FEATURE-BASED SOFTWARE ASSET MODELING WITH DOMAIN SPECIFIC KITS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

NESİP İLKER ALTINTAŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER ENGINEERING

AUGUST 2007

Approval of the thesis

FEATURE-BASED SOFTWARE ASSET MODELING WITH DOMAIN SPECIFIC KITS

submitted by **Nesip İlker Altıntaş** in partial fullfillment of the requirements for the degree of **Doctor of Philosophy in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen ________________
Dean, **Graduate School of Natural and Applied Sciences**

Prof. Dr. Volkan Atalay ________________
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Ali H. Doğru ________________
Supervisor, **Computer Engineering Dept., METU**

Dr. Semih Çetin ________________
Co-Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. A. Ziya Aktaş ________________
Computer Engineering Dept., Çankaya University

Assoc. Prof. Dr. Ali H. Doğru ________________
Computer Engineering Dept., METU

Prof. Dr. Semih Bilgen ________________
Electrical and Electronics Engineering Dept., METU

Prof. Dr. İ. Hakkı Toroslu ________________
Computer Engineering Dept., METU

Assoc. Prof. Dr. Halit Oğuztüzün ________________
Computer Engineering Dept., METU

Date: 10.08.2007

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Nesip İlker Altıntaş

Signature :

ABSTRACT

FEATURE-BASED SOFTWARE ASSET MODELING WITH DOMAIN SPECIFIC KITS

Altıntaş, Nesip İlker
Ph.D., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Ali H. Doğru
Co-Supervisor: Dr. Semih Çetin

August 2007, 172 pages

This study proposes an industrialization model, Software Factory Automation, for establishing software product lines. Major contributions of this thesis are the conceptualization of Domain Specific Kits (DSKs) and a domain design model for software product lines based on DSKs. The concept of DSK has been inspired by the way other industries have been successfully realizing factory automation for decades. DSKs, as fundamental building blocks, have been deeply elaborated with their characteristic properties and with several examples.

The constructed domain design model has two major activities: first, building the product line reference architecture using DSK abstraction; and second, constructing reusable asset model again based on DSK concept. Both activities depend on outputs of feature-oriented analysis of product line domain. The outcome of these coupled modeling activities is the reference architecture and asset model of the product line.

The approach has been validated by constructing software product lines for two product families. The reusability of DSKs and software assets has also been discussed with examples. Finally, the constructed model has been evaluated in terms of quality improvements, and it has been compared with other software product line engineering approaches.

Keywords: Asset Modeling, Domain Specific Kits, Feature-Based Software Development, Software Architectures, Software Factories, Software Product Lines

ÖZ

ALANA ÖZGÜ KİTLER İLE ÖZELLİK BAZLI YAZILIM VARLIK MODELLEMESİ

Altıntaş, Nesip İlker
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Ali H. Doğru
Ortak Tez Yöneticisi: Dr. Semih Çetin

Ağustos 2007, 172 sayfa

Bu çalışma yazılım ürün hatları kurulmasına yönelik Yazılım Fabrika Otomasyonu olarak adlandırılan bir endüstrileşme modeli önermektedir. Bu tezin ana katkısı Alana Özgü Kitlerin (AÖK) kavramsallaştırılması ve buna dayalı olarak yazılım ürün bantlarına yönelik bir alan tasarım modelidir. AÖK kavramı diğer endüstrilerde yıllardır uygulanmakta olan fabrika otomasyon modelinden esinlenmiştir. Ana yapı taşı olarak AÖK'ler, temel nitelikleri ve örnekleri ile detaylı olarak incelenmiştir.

Geliştirilen alan tasarım modeli iki ana aktivite içermektedir: Birincisi, AÖK'ler kullanılarak ürün hattı referans mimarisinin oluşturulması ve ikinci olarak yine AÖK kavramına dayanarak yeniden kullanılabilir varlık modelinin geliştirilmesidir. Her iki modelleme aktivitesi de özellik bazlı alan analizi çıktıları üzerine kurgulanmıştır. Bu iki modelleme aktivitesi çıktıları ürün hattı referens mimarisi ve varlık modelidir.

Yaklaşım, iki farklı ürün ailesi için yazılım ürün hattı kurularak denenmiş ve geçerlenmiştir. AÖK ve yazılım varlıklarının tekrar kullanılabilirliği örnekler ile tartışılmıştır. Son olarak, geliştirilen model sağladığı kalite iyileştirmeleri açısından değerlendirilmiş ve diğer ürün hattı mühendislik yaklaşımları ile karşılaştırılmıştır.

Anahtar Kelimeler: Alana Özgü Kitler, Özellik Bazlı Yazılım Geliştirme, Varlık Modelleme, Yazılım Fabrikaları, Yazılım Mimarileri, Yazılım Ürün Bantları

To the Memory of my Mother and my Father...

