) takes place. Integration decisions on this dimension inherit from higher dimensions, or are driven by technological changes, such as introducing or replacing applications.

### Common problems in systems integration

Despite systems integration supports and simplifies the execution of business processes, it comes with several particularities and challenges during implementation. Based on Vogler in [Vog06, S. 24–31], the following sections briefly describe the potential problems per integration domain:

- **Business Process domain:** New business processes are often defined from a semantic point of view or are based on preconceptions from earlier projects, regarding their representation in IT systems. As the IT landscape has direct implications on the business processes, it is important to understand the integration relationships in order to *identify and choose an optimal solution*. In many companies however, the particular situation is hardly known. Similarly the unawareness of the particular IT landscape may lead to *unforeseen consequences*. Only minor changes in a process may require adaptations of the underlying systems, and thus other processes due to changed interfaces or data structures. If these consequences were known early enough, business processes could be designed around them.
- **Workflow domain:** Subdividing a business process into workflows and depicting them ad hoc on different IT systems may lead to a *suboptimal degree of integration*. Such a point to point instead of shared integration architecture leads to  $n*(n-1)$  relationships, making later changes more complex. In such an environment the *integration relationships may be unknown* due to insufficient documentation and the lack of a big picture. New implementations may be redundant and the consistency and integrity of interfaces cannot be ensured. A

prominent example was the Y2K problem where it was hardly known which systems relied on the data to be changed.

Furthermore, a *methodological approach* for systems integration projects is hardly used. Despite suitable methodology has been defined in literature recently, it has not yet been adopted in the industry. This also becomes evident as SI is not sufficiently considered in current software development models [Gas96, S. 32].

Due to uncoordinated efforts and the lack of methodologies, integrated systems show a *high complexity*, leading to increased time and cost for future adaptations.

Heterogeneity, caused by the previous problems, prevents the implementation of holistic *integration platforms or architectures* within enterprises. Although there are certain middleware systems or transaction monitors in place, these are often not part of a bigger picture or integration concept.

- **Technology domain:** From a technical point of view, *heterogeneity* is the major issue in systems integration. Depending on the differences, data representation and functionality, as well as underlying technologies, must be aligned. The required effort disproportionately rises with the number of systems to be integrated, unless a common architecture or platform is used.

Another big problem is the integration with *legacy applications*. These were often designed as stand alone solutions with no integration in mind. Obsolete data management, interfaces, or a lack of documentation or maintenance make integration extremely difficult. Furthermore these systems often cannot be altered or replaced and therefore impose restrictions on the overall integration concept.

The final issue lies in the *redundancy of data*. In integrated environments it becomes difficult to define which data resides where, how it is accessed and how redundancy is managed. Without such management, information may easily become outdated and inconsistent, leading to serious issues in business process execution.

## Industrialized Systems Integration

As discussed in the first chapter, Software Product Lines, Component Based Development and Model Driven Engineering represent specialization, standardization & systematic reuse, and automation, for software development. The respective concepts are well understood and first literature is available on combining them in factory like development environments [GSC04].

The second chapter shows the particularities systems integration comes with, distinguishing it from the domain of conventional software development. It has to challenge a multiplicity of technologies, inflexible legacy systems, once only technology combinations and a very high complexity. It seems disputable whether the concepts for industrialized software development in their original form can be applied to the field of systems integration, as discussed in the following:

- **Software Product Lines:** In Software Product Lines, design and development occur in a particular context, sharing common features and solving common problems. Product families may either be tailored around complete business

solutions or a series of related products. They concentrate on reusable implementation artefacts, as well as frameworks, processes and tools.

With reference to systems integration, the multiplicity of different technologies, caused by high heterogeneity, inflexible legacy systems and different data sources, seems to be a major drawback to the definition of distinguished product lines. In a product line covering Customer Relationship Management (CRM) systems for example, products may be highly integrated with third party logistics and finance systems. Including support for any potentially attached systems undermines the advantages of a delimited context, while excluding them will force development to occur outside the industrialized concepts. An additional drawback is the de-facto development of one-off solutions per customer. Barely any solution operates in the same environment or is interconnected with the same type of systems. The initial set-up cost for software product lines may therefore be contraindicative as the return of investment cannot be ensured.

- **Component Based Development:** According to Greenfield & Short [GSC04, S. 130], development by assembly with software components has certain requirements which must be met: Platform independent protocols (e.g. XML), self-description of components (formalized and enhanced meta-data within components), deferred encapsulation (allowing to interweave additional functionality), assembly by orchestration (machine controlled interaction and management of components), and architecture driven development (to promote the availability of well-matched components).

With regard to systems integration, the author does not see any major difficulties to technically apply development by assembly. However, the assembly approach relies on systematic reuse and thus on a methodical approach in a clearly delimited context, which may not be easy to define. This context also has an influence on the availability of predefined software architectures, as well as the number of reusable components. Furthermore systems integration standards, for instance the TOGAF framework, are not common as of today. The most important challenge to be met is the definition of a component based systems integration architecture in which development by assembly may occur.

- **Model Driven Engineering:** Model Driven Development, and in a greater sense Model Driven Engineering, raises the level of abstraction to reduce complexity and express business concepts more efficiently. It consists of domain specific modelling languages and model transformation engines & code generators. The former allow a context free description of the intended products of a product line, whereas the latter provide model transformation to a lower, more specific model or eventually the generation of source code.

For systems integration, the efforts required to define a domain specific language (DSL) could become an obstacle, especially if applied to product lines with a limited number of expected products. With reference to Software Product Lines, the scope of a DSL cannot be clearly delimited as each product may need to be integrated with other external systems. Furthermore, to automate the development process by transforming models to a lower level or generating source code, transformation engines and code generators have to be implemented which also impose high set up cost.

## Areas requiring Further Research

As shown above, traditional software development processes neither sufficiently support systems integration, nor the application of specialization, standardization, systematic reuse and automation. In turn, implementations of industrial key concepts in their basic occurrence, such as CBD, MDE or SPLs, also don't seem to be suitable for systems integration as discussed in the previous section.

Further research is therefore required to elaborate a software development approach, which allows following the industrial paradigm while considering the particularities of systems integration. From the author's point of view, the topic is best approached from the industrialization perspective, as it depicts the fundamental concepts to advance from handcrafted, singular manufacturing to automated, mass-market production. As this involves fundamental changes in software development methods as well as software architectures, these methods must be applied first. In a subsequent and iterative approach, the requirements of systems integration can then be woven in.

The previous considerations lead to the following major research questions:

1. Migrating from project and technology oriented development to Software Product Lines will require a reorganization of the company [PBL05, S. 9]. With specialization in mind, how should a systems integration provider be organized to develop its products in an industrialized manner? Can enterprises afford to organize themselves in fully featured Software Product Lines, or are other forms of organization more feasible?
2. In a second step the assessment of CBD for systems integration within software product lines seems viable. Given that an expedient classification of software products into product lines or families has taken place, is it possible to define software components to be reused in different integration solutions for different customers? With reference to the common problems of systems integration, is it possible to combine component based architectures with systems integration frameworks such as TOGAF for example?
3. The probably most ambitious objective of an industrialized software development is the automated creation of artefacts. It offers interesting possibilities to resolve problems related to the business process and workflow domain of SI, such as integration consequences and depicting intersystem relationships. But to which extend can a systems integration provider economically implement such a concept, and can it be used for different customers? Which advantages do domain specific languages offer in the given context? Are separate tools such as model transformers or code generators required for each product line or can their foundations be reused?

The present research can be classified into the scientific area of business informatics as it covers matters from business (organizational forms of enterprises and product family management), and computer sciences (implementation of CBD and MDE). It will be conducted in close collaboration with the industry, meeting concerns about the fact that very little of software engineering research finds its way into

practice [Pot93, S. 19]. The obtained results of the research questions will be presented within scientific papers and conference contributions, whereas the concluding dissertation will draw a holistic picture of industrialized systems integration and provide a software development approach that addresses the application of industrial concepts in systems integration.

## Bibliography

- [Bal08] Balzert, Helmut (2008): Lehrbuch der Softwaretechnik. 2. Aufl. Heidelberg: Spektrum Akad. Verl.
- [CHK06] Conrad, Stefan; Hasselbring, Wilhelm; Koschel, Arne (2006): Enterprise Application Integration. Grundlagen Konzepte Entwurfsmuster Praxisbeispiele. 1. Aufl. Herausgegeben von Stefan Conrad. München, Heidelberg: Elsevier Spektrum Akad. Verl.
- [CN07] Clements, Paul; Northrop, Linda (2007): Software product lines. Practices and patterns. [Nachdr.]. Boston: Addison-Wesley.
- [Enc91] N.N. (1991): Industrial Revolution. In: The new encyclopaedia Britannica. In 32 volumes. 15. ed., 1991. Chicago, Auckland, Geneva, London, Madrid, Manila, Paris, Rome, Seoul, Sydney, Tokyo, Toronto: Encyclopaedia Britannica, Bd. 6, S. 304–305.
- [Fis99] Fischer, Joachim (1999): Informationswirtschaft Anwendungsmanagement. München, Wien: Oldenbourg.
- [Gas96] Gassner, Christian (1996): Konzeptionelle Integration heterogener Transaktionssysteme. Dissertation. St. Gallen: Universität St. Gallen, Institut für Wirtschaftsinformatik.
- [GSC04] Greenfield, Jack; Short, Keith; Cook, Steve (2004): Software factories. Assembling applications with patterns, models, frameworks, and tools. Indianapolis: Wiley.
- [Has00] Hasselbring, Wilhelm (2000): Information System Integration. In: Communications of the ACM, Jg. 43, H. 6, S. 32–38.
- [LN07] Leser, Ulf; Naumann, Felix (2007): Informationsintegration. Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen. 1. Aufl. Heidelberg: dpunkt-Verl.
- [McI69] McIlroy, Douglas (1969): Mass Produced Software Components. In: Naur, Peter; Randell, Brian (Hg.): Software Engineering. Report on a conference sponsored by the NATO SCIENCE COMMITTEE. Bussels, S. 138–156.
- [PBL05] Pohl, Klaus; Böckle, Günter; Linden, Frank (2005): Software product line engineering. Foundations, principles, and techniques ; with 10 tables. Berlin: Springer.

- [Pot93] Potts, Colin (1993): Software-Engineering Research Revisited. In: IEEE Software, Jg. 10, H. 5, S. 19–28.
- [Rie97] Riehm, Rainer (1997): Integration von heterogenen Applikationen. Dissertation. St. GallenUniversität St. Gallen.
- [Sch06] Schmidt, Douglas (2006): Model-Driven Engineering. In: IEEE Computer, Jg. 39, H. 2, S. 25–31.
- [Sne87] Sneed, Harry M. (1987): Software-Management. Köln: Müller.
- [Szy99] Szyperski, Clemens (1999): Component software. Beyond object-oriented programming. Harlow: Addison-Wesley.
- [Vog06] Vogler, Petra (2006): Prozess- und Systemintegration. Evolutionäre Weiterentwicklung bestehender Informationssysteme mit Hilfe von enterprise application integration. 1. Aufl. Wiesbaden: Dt. Univ.-Verl.

# Software Industrialization in Systems Integration

Matthias Minich  
Prof. Dr. B. Harriehausen-Muehlbauer  
Prof. Dr. C. Wentzel

**Abstract**— Today's economy is in a permanent change, causing merger and acquisitions and co operations between enterprises. As a consequence, process adaptations and realignments result in systems integration and software development projects. Processes and procedures to execute such projects are still reliant on craftsmanship of highly skilled workers. A generally accepted, industrialized production, characterized by high efficiency and quality, seems inevitable.

In spite of this, current concepts of software industrialization are aimed at traditional software engineering and do not consider the characteristics of systems integration. The present work points out these particularities and discusses the applicability of existing industrial concepts in the systems integration domain. Consequently it defines further areas of research necessary to bring the field of systems integration closer to an industrialized production, allowing a higher efficiency, quality and return on investment.

**Keywords**—Software Industrialization, Systems Integration, Software Product Lines, Component Based Development, Model Driven Development.

## I. INDUSTRIALIZATION

FROM a generic point of view, the term industrialization is defined as the dissemination of industries within an economy, in proportion to agriculture, handicraft and small trade. Relating to the production of goods and services, it is defined as the implementation of standardized and highly productive methods in order to increase efficiency and reduce cost [1, 2]. The process of industrialization began at the end of the 18th century in Great Britain and was characterized by an increasing division and specialization of labor, capital intensive technologies, mass production, rationalization and the application of new energy sources [2]. Industrialization is seen as a necessary step for economic growth, technological advances and increasing wealth. Only industrial production methods allow to produce a multiplicity of goods in a

sufficient amount and quality [2].

The present paper will focus on the application of standardized and highly productive methods to the field of software development in systems integration. Applied to the process of industrialization as introduced above, the key concepts of such methods can be summarized in specialization, standardization & systematic reuse, and automation.

As of today, the above principles can be found in almost all industries at different levels of penetration. Standardization and specialization advance the level of reuse and enable automation of rote and menial tasks, whereas creative tasks (that cannot be standardized), such as product design, are still performed by highly skilled workers. Omitting the availability of the required commodities and energy, the fundamental principles of industrialization can be described as follows.

### A. Specialization

In the given context, the term specialization describes the concentration of an economic subject (worker, business, society, etc.) to a particular area within a larger scope, such as certain industries, product families, technologies or skills. A production process is subdivided into less complex functions that can be assigned to well-trained workers or purpose-built machinery. This division of labor allows the specialization of individuals, expanding their knowledge and abilities in a particular area. In turn, they achieve a higher efficiency and quality. Specialization also allows reusing production or product artifacts. The former for example include processes, tools and machinery, while the latter include architectures, frameworks and components. Systematic reuse can only occur in a precisely delimited scope, defined by specialization and standardization [3].

The disadvantages of specialization lie in a reduced flexibility and thus the dependency on market demand of the area or skill in scope, as well as the dependency of upstream production. A highly specialized economic subject cannot quickly change its area of focus. A farsighted, strategic planning of specialization is mandatory.

Well known implementations of specialization can, for instance, be found in the automotive sector. A whole industry subcontracting to automotive manufacturers emerged, specializing in certain product families such as engines, brake systems or electronic control units. Furthermore, employees

M. Minich is with the University of Plymouth, Plymouth, PL4 8AA United Kingdom and the University of Applied Sciences Darmstadt, 64283 Darmstadt, Germany (phone: +49 1713885673; e-mail: [matthias.minich@plymouth.ac.uk](mailto:matthias.minich@plymouth.ac.uk)).

Prof. Dr. B. Harriehausen-Muehlbauer is with the University of Applied Sciences Darmstadt, 64283 Darmstadt, Germany (phone: +49 6151 16-8485; e-mail: [b.harriehausen@fbi.h-da.de](mailto:b.harriehausen@fbi.h-da.de)).

Prof. Dr. C. Wentzel is with the University of Applied Sciences Darmstadt, 64283 Darmstadt, Germany (phone: +49 6151 16-8459; e-mail: [c.wentzel@fbi.h-da.de](mailto:c.wentzel@fbi.h-da.de)).

specialize in particular skills and tasks in the production process.

### B. Standardization & Systematic Reuse

Standardization describes the unification of specific attributes of production or product artifacts. The objective is to establish a common understanding of these attributes in order to exchange artifacts, integrate upstream work products, align production processes or simplify information exchange [4]. Together with specialization, standards provide the base for systematic reuse. Only if an artifact follows clearly defined principles, it can be reused as is in another product. Standards can be officially defined (by a binding regulation or contract), de facto (by market position or dominant usage), or voluntary.

With regard to market position or profit margin they can also be disadvantageous as standards encourage competition between suppliers. Furthermore, they may require tradeoffs in functionality that may affect a unique selling point of the own product or the lack of customization possibilities.

A good example of standardization can be found in the modular construction system of automotive manufacturers. Uniquely designed product artifacts such as axles or suspensions can be reused in many different models of a product family. Likewise, production artifacts such as assembly lines, tools or machinery, can be reutilized to produce many different products.

### C. Automation

By division of labor, standardization, and systematic reuse, rote, menial or dangerous tasks can be taken over by purpose-built machinery. The operational sequence, regulation and monitoring of the production process is also performed by technical equipment. Such machines often are more precise and time & cost efficient as compared to human workers. Important prerequisites are specialization and standardization, as machinery cannot solve unknown problems. In an industrialized production, the worker's role shifts towards planning, monitoring and correction of the production process. The objective here also is to reduce cost and time and increase quality.

Drawbacks in automation inherit from the previously mentioned principles. High upfront investments require a minimum utilization rate to break even. Reduced flexibility implicates a high market dependency of the segment in scope.

Automation is as well an important factor in the automotive sector. The industry heavily relies on automated production such as welding robots or automated assembly lines.