ACKNOWLEDGMENTS

Finally, this long process has come to an end. I have to thank those people who have helped me along the way. I really don't know how to express my deep gratitude to them.

First, I would like to thank Dr. Semih Çetin who is my co-supervisor, my boss and my friend all at the same time. Your vision and enthusiasm, in search of better, faster and cheaper, have contributed a lot to this study. It would have been almost impossible to finish without your friendship, motivation, guidance and encouragement. Thank you!

I express my deep appreciation to Assoc. Prof. Dr. Ali Doğru, as my supervisor, for letting me freely come up with new ideas each time, for his constant positive outlook and his constructive comments. Prof. Dr. Hakkı Toroslu and Prof. Dr. Semih Bilgen guided me with their feedback during the progress meetings throughout the study. Thank you!

Dr. Onur T. Şehitoğlu, my old friend, supported me a lot during the writing of my dissertation. My problems in latex were solved by his intelligent tricks. Thank you!

To my former supervisors, Assoc. Prof. Dr. Cem Bozşahin and Assoc. Prof. Dr. Halit Oğuztüzün, and my former colleagues in Department of Computer Engineering: thank you for your contributions to my academic background.

I feel lucky to work at Cybersoft, a fruitful environment for technology and engineering. My special thanks go to my colleagues at Cybersoft: especially to Mr. Mehmet Surav for many discussions on the drafts of this work. Thanks to Mrs. Özgür Tüfekçi for reading drafts of the manuscript and her constructive comments. I am also grateful to my other colleagues which helped to overcome the challenges and hard times in our projects during which many ideas have appeared. Thank you!

Finally to my family; your love, support and patience have smoothed my way throughout the study. Thank you İlkay, you have stood by me all along this long road with your love, support and encouragement. Thank you Cansu, my little daughter, for your patience during these long days of work- especially since some of our playtime has been sacrificed for my research. And last but not the least, thank you Pelin, for sharing the life.

Thank you all!

TABLE OF CONTENTS

APPENDIX

LIST OF FIGURES

FIGURES

LIST OF TABLES

TABLE

LIST OF ABBREVIATIONS

ABD	Architecture-Based Development
ACM	Asset Capability Model
ACS	Architectural Concern Spaces
ADD	Attribute-Driven Design
ADL	Architecture Description Language
ADO	ActiveX Data Object
AJAX	Asynchronous JavaScript and XML
AML	Asset Modeling Language
AMM	Asset Meta Model
AOM	Adaptive Object Model
AOP	Aspect-Oriented Programming
API	Application Programming Interface
AQAP	Allied Quality Assurance Publications
ARID	Active Reviews for Intermediate Designs
ATAM	Architecture Tradeoff Analysis Method
B2B	Business-to-Business
B2C	Business-to-Customer
BPE	Business Process Engine
BPEL	Business Process Execution Language
BPM	Business Process Management
BPP	Business Process Platform
BRE	Business Rule Engine
BRMS	Business Rule Management System
BSE	Business Service Engine
CAF	Composite Application Framework
CBAM	Cost-Benefit Analysis Method
CBD	Component-Based Development
CDL	Choreography Description Language
CMMI	Capability Maturity Model Integration
CORBA	Common Object Request Broker Architecture
CRA	Central Registry Agency
DAO	Data Access Object
DFM	Domain Feature Model
DSA	Domain Specific Artifact
DSAT	Domain Specific Artifact Type
DSE	Domain Specific Engine
DSK	Domain Specific Kit
DSL	Domain Specific Language
DSM	Domain Specific Modeling
EBML	Enhance Bean Markup Language
EDS	EBML Development Studio
ERE	EBML Rendering Engine
ERP	Enterprise Resource Planning
EIA	Enterprise Internet Application
FCL	Feature-Constraint Language
FGW	Financial Gateways Product Line

FODA	Feature-Oriented Domain Analysis
FODM	Feature-Oriented Domain Modeling Method
FORE	Family-Oriented Requirements Engineering
FORM	Feature-Oriented Reuse Method
GP	Generative Programming
GPD	Graphical Process Designer
GUI	Graphical User Interface
HL7	Health Level Seven
HTTP	Hypertext Transmission Protocol
IDE	Integrated Development Environment
INV	Investment Banking Product Line
ISO	International Standards Organization
IT	Information Technology
JDBC	Java Database Connectivity
JPDL	jBPM Process Definition Language
LRE	Listing and Reporting Engine
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MOF	Meta Object Facility
MVC	Model-View-Controller
MQ	Message Queue
ODBC	Open Database Connectivity
OO	Object Orientation
OOP	Object-Oriented Programming
O2R	Object-to-Relational
OMG	Object Management Group
OVM	Orthogonal Variability Model
PIM	Platform Independent Model
PLC	Programmable Logic Controller
PME	Persistence Management Engine
POM	Persistent Object Model
POJO	Plain Old Java Objects
PSM	Platform Specific Model
QAW	Quality Attribute Workshop
QoS	Quality-of-Service
RAS	Reusable Asset Specification
RDBMS	Relational Database Management System
RE	Requirements Engineering
RIA	Rich Internet Application
RISC	Reduced Instruction Set Computer
RUMBA	Rule-based Model for Basic Aspects
SAAM	Software Architecture Analysis Method
SCA	Service Component Architecture
SCADA	Supervisory Control and Data Acquisition
SDLC	Software Development Life Cycle
SDO	Service Data Objects
SFA	Software Factory Automation
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service-Oriented Computing
SPI	Software Process Improvement
SPICE	Software Process Improvement and Capability Determination
SPL	Software Product Line
SQL	Structured Query Language
UCS	Utility Concern Space
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language