## II. INDUSTRIALIZED SOFTWARE DEVELOPMENT

Software development is “[...] slow and expensive, and yields products containing serious defects that cause problems of usability, reliability, performance and security” [3]. At the beginning of this chapter, industrialization was defined as a method to increase efficiency and quality and to reduce cost by implementing standardized and highly productive methods. The objectives of every software project can be categorized

into quality, quantity, time and cost [5, 6]. Harry M. Sneed depicted their interaction as the Devil's Square [7], in which the four factors are in an antagonistic relationship. As the available productivity of the performing organization is limited and cannot satisfy all needs, tradeoffs have to be made. For example, doing more work in a higher quality will result in higher cost and a longer development time. However, by applying industrial methods and thus increasing productivity, quality and product complexity can possibly be increased and at the same time cost and production time reduced.

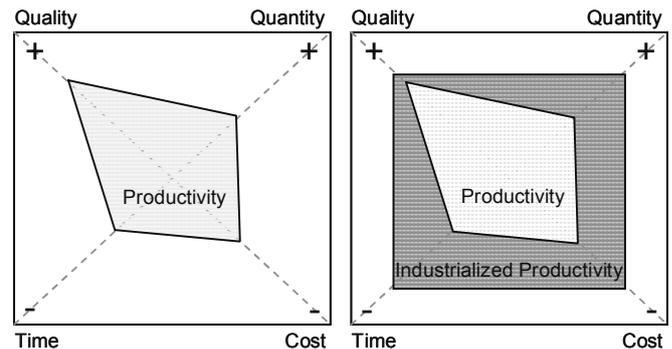


Fig. 1 Sneed's Devil's Square: Dimensional tradeoff versus industrialized productivity

Several efforts have been taken to apply such methods to software development. Referring back to the previous chapter, the key industrial principles now can also be found in the field of software engineering. *Specialization* is represented by Software Product Lines, *Standardization & systematic reuse* may be found in Component Based Development, and *Automation* can be achieved with Model Driven Engineering. Unfortunately, the most important concept, specialization, was invented last. As of gracious generality, caused by the lack of a clearly delimited scope, Component Based Development and Model Driven Engineering in their initial occurrence seem to have failed [3]. Only recently all the concepts are in place and can be used to facilitate industrialized software development, as for example described in Greenfield & Short's book "Software Factories" [3]. The referenced concepts will be briefly described in the following.

### A. Software Product Lines

The latest and maybe most important concept is the one of Software Product Lines that maps to the industrial principle of specialization. It seems to be very difficult or even impossible to determine how mechanisms for reuse or automation should be implemented in an arbitrary context. Systematic reuse must be planned for and cannot occur coincidentally. A Software Product Line (SPL) therefore spans a clearly delimited frame around a family of software products, sharing “[...] a common, managed set of features satisfying the specific needs of a particular segment or mission” [8]. It first emerged in 1995 in a Swedish naval software firm and was further developed at the Carnegie Mellon Software Engineering Institute [8]. The concept requires to separate product development from

product line development. The former produces the actual software product, while the latter produces the required assets to support the development process. By concentrating on a clearly delimited scope, production assets can be much more powerful as, for instance, reusable components or frameworks and architectures. However, specialization alone would not allow for any diversification as required by different customers. Software Product Lines therefore identify recurring functionality and points of variation to define a common framework or architecture in which customer specific requirements can be considered. This concept is also known as mass customization in other industries.

During actual product development, knowledge and reusable assets, such as business functionality components, are captured to include them in the Software Product Line for future products. The following figure depicts the described concept:

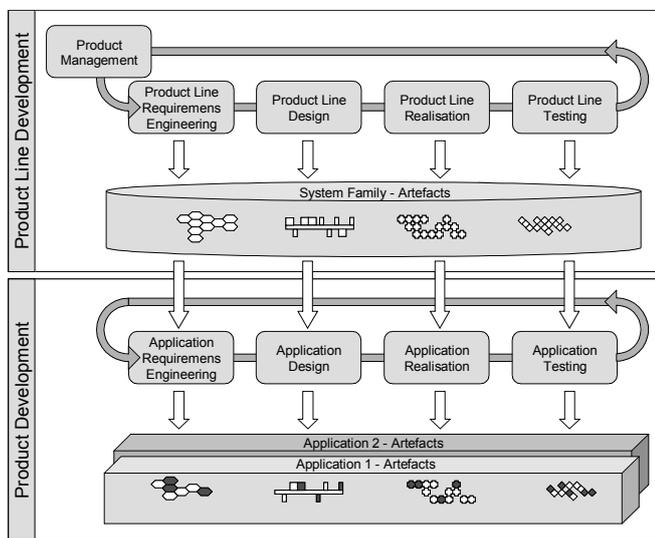


Fig. 2 Software development in a Software Product Line (q.v. [9])

*Product line* developers produce product and production assets, utilized by *product* developers to produce the particular family member. During product development new assets are created and fed back into the product line. It is therefore very important that any customer specific variability is developed with a potential reuse in mind. Further advantages can also be found in a higher quality and a shorter time to market: As reusable product assets are reviewed, implemented and tested in many different products, chances to find faults and correcting them are significantly higher [10]. Furthermore, a once identified fault can be corrected before it becomes evident in other products. Although time to market is higher in the beginning due to product line development, it decreases significantly once assets are in place that can be reused for each new product [10].

Of course, this specialization to a particular segment or mission is not for free. Upfront investments are required to define the scope and the initial asset base for the Software Product Line. Unlike in manufacturing industries, these costs

cannot be recovered by economies of scale as software can be copied very easily and customer requirements are hardly the same. SPL must therefore focus on *economies of scope*, producing distinct but similar products, all based on a common set of functionality. Literature suggests about three systems to reach the break-even-point as compared to conventional, one-off development [8].

### B. Component Based Development

One of the first ideas of using industrial principles came up in October 1986 on the NATO conference on software engineering. It can be mapped to the industrial principle of standardization which is the foundation for the exchange of artifacts and systematic reuse. In his contribution “Mass produced Software Components” [11], M.D. McIlroy suggested to develop applications by assembling previously produced components, as most of a software’s functionality has already been developed or will be required in many other applications as well. In his book about component software [12], Szyperski defines such a component as follows:

*“A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”*

As in manufacturing industries, systematic reuse requires a clearly delimited context. It is for example much easier to build GUI components for Microsoft’s .NET platform than within an arbitrary context [3]. By using current component standards it is possible to encapsulate business logic within reusable software building blocks. The context in which this occurs can be set by Software Product Lines. They define a delimited scope to employ reusable components for the development of new product line members. Current CBD standards define the requirements such a component has to fulfill from a syntactic and semantic point of view [13]. They furthermore define interface specifications, component allocation and component interaction across different programming languages and platforms. The underlying architecture can also be provided by these technologies in a way that it becomes possible to completely implement the commonalities of a certain product line, while allowing to “plug in” customer specific requirements [3].

The four most widely adopted component standards today are Sun’s Java Platform Enterprise Edition (Java EE), the Corba Component Model (CCM) by the Object Management Group, and Microsoft’s Distributed Component Object Model (DCOM), as well as their .NET Framework. All of them support language independent integration of other components or systems by providing clearly defined interfaces and data types which can be accessed on a binary level. CCM and Java EE are also platform independent.

### C. Model Driven Development

The final aspect of industrialization, automating certain tasks, can be achieved with Model Driven Development (MDD) and was initiated by Computer Aided Software

Engineering (CASE) in the 1980s. It encouraged development methods based on graphical representations of software (models) with state machines, structure diagrams or dataflow diagrams [14] to generate source code. The graphical representations, however, were too generic to precisely describe the intended solution and did poorly map to the underlying technologies. The result was highly complex source code which had to be altered by hand. The corresponding models were out of date very soon as the CASE tools could hardly depict manual changes to the code. Today, model driven engineering has been further advanced, overcoming the problems discovered with CASE tools.

Using visually represented models as a description of software, it aims to raise the level of abstraction in order to fill the gap between the problem solution and the technical implementation, bringing the latter closer to a vocabulary understood by subject matter experts [3]. Omitted details are subsequently added until executable software is available. Of course, this process is by far not trivial: The extensive degree of freedom and context sensitivity becomes an issue if the model is to be interpreted by a code generator. To overcome this issue, MDD combines Domain Specific Languages and Transformation Engines & Generators [14]. Both are uniquely designed for a particular application domain, reducing the degree of freedom and possible contexts by providing a clearly specialized vocabulary and grammar. Once a system has been defined with an appropriate DSL, the resulting set of models may be transformed into either intermediate models or directly generated into source code. Similar to component based software development, MDD requires a clearly defined context in which it occurs.

### III. SYSTEMS INTEGRATION

Software systems are being developed and used for more than 40 years now and become more and more important in day to day business. At the same time, IT faces high demands in quickly adapting to new business requirements. As legacy systems often do not offer the flexibility to do so, new systems are implemented which need to interact with the existing IT landscape. It is often not possible to simply replace legacy applications due to the extreme cost involved. This situation inevitably leads to systems integration efforts, joining the different subsystems into a cohesive whole, in order to alleviate functionality or data access via a common interface [15, 16].

The term integration can either be defined as a *state* in which entities continue to exist after being integrated, or as the *process* of integrating them into a larger entity. *Integration as a state* defines classes by which the degree of integration of IT systems can be differentiated and evaluated. *Integration as a process* deals with the steps required to move an IT system from a given degree of integration to a higher one, which is done by merging distinct entities into a cohesive whole or integrating them into already existing systems [15, 17]. The present work follows the latter definition of the term

integration, i.e. the process of integrating distinct entities into a cohesive whole.

This process of integration can be further divided into data integration and application integration [16, 18]. Data integration concentrates on the integration of different data sources, for instance, by data consolidation or data warehousing. Application integration in turn covers the combination of different software systems that support business processes. The integration of such systems is also referred to as Enterprise Application Integration (EAI) and depicts a core area in today's business engineering. However, several interpretations exist for the term [19]:

- EAI as an integration middleware solution
- EAI as a high level integration on a semantic or process level
- EAI as an integration framework architecture
- EAI as an approach to implement business requirements, including strategic and process related considerations, utilizing different integration techniques

The present work focuses on Enterprise Application Integration and adheres to the fourth definition of the term, i.e. EAI as an integration approach from a strategic, process and technology related perspective. It does so because each layer may have severe influence on its neighboring ones and thus may not be considered in isolation, as suggested by the first three interpretations.

#### A. Dimensions of integration

Within the previously adopted definition of the term EAI, literature usually defines several layers or dimensions of integration. They start from a strategic and business process point of view, through process partitioning for different systems, to the actual data and functionality management on an implementation level.

In her book "Prozess- und Systemintegration", Vogler for example defines process, desktop and systems as the three sub domains of integration [19]. The process domain defines how business processes are depicted onto the IT landscape and how they support the overall workflow from a more strategic point of view. The second domain (desktop) defines when and how different (heterogeneous) applications are involved, and how they exchange information with the user (e.g., via a common user interface) or with each other. The underlying systems domain then defines which application accesses which data, how data exchange takes place, and how data redundancy is managed.

Hasselbring offers a similar classification in [20] by defining a business, application and technology architecture. He limits the term EAI to the second layer only, while applying interorganizational process engineering and middleware integration to the first and last layer, respectively. Despite the different interpretation of the EAI term, Hasselbring indeed considers the remaining aspects in his work.

A comparable classification can be found with Fischer in

[15], who identifies a business, organizational, functional and technical dimension. The business dimension defines which IT systems are required based on the strategic business needs. The organizational dimension aligns IT systems and workflows, and optionally adapts either. Data collection and storage of information (data integration), as well as controlling intermeshing activities (process integration) is done within the functional dimension. The fourth dimension (technology) aims at proper coupling of the different IT systems, independent of their location or underlying technology (systems interconnection).

Taking the previous definitions and explanations into consideration, the present paper defines the following three dimensions of integration:

#### 1) *Business Process*

On the business process dimension the organizational objectives, structure and core business processes of an enterprise are characterized. They define which business functionality and information is required and how the involved IT systems must interact from a semantic point of view. Integration decisions on this dimension are usually driven by mergers & acquisitions, collaboration agreements, or realignment of company objectives.

#### 2) *Workflow*

The workflow dimension subdivides a business process into distinct activities and maps these to the different IT systems. It defines the data sources and functionality required from the available IT systems from a technical point of view, as well as the interaction among each other and with the end users. On the business process dimension these data sources and functionalities map to the semantic steps of the business process. Integration decisions on this dimension may inherit from higher or lower dimensions, or are driven by process adaptations due to regulatory influences or improvement activities, for instance.

#### 3) *Technology*

Information and communication infrastructure of an integrated systems landscape is implemented at the technology domain. It defines which applications may access which data or functionality, how this is done, and how data management (e.g., redundancy) takes place. Integration decisions on this dimension inherit from higher dimensions, or are driven by technological changes, such as introducing or replacing applications.

### *B. Common problems in systems integration*

Despite the fact that systems integration supports and simplifies the execution of business processes, it involves several particularities and challenges during implementation. Based on Vogler in [19], the following sections briefly describe the potential problems per integration domain:

#### 1) *Business Process domain*

New business processes are often defined from a semantic point of view or are based on preconceptions from earlier projects regarding their representation in IT systems. As the IT landscape has direct implications on the business processes, it

is important to understand the integration relationships in order to *identify and choose an optimal solution*. In many companies, however, the particular situation is hardly known. Furthermore, business processes designers may not know about the solutions available on the market and often do not have the knowledge to design an overall concept.

Similarly, the unawareness of the particular IT landscape may lead to *unforeseen consequences*. Only minor changes in a process may lead to adaptations of the underlying systems, which in turn may require the adaptation of other processes due to changed interfaces or data structures. If these consequences were known early enough, business processes could be designed around them.

#### 2) *Workflow domain*

Subdividing a business process into workflows and depicting them ad hoc on different IT systems may lead to a *suboptimal degree of integration*. Due to time or cost constraints, these systems are often interconnected on a point to point basis instead of using shared integration architecture. In extreme circumstances this leads to  $n*(n-1)$  relationships, making later changes more and more complex.

In such an environment the *integration relationships may be unknown* due to insufficient documentation and the lack of a big picture. New implementations may be redundant and the consistency and integrity of interfaces cannot be ensured, which leads to unforeseen consequences for other systems. A prominent example was the Y2K problem where it was hardly known which systems rely on the data to be changed.

Enterprises still do not use a *methodological approach* with best practices or standardized processes for their systems integration projects. However, suitable methodology has been defined in literature during the last years but is not yet known or adopted in the industry. This also becomes evident as SI is not sufficiently considered in current software development models [21]. Integration projects are often done ad hoc and for a single purpose only that leads to the initially mentioned suboptimal degree of integration.

Due to uncoordinated efforts and the lack of methodologies, integrated systems show a *high complexity*, leading to increased time and cost for future adaptations.

Heterogeneity caused by the previous problems prevents the implementation of holistic *integration platforms or architectures* within enterprises. Although there are certain middleware systems or transaction monitors in place, these are usually not part of a bigger picture.

#### 3) *Technology domain*

From a technical point of view, *heterogeneity* is the major issue in systems integration. Depending on the differences, data representation and functionality, as well as underlying technologies must be aligned. The required effort thus disproportionately rises with the number of systems to be integrated, unless a common architecture or platform is used.

Another big problem is the integration with *legacy applications*. These were often designed as stand alone solutions with no integration in mind. Obsolete data management, interfaces, or a lack of documentation or

maintenance make integration extremely difficult. Furthermore these systems often cannot be altered or replaced and therefore impose restrictions on the overall integration concept.

The final issue lies in the *redundancy of data*. In integrated environments it becomes difficult to define which data resides where, how it is accessed and how redundancy is managed. Without such management, information may easily become outdated and inconsistent, leading to serious issues in business process execution.

#### IV. THE INDUSTRIALIZATION OF SYSTEMS INTEGRATION

Today systems integration solutions are still implemented from scratch by utilizing traditional software development methods, such as the Waterfall Model or the V-Model. These however were designed with regard to monolithic systems, as integration was not of interest at the time of their development. Recent works such as the V-Model XT briefly reference integration with external environments [5] but still do not pursue a standardized and methodological approach. The result may be an “integrated monolithic system” with highly complex dependencies as described in section B above. Moreover, these development models do not incorporate the basic principles of industrialization and thus may not leverage potential improvements in cost, efficiency and quality as initially stated.

As discussed in chapter II, Software Product Lines, Component Based Development and Model Driven Engineering represent specialization, standardization & systematic reuse, and automation for software development. The respective concepts are well understood and first literature is available on combining them in factory like development environments, as for example in Greenfield and Short’s book on Software Factories [3].

As shown in chapter III, SI comes with several particularities, distinguishing it from the domain of conventional software development. It has to challenge a multiplicity of technologies, inflexible legacy systems, once only technology combinations and a very high complexity. It seems disputable whether the concepts for industrialized software development in their original form can be applied to the field of systems integration, as depicted in the following:

##### *A. Software industrialization concepts with regard to SI particularities*

###### *1) Software Product Lines*

In Software Product Lines, design and development occur in a particular context, sharing common features and solving common problems. Product families may either be tailored around complete business solutions or a series of related products. They concentrate on reusable implementation artifacts, as well as frameworks, processes and tools.

With reference to systems integration, the multiplicity of different technologies, caused by high heterogeneity, inflexible legacy systems and different data sources, seems to be a major drawback to the definition of distinguished product lines. In a product line covering Customer Relationship Management

(CRM) systems for example, products may be highly integrated with third party logistics and finance systems. Including support for any potentially attached systems undermines the advantages of a delimited context, while excluding them will force development to occur outside the industrialized concepts. An additional drawback is the de-facto development of one-off solutions per customer. Barely any solution operates in the same environment or is interconnected with the same type of systems. The initial set-up cost for software product lines may therefore be contraindicative as the return of investment cannot be ensured.

###### *2) Component Based Development*

According to Greenfield & Short [3], development by assembly with software components has certain requirements that must be met: Platform independent protocols (e.g., XML), self-description of components (formalized and enhanced meta-data within components), deferred encapsulation (allowing to interweave additional functionality), assembly by orchestration (machine controlled interaction and management of components), and architecture driven development (to promote the availability of well-matched components).

With regard to systems integration, the author does not see any major difficulties to technically apply development by assembly. However, the assembly approach relies on systematic reuse and thus on a methodical approach in a clearly delimited context that may not be easy to define as shown in 1). This context also has an influence on the availability of predefined software architectures, as well as the number of reusable components. Furthermore systems integration standards are not common as of today [19]. The most important challenge to be met is the definition of a component based systems integration architecture in which development by assembly may occur.

###### *3) Model Driven Engineering*

Model Driven Development, and in a greater sense Model Driven Engineering, raises the level of abstraction to reduce complexity and express business concepts more efficiently. It consists of domain specific modeling languages and model transformation engines & code generators. The former allow a context free description of the intended products of a product line, whereas the latter provide model transformation to a lower, more specific model or eventually the generation of source code.

For systems integration, the efforts required to define a domain specific language (DSL) could become an obstacle, especially if applied to product lines with a limited number of expected products. With reference to Software Product Lines, the scope of a DSL cannot be clearly delimited as each product may need to be integrated with other external systems. Furthermore, to automate the development process by transforming models to a lower level or generating source code, transformation engines and code generators have to be implemented which also impose high set up cost.

##### *B. Areas requiring further research*

As can be seen in section A, existing concepts of software

industrialization may not necessarily suit the particularities found in the field of systems integration. Thus further research is required to either adapt or enhance existing concepts, while considering how to align organizational structures to support the application of industrial production paradigms.

The focus of the present research is therefore aimed at the application of industrial production principles in the specific domain of systems integration from a solution provider's point of view. The research deals with the following areas.

#### 1) *Organizational aspects*

Organizational aspects focus on the surrounding conditions of industrialization in SI. They are reflected in roles, responsibilities, and corporate structures, and should be carefully considered before performing a paradigm shift throughout the organization. With specialization and SI particularities in mind, an organizational structure needs to be developed which enables systems integration providers to implement industrial concepts. This subsequently imposes the question whether enterprises can afford to organize themselves in fully featured Software Product Lines or if other forms of organization, for instance, shared service centers for product line definition and management, or a combination of both, are more feasible.

Therefore an organizational concept, describing the definition of divisions and departments of a systems integration provider, may shape up to be useful as foundation for industrialization.

#### 2) *Software Product Lines*

Given a typical systems integration provider, an approach to implement software product lines in a way that they are neither too small nor too large, has to be developed. How can the wide variety of customer requirements, heterogeneity of integrated systems, and one-off developments be covered, without endangering the return on investment? As it delineates their scope, product line design for systems integration also has to bear the concepts of systematic reuse and automation in mind.

Further research must discuss the detailed requirements of Software Product Line implementation and identify ways of applying or adapting them in a systems integration context.

#### 3) *Component Based Development*

Given that an expedient classification of software products into product lines or families has taken place, is it possible to define software components to be reused in different integration solutions for different customers? As shown before, component based development requires an adequate architecture in which it takes place. With reference to the common problems of systems integration, a combination of component based architectures and systems integration frameworks seems necessary.

In a joint analysis of existing component architectures and SI frameworks, it should be figure out if a combination of both is feasible and may be used as the technical foundation for software product lines.

#### 4) *Model Driven Engineering*

The probably most ambitious objective of an industrialized software development is the automated creation of artifacts

such as model transformations or code generation. For SI it offers interesting possibilities to resolve problems related to the business process and workflow domain of SI, such as integration consequences and depicting intersystem relationships. However, it is unclear to which degree an SI service provider can economically implement such a concept and if it can be used for different customers. The role and potential advantages of domain specific languages in the given context is also unknown. Are separate tools such as model transformers or code generators required for each product line or can their foundations be reused?

Based on the previous three aspects, the feasibility of Model Driven Engineering in systems integration should be analyzed and suggestions for the degree of its implementation derived. In this context MDE may shape up to be useful to solve SI related problems such as unknown integration consequences or intersystem relationships.

## V. CONCLUSION & RESEARCH APPROACH

Systematic reuse of existing software artifacts hardly takes place and the majority of goods is still produced from scratch. With increasing complexity and size of today's IT systems, a generally accepted and industrialized production principle becomes necessary. Promising approaches, notably Software Product Lines, Component Based Development, and Model Driven Engineering, are currently being developed and implemented in practice, as described in chapter II.

However, as software engineering takes place in a wide variety of application domains, it cannot be assured whether the available industrialization models can be applied to every one of them. One of these domains is systems integration in which IT systems are adapted and interconnected to support new or changing business processes or requirements. To better understand the particularities of this field, chapter III depicts its substantial differences that are primarily the lack of knowledge about the integrated IT landscape of an enterprise, the lack of a methodological approach and integration framework, and a high heterogeneity of systems.

Chapter IV picks up these particularities and maps them to the introduced concepts of industrialized software engineering. The first section shows why these concepts cannot be applied to systems integration in their initial occurrence, while the second suggests further research to advance the field of software engineering in systems integration towards an industrialized production process:

- Organizational aspects: Which changes are required to the organizational structure of a systems integration provider in order to implement industrial production methods in an economically feasible way?
- Software Product Lines: Is the concept of SPL in its original form viable for systems integration providers? How can the gap between a standardized product family and customer specific requirements in a highly heterogeneous environment be bridged at feasible cost?
- Component Based Development: CBD and SI require a

specific architecture or framework. Can both be combined to form a basis on which Software Product Line development and systematic reuse can be built on? How can the high heterogeneity of systems to be integrated taken into account?

- Model Driven Engineering: To what extent does it economically make sense to implement MDE in systems integration? Does MDE offer additional benefits to SI as, for instance, an integration management approach?

The present work can be classified into the scientific area of business informatics as it covers matters from business (organizational forms of enterprises and product family management) and computer sciences (implementation of CBD and MDE). To meet concerns about the fact that very little of software engineering research finds its way into practice [22], research in the described area could be conducted in close collaboration with the industry. Thereby the approach of action research, aiming at the retrieval of scientifically proven procedures and guidelines and applying them in practice, seems to be suitable. The approach consists of three major phases: During the first phase, scientists and practitioners outline the problem definition and a first concept is developed, based on domain analysis and theoretical research. In the second phase the derived concepts are discussed with subject matter expert and subsequently implemented in practice. The third phase then reflects the results of the implemented solution and derives suggestions for improvement and further research, out of which a new cycle of action research can be initiated. For each of the above aspects at least one action research cycle will be accomplished, further ones may be added as needed.

The overall objective of the depicted areas of research may be a guideline which will draw a holistic picture of industrialized systems integration and provide a software development approach that addresses the application of industrial concepts in systems integration.

#### REFERENCES

- [1] Butschek, F., *Industrialisierung*. 2007, Ulm: Ebner & Spiegel.
- [2] Brockhaus-Enzyklopädie, in *Brockhaus-Enzyklopädie*. 2005, Brockhaus: Mannheim.
- [3] Greenfield, J. and K. Short, *Software Factories - Assembling Applications with Patterns, Models, Frameworks, and Tools*. 1 ed. 2004, Indianapolis: John Wiley & Sons.
- [4] Brockhaus-Enzyklopädie, in *Brockhaus-Enzyklopädie*. 2005, F.A. Brockhaus: Mannheim.
- [5] Balzert, H., *Lehrbuch der Software-Technik: Software Management*. 2 ed. 2008, Heidelberg: Spektrum Verlag.
- [6] Sneed, H.M., *Software-Qualitätssicherung für kommerzielle Anwendungssysteme*. 1 ed. 1983, Bergisch Gladbach: Verlagsgesellschaft Rudolf Müller.
- [7] Sneed, H.M., *Software Management*. 1987, Cologne: Müller GmbH.
- [8] Clements, P. and L. Northrop, *Software Product Lines*. 2007, Boston: Addison-Wesley.
- [9] Linden, F.v.d., K. Schmid, and E. Rommes, *Software Product Lines in Action*. 2007, Berlin, Heidelberg, New York: Springer.
- [10] Pohl, K., G. Böckle, and F. van der Linden, *Software Product Line Engineering*. 1 ed. 2005, Berlin, Heidelberg, New York: Springer.
- [11] Software Engineering. in *NATO Software Engineering Conference*. 1968, Garmisch Partenkirchen: NATO Science Committee.
- [12] Szyperski, C., *Component Software: Beyond Object-Oriented Programming*. 1998, Massachusetts: Addison-Wesley.
- [13] Andresen, A., *Komponentenbasierte Softwareentwicklung mit MDA, UML 2 und XML*. 2 ed. 2004, Munich, Vienna: Carl Hanser Verlag.
- [14] Schmidt, D.C., *Model Driven Engineering*. *IEEE Computer*, 2006. 39(2): p. 7.
- [15] Fischer, J., *Informationswirtschaft: Anwendungsmanagement. Lehr- und Handbücher zu Controlling, Informationsmanagement und Wirtschaftsinformatik 1999*, Munich, Vienna: Oldenbourg.
- [16] Leser, U. and F. Naumann, *Informationsintegration*. 2007, dpunkt: Heidelberg.
- [17] Riehm, R., *Integration von heterogenen Applikationen*. 1997, Universität St. Gallen: St. Gallen.
- [18] Conrad, S., et al., *Enterprise Application Integration - Grundlagen, Konzepte, Entwurfsmuster, Praxisbeispiele*. 2006, Elsevier: Munich.
- [19] Vogler, P., *Prozess- und Systemintegration - Evolutionäre Weiterentwicklung bestehender Informationssysteme mit Hilfe von Enterprise Application Integration*. 2004, Wiesbaden: Deutscher Universitäts-Verlag.
- [20] Hasselbring, W., *Information System Integration*. *Communications of the ACM*, 2000. 43(6): p. 7.
- [21] Gassner, C., *Konzeptionelle Integration heterogener Transaktionssysteme*. 1996, Universität St. Gallen: St. Gallen.
- [22] Potts, C., *Software-Engineering Research Revisited*. *IEEE Software*, 1993. 10(5): p. 9.

# Industrielle Softwareentwicklung

*Leitfaden und Orientierungshilfe*

For copyright reasons please refer to the following address or contact the author.

[http://www.bitkom.org/de/publikationen/38337\\_65192.aspx](http://www.bitkom.org/de/publikationen/38337_65192.aspx)



Industrielle Softwareentwicklung  
*Leitfaden und Orientierungshilfe*

# An Organizational Approach for Industrialized Systems Integration

Matthias Minich, Bettina Harriehausen-Mühlbauer, and Christoph Wentzel

Fachbereich Informatik  
University of Applied Sciences Darmstadt  
Haardtring 100  
64295 Darmstadt, Germany  
matthias.minich@web.de  
b.harriehausen@fbi.h-da.de  
c.wentzel@fbi.h-da.de

**Abstract:** Software development in systems integration projects is still reliant on craftsmanship of highly skilled workers. To make such projects more profitable, an industrialized production, characterized by high efficiency and quality, seems inevitable. While first milestones of software industrialization have recently been achieved, it is questionable if these can be applied to the field of systems integration as well. One of the most important concepts herein is specialization, represented by Software Product Lines. The present work analyses this concept against the particularities of systems integration and subsequently develops an alternative approach suitable for its implementation. The outcome is a three-layered organizational model that adapts and distributes the processes of Software Product Line Engineering and Product Development in accordance with the requirements of systems integration.

## 1 Introduction

Software Engineering offers several methodologies for industrialized software development, representing specialization, standardization, systematic reuse, and automation. The latter three are based on standardization to be most effective, which is represented by Software Product Lines (SPL). It seems to be very difficult or even impossible to create reusable artefacts or automate development in an arbitrary context. A Software Product Line therefore spans a clearly delimited frame around a family of software products, sharing a common set of features and artefacts within a particular segment [CN07]. By concentrating on a limited scope, reusable production assets can be much more powerful than in a generic one. Unfortunately this does not apply to all fields of software engineering.

With reference to systems integration, the multiplicity of different technologies, caused by high heterogeneity, inflexible legacy systems and different data sources, seems to be a major drawback to the definition of distinguished product lines. In a product line

covering Supply Chain Management systems, products may be highly integrated with third party shopfloor or finance systems, for instance. Including support for any potentially attached system undermines the advantages of a delimited context, while excluding them will force development to occur outside the industrialized concepts. Another major drawback is the de-facto development of one-off products per customer. Barely any solution operates in the same environment or is interconnected with the same type of systems. The initial setup cost for software product lines may therefore be contraindicative as the return of investment cannot be ensured.

## **2 Organization of Product Lines**

Software Product Lines separate between product and product line development, represented in the following two scenarios:

A) Quality oriented development of standard software in a complex environment with a high degree of novelty. Considering the impact of a defect introduced into the underlying product line, a high level of quality outweighs the objective of lower cost. It is assumed that developing a new software product line is highly complex [Lin07] and comes with a high degree of novelty. As of a product line's fundamental and long lasting nature, a low dynamic is assumed.

B) Cost oriented development of customized software in an established environment with low dynamics and low complexity, which applies to product development. As most motives for software product lines are based on economic considerations [Lin07, CN07, PBL05], cost outweighs quality in this scenario. Furthermore, a certain level of quality is automatically ensured by utilizing artefacts of the respective product line. As products are developed within the boundaries of a well-known environment, novelty and dynamics can be considered as low. Likewise, complexity is reduced due to the reuse of common parts in a predefined platform and architecture.

Out of these scenarios, the present work developed a generic organizational structure for software product lines. It is thereby considering several works on organizational structures from an economic and software development point of view [Gro95, WD08, Töp85, Fre98, Lan04].

### **2.1 Software Product Line Engineering**

Software product line engineering consists of several processes to define its scope and develop the infrastructure and core assets to be utilized in future products. Considering literature on software product line engineering [PBL05, CN07, Lin07, Bal08], the primary processes can be subsumed as follows:

- *Business Domain Analysis* identifies typical business processes, associated problems and solutions, and evaluates and prepares this knowledge for further processing. It also specifies a product's features within a software product line.

- *Domain Requirements Engineering* defines a product line's scope by identifying products and documenting their commonalities and variabilities. This scope may evolve over time [CN07].
- *Architecture Design & Development* transforms the scope into a technical architecture for the product line and its products. The architecture decomposes a software system into common and variable functional parts, defines relationships and interfaces, and establishes rules for their implementation.
- *Core Asset Development* provides detailed design and implementation of reusable components based on the reference architecture [PBL05]. It includes executable code, variability mechanisms, common processes, development tools, and any other reusable assets.
- *Domain Testing* develops test cases and inspects all core assets and their interactions against the requirements and contexts defined by the product line architecture. This also includes validation of non-software core assets.
- *Software Integration* occurs during preintegration of several software components. They form blocks of functionality common to all products and contexts of a product line. It also ensures the interoperability of all reusable assets and provides the required integration mechanisms.

Implementing a delimited software product line is a singular undertaking for an enterprise. Although the primary processes remain the same throughout the development of other domains, work objects (architecture or core assets for instance) are unique and require different production steps for their completion. A decomposition based on work objects thus seems more appropriate than an activity based breakdown. It results in divisional units, responsible for all tasks required to create or modify a heterogeneous and potentially dynamic work object [Gro95]. This decomposition however, is limited by the interdependencies of the underlying work objects. It must be decided if interacting objects should be merged and represented in a single organizational unit, or if they can remain separate in favour of smaller and more efficient units. For software product line engineering, the present work suggests a structure as depicted in Figure 1. It assumes a low interdependency of business domain analysis due to its observing and strategic focus. It is therefore represented in its own organizational unit. Requirements Engineering and Architecture Design & Development are joined as of their intense interaction during problem definition and solution development within a new problem domain [Lan04]. Once requirements and architecture are defined, core asset development may occur. It is expected to have only little interdependencies as their overall structure and requirements have been defined in the previous two processes. They are implemented in their own unit and may be parallelized. Integration testing requires a certain degree of integration to be performed. It is therefore joined with the software integration process.