VP	Variability Point
WS-CTX	Web Service Context
WS-CF	Web Service Coordination Framework
WS-TXM	Web Services Transaction Management
WSDL	Web Services Description Language
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION

The software development for enterprises becomes increasingly challenging with the unbridled evolution of Web-based presentation techniques, mobile and ambient environments, different access channels, B2B and B2C system integration requirements (e.g., a typical banking software, an airline reservation system, or an e-government application may require the connectivity to tens of systems and might be accessed through diverse channels), and concurrent use of several development environments (e.g., a 30 years old CICS software should be maintained together with a brand new mobile device software). In spite of these, organizations need to achieve large productivity gains, improve time-to-market, maintain market presence and sustain unprecedented growth, improve product quality and customer satisfaction, achieve reuse goals, and enable mass customization.

1.1 Statement of the Problem

The common characteristics of contemporary IT solutions can be overviewed as follows:

- They are usually technology or platform driven, so their focus is on the accidental difficulties, rather than essential difficulties in software engineering. Brooks defined the four inherent difficulties of modern software as: “complexity”, “conformity”, “changeability”, and “invisibility”. Much of the “complexity” that software engineer has to master is arbitrary and forced by the differing human institutions and systems, in other words the atypical business environment to which the software interfaces must “conform”. The “invisibility” of software hinders the communication among minds, particularly the minds of business and IT professionals. Worse than that software is embedded in a cultural mix of applications, users, laws, and machine vehicles that continually change, inexorably forcing change upon the software product. IT departments are overburdened with intolerable costs of “changeability” resulted from the everlasting demands of business departments [23].

- They are labor-intensive which limits the repeatability of solutions without depending on the man-power. The methods, techniques, processes, and best-practices usually come and go away with the team; they are volatile.
- They are extremely costly with high percentage of hidden costs rather than direct implementation costs. Understanding the business, capturing the requirements, design with customizability and reusability in mind, coping with technology waves and adaptations to the most recent standards creates all hidden costs that a software product or project must face.
- They are based on the abstractions from IT perspective. The entities, objects, services, components, aspects, models, etc. are all the building blocks of technical staff.
- They are usually difficult and expensive to adapt. Flexibility of software to new requirements, adaptability of architectures, parameterization, managing the variability and commonality can not simply be addressed with ad-hoc methods or on-demand development of such abilities is excessively expensive.
- They are not tolerable to shifts and changes in business, where business needs hand-on flexibility in accordance with the market changes.
- They usually lack in satisfying the quality targets. Quality targets for functional and non-functional requirements are equally important; hence the product-oriented and process-oriented quality models need to be in place.

In summary, one-fits-all approach with generic processes, methods, models, architectures, frameworks and tools depends on a real craftsmanship and is labor-intensive; hence it provides minimal reuse.

For the past two decades, software industry has mainly demanded personal productivity. Instead, the software industry is now switching gears to explore technologies which automate business processes. As the industry matures, businesses look for much richer functionalities and quicker response times. Accordingly, software industry should surpass the techniques that brought it to this point, and embrace the industrialization best practices achieved by manufacturing. These include product assembly from components, reducing labor-intensive tasks with automation, setting up the software product lines and supply chains, formalizing the interfaces, and standardizing architectures and processes.

The vision of improving reusability is in the heart of the problem. Improving reuse and hence quality is critical for increasing the productivity of software teams as well as decreasing the cost and time to market of software products. Boehm puts special emphasis on software productivity management through systematic reuse leveraged by three basic strategies: working faster via tools to automate the labor-intensive tasks, working smarter with process improvement, and working less via reuse of software artifacts [20]. The question is which strategy will produce the highest payoff? An extensive analysis addressed this question for the US Department of Defense and concluded that *working less* is more valuable three times than *working smarter* and six times than *working faster* [19].