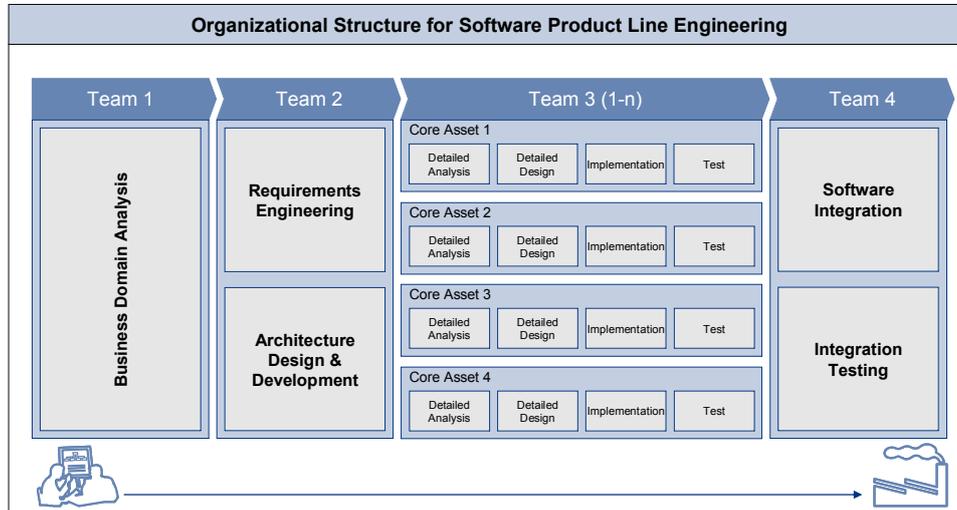


Figure 1: Organizational Structure for SPL Engineering

## 2.2 Product Development

In product development, the applications of the product line are built by reusing the core assets developed in software product line engineering. Considering literature [PBL05, CN07, Lin07, Bal08], the primary processes of product development can be subsumed as follows:

- *Product Requirements Engineering* analyses the variance of product requirements from the product line's core assets and decides whether to implement application specific assets or accept a functional trade-off.
- *Product Design* derives a particular product architecture from the overall product line architecture. Abstract variation points are instantiated and product specific requirements added. It defines how the product will be realised and may be compared to a detailed technical concept in single system development.
- *Product Realisation* assembles the application from core assets within the product architecture, binds their variability points according to requirements and design, and implements product specific assets. Compared to single system development, integration efforts are decreased due to predefined architectures and integration mechanisms [CN07].
- *Product Testing* ensures sufficient quality of the end product. Although the components have been tested during product line development, instantiated variability points and interaction with other components must also be covered. Furthermore, during domain testing it is impossible to cover all potential combinations of core assets.

Product development within an SPL is a recurring activity. The objects of work (i.e. products) are homogeneous and stable. It can be assumed that similar products have been developed before and that product architecture and technology are well understood. Interdependencies between the primary processes are therefore expected to be low. Compared to scenario A, product development is characterised by low dynamics and low complexity. Consequently, an activity-based decomposition of work seems most appropriate, which will result in divisional units [Gro95]. The degree of decomposition is determined by a preferably small team size [Bal08], and the required interaction between the resulting functional units. For product development, the present work suggests a structure as depicted in the following.

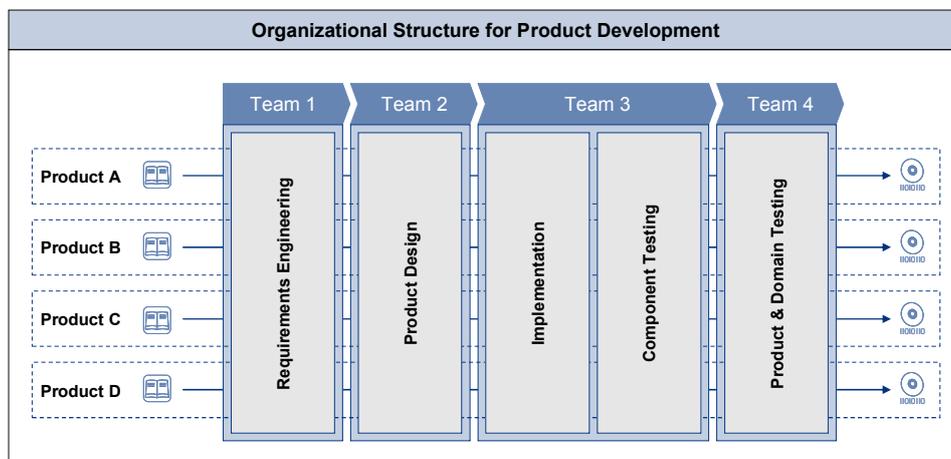


Figure 2: Organizational Structure for Product Development

The above structure assumes a low interdependency between the five primary processes of product development, except for implementation and component testing. This is due to the fact that the product line requires development to occur within clear boundaries with precisely defined interfaces between core assets and architectures. It however does not predefine requirements for component testing, which is therefore joined with the implementation process due to presumably interactive interactions between both.

### 3 A Three Layered Structure for SPL in Systems Integration

As described before, it seems disputable whether the concept of software product lines in its original form can be applied to the field of systems integration (SI). Revisiting chapter 1, the major issues can be subsumed as follows:

- *Integration across software product line boundaries:* SPLs have a preferably narrow scope to be most powerful. SI however requires considering a variety of different products with a very different scope. If the product lines of a system integrator are not compatible with each other, integration will inevitably occur outside their boundaries.

- *Multiplicity of technologies*: The multiplicity of different technologies in SI, caused by high heterogeneity, inflexible legacy systems, and different data sources, seems to be a major drawback to distinct product lines. Too much scope, which will most likely be used only once, would have to be added.
- *Uncertain return of investment*: As systems integration produces very customer specific solutions, the minimum number of products to break even cannot be ensured. The cost for a substantial software product line may outweigh its savings.

It is assumed that an integration of different IT systems mostly occurs within the boundaries of a particular industry. An automotive supplier for instance will hardly need to integrate any of his systems with an e-government solution from the public sector. Yet he may require integrating his SAP accounting system with a logistics application of one of his suppliers. This assumption is backed by current organizational structures of major systems integration companies, which are organized in a vertical structure based on certain industries [Pie09]. Any integration architecture should therefore at least support the typical systems of the respective industry. Implementing such an architecture within an software product line however, would broaden its scope far beyond being efficient and thus feasible for industrialization. This especially applies to reusable core assets, which would be too generic to provide any benefit.

To overcome these problems, a three layered approach for software product lines in systems integration has been developed. It essentially adds a layer of abstraction on top of the software product lines. The contents of each are suggested as follows:

- The *Business Domain Layer* is a new super ordinate layer that spans over a complete division or business segment within a system integrator's organizational structure. It identifies the major requirements of the business domain in scope and conceptually defines fundamental core assets, technologies, and systems typically used therein. The development of an abstract system landscape and integration architecture ensures the interoperability of different systems and product lines within the business domain. It consists of the four core processes Business Domain Analysis, Portfolio Definition, Architecture & Roadmap Definition, and Core Asset Development.
- The *Product Line Layer* consists of several software product lines identified in the Portfolio Definition process of the Business Domain Layer. The Engineering processes of these software product lines differ only marginally. The most obvious variance to conventional software product line engineering is the lack of the Business Domain Analysis process, and a reduced Domain Requirements Engineering process. These functions are now incorporated in the Business Domain Layer and provide their findings to the subsequent product lines. All other processes remain the same but must adhere to the specifications and utilize the provided core assets from the business domain layer.
- The *Production Layer* contains the actual development of a product within a software product line. It does not differ from the conventional concept of software product lines depicted in section section 2.2. This of course only applies to the software development process. Any systems integration specific architecture or solution design was already taken care of in the previous two layers.

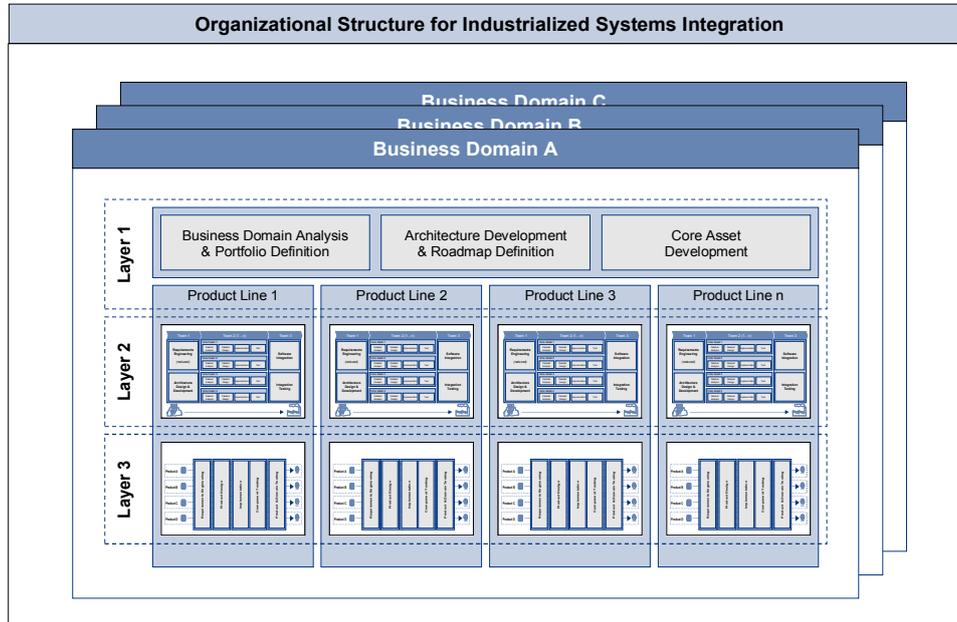


Figure 3: Three Layered Approach for Industrialized SI

The work objects of the above processes can be combined into a product line skeleton, which will be instantiated by a particular software product line. This rather abstract layer for industrialized software development is expected to have a positive effect on the previously mentioned major issues of industrialized systems integration. The first concern, *integrating products from different product lines*, may be solved by the Portfolio Definition and Architecture & Roadmap Definition processes. The abstract architecture applicable to all software product lines ensures the compatibility of products within a given business domain. Integrating the products from a supply chain management product line with those from a shopfloor systems product line, for instance, will be much easier due to compatible architectures and technologies. The second one, *multiplicity of technologies*, can be alleviated by a joint technology roadmap. It will limit the number of utilized technologies within the software product lines and thus reduce their heterogeneity. While differences due to legacy systems or third party applications will prevail, these may be alleviated by joint interface components across multiple product lines. Within the product line boundaries, heterogeneity is thereby reduced to standardized interfaces, while externally a wide variety of products may be supported. Furthermore, if products from such product lines are integrated with each other, these issues will resolve over time. The third concern, *ensuring the return of investment*, can also be attenuated. Software product line engineering may instantiate the predefined skeleton and has a greatly reduced effort in the processes Business Domain Analysis, Business Domain Architecture, Architecture Design & Development, and Core Asset Development. Due to reduced efforts and thus cost, the break even point of a SPL may be reached earlier. Although this approach may not be as efficient as traditional software product lines, the author assumes that it still helps to advance their break even point in SI

and that it will have a positive effect on the overall product quality. Case studies and real world implementations in traditional software development have shown a break even after 2–3 products of a product line [CN07]. This value of course depends on the efficiency and reusability of the underlying core assets. However, if these characteristics can also be achieved for the core assets of a systems integration product line still has to be proven in practice.

As implementing the business domain layer is a singular and novel undertaking, work is decomposed based on work objects. Thereby the processes Business Domain Analysis and Portfolio Definition are combined due to a presumably close interaction. Architecture Development and Roadmap Definition, as well as Core Asset Development remain separate as they only rely on their predecessor's outcomes but do not significantly influence them. Based on the nature of the core assets to be developed, it is also conceivable to break it down into different teams. These teams may then be responsible for particular assets throughout their lifetime and also take over their maintenance.

The resulting structure of the three layered approach is depicted in Figure 3. It should be noted that the Product Line layer does no longer contain the Business Domain Analysis Process from Software Product Line Engineering and also reduces the responsibilities of the Domain Requirements Engineering Process. These functions are now incorporated in the Business Domain Layer and provide their findings to the subsequent product lines. All other processes remain the same but must adhere to the specifications and utilize the provided core assets from the business domain layer. The internalization and thus organizational structure of the remaining product line engineering processes remains the same.

#### **4 Further Research**

The present work introduced into the fundamentals of software industrialization and systems integration, and showed that both cannot be combined easily. The reasons therefore are integration issues across product line boundaries, a high heterogeneity, and an unsure return on invest. In spite of these issues, the paper developed an alternative approach to implement software product lines as one of the industrial key concepts in the field of systems integration. As the present work has a rather conceptual character, further research is required to move it closer to industrial practice. This especially includes developing a more detailed process and role model to give practitioners a starting point for realisation. Furthermore, the exemplary implementation of a Business Domain Layer as a proof of concept, especially with regard to abstract, reusable assets, would be helpful to promote industrialization in systems integration.

Besides implementing software product lines as the industrial key principle of specialization, standardization, systematic reuse, and automation represent the next key milestones on the way towards fully industrialized systems integration. Further research is required to identify their current representation in software engineering and eventually apply them to the field of systems integration.

## Bibliography

- [Bal08] Balzert, H.: Lehrbuch der Softwaretechnik. Softwaremanagement. Spektrum Akad. Verl., Heidelberg, 2008.
- [CN07] Clements, P.; Northrop, L.: Software product lines. Practices and patterns. Addison-Wesley, Boston, 2007.
- [Fre98] Frese, E.: Grundlagen der Organisation. Konzept, Prinzipien, Strukturen. Gabler, Wiesbaden, 1998.
- [Gro95] Grochla, E.: Grundlagen der organisatorischen Gestaltung. Schäffer-Poeschel, Stuttgart, 1995.
- [Lan04] Lang, C.: Organisation der Software-Entwicklung. Probleme Konzepte Lösungen. Dt. Univ.-Verl., Wiesbaden, 2004.
- [Lin07] Linden, F.: Software product lines in action. The best industrial practice in product line engineering. Springer, Berlin, Heidelberg, New York, 2007.
- [PBL05] Pohl, K.; Böckle, G.; Linden, F.: Software product line engineering. Foundations, principles, and techniques ; with 10 tables. Springer, Berlin, 2005.
- [Pie09] Pierre Audoin Consultants: Software and IT Services Industry (SITSI) Report, Paris, 2009.
- [Töp85] Töpfer, A.: Umwelt- und Benutzerfreundlichkeit von Produkten als strategische Unternehmensziele. In Marketing ZFP, 1985, 7; pp. 241–251.
- [WD08] Wöhe, G.; Döring, U.: Einführung in die allgemeine Betriebswirtschaftslehre. Vahlen, München, 2008.

# Component Based Development in Systems Integration

Matthias Minich, Bettina Harriehausen-Mühlbauer, Christoph Wentzel

School of Computing and Mathematics  
University of Plymouth  
Drake Circus  
PL4 8AA Plymouth  
matthias.minich@plymouth.ac.uk

Fachbereich Informatik  
Hochschule Darmstadt  
Haardtring 100  
64293 Darmstadt  
christoph.wentzel@h-da.de  
b.harriehausen@h-da.de

**Abstract:** Software development in systems integration projects is still reliant on craftsmanship of highly skilled workers. To make such projects more profitable, an industrialized production, characterized by high efficiency and quality, seems inevitable. While first milestones of software industrialization have recently been achieved, it is questionable if these can be applied to the field of systems integration as well. Besides specialization, standardization and systematic reuse is one of the key concepts of industrialization, represented by component-based development (CBD). The present work analyses a CBD approach suitable for large scale, enterprise wide systems, while considering the particularities found in the field of systems integration. The outcome is an alignment of this slightly adapted approach with organizational requirements found in the industrial principle of specialization, as well as entities found in a typical enterprise application integration project.

## 1 Introduction

The key concepts of industrialization can be defined as specialization, standardization, systematic reuse, and automation, and are found in most industries at different levels of penetration [Enc91]. They usually evolve around a market, which has enough demand to satisfy the efforts required to industrialize production. The field of software development however is still reliant on craftsmanship and highly skilled workers [GSC04]. By applying industrial methods and thus enhancing an organization's productivity, we possibly can increase quality and product complexity, and at the same time reduce cost and production time. For software engineering, Software Product Lines (SPL) represent the industrial principle of specialization. It seems to be very difficult or even impossible to determine how mechanisms for reuse or automation should be implemented in an arbitrary context. A Software Product Line (SPL) therefore spans a clearly delimited frame around a family of software products, sharing "[...] a common, managed set of features satisfying the specific needs of a particular segment or mission" [CN07]. Standardization and systematic reuse are available within Component Based Development (CBD), an approach to exchange and systematically reuse software artifacts. Components can be independently utilized and composed to applications. The final aspect of industrializa-

tion, automation, can be achieved with Model Driven Engineering (MDE). Using visual models as a description of software and utilizing domain specific languages, the degree of freedom and possible contexts is reduced. This allows using model transformation engines and code generators to automatically advance the development process.

In today's business world, IT faces high demands in quickly adopting to new requirements. As legacy systems often do not offer the flexibility to do so, new systems are implemented which need to interact with the existing IT landscape. This situation inevitably leads to systems integration efforts, joining the different subsystems into a cohesive whole, in order to alleviate functionality or data access [Fis99; LN07]. Systems integration deals with the steps required to move an IT system from a given degree of integration to a higher one by merging distinct entities into a cohesive whole, or integrating them into already existing systems [Rie97; Fis99]. This process of integration can be further divided into data integration and enterprise application integration [CHK06; LN07]. Data integration concentrates on the integration of different data sources, for instance, consolidation of previously separate databases or building a data warehouse from many different enterprise wide data sources. Application integration in turn covers the combination of different software systems supporting business processes. The present research focuses on enterprise application integration as an integration approach from a strategic, process and technology oriented perspective.

## **2 Challenges of CBD in Systems Integration**

Systems Integration comes with certain particularities, distinguishing it from conventional or single-system software development. It has to challenge a multiplicity of technologies, once only combinations thereof, and thus a very high complexity of to be integrated systems. The present chapter will describe these particularities in further detail and their implications on Component Based Development. The present work thereby takes the position of a large systems integrator, who provides enterprise application integration (EAI) services and solutions to his customer. A typical project can for instance be the implementation of a complex business process across several enterprise resource planning (ERP) systems. Taking into account that a large systems integrator is usually active in several industries and that for these industries a multiplicity of different solutions exist, a high heterogeneity must be assumed. Such heterogeneity may occur on a logical, physical, or technical layer. It anticipates the formation of standards as, for instance, companywide integration architectures. Furthermore, heterogeneity is reinforced by the fact that integrated systems are often connected on a peer-to-peer basis with each other, leading to  $n*(n-1)$  relationships. As of the high costs of these, they are also not replaced frequently: "[...] SI aims at building applications that are adaptable to business and technology changes while retaining legacy applications and legacy technology as long as possible" [Has00]. As can be seen from the explanations before, a great variety of combinations from different technologies, business processes, or regulatory implications are possible. Considering the fact that most system integrators are active in multiple industries with multiple customers, chances that one project is similar to another are extremely small. However, as introduced in section 1.1, standardization and systematic reuse are considered as one of the key principles for industrialization. Fur-

thermore, CBD suggests to encapsulate business and application logic into standardized and reusable units of composition. As current SI projects practically are one-off developments, CBD would only create additional overhead, as components could hardly be reused. Furthermore, implementation cost are associated with component-based development. First, one has to define a component architecture on which the different applications will be based on. Subsequently, a suitable component framework as the technical basis has to be selected and adapted to the product line architecture. Once the required infrastructure is in place, component development may begin. In the context of systems integration with its high heterogeneity and one-off development projects, implementation cost seem contradictory to component-based development. With the given situation and existing CBD concepts, it must be assumed that such intent will never break even, as no considerable economies of scale or scope exist to justify expenses for component framework creation or deriving components from existing code for later reuse. To overcome this challenge, either reusability or cost efficiency must significantly be increased.

Considering the above, an efficient approach for component based systems integration must be developed. Based on the underlying research, the present paper suggests combining already existing models for systems integration and component based development. It thereby considers the industrial key concept of specialization, as well as the SI particularities heterogeneity, one-off development, and return on invest.

### 3 The Organizational Model for Industrialized Systems Integration

In our previous work [MHW10] we developed an organizational model for software product lines in systems integration. This was done as a first step towards industrialized SI and represents the industrial key concept of standardization. It assumes that integration of different IT systems mostly occurs within the boundaries of a particular industry. An automotive supplier for instance will hardly need to integrate any of his systems with an e-government solution from the public sector. Yet he may require integrating his SAP accounting system with a logistics application from one of his suppliers. Any integration architecture should therefore at least support the typical systems of the respective industry. Implementing such architecture within a software product line however, would broaden its scope far beyond being efficient and thus feasible for industrialization. This especially applies to reusable core assets of a software product line, which would be too generic to provide any benefit. To overcome these shortcomings, a three-layered approach for software product lines in systems integration has been developed. The contents of each layer are defined as follows:

The **business domain layer** as a new super ordinate layer spans over a complete division or business segment within a system integrator's organizational structure. It identifies the major requirements of the business domain in scope and conceptually defines fundamental entities, core assets, technologies, and systems typically used therein. The development of an abstract system landscape and integration architecture ensures the interoperability of different systems and product lines within the business domain. It consists of the four core processes business domain analysis, portfolio definition, architecture & roadmap definition, and core asset development. The **SPL engineering layer** consists of

several software product lines identified in the portfolio definition process of the business domain layer. The engineering processes of these software product lines differ only marginally from conventional ones, but lack business domain analysis, and have a reduced domain requirements engineering process. These functions are now incorporated in the business domain layer and provide their deliverables to the subsequent product lines. All other processes remain the same but must adhere to the specifications from the business domain layer. The **product development layer** contains the actual development of a product within a software product line. It does not differ from the conventional concept of software product lines.

This organizational model is expected to have a positive effect on industrializing systems integration: Integrating products from different product lines is solved by an aligned product portfolio and common integration architectures. The multiplicity of technologies can be alleviated by a joint technology roadmap. Unfortunately, this does not reduce heterogeneity introduced by legacy systems. It may however be improved by joint interface components across multiple product lines. The return of investment can also be attenuated: SPL engineering benefits from predefined architectures and infrastructure and has a greatly reduced effort in the processes business domain analysis, business domain architecture, architecture design & development, and core asset development. Due to reduced efforts, the breakeven point of an SPL can be reached earlier. For additional information please refer to [MHW10].

## 4 The Integration Metamodel

The integration metamodel developed by Vogler [Vog06] shows the different objects of integration, their relationships to each other, and possible variants of integration. It currently is the only describing model available in literature, as most others focus on implementation concepts or technologies. As our approach aims at being independent of any implementation technology, a metamodel provides the most suitable foundation therefore. It is based on the following four viewpoints:

The fundamental one is the **information system**. It serves as the base for different applications, programs and data sources within a company. Applications provide interfaces for functionality and data, consisting of different programs, data structures, and data collections. Programs are responsible for data collection and provide specific business functionality via interfaces. Applications with their underlying programs and data collections are located on information systems. **Process integration** is responsible for the conceptual design and realization of a process. A managing entity implements a business process in one or more workflows. Workflows are based on logical states to control their behaviour, are implemented on an information system, and consist of one or more activities. Activities are based on states and control data. We omit additional organizational entities of the model, as they are specific to a particular instance of a business process and thus not relevant for our generic model. **Desktop integration** combines required applications and programs from different enterprise systems and presents them to the user as sequence of tasks to be completed. Presentation usually occurs through graphical user interfaces, data entry masks, lists, reports, or similar. In this context, desktop inte-

gration is also responsible for data flow between the underlying applications and programs, which is defined by its data structure and, in case of a non-desktop data exchange, realized as a data transfer between different systems. Tasks are embedded within executable activities, which may contain different tasks for different users. **Systems integration** represents the third viewpoint of the integration metamodel and represents technical aspects of a system, such as distinct applications or middlewares. This is important if a certain piece of software is to be removed, for instance. In this case, all depending business processes must be adapted. An integration relationship can therefore be seen as the sum of all data transfers between two different applications. Interfaces of an application can be differentiated into program and data interfaces. Depending on technical and logical requirements, different integration variants are applied and supported by a middleware (e.g. frontend or data integration, method invocation, or a dedicated integration application). Programs, data collections, and middleware can be located on one or more information systems.

Figure 3 shows the overall integration metamodel, representing the three integration viewpoints (due to limited space including later introduced distribution tiers). It allows us to describe any integration relationship independently from a specific architecture or technology. For further and more detailed information please refer to [Vog06].

## 5 The Business Component Model

The Business Component Model is a methodology to model, analyse, design, construct, validate, deploy, customize, and maintain large scale distributed systems, developed by Herzum and Sims [HS00]. One of its principles is “that any software artefact in a system should be defined in one and only one place, and it should be reused as many times as necessary” [HS00]. This principle perfectly fits the concept of Software Product Lines, in which a distinctive unit or team is responsible for a particular set of core assets, such as reusable business components. It has five dimensions:

The first dimension are **five levels of component granularity**, which are the language class (which is not considered a component itself as it is not independently deployable nor does it have run-time or network interfaces), the distributed component (a component in its common sense, e.g. an EJB or CORBA component), the business component (still independently deployable, consisting of distributed components and glue code), and the system level component (a set of business components providing business functionality). The highest level of granularity is the federation of system-level components (i.e. system level components federated to provide multiple complex business services).

The second dimension consists of **four architectural viewpoints**: The technical architecture is concerned with the component execution environment and fundamental services such as activation, persistence, transaction services, and also describes an application independent integrated development environment. The application architecture defines the specific characteristics of the application to be built, such as design principles or application structures. A less obvious viewpoint is the project management architecture, which addresses principles, guidelines, policies, and tools required to build a scalable

and high performance system with a large team. The functional architecture describes the actual application to be built and defines business modelling, component design and development, as well as persistence management of component and business data.

An **aligned development process** as the third dimension contains rapid component development for designing, building, and testing an individual business component; system architecture and assembly for architecting, assembling, and testing a complete system; as well as federation architecture and assembly for architecting, assembling, and testing a federation of systems using system level components. The process is based on the well-known V-Model, while all phases and activities are component centric, i.e. each functionality or other aspect of the system belongs to one business component only.

The anatomy of a component is separated into user, workspace, enterprise, and resource **distribution tier** as the fourth dimension. The user tier presents the component on the screen and communicates with the user. It may be stand-alone, plug in, or non-existent at all and has no business logic. The workspace tier implements local business logic and is responsible to interact with the enterprise tier, serving as a broker for the user tier. The enterprise tier implements enterprise-level business rules, validation, interaction between enterprise components, as well as data integrity. It typically forms the core functionality of business components. The resource tier manages access to shared resources, such as databases or files, and shields all higher layers from the technical implementation.

The fifth dimension defines four broad **functional categories**, which are utility business components, entity business components, process business components, and auxiliary business components. Utility components can most generally be reused and represent simple autonomous concepts, such as a number generator or currency converter. Entity business components represent the logical entities on which a business process operates and is rather specific to a particular business domain. Examples are item, invoice, address, or customer. The actual business process is implemented within a process business component and is usually unique and hardly reusable. Based on business process descriptions or use cases, they implement an actual process by utilizing utility, entity, and auxiliary business components. The last category, auxiliary business components, provides services usually not found within a business process description. Such services may be performance monitoring, messaging, or middleware services.

## 6 Component Based Systems Integration

While each of the previous concepts describe the process of systems integration and component based development individually, we currently do not know how they can be combined to achieve component based systems integration. The present work contributes to this question by organizationally and conceptually aligning their different viewpoints, resulting in a new approach to component based systems integration. We do so by taking our previously developed organizational model for industrialized systems integration [MHW10], as well as Vogler's metamodel for systems integration [Vog06] as a foundation. Depending on their nature, the five dimensions of the business component approach will be aligned to either one of the two models.

## 6.1 Architectural Viewpoint Alignment

Herzum and Sims suggest four architectural viewpoints in their approach, defining the execution environment, development patterns and standards, functional design and scope, as well as organizational decisions, including tools and guidelines. In the context of systematic reuse, these viewpoints are defined once and are then applied to different products. It is therefore most feasible to align them with the different layers of our organizational model for industrialized systems integration. This also allows us to distinguish architectures based on business domains, SPLs, and customer specific products.

The project management architecture (PMA) is concerned with architectural and organizational decisions, and associated tools and guidelines to implement a CBD project. Such processes, tools and guidelines are also part of software product lines and considered core assets in SPL development [CN07]. As suggested in our previous work, these joint core assets should be implemented within the business domain layer, which “ensures the interoperability of different systems and product lines within the business domain” [MHW10]. With respect to systems integration, it furthermore allows to consolidate efforts from different product lines of a certain business domain, ensuring ROI also in a heterogeneous environment. The PMA is therefore aligned with the business domain layer. Within the PMA, the business domain layer defines mandatory development tools, processes, and guidelines for all underlying software product lines.

The technical architecture (TA), defines the execution environment, tools, the user-interface framework, and other technical facilities required to develop and run a system [HS00]. With reference to software product line engineering, we can find very similar activities that can be summarized under the term architecture design & development [PBL05; CN07; Lin07]. Although an SPL architecture is more comprehensive than what can be found within a TA, the TA’s reduced scope perfectly fits into our organizational model: The business domain layer defines mandatory technologies, architectures, and systems. These will be further refined within the actual SPL [MHW10]. Having a joint technical architecture ensures the interoperability of different systems and SPLs, reduces technical heterogeneity, and helps to achieve a positive ROI by consolidating architectural efforts. Such architecture would support different technologies as needed; however, it still ensures interoperability from a technical point of view. We therefore align the TA of the component-based approach with the business domain and SPL engineering layer of our organizational model. The business domain layer defines cross product line requirements and standards, whereas the SPL layer defines more detailed technical concepts, aligned with the requirements of the products to be developed therein.

The third viewpoint is the application architecture (AA) and is concerned with “the set of architectural decisions, patterns, guidelines, and standards required to build a component-based system” [HS00]. Such can be architectural principles (e.g. noncircularity) and styles (e.g. type-based vs. instance-based), collaboration patterns for transactions, or a system wide error handling mechanism [HS00]. With regard to the organizational model for industrialized SI, an application architecture is too specific for a whole business domain due to the variety of many different product lines and products. There may be, for instance, shop floor systems in a factory, which require almost real-time perform-

ance, whereas an order management system within the same business domain may be focussed on high scalability. An AA only makes sense for a clearly delimited scope, which can be found in a software product line. We therefore align the application architecture with the SPL Engineering layer of our organizational model. Breaking it further down to the production layer of a single family member would undermine the principles of standardization and systematic reuse and reduce economies of scope.

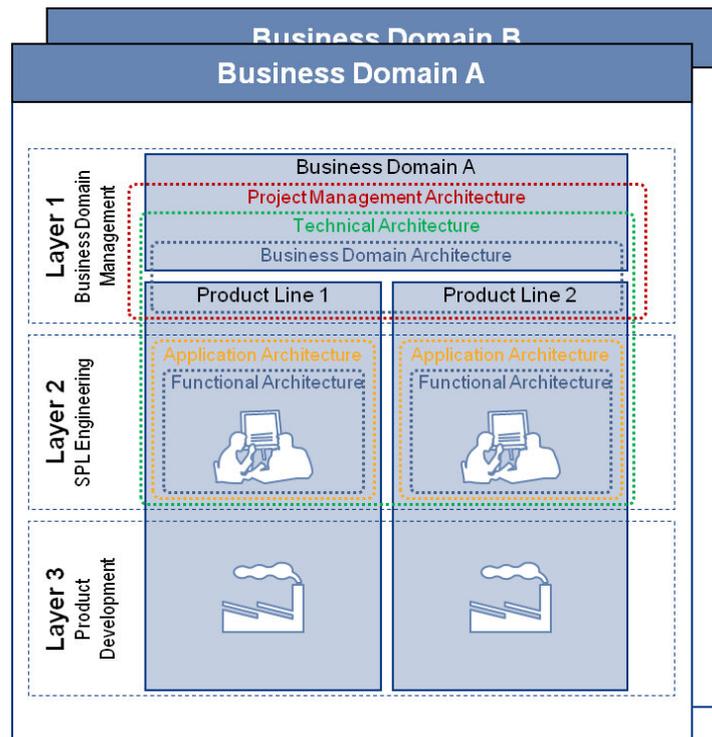


Figure 1: Architectural Viewpoint Alignment

The functional architecture (FA) is the most detailed architectural viewpoint and is “concerned with the functional aspects of the system, including the actual specification and implementation of a system” [HS00]. It consists of component-based business modelling and component-based design. The former can be further broken down into business modelling and functional modelling: “The business modeller’s objective is to produce a model of the problem space that can help identifying business challenges and business opportunities.” [HS00]. “The functional architect, on the other hand, when modelling the business, aims to support the production of a software application” [HS00]. Comparing this separation of concerns with the organizational model for industrialized SI, business modelling represents the ‘business domain analysis & portfolio definition’ process within the business domain layer [MHW10]. Functional modelling in turn, fits into the ‘architecture design & development’ process with the SPL engineering layer, as does the second core process of the FA, i.e. component-based design [MHW10]. We therefore

suggest to separate business modelling from the rest of the functional architecture and align it with the business domain layer of our organizational model. Functional modeling and component-based design should be aligned with the SPL engineering layer to define the functional architecture of the software product line, based on the business model developed in the business domain layer. The alignment is shown in Figure 1.