Reuse must be maximized; however managing the software reuse is not trivial. The classes, templates, programs, executables, frameworks, architectures, domain know-how, assets, features, etc. are all candidates to be reused, if possible all at the same time. The higher the abstraction level of reusable item is the higher the gain in benefits and improvement in the productivity in software development since it reduces the complexity and brittleness of the software [62]. Contrarily, the powerful abstractions also narrow down the scope of application. Since no two software installations are exactly the same, reuse requires the management of variability and commonality, ease of configurability, and effective means to manage the software configurations.

The abstraction level in software design plays a major role in the specification of software assets for maximizing the reuse [19]. It has also been revealed that reuse maximization needs the separation of both inter-asset and intra-asset concerns, and their subsequent composition in software design. The strategy is to model the functional (business) and non-functional (architectural) requirements of the problem domain and designing the solutions domain by reusing the solution domain artifacts. Note that solution domain artifacts are not one-to-one mapped from the problem domain, and even their terminologies are different.

The problem statement, in short, is as follows:

> How can we incorporate domain specific abstractions to improve systematic reuse of software assets on the basis of an industrialization model in order to enhance the software development productivity and product quality while reducing the per product development and maintenance costs?

1.2 Research Method

Typically, a research method has three phases: determination of the research question; conducting the research and obtaining the research results; and validating the results. For conducting research, several sub-questions can be raised in the light of above research statement. For each sub-questions, we have employed the following particular research steps:

Q1. *Can domain specific know-how be abstracted and reused across different business domains?* To incorporate domain specific abstraction, Domain Specific Kits (DSKs) have been devised as a basic building block to express the domain specific types and artifacts. It has been validated by modeling several DSKs from different domains and they are checked whether they can be reused across different product families.

Q2. *Can such abstractions increase the reusability of software assets?* An asset modeling approach has been constructed, and two product families have been modeled using this approach. The reuse scope and reuse rates in these families have been investigated for validation purposes.

Q3. *What should be the content of reusable assets for increased reuse scope?* The DSK abstraction has been used for product line asset modeling. The domain specific artifact types, their instances, their dependencies, variability mechanisms, and contextual information have been abstracted as software assets. We have later checked that the assets can be reused not only within a product family but also in multiple product families as long as their DSK and contextual dependencies are provided.

Q4. *How do those domain specific abstractions be employed and helpful in modeling the reference architecture of a product family?* A reference architecture modeling approach has been constructed based on the separation of concerns both in problem and solution domain. The concept of DSK has been successfully utilized in product line reference architecture. This has been validated by constructing reference architectures of different domains.

Q5. *Can we construct a roadmap for setting up software product lines for different domains out of a reference model?* The whole model has been named as Software Factory Automation (SFA) approach, and it has been used experimentally to model two distinct product families.

1.3 Publications

The results of research in this thesis have been presented in several papers in various workshops and conferences. Figure 1.1 presents the roadmap of publications which paves the way to the SFA approach.

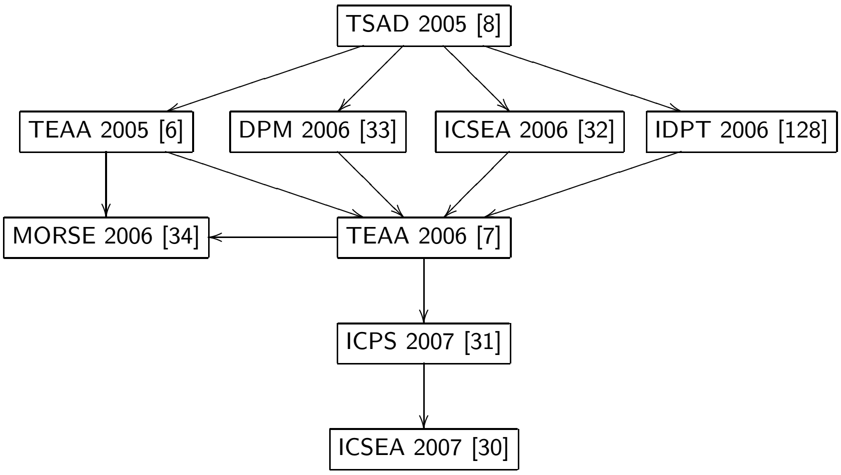


Figure 1.1: Roadmap of publications

In [8], an in-house Software Product Line (SPL) of Cybersoft[1], so-called Aurora, has been introduced as a platform independent, service-oriented, and multi-tier Web

[1] Cybersoft C/S Information Technologies Co., http://www.cybersoft.com.tr/

application development environment including the core infrastructure based on Rich Internet Application (RIA) and Enterprise Internet Application (EIA) models. Besides Aurora provides a complete roadmap to enterprise scale Web-based applications, it also embodies the SPL by providing software process management methodology, design and development environments, software lifecycle management techniques and quality management tools. Essential SPL activities, such as management, core asset development and product development, have been evaluated briefly.

The [6] and [33] are initial steps of the constructed approach to segregate and compose the artifacts (concerns) during software design.