## 6.2 Component Granularity Alignment

As introduced before, the business component approach is based on five levels of component granularity. In our approach, we omit the smallest and largest one, i.e. the language class and the federation of system level components as they are too small to provide an actual benefit, or too large to be treated as a distinct component for its mere size, respectively.

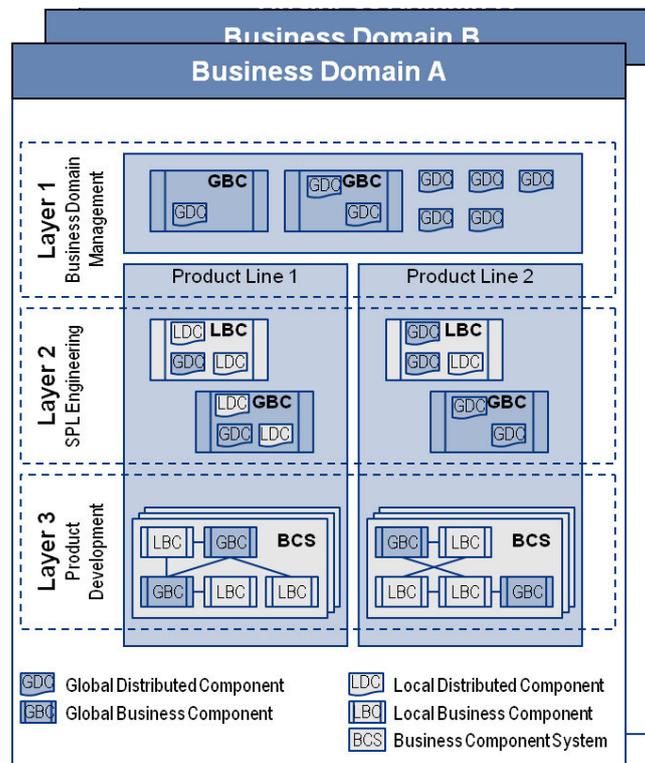


Figure 2: Component Granularity Alignment

Differentiating the three granularity levels based on the entity types of the integration metamodel does not make sense. A distributed or business component may be used in any of these entity types, such as middleware, applications, or workflows. We therefore chose to align the granularity levels with our organisational model for industrialised SI. This allows us to have clear responsibilities for each component, as demanded by the

business component approach [HS00], as well as for core assets in software product lines [PBL05]. Additionally, we can depict definition and refinement of reusable components throughout the hierarchical structure of software product lines, one of the key benefits presented in our previous work [MHW10]. To do so, we enhance Herzum and Sims' approach by differentiating between global distributed components and global business components (GDC and GBC), and local distributed components and local business components (LDC and LBC).

Global components are developed and maintained on the Business Domain Layer and provide reusable functionality for all or some of the underlying SPL. An example would be entities of a certain business domain, such as invoice, order, or bill of materials. In addition, GDCs may provide standardized interfaces to other systems outside of the product line, such as a financial SAP installation or a resource-planning tool. Based on the technical or application architecture introduced in 5, some of these GDCs or GBCs can be made mandatory in underlying SPLs to ensure compatibility between products from different product lines of a given business domain. Local distributed components and local business components are developed and maintained on the SPL level. As shown in Figure 2, they are either developed individually, or partially inherited from the business domain. In both cases, they represent a business concept (for LBCs) or functionality (for LDCs) which is unique for the respective product line. Among other non tangible assets, an appropriate choice of local and global distributed and business components represent the reusable core assets of a software product line.

These core assets can then be used in the product development layer to actually produce an application within an SPL. With reference to CBD, this application represents a business component system (BCS) and therefore the third level of component granularity. As every other component, a BCS must also provide run-time interfaces, be independently deployable, and network addressable.

### **6.3 Development Process Alignment**

The development process dimension of the business component approach defines the chronological sequence in which the activities of the other four dimensions are carried out. It consists of three basic manufacturing processes, which will be aligned with our organisational model for industrialized systems integration as follows.

The first process, rapid component development (RCD), covers definition, building, and testing of individual business components [HS00]. As shown in 6.2, such components may be developed on the business domain, as well as the SPL engineering layer of our organizational model. Rapid component development is therefore aligned with both, the business domain layer, as well as the SPL engineering layer. System architecture and assembly (SAA) represents the second manufacturing process. It covers "architecting, assembling, and testing a system using business components" [HS00]. Compared to the organizational model, these activities occur within the product development layer processes, which are product requirements engineering, product design, product realisation, and product testing [MHW10]. It is important to notice that the SAA is not related to the architectural viewpoints discussed in 6.1. SAA rather selects, adapts, and deploys al-

ready existing distributed and business components according to customer specific requirements. It is the actual manufacturing process in its original sense. We therefore align system architecture and assembly with the product development layer of the organizational model. The third development process, federation architecture and assembly (FAA), is the most advanced activity in business component based development. It designs, assembles, and tests a federation of system level components, i.e. complete business component systems (BCS). As a BCS represents a business component itself, FAA selects, adapts, and deploys already existing business component systems according to customer specific requirements. We therefore also align federation architecture and assembly with the product development layer of the organizational model, although it is far from being trivial and depends on extensive experience and supporting architectures.

#### **6.4 Distribution Domain Alignment**

Herzum and Sims differentiate four different distribution domains, which are the user, workspace, enterprise, and resource tier. “Each tier corresponds to a different logical area of responsibility [...], and each addresses a separate area of concern” [HS00]. They are furthermore grouped into the user workspace, and the enterprise resource distribution domain. This is because local and enterprise wide functionality is usually separated from each other and treated differently in large-scale systems.

The user workspace domain is responsible to “support a single human being’s view of system facilities through some user interaction/interface technology” [HS00]. It represents a typical client application that may also contain locally delimited business functionality, such as interaction with other local tasks. The enterprise-resource domain in turn, implements “a set of computing facilities within which state changes to important (probably concurrently shared) resources can reliably be made” [HS00]. It represents a typical server based application and data source which is used by multiple client applications and users (the user workspace domain), but also other business components from the enterprise resource domain. As this architectural viewpoint is more focussed on the logical structure of a component based system and how to scope and distribute functionality, aligning it with the organizational model would not make sense. We therefore refer to Vogler’s integration metamodel, which can be taken as a foundation when planning and designing an integrated system [Vog06]. This alignment will thus help to decide how certain entities of the integration metamodel may be implemented within a business component approach and vice versa. The integration metamodel defines similar layers or tiers, which are desktop, process, and systems integration. Desktop integration takes distinct tasks as its reference point. It is responsible to present them to the user, interact with him, exchange data with other tasks, and provide an interface to enterprise applications and resources where required [Vog06]. This exactly describes, in other words, the responsibilities of the user workspace domain, which is also responsible to interact with the user and provide local business functionality. However, there may be circumstances in which the metamodel entities workflow, activity, control data, and state also find their way into the user workspace domain. This is the case if an end user application independently represents a complete business process or workflow, including local process integration activities.

The process integration layer of the integration metamodel is concerned with process control by defining one or more workflows, which are hierarchically structurable and consist of different activities and transactions [Vog06]. Such processes usually involve multiple other enterprise resources and end users. The system integration tier is concerned with interfaces and data transfers between different applications and resources. It may also use a middleware, for instance, to support such activities [Vog06]. With reference to the business component model, the enterprise tier implements “enterprise-level business rules, validation and interaction between enterprise components, and it also manages the business aspects of data integrity” [HS00]. These activities reflect those of the process integration tier of the integration metamodel. The resource tier “manages the physical access to shared resources” [HS00] and shields the business logic from technical aspects. They represent the activities of the system integration tier. We therefore suggest aligning enterprise resource domain of the business component model with the process and systems integration layer of the integration metamodel.

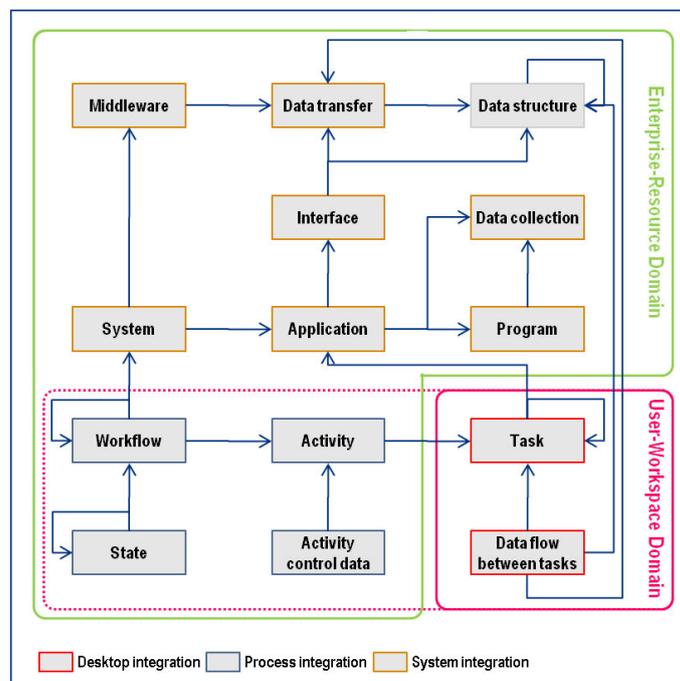


Figure 3: Distribution Domain Alignment

## 6.5 Functional Category Alignment

As with distribution domains, functional categories describe a concept realized separately for each product line and product. They are “concerned with the functional aspects of the system, including the actual specification and implementation of a system that

satisfies the functional requirements” [HS00]. In the following, they will therefore also be aligned with Vogler’s integration metamodel.

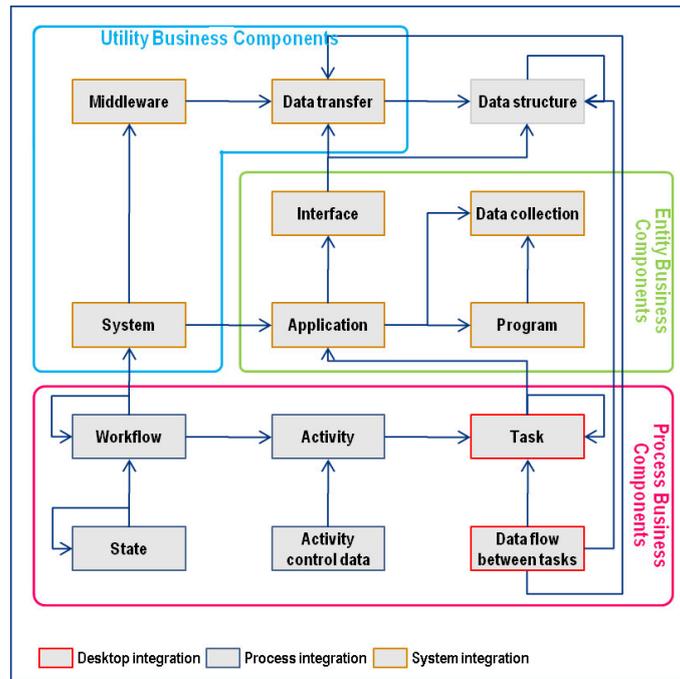


Figure 4: Functional Category Alignment

Herzum and Sims define three broad functional categories for business components, which are utility business components, entity business components, and process business components. The first category defines supporting concepts that are generally available to other components and may be used in a variety of systems and product lines. Examples for utility business components are a middleware system, print services, or a currency converter, for instance. As such, we align the utility functional category with parts of the systems integration layer of the integration metamodel. As the model does not differentiate between technical and functional aspects, we suggest mapping this functional category with the system, middleware, and data transfer entities, although a wide variety of other utility business components is conceivable (but not represented in the integration metamodel). Entity business components represent logical concepts and entities within a system. They often contain persistent information, which may be altered by tasks. Such concepts are usually specific to a business domain (e.g. the automotive industry) or a product line (e.g. a product line for shop floor solutions). They are most likely to be reused in “software systems and component frameworks aimed at supporting a particular collection of business activities” [HS00]. Examples for entity business components are a work order, a bill of materials, or a customer. Concerning the integration metamodel, a precise point of demarcation cannot be found. We therefore assume that applications and programs (together with their data collections and interfaces) can be

seen as system level components as described in 6.2. For new systems, however, these would be replaced by according entity business components, which rely on utility components for data transfer and are being utilized by process business components. “Process business components represent business processes and business activities” [HS00]. They define a business process as a workflow with different activities and tasks to be performed, and also control these to ensure process completion. Such processes are very customer specific and can hardly be reused. Examples for process business components are order management or payment processing. We align process business components with the metamodel entities of the process and desktop integration layer. Both together exactly represent the responsibilities of the process business component category by defining workflows, activities, and tasks together with their respective control data and data flows.

## **7 Conclusion and Further Research**

The present paper introduced into the requirements of industrialized software development in the field of systems integration. It thereby took the position of a typical systems integrator who develops large-scale EAI solutions for a broad variety of customers from different industries. While implementing the three industrial key principles, specialization, standardization, and automation, they encounter certain particularities typical for their field. For SI these are a high heterogeneity due to different technologies, vendors, and legacy systems; one-off developments due to requirements based on a specific systems landscape of a single customer; and an unsure return on investment due to implementation efforts for industrialized development without sufficient throughput to break even.

During the course of research, we identified the business component approach developed by Herzum and Sims as a feasible concept for component based development of EAI solutions on an industrial scale. In its original form, however, it is still affected by the particularities of systems integration. To overcome these particularities, we aligned their approach with one of our previous works regarding the implementation of software product lines, as well as a generic metamodel for integration developed by Vogler. The former allowed us to consolidate recurring tasks of CBD into the business domain layer of the organizational model for industrialized systems integration. We could furthermore show how the different viewpoints of the business component model can be aligned with the organizational units of a software product line in systems integration. In turn, Vogler’s model allowed us to define how different entities of the integration metamodel can be represented by business components (e.g. utility, process, or entity) and how different entities can be distributed in a large-scale system (i.e. user workspace domain, and the enterprise resource domain).

The present work has shown that, with slight adaptations, the business component approach can be applied to the field of software development in systems integration. It furthermore has shown that the approach also is in line with the first industrial principle, specialization.

To further advance industrialized systems integration, further research is needed. First, the approach presented in this paper must be implemented and validated in practice. For the time being, we can only rely on logical derivation of concepts and discussions with subject matter experts from the industry. Due to high implementation cost and the risk of failure, we suggest implementing a small-scale pilot project to test the concept from an organizational and technological point of view. Based on the resulting experiences, a business case can be calculated as the basis for a launch decision.

Besides specialization and standardization, automation as the third key principle of industrialized software development must be researched. Although it generally exists in the form of model driven engineering (MDE), we do not know if it can be applied in all areas of software development. As with CBD, we need to find out if MDE is suitable for systems integration and, if not, how MDE can be adapted to overcome the particularities of systems integration.

## References

- [CHK06] Conrad, S.; Hasselbring, W.; Koschel, A.: Enterprise Application Integration. Grundlagen Konzepte Entwurfsmuster Praxisbeispiele. Elsevier Spektrum Akad. Verl., München, Heidelberg, 2006.
- [CN07] Clements, P.; Northrop, L.: Software product lines. Practices and patterns. Addison-Wesley, Boston, 2007.
- [Enc91] Industrial Revolution: The new Encyclopaedia Britannica. In 32 volumes. Encyclopaedia Britannica, Chicago, London, New Delhi, Paris, Seoul, Sydney, Taipei, Tokyo, 2005; S. 304–305.
- [Fis99] Fischer, J.: Informationswirtschaft Anwendungsmanagement. Oldenbourg, München, Wien, 1999.
- [Has00] Hasselbring, W.: Information System Integration. In Communications of the ACM, 2000, 43; S. 32–38.
- [HS00] Herzum, P.; Sims, O.: Business component factory. A comprehensive overview of component-based development for the enterprise. John Wiley, New York, 2000.
- [Lin07] Linden, F.: Software product lines in action. The best industrial practice in product line engineering. Springer, Berlin, Heidelberg, New York, 2007.
- [LN07] Leser, U.; Naumann, F.: Informationsintegration. Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen. dpunkt-Verl., Heidelberg, 2007.
- [MHW10] Minich, M.; Harriehausen-Mühlbauer, B.; Wentzel, C.: An Organizational Approach for Industrialized Systems Integration. In (Bleimann, U. G.; Dowland, P. S.; Furnell, S. M.; Schneider, O. Ed.): Proceedings of the Eighth International Network Conference, Plymouth, 2010; S. 399–407.
- [PBL05] Pohl, K.; Böckle, G.; Linden, F.: Software product line engineering. Foundations, principles, and techniques ; with 10 tables. Springer, Berlin, 2005.
- [Rie97] Riehm, R.: Integration von heterogenen Applikationen. Dissertation, St. Gallen, 1997.
- [Vog06] Vogler, P.: Prozess- und Systemintegration. Evolutionäre Weiterentwicklung bestehender Informationssysteme mit Hilfe von enterprise application integration. Dt. Univ.-Verl., Wiesbaden, 2006.