In [6], an approach has been proposed to integrate Aurora with reflective rule-based business process modeling (RUMBA). RUMBA is a rule-based model in which rules and rule-sets can be expressed in terms of dynamic aspects and delegated facts. The approach mainly addresses "Reflective Aspect" and "Reflective Rule" patterns for the seamless integration of Aurora and RUMBA. Both architectural patterns introduce a "generative" approach for developing the basic aspects, dynamic rules and rule-sets so that all can be implemented with Adaptive Object Model (AOM).

In [33], the segregation of business rules that crosscut several parts, such as workflows, task assignments, and business transactions, at almost every tier of software architecture has been further explored. Seamless integration with the rest of the picture has been presented with a practical Aspect-Oriented Framework for rule-based business process management where all aspects, facts, rules and rule-sets can be defined and managed dynamically by means of a GUI console. Moreover, this lightweight framework has been implemented in conformance to Adaptive Object Model to facilitate the process dynamism through declarative techniques and bytecode engineering.

In [32], another dimension, i.e. Architectural Modeling, of the study has been initiated. Architectural modeling identifies several concerns in problem domain and associates them with design decisions in solution domain. This paper proposed a modeling approach to address the architectural concerns in multiple concern spaces both for problem and solution domains, and align them symmetrically. Chapter 4 is primarily based on this paper.

In [128], the problem of business modeling has been elaborated with a new outlook based on business process categorization, separation of concerns and loose coupling, patterns of business process capturing, conceptualization of business assets in terms of processes, services and rules, declarative definition and vibrant management of business assets, and finally proper architecture over which the quality of business assets can be best matched with the functionality of business processes. The primary focus of the study was how the concerns of business and IT departments can be separated so that business goals and IT goals can be fulfilled independently; and the term “discrepancy of the perspectives” has been defined for business modeling.

In [7], the SFA model has been proposed as an industrialization model for software development. The study basically introduced the concept of Domain Specific Kit (DSK) and defined the concept of “software factory automation” for setting up product lines and managing reusable assets across distinct software product lines. Chapter 3 and Chapter 5 are primarily based on this paper.

In [34], the concept of DSK has been elaborated within the scope of reuse, and the similarities of software modeling and manufacturing industries have been identified. It has tried to show how the former can benefit from the latter for the systematic reuse of domain specific models.

In [31] and [30], the DSK and choreography engine concepts have been reshaped as an enabling technology towards migrating to the service harmonization platform in the context of Service-Oriented Computing. This work provides a roadmap for the migration of legacy software to Service-Oriented Computing by means of right levels of abstraction. The proposed approach has also been exemplified on a simple case problem. It has been briefly discussed in Section 6.7 as an application of the concepts introduced here.

1.4 Organization of the Thesis

The rest of the manuscript has been organized as follows:

In Chapter 2, the background work on software reuse and abstraction has been studied and briefly discussed. The discussion covers a historical view starting from Object-Oriented Development to today’s Software Factory approaches. In the meantime, Component-Based Development, Model-Driven Development, Service-Oriented Computing, Architecture-Based Development, Asset-Based Development, Feature-Based Approaches and Software Product Lines have been discussed within the scope of improving reuse and abstraction in software engineering.

Software Factory Automation (SFA), as a software industrialization model, has been introduced in Chapter 3. The model depends on a key conceptualization, named Domain Specific Kit (DSK), to construct and reuse the domain specific artifacts for a product family.

The reference architecture modeling, which is crucial for the approach, has been introduced in Chapter 4. It includes the details of six-step roadmap and a case study to demonstrate the applicability of the approach in Web security framework modeling. The chapter also includes a brief survey of architecture modeling techniques in relevance to the proposed approach.

The accompanied asset modeling approach has been introduced in Chapter 5. The five-step feature-based software asset modeling roadmap depends on the architectural model of the previous chapter. The key idea is to specify asset models with reusable Domain Specific Artifacts abstracted by DSKs (composed of a domain specific language, engine, and the toolset). This approach encapsulates correlated features within more cohesive asset models and composes them through a choreography engine.

Chapter 6 presents the evaluation of the study from different perspectives. For the validation purposes, there are two case studies modeled using the proposed approach. The results have been validated with respect to the problem definition and research questions. The previous chapters also include several simpler and partial examples for clarifying the concepts and ease of understanding.

Finally, Chapter 7 includes the summary of the work, concluding remarks and further research areas.

CHAPTER 2

BACKGROUND

This chapter discusses the existing approaches for improving reuse and abstraction on the way to industrialization of software development.