# Model Driven Engineering in Systems Integration

Matthias Minich

Bettina Harriehausen

Christoph Wentzel

University of Plymouth  
matthias.minich@acm.org

University of Applied  
Sciences Darmstadt  
b.harriehausen@h-da.de

University of Applied  
Sciences Darmstadt  
c.wentzel@h-da.de

## Abstract

Software development in systems integration projects is still reliant on craftsmanship of highly skilled workers. To make such projects more profitable, an industrialized production, characterized by high efficiency, quality, and automation seems inevitable. While first milestones of software industrialization have recently been achieved, it is questionable if these can be applied to the field of systems integration as well. Besides specialization, standardization and systematic reuse, automation represents the final and most sophisticated key concept of industrialization, represented by Model Driven Engineering. The present work discusses the most prominent approaches, while considering the particularities of systems integration. It identifies Generative Programming as being most suitable and integrates it into previous works on Software Product Lines and Component Based Development in Systems Integration.

## Keywords

Software Industrialization, Automation, Systems Integration, Software Product Lines, Generative Programming, Model Driven Engineering

## 1. Introduction

Compared to other high tech industries, software engineering shows only marginal improvement in terms of productivity, quality, and cost efficiency. It is still characterised by a high degree of craftsmanship to develop software from scratch with labour-intensive methods. By applying industrial methods and thus enhancing an organization's productivity, we possibly can increase quality and product complexity, and at the same time reduce cost and production time. Key industrial methods can be defined as specialization, standardization, systematic reuse, and automation [Enc05]. In the field of software engineering, Software Product Lines (SPL) represent specialization as the first and probably most important principle. By limiting the scope, production assets can be more power- and useful, which is especially important for standardization and systematic reuse as the second industrial principle. Both are available within Component Based Development (CBD), an approach to exchange and systematically reuse software artefacts in a standardized fashion. The final aspect, automation, can be achieved with Model Driven Engineering (MDE). Using models as a description of software written in domain specific languages, the degree of freedom and possible contexts available to a software developer is standardized. Without such standardization it would hardly be possible to provide formal model transformers and code generators. They would

have to cover an indefinite number of possible implementations for e.g. a single business concept.

In today's business world, IT faces high demands in quickly adopting to new requirements. As legacy systems often do not offer the flexibility to do so, new systems are implemented which need to interact with the existing IT landscape. This situation inevitably leads to systems integration efforts, joining the different subsystems into a cohesive whole, in order to provide new functionality or data access [Fis99; LN07]. Systems integration deals with the steps required to move an IT system from a given degree of integration to a higher one by merging distinct entities into a cohesive whole, or integrating them into already existing systems [Rie97; Fis99].

Although several literature on the different industrialization concepts and their practical implementation is available [CN07; HS00; SB07], it seems questionable if they are suitable for all areas of software development, such as systems integration with its high heterogeneity or single-use development projects. The present work therefore takes the position of a large systems integrator, who provides enterprise application integration (EAI) services and solutions to his customers. Research was done with support of a German company active in the field, providing a variety of integration solutions to its customers. The objective was to identify different possibilities for model driven engineering while considering the particularities of the company: Being involved in different industries, a high heterogeneity must be assumed. This anticipates the formation of standards and is reinforced by the fact that integrated systems are often connected on a peer-to-peer basis with each other. Due to high acquisition cost, they are also not replaced frequently [Has00].

We believe that for such heterogeneous, volatile, and customer specific projects, conventional industrialization approaches are hardly feasible. For Software Product Lines and Component Based Development, we have developed a methodology in our previous works about an Organizational Approach for Industrialized Systems Integration [MHW10], and Component Based Development in Systems Integration [MHW11]. The present work deals with the implementation of the third industrial principle, i.e. automating development with the help of Model Driven Engineering. With the given situation and existing MDE concepts, it must be assumed that such intent will never break even, as no considerable economies of scale or scope exist to justify expenses for domain specific language, transformer, and generator development. To overcome this challenge, either reusability or cost efficiency must significantly be increased.

## **2. Automating Software Development**

In automated software development, software engineers specify what to do, but not how. It is up to model transformers or code generators to interpret descriptive models of the intended system and create either intermediate models to be further refined, or source code. Different approaches exist or are currently being researched. The following are the most discussed ones in literature:

- **Model Driven Architecture (MDA):** An initiative from the Object Management Group (OMG), MDA defines a model driven development approach which is based on a separation of functional and technical concerns [OMG03]. It therefore specifies UML as its modelling language, and the Meta Object Facility as its describing model (meta model) for all specification models. These are the Computation Independent Model (CIM), the Platform Independent Model (PIM), the Platform Specific Model (PSM), and the Platform Specific Implementation (PSI). The CIM describes the required systems from hard- and software independent point of view. It can be represented as a high level UML class diagram containing the key concepts and terms of the respective domain. The CIM is further elaborated with conceptual information and transforms into a PIM, describing the required system on a formal and precise level, containing elements like entities, attributes, or data types [PM06]. The PIM is the first model which may automatically be transformed by transformation engines or code generators and thus needs to be as precise as possible [SS09]. Subsequently, it is transformed into the PSM, formally describing the application for the specified platform. Several iterations are possible, until the final result is the Platform Specific Implementation, i.e. an executable artefact reflecting the requirements previously depicted in the CIM.
- **Generative Programming (GP):** Based on the work of Czarnecki and Eisenecker [Cza05], Generative Programming aims at automating the development of a family member within a Software Product Line. It therefore defines a problem space expressed by a Domain Specific Language and the solution space consisting of “implementation-oriented abstractions, which can be instantiated to create implementations of the specifications expressed using the domain-specific abstractions from the problem space” [Cza05]. The mapping between both contains the configuration knowledge such as illegal feature combinations, default settings, default dependencies, construction rules and grammar, or optimizations. These mapping rules are implemented within a generator returning the solution space, which may either be an intermediate model or executable program code.
- **Software Factories (SF):** An approach introduced at Microsoft by Greenfield and Short [GSC04] which, similar to GP, utilizes Software Product Lines and Component Based Development, along with a highly customized IDE. It is based on Software Factory Schemes, which describe certain viewpoints required to develop a system. Such viewpoints express concerns regarding the business logic and workflows, data model and data messaging, application architecture, and technology, and may be present on all levels of abstraction. All together the schemes with their viewpoints exactly define what needs to be done and how to manufacture a family member. In order to provide a customized IDE, the schema with its viewpoints is represented by a Software Factory Template. The template can be loaded into an IDE, providing wizards, patterns, frameworks, templates, domain specific languages, and editors. Complete definitions of domain specific languages furthermore allow (semi-) automatic model to model transformations and code generation.

Compared to MDA, Generative Programming has a domain oriented focus which is usually found in Software Product Lines. MDA in turn does not necessarily rely on a

clearly delimited problem domain. GP furthermore allows to create DSL, generator, and other artefacts required “on the fly” during regular software development. This reduces the necessity of high upfront investments and leads to artefacts tailored exactly to the needs of the implementing company. In contrast to GP and MDA, Software Factories are currently based on proprietary IDEs and modelling frameworks from Microsoft. Furthermore, most of the infrastructure needs to be in place before software development may start, leading to high upfront investments.

Comparing MDE with previous advances of software development, such as compilation technology or 3<sup>rd</sup> generation languages, further advancing the level of abstraction and thus increasing automation seems obvious. However, even after almost 30 years of research in Computer Aided Software Engineering (CASE) and similar approaches as the ones introduced above, this has not yet happened. In an article on automation and model based software engineering [Sel08], Bran Selic names some of the most significant reasons for the lack of acceptance of automated software development in the industry. Foremost, the biggest advantage of fourth generation programming languages (i.e. Domain Specific Languages) is also their biggest drawback: A limited scope makes them very powerful, but also reduces the economies of scale for any infrastructure development such as IDEs, transformation engines, or code generators. Development tools are either built in-house and commercially hardly break even, or by a very small number of vendors, leading to a vendor lock-in. In addition, software developers sufficiently skilled in a particular language or toolset are highly specialized and not easily available on the market. However, even with such available, there are still some more pragmatic issues such as usability of large graphical models, interoperability between tools, or current development culture [Sel08].

In conclusion it can be said that with Model Driven Architecture, Generative Programming, and Software Factories, there are some interesting and promising approaches being developed. However, their way into industrial practice is still prone to “a great deal of improvisation, invention, and experimentation and still carries with significant risk” [Sel08]. Major improvements in standardization and availability of tools must be made to further advance model driven engineering beyond academia. The authors therefore do not believe that for the time being a full-fledged model driven engineering approach in an industrial setting is feasible. This especially applies to the field of systems integration with particularities like one-off development, high heterogeneity, and multiple systems to be integrated. These and their implications on automated software development will be discussed in the following.

### **3. Characteristics of Systems Integration**

Systems Integration comes with certain particularities, distinguishing it from conventional or single-system software development. It has to challenge a multiplicity of technologies, business processes, and other aspects, such as regulatory requirements. Considering the fact that most system integrators are active in multiple industries with multiple customers, chances that one project is similar to another are extremely small. However, the industrialization of software development requires

some sort of specialization, standardization, and automation to be beneficial. While specialization can be found in Software Product Lines and standardization in Component Based Development, automation requires an approach similar to the ones introduced in chapter 2.

As with every new technology, implementation cost are associated with model driven engineering. First, one has to define a domain specific language in which the different applications of a product line will be modelled in. For systems integration, such a DSL needs to represent not only the system that is to be modelled, but also parts of those systems the new one is to be integrated with. Subsequently, respective model transformation engines and code generators must be developed, a task far from being trivial. Depending on the type of integration, such generators need to generate code for different platforms. Once all this is in place, automated software development may begin. Preparations therefore require a certain effort to be completed and must be considered from a cost benefit analysis. However, in the context of systems integration, implementation costs seem contradictory to model driven engineering. With the given situation and existing MDE concepts, it must be assumed that such intent will never break even, as no considerable economies of scale or scope exist to justify expenses for DSL, transformation engine and code generator, and IDE development. Furthermore, one has to consider shortcomings of current tools and development culture as introduced at the end of chapter 2. To overcome these challenges, either reusability or cost efficiency must significantly be increased, as well as suitable tools need to be available.

#### **4. Combining MDE with Industrial Systems Integration**

In our previous work [MHW10], we presented an organizational model for industrialized systems integration, which was done as a first step towards industrialization. It assumes that integration of different IT systems mostly occurs within the boundaries of a certain business domain, as the automotive industry, for instance. Herein, a large number of concepts, such as the logical entities car, supplier, or customer, remain the same for all applications and product lines. The model therefore consolidates similar activities of different product lines within a super ordinate layer, i.e. the Business Domain Layer. The advantage of this consolidation lies in a simplified integration of products from the underlying product lines, and a more efficient implementation approach due to the consolidation of redundant activities. In a subsequent step, we adapted the Business Component Model by Herzum and Sims [HS00] as the second key principle of industrialization [MHW11]. Herein we have shown how the different aspects of the model can be adapted to systems integration by matching them to an integration meta model. In addition we have shown where the required process steps of the Business Component Model are best situated within our Organizational Model for Industrialized Systems Integration.

This leaves us with automation as the final step, represented by model driven engineering. Given the MDE approaches introduced above, we chose Generative Programming as the basis for our work due to its focus on automating the development of a family member within a software product line [Cza05] and its

ability to be implemented concurrently with the actual product being developed. Development within a product line allows for specialization as one of the key principles of industrialization. Advancing the approach while developing an actual product removes the necessity of high upfront investments. In the following sections we will show where in our previously developed organizational model the GP processes are best situated and how they relate to the Business Component Model.

#### 4.1. Development Processes of Generative Programming

Generative programming (GP) includes the following eight main development processes [CE00] to define scope and functionality, infrastructure and core assets, as well as automation artefacts:

1. **Domain Scoping** identifies the domain of interest, stakeholders, goals, and defines the scope of the GP approach. It is influenced by e.g. the stability and maturity of potential solutions, available resources to implement them, and the potential for reuse during production [CE00].
2. **Feature & concept modelling** identifies the distinguishable characteristics of a system within a certain domain and models them within a feature model [CE00].
3. **Common architecture & component definition** depends on the previously developed feature model. Each identified area of functionality requires one or more components, whereas their component model, interaction, type, and distribution will depend on the architecture chosen for the system [CE00].
4. **Domain Specific Language design** specifies a language by defining its syntax and semantics. This may be done in different ways, ranging from simple translational semantics (i.e. defining a translation scheme to an implementation language) to complex axiomatic semantics (i.e. defining a mathematical theory for proving programs written in a given programming language) [CE00].
5. **Specification of configuration knowledge** defines how the problem space will be transformed into the solution space by utilizing the features and concepts identified above. It shields the developer from knowing all components and features by specifying illegal combinations, default settings, dependencies, or construction rules.
6. **Architecture & component implementation** implements the architecture and components identified above. The technology in which both are implemented depends on the scope of the domain.
7. **Domain Specific Language implementation** takes the DSL specification from the DSL design process and derives a concrete implementation. Here GP differentiates between separate DSLs (e.g. SQL or T<sub>E</sub>X), embedded DSLs (e.g. template meta programming in C++), and modularly composable DSLs (e.g. embedded SQL, or aspect oriented programming) [CE00].
8. **Configuration knowledge implementation in generators** allows advancing the problem specified with the help of a Domain Specific Language into executable program code. To do so, generators apply validation of the input specification, complete a given specification with default settings, perform optimizations, and eventually generate the implementation. Generators may be implemented as stand-alone programs, using built-in meta programming capabilities of a programming language, or by using a predefined generator infrastructure [CE00].

Comparing the eight process steps with the concepts of Software Product Line and Component Based Development, Generative Programming can be clearly subdivided into the industrial key concepts of specialization (steps 1 and 2), standardization (steps 3, 5 and 6), and automation (steps 4, 7, and 8).

## 4.2. GP and the Organizational Model for Industrialized Systems Integration

Software Product Lines and Component Based Development already cover the large parts of the GP processes. In the following we will therefore describe how our previously developed approach for Software Product Lines in systems integration needs to be adjusted to incorporate the requirements of Generative Programming.

### 4.2.1. The Business Domain Layer

The Business Domain Layer was developed to align domain wide functionality and utilize economies of scope due to similar concepts and core assets among different product lines of a given domain. It therefore contains the Software Product Line processes domain analysis & portfolio definition, architecture development & roadmap definition, and core asset development.

As to Generative Programming, the above processes already cover the GP processes 1 and 2, such as development of a domain or feature model [MHW10]. Furthermore, the activities of GP processes 3 and 4 are already enclosed in Architecture Development & Roadmap Definition, and Core Asset Development. However, as the Business Domain Layer only features concepts suitable for more than one product line, we have to differentiate between global (business domain wide) and local (product line specific) aspects of GP. This means that there will for instance be DSL design activities in both, the Business Domain and the Software Product Line Layer. In the former, the overall structure and domain wide syntax and semantics are defined, whereas the latter covers product line specific syntax and semantics, such as “bill of materials” for a shop floor system produced in a particular software product line. The distribution is illustrated in Figure 1. Combining the activities introduced in [MHW10] with the respective ones from Generative Programming, the Business Domain Layer in its final stage consists of the following core processes:

- **Business Domain Analysis** explores the typical IT landscape of the business domain in scope and identifies areas of expertise required to develop and provide the products and services under consideration. Similar to software product lines but on a higher level, it identifies recurring problems and known solutions.
- **Portfolio Definition & Domain Scoping** evaluates the information from the domain model and develops a product portfolio for the particular business segment. The portfolio covers typical applications and solutions for the most important business services of the segment and identifies the portfolio elements and resulting software product lines.
- **Architecture & Feature Definition.** Once the scope is defined, a basic product line and integration architecture, a component framework, and an overall feature model, applicable for all product lines are developed. As different product lines have different functional and technical requirements, this architecture may also

exist in an abstract form and be instantiated within the product line subsequently. This approach allows for a later integration of products from different product lines of the same business domain.

- **Core Asset Development** develops reusable assets, applicable to all or many software product lines within the business segment. Such joint core assets may for instance be development tools and processes, or joint software development patterns. Core Asset Development may also include the production of reusable software components equal to each product line. To additionally support Generative Programming, Core Asset Development now also contains the definition of an abstract syntax for a domain wide specification language. This DSL may then be extended within the underlying software product lines in order to support more specific concepts.