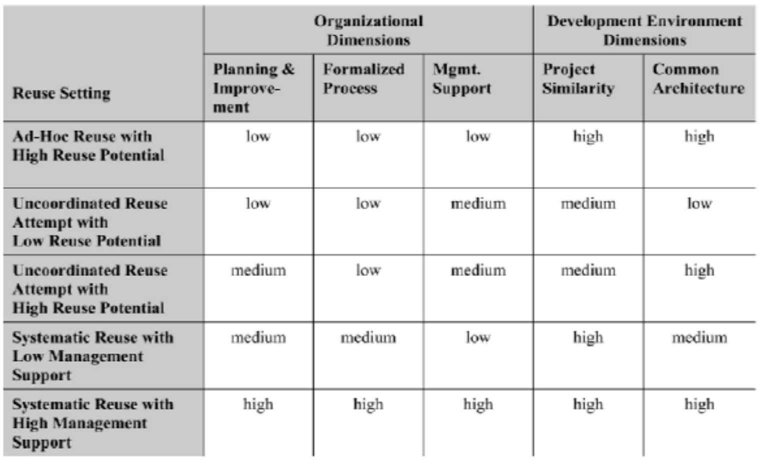
2.1 Reuse in Software Engineering

In a recent analysis of reuse strategies, Rothenberger et al. [114] have investigated the practical reuse strategy alternatives and their effectiveness for a successful reuse program. Based on the data collected from 71 software development groups all over the world, they have performed a principal component analysis to identify the dimensions that best describe the characteristics of software reuse settings and their potential for reuse success. The study establishes the dimensions and classifies the reuse settings in five reuse archetypes as given in Table 2.1.

In this study, it has been concluded that the success of reuse is independent from the choice of technology. The results are summarized as follows:

- Performing well in all reuse dimensions leads to all of the reuse benefits.
- Software quality can be realized by a focus on project similarity and common architecture.
- Performing only moderately well, or poorly, across all of the dimensions only leads to moderate or poor reuse success.
- Focusing on formalized process and project similarity can have good overall performance, but not the best without the other dimensions.

Table 2.1: Reuse archetypes (from Rothenberger et al. [114])



Reuse Setting	Organizational Dimensions			Development Environment Dimensions	
	Planning & Improvement	Formalized Process	Mgmt. Support	Project Similarity	Common Architecture
Ad-Hoc Reuse with High Reuse Potential	low	low	low	high	high
Uncoordinated Reuse Attempt with Low Reuse Potential	low	low	medium	medium	low
Uncoordinated Reuse Attempt with High Reuse Potential	medium	low	medium	medium	high
Systematic Reuse with Low Management Support	medium	medium	low	high	medium
Systematic Reuse with High Management Support	high	high	high	high	high

Throughout the years, several proposals have been on the stage for improving the reuse. Transition from procedures and data to object encapsulation [21, 40], later to components [24, 50, 89], now to service and business processes are well-known examples. Recent approaches like Service-Oriented Computing [30, 107], Model Driven Development [59, 99, 115], Architecture-Based Development [12, 13, 78, 80, 81, 117], Asset-Based Development [88, 112], Feature-Based Approaches [45, 64, 75] and Software Product Lines (SPL) [38, 103, 109, 111, 136], Software Factories [63, 84, 87, 94, 98] identify the same problem with different perspectives.

2.2 Object-Oriented Development

Object Orientation (OO), as a paradigm, has been used increasingly as an approach to facilitate the reuse. The use of object-oriented programming languages, object-oriented analysis and design methodologies, distributed object computing techniques, and object-oriented domain modeling languages have come to scene for better quality software and improved reuse. During the last decades, it had been sternly advocated that OO paradigm encompassed the complete view of software engineering without the loss of communication [21].

The idea behind object orientation assumes that we have been living in a world of objects [3]. Modeling, understanding, and developing objects are easier since they constitute a common vocabulary. The objects take place in nature, in human made entities, in businesses, and in the products that we use. Both data and the processing applied to that data have been encapsulated by objects. The practice of defining data structures and code in the same class keeps the elements that need to be reused as a unit within one framework, and encapsulation forces to clearly define the interfaces of each class to the outside world.

The object-oriented paradigm has been attractive to many software development organizations with the expectation that it yields reusable classes and objects. While, at the same time, the software components derived using the object-oriented paradigm exhibit design characteristics (e.g. proper decomposition, functional independence, information hiding etc.) that are associated with high-quality software [40].

Rothenberger et al. [114] have checked whether "higher levels of object technologies are associated with higher levels of reuse program success". Although the use of "Object Technologies" was initially a candidate for a reuse dimension, it was determined to be insignificant in explaining the reuse success. They concluded that an organization's reuse success is not dependent on the use of object-oriented techniques. This result is also consistent with object technology practice and research [53, 106]. Both indicate that object-oriented methods do not always lead to high reuse. An organization may succeed at reuse without employing object-oriented methods. A reuse program may benefit object-oriented methods, but it takes more than just object orientation to succeed.