#### 4.2.2. The Software Product Line Layer

The Software Product Line Layer consists of several software product lines identified in business domain analysis and portfolio definition processes of the business domain layer [MHW10]. The most obvious variance to a conventional software product line is the lack of the business domain analysis process, and a simplified domain requirements engineering process. These functions are now incorporated in the business domain layer and provide their findings to the subsequent product lines. All other processes remain the same but must adhere to the specifications and utilize the provided core assets from the business domain layer.

As to Generative Programming, we can find all but the first development process within the Software Product Line Layer. However, due to the separation of domain wide and product line specific concerns, the GP processes 2 to 4 only handle product line related concerns. A systems integrator's feature model for the automotive industry may for instance define the entity car with several features, such as model, engine, transmission, colour, price, owner, and so on. These features exist in all products of the underlying product lines. A product line for shop floor systems may however extend this feature model by adding features like electronic control unit (ECU) type, brake type, or parts list. As this has no implication on the functionality of the car itself or the customer, these features are not necessary to be known in other product lines. A financial system does not need to know what type of ECU is built into a car, but it does need to know the price and the owner of the car. This same principle applies to Common Architecture & Component Definition and Domain Specific Language Design. GP processes 5 to 5 are carried out in the software product lines only. Combining the activities introduced in our Organizational Model for Industrialized Systems Integration with the respective ones from Generative Programming, the Product Line Layer in its final stage consists of the following core processes:

- **Requirements Engineering & Feature Modelling** defines the scope of the intended software product line by identifying its products and documenting their commonalities and variability within a feature model. The process has to conform to the Portfolio Definition & Domain Scoping artefacts of the superior business domain layer, but may extend them with product line specific features.

- **Architecture, Component & DSL Design** transforms the scope defined in requirements engineering into a technical architecture and specification for the product line and its products. The architecture decomposes a software system into common and variable functional parts, and specifies the configuration knowledge in terms of component dependencies, default configurations, construction rules, illegal combinations, and rules for their implementation. Each identified area of functionality requires one or more components with an architecture specific component model, interaction scheme, and distribution mechanism. All programming artefacts are finally described within a Domain Specific Language. The process' activities must adhere to the specifications from the business domain layer, but may extend it with product line specific features.
- **Core Asset Development** provides the design and the implementation of reusable software assets [PBL05]. This implementation includes the overall framework, software components, executable code, and other product line assets, such as development processes and tools. In terms of Generative Programming, core asset development is also responsible for the implementation of the DSL as specified in the previous process. In a later and more mature stage, Core Asset Development will implement the configuration knowledge within generators to advance the system specified with the help of a DSL into intermediate models or executable code. As this can be extremely complex, we suggest postponing this activity until reasonable experience with the DSL and the product lines has been gathered.
- **Domain Testing** develops test cases and inspects all core assets and their interactions against the requirements and contexts defined by the product line architecture. Domain testing also includes validation of non-software core assets, such as business processes, product line architecture or development policies.
- **Software Integration** in the context of product line development occurs during pre-integration of several software components. They form blocks of functionality common to all products and contexts of a product line. Furthermore, the integration process ensures the interoperability of all reusable assets and provides the required integration mechanisms.

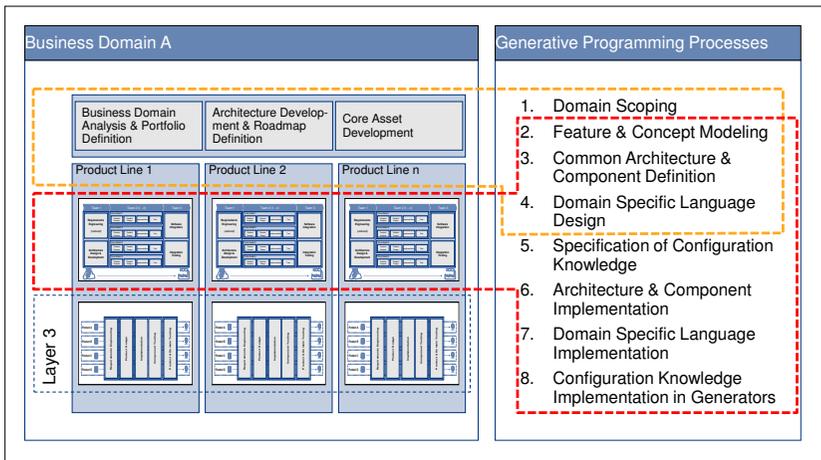


Figure 1: Mapping of GP Processes to Organizational Structure

### 4.3. GP and the Business Component Model

The Business Component Model is a methodology to model, analyse, design, construct, validate, deploy, customize, and maintain large scale distributed systems, developed by Herzum and Sims [HS00]. It consists of five dimensions: Architectural Viewpoints, Component Granularity, Development Process, Distribution Tier, and Functional Categories. In our previous work we have already shown how to align the Business Component Model with our Organizational Model to reflect the particularities of Systems Integration. The following sections will show how these five dimensions fit together with Generative Programming, assuming that development of components occurs with GP.

#### 4.3.1. Architectural Viewpoints

The first dimension consists of four architectural viewpoints, which are the Project Management Architecture (PMA, concerned with organizational decisions, tools, and guidelines), the Technical Architecture (TA, defining the execution environment, component and user interface frameworks, and other technical facilities), the Application Architecture (AA, describing development patterns, guidelines, or standards), as well as the Functional Architecture (FA, identifying the features and functional aspects of a system and their relationships).

With regards to the Project Management Architecture, Generative Programming does not make any statements about the organization or structure of a development project within its processes. The PMA from the Business Component Model is therefore regarded beneficial to the GP approach. In our organizational model, the PMA is found in the Business Domain Layer, whose organizational decisions, tools, and guidelines will influence the development in GP. The remaining three, rather technical viewpoints, are concerned with the execution infrastructure and programming frameworks (Technical Architecture), development patterns, guidelines, and programming standards (Application Architecture), as well as the functional aspects of a system including its implementation (Functional Architecture). Generative Programming in turn only offers the generic process common architecture & component definition. We therefore suggest replacing the respective GP process with the actual implementation of the much more detailed architectural viewpoints from the Business Component Model. For Generative Programming, this replacement offers a more comprehensive view on different aspects of the architecture, while for CBD it ensures coverage of more component related artefacts, such as the component infrastructure or execution environment.

#### 4.3.2. Component Granularity

Generative Programming does not explicitly refer to well defined components as known from e.g. Enterprise Java Beans or Corba. Also it doesn't conceptually concentrate on business processes and therefore does not know reasonable levels of granularity. An artefact may for instance be a generic and reusable data container for C++, allowing handling domain specific types of information. It may also be a reusable programming library providing a complex business concept like a bank

account. Generative Programming rather concentrates on technologies and means to develop reusable artefacts of variable sizes, depending on the intended usage. This way of partitioning a problem into reusable artefacts is known as continuous recursion. One iteratively partitions a problem into different but reasonable granularities. The Business Component model in turn follows a discrete recursion approach. It therefore defines five levels of granularity: the language class, the distributed component (a component in its common sense, e.g. an EJB or CORBA component), the business component (still independently deployable, consisting of distributed components and glue code, representing a business process), and the system level component (a set of business components providing business functionality). The highest level of granularity is the federation of system-level components (i.e. system level components federated to provide multiple complex business services).

We believe that discrete recursion and thus partitioning of the problem is more beneficial in an environment with systematic reuse. For each layer of recursion, a developer has to define scope, characteristics, packaging, and deployment [HS00]. In an environment where components are to be reused as much as possible, it seems more beneficial to define these layers of recursion on a common basis. A middleware messaging adaptor for a specific ERP system will most likely exist as a distributed component as introduced above. A developer can rely on this concept and build his application accordingly. We therefore suggest to introduce discrete recursion to the Generative Programming approach if it is to be used within component based development and systematic reuse in mind.

#### 4.3.3. Development Process

The Business Component Model encompasses a set of manufacturing processes, which support component, system, and federation of systems development. However, as most organizations are in a transitive state towards CBD, Herzum and Sims suggest a process called rapid system development (RSD). It is following the well known V-Model, whereas requirements to implementation denote the left, and component, system, and acceptance testing the right side of the V [HS00]. RSD allows subsequently engineering reusable artefacts based on customer specific requirements and eventually building the respective end product. The advantage is that reusable artefacts evolve on the fly. The disadvantage is that, beginning with the requirements of one specific customer, one may easily miss important variation points or even take architectural decisions which may conflict with the overall scope of the product line. Generative programming in turn focuses much more on domain engineering activities and the technical implementation of reusable artefacts, rather than development of the end product. It puts explicit focus on feature modelling processes such as FODA or FeatuRSEB [CE00], as all GP artefacts rely on a detailed domain model. As research in the field has progressed, we also considered PLUS (Product Line Use Case Modelling for Systems and Software engineering) [EBB06] being a viable alternative for precise domain modelling. The advantage of PLUS over FODA or FeatuRSEB is that besides a feature model it also allows to allocate use cases, use case variations, and cross-cutting concerns to each feature. In the context of the present work we follow the rationale of Generative Programming to

define a precise model of the product domain before implementing any reusable artefacts. This seems especially important if domain specific languages and generators are to be built, although they will be rather simple in the beginning. We therefore suggest to enhance the Requirements, Analysis, and Design activities of Herzum and Sims' rapid system development process with Feature Modelling and Use Case Development of Eriksson et.al.'s PLUSS approach [EBB05]. The result will be a detailed feature model, including a variety of use cases for the required feature combinations. Based on these artefacts, the customer specific application can be built and reusable components derived.

#### 4.3.4. Distribution Tier

In their model, Herzum and Sims separate between user, workspace, enterprise, and resource tier. The user tier presents the component on the screen and communicates with the user. It may be stand-alone, plug in, or non-existent at all. The local business logic is implemented by the workspace tier, which will interact with the enterprise tier. Typical business logic may for instance include transaction management utilizing several enterprise-level resources. The latter are implemented by the enterprise tier, providing business rules, validation, and interaction between components. It typically forms the core functionality of business components of a complex, large-scale component based system. The resource tier manages access to shared resources, such as databases, files, or communication infrastructures and shields all higher layers from their technical implementation.

Such detailed differentiation of reusable components and their internal structure is not provided by the Generative Programming approach. Being more generic, GP leaves such decisions on the target architecture of the product line, which is in turn depending on the overall feature model [CE00]. With regard to the Business Component model, feature model and architecture will already be available and are furthermore influenced by the conceptual structure of business components. In combination with GP, we see no issues when implementing the four distribution tiers with the means of Generative Programming.

#### 4.3.5. Functional Categories

The final dimension defines utility, entity, process, and auxiliary business components [HS00]. Utility components can most generally be reused and represent autonomous concepts, such as unique number generators, currency converters, or an address book. Entity business components represent the logical entities on which a business process operates and are specific to a particular business domain. Examples are item, invoice, address, or customer. The actual business process is implemented within a process business component. Usually unique for one industry or customer, it is hardly reusable. The fourth category, auxiliary business components, provides services usually not found within a process description. Such may be performance monitoring, messaging, or middleware services.

As with the distribution tier above, Generative Programming does not know any functional categories. However, a detailed feature model in connection with

component granularity, distribution tiers, and functional categories, will provide a structured and standardized approach to generative development of business components. As such we believe it is more likely to yield systematic reuse than a structure that is flexible from component to component.

## **5. Conclusion & Further Research**

As we have explained in chapter 3, systems integration comes with certain particularities requiring a highly efficient and cost effective way of implementing industrial key concepts. The low number of similar products in SI seems contradictive to Model Driven Engineering with its Domain Specific Languages, Model Transformers, and Code Generators. However, with integrating GP into our organizational model and combining it with CBD, we can save efforts for domain scoping, feature and concept modelling, architecture and component definition, configuration knowledge specification, and component implementation. All these activities, although slightly adapted, have already been completed, once it comes to the implementation of MDE.

Together with the Business Component Model, a standardized component and implementation architecture is available which allows us to systematically reuse functionality already developed. If a middleware adaptor will always be implemented as an auxiliary business component at the resource distribution tier, it is much more likely to be reused than a freely implemented one. We therefore believe that in order to get the most out of Generative Programming, it must be combined with a component based development approach. Based on our previous work [MHW11], we found the Business Component Model to be most beneficial, especially in the context of systems integration.

The present paper completes the development of a concept for industrialized systems integration. It consists of the organizational model for software product lines in systems integration, reflecting specialization as the first industrial key principle. Subsequently, the alignment of Herzum and Sims' Business Component Model with Vogler's Integration Meta Model describes how to divide a system into a set of reusable artefacts, reflecting the particularities of systems integration. With the present work, Generative Programming has been identified as a potential way towards automation as the final industrial key principle. What is left to be done is a concluding description of the overall concept including the presentation of a field study across all three principles: Beginning with Business Domain design and subsequent Software Product Line definition, over the development of a detailed feature model and component structure, up to the definition and implementation of an initial Domain Specific Language and the according generators with the help of Generative Programming. It is intended to exemplarily develop at least one example of each artefact required for a successful industrialization of systems integration.

## 6. References

- [CE00] Czarnecki, K.; Eisenecker, U.: Generative programming. Methods, tools, and applications. Addison Wesley, Boston, 2000.
- [CN07] Clements, P.; Northrop, L.: Software product lines. Practices and patterns. Addison-Wesley, Boston, 2007.
- [Cza05] Czarnecki, K.: Overview of Generative Software Development. In (Banâtre, J.-P.; Fradet, P.; Giavitto, J.-L.; Michel, O. Ed.): Unconventional Programming Paradigms. Springer Berlin, Heidelberg, 2005; S. 97-97.
- [EBB05] Eriksson, M.; Börstler, J.; Borg, K.: The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In (Obbink, H.; Pohl, K. Ed.): Software product lines. 9th international conference, SPLC 2005, Rennes, France, September 26-29, 2005 : proceedings. Springer, Berlin, New York, op. 2005; S. 33-44.
- [EBB06] Eriksson, M.; Börstler, J.; Borg, K.: Software Product Line Modeling Made Practical. In Communications of the ACM, 2006, 49; S. 49-53.
- [Enc05] Industrial Revolution: The new Encyclopaedia Britannica. In 32 volumes. Encyclopaedia Britannica, Chicago, London, New Delhi, Paris, Seoul, Sydney, Taipei, Tokyo, 2005; S. 304-305.
- [Fis99] Fischer, J.: Informationswirtschaft Anwendungsmanagement. Oldenbourg, München, Wien, 1999.
- [GSC04] Greenfield, J.; Short, K.; Cook, S.: Software factories. Assembling applications with patterns, models, frameworks, and tools. Wiley, Indianapolis, Ind., 2004.
- [Has00] Hasselbring, W.: Information System Integration. In Communications of the ACM, 2000, 43; S. 32-38.
- [HS00] Herzum, P.; Sims, O.: Business component factory. A comprehensive overview of component-based development for the enterprise. John Wiley, New York, 2000.
- [LN07] Leser, U.; Naumann, F.: Informationsintegration. Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen. dpunkt-Verl., Heidelberg, 2007.
- [MHW10] Minich, M.; Harriehausen-Mühlbauer, B.; Wentzel, C.: An Organizational Approach for Industrialized Systems Integration. In (Bleimann, U. G.; Dowland, P. S.; Furnell, S. M.; Schneider, O. Ed.): Proceedings of the Eighth International Network Conference, Plymouth, 2010; S. 399-407.
- [MHW11] Minich, M.; Harriehausen-Mühlbauer, B.; Wentzel, C.: Component Based Development in Systems Integration: GI Lecture Notes in Informatics 2011. Informatik schafft Communities. Ges. für Informatik, Bonn, 2011; S. 470.

- [OMG03] Object Management Group: MDA Guide Version 1.0.1, 2003.
- [PBL05] Pohl, K.; Böckle, G.; Linden, F.: Software product line engineering. Foundations, principles, and techniques ; with 10 tables. Springer, Berlin, 2005.
- [PM06] Petrasch, R.; Meimberg, O.: Model Driven Architecture. Eine praxisorientierte Einführung in die MDA. dpunkt, Heidelberg, 2006.
- [Rie97] Riehm, R.: Integration von heterogenen Applikationen. Dissertation, St. Gallen, 1997.
- [SB07] Stahl, T.; Bettin, J.: Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management. dpunkt-Verl., Heidelberg, 2007.
- [Sel08] Selic, B.: Personal reflections on automation, programming culture, and model-based software engineering. In Automated Software Engineering, 2008, 15; S. 379-391.
- [SS09] Singh, Y.; Sood, M.: Model Driven Architecture: A Perspective. In (IEEE Computer Society Ed.): Proceedings of the 2009 IEEE International Advance Computing Conference. IEEE Computer Society, Patiala, 2009; S. 1644–1652.