2.3 Component-Based Development

The demand for low production costs, short time to market and high quality is also addressed by means of the rapidly emerging Component-Based Development (CBD) approach. In CBD, software systems are built as an assembly of components already developed and prepared for integration. This aims the development of components as reusable entities as well as the maintenance and upgrading of systems by customizing and replacing their components. The main advantages of the CBD approach include effective management of complexity, reduced time to market, increased productivity, a greater degree of consistency, and a wider range of usability [24].

The distinction between components and objects are as follows: in addition to many borrowed concepts from objects, components integration capabilities are far more improved, their interfaces have more power with a protocol plus lists of events in addition to properties and methods. On the other hand, components are limited to composition whereas objects can use inheritance [50]. Components are in general considered as black boxes with little or no information easily accessible.

The development processes of component-based systems are separated from development processes of the components; the components should already been developed and possibly used in other products when the system development process starts [43].

A general process model for component-based software development starts with system specification; goes on by decomposition of system into components; proceeds with specification, search, modification, and creation of components; and finally concludes with integration [50]. System decomposition is an iterative process through alternate decomposition and composition activities until the specifications of modules agree with a set of components. This also requires a new activity of finding and evaluating the components.

Despite many foreseen advantages, there is a number of reuse challenges using components [89]:

- Component-based applications are sensitive to evolution of the system. As components and applications have separate lifecycles and different kinds of requirements, there is some risk that a component will not completely satisfy the particular requirements of certain applications or that it may have characteristics not known to the application developer. One of the most important factors for successful reusability, in an evolving software system, is the compatibility between different versions of the components. Evolution of system requirements (functional and non-functional), evolution of technology used in software products, evolution of technology related to different domains, and evolution of technology used for the product development are all affect the long life products. In order to cope with these evolutions, the components must be updated more rapidly and the new versions must be released more frequently than the products using them.

- When developing reusable components, the development process must consider the development of components on different platforms; development of different variants of components for different products; independent development of components and products. In order to cope with these types of problems, complicated development processes are essential as well as an appropriate product architecture and component design.

- The maintenance of reusable components process is also complex, because the relations between components, products and systems must be carefully registered to make possible the tracing of errors on all levels. It is even more complicated in case of external components.

Finally, while component-based models successfully deal with functional attributes, they provide limited support for managing quality attributes of systems or components. The quality aspects of software products are not, however, addressed adequately by component-based development.

2.4 Model-Driven Development

Model-Driven Development (MDD) is a model-centric software engineering approach which aims at improving the quality and lifespan of software artifacts by focusing on models instead of code [59]. Models are considered as first class entities. A system is described by a family of models, each representing the system from a specific perspective and at a specific level of abstraction. Thus, working with models by means of refinement and transformation provides traceability between elements in different models.

The most important realization of MDD is definitely OMG's Model-Driven Architecture (MDA) [99]. The MDA approach comprises the creation of a "Platform Independent Model" (PIM), which is based on a suitable UML profile and represents business functionality and behavior and, subsequently, the semi-automatic or fully automatic transformation of the PIM into a "Platform Specific Model" (PSM). In the next step, code can then be generated from the PSM.

MDD brings a number of advantages. First, platform independent models hold business semantics and functionality. Second, higher level of abstraction reduces custom code quantity and complexity. Developers only add code to specialize operations, rules, and constraints. Third, platform specific design and implementation models are more precise. Fourth, small amounts of metadata replace large amounts of custom code; the generators are capable of transforming design, implementation and deployment elements into code. Fifth, the change will be in system or business configuration instead of writing or editing custom code.

On the other hand, there is a number of critics for MDD approach (the discussion has been taken from [59]): it is considered to be an inadequate starting basis for automatic code generation due to expressional weaknesses of UML and other existing modeling languages. Secondly, similar to source code, the models have to be verified when they are built, transformed, and used for code generation. Currently, such a "model compiler" does not exist yet, and because of the resulting necessity to revise and extend the generated code the desired maximum degree of reuse is not yet achieved at all. This deficiency can be traced back to the semi-formal nature of UML and shortcomings with respect to modeling dynamic behavior.

Another open problem with MDD is the level of abstraction of the models. Depending on the focus of the approach, it ranges from concrete and fine-grained models to very abstract models that let business experts to build models. Finally, the problem of working with large models for practical cases is still a major problem. It is inevitable to describe a system by several models presenting different views and to integrate them. However, a thorough analysis of how such a "super-model" can be created in a generic way does not exist [59].

2.5 Service-Oriented Computing

Service-Oriented Computing (SOC) is a new computing paradigm that takes services as basic elements. SOC relies on Service-Oriented Architecture (SOA) when constituting the service model. Basic tenets of SOC are loosely coupled asynchronous interactions on the basis of open standards to support complex business processes and transactions as reusable and accessible services, in contrast to tightly integrated monolithic applications [107]. SOC will be presented a little bit in detail here since it offers higher reuse potential and the proposed approach in this thesis relies on SOA as a paradigm for composition.

The constituents of SOA can be providers (basic service providers and aggregators), consumers (service aggregators and end users), and brokers (middleware and registries). Providers do advertise their services to registries and consumers query registries in order to discover required services that satisfy their goals.

Adapted from [107], the crosscutting concerns of SOA can be described at three different service levels:

- *Basic level* includes service description portions (capability, interface, behavior, Quality-of-Service – QoS) and basic operations on services (publication, discovery, selection, binding, and invocation) for offering reusable, adaptable and context-aware services to conform a constructable model.
- *Composite level* includes coordination activity (orchestration of services), conformance (integrity insurance of interfaces), monitoring, QoS for offering static/dynamically composable, verified with regard to quality concerns, and seamlessly integrated services to conform a composable model.
- *Managed level* includes operations (providing control and feedback) and market considerations for offering satisfied regarding market needs, correlated and controllable services to conform a canonical model.

The fundamental challenges of SOC are finding the effective and efficient ways of service description, discovery, selection, composition, monitoring, and integration while focusing on semantics those point out intelligent, dynamically adaptive, and context sensitive services.

In the context of service composition challenge, the definition of service component adds an abstraction layer to facilitate the representation of modularized service based applications to overcome complexity. Service Component Architecture (SCA[1]) emerges with a set of specifications describing a model for service component as a cohesive and conceptual module which includes services assembled by wiring of service-oriented interfaces and orchestrated according to stated business logic. SCA can be coupled with Service Data Objects (SDO[2]) to provide uniform representation of business data for accessing the messages that arrive at or are sent from components.

Internet standards assist realizing SOA with Web services through exposing them as services that can be described, advertised, discovered, and interoperated [26]. Web services can be described using Web Service Description Language (WSDL[3]), which

[1] OSOA, http://www.osoa.org/display/Main/Service+Component+Architecture+Home

[2] OSOA, http://www.osoa.org/display/Main/Service+Data+Objects+Home

[3] WSDL Ver1.1, http://www.w3.org/TR/wsdl

defines operations along with input/output messages and data residing in messages. The interaction between services can be achieved by an XML document whose schema is specified by Simple Object Access Protocol (SOAP[4]) using HTTP at transportation layer. The universally accepted standard to facilitate the discovery of Web services is Universal Description, Discovery, and Integration (UDDI[5]).

Composite services have two views complementing each other, namely, orchestration and choreography. Orchestration of Web services enables coordination of services by assigning an orchestrator, which is a central manager responsible for invoking and combining subactivities. However, Web service choreography defines inter and intra collaboration of each service to realize the system target goal without a central mechanism. But any of the service composition language is sufficient to represent business agreement support, which defines the contract between two parties on QoS [26]. Further details on choreography have been discussed in Section 3.5.1.

SOA and Business Process Management (BPM) are two key technologies for Service-Oriented Computing. BPM involves a control mechanism for defining, altering, orchestrating, executing, and monitoring business processes taking business rules into account. BPM defines behavioral roles of business processes, which are seen as assembly of activities realized through workflow and business rules with the human intervention.

Within SOC, business processes act as a conceptual player, whereas services spread over logical layer of the picture. When appropriately represented and put into development, business processes can provide the application-wide glue in composing Web services. Without such processes, the SOA cannot account for the sequencing of the service activations. The new trend currently points out networks of orchestration for collaborating different enterprise applications within and across organizational boundaries; context adaptive, ambient intelligence type services are spread over the network and are accessed potentially from any device and any location [30]. The notations for business processes modeling, visualizing, and execution have been discussed in Section 3.5.1 during explanation of the choreography of domains.

SOC brings the services as flexible abstractions encapsulating piece of software (algorithm, computation, etc.) which can be reused across different compositions to form

[4] SOAP Ver1.2 Working draft, http://www.w3.org/TR/soap12-part0/

[5] http://www.uddi.org/

higher level abstractions (services). Therefore, either created brand new or wrapped an already existing computation, services embody a high potential for reuse. Although great achievements done, there is still a huge research agenda on service-oriented architecture and engineering, enabling technologies, methodologies, programming tools, management tools, the economic models, and on reducing the complexity [123].

The model presented in this thesis has been collaboratively applied on legacy migration to service-oriented computing [30, 31]. Section 6.7 includes the discussion of how it can help in migration to service-orientation.

2.6 Architecture-Based Development

Although there are many definitions of software architecture, generally accepted definition is "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" [13]. An architecture defines the rationale behind the components and the structure in relation to system stakeholders' requirement statements. Software architecture documentation facilitates communication between stakeholders, identifies early decisions about high-level design, and allows reuse of design components and patterns [13].

Garlan defines six aspects of software development that software architecture can have significant impact [57]: it simplifies the "understanding of large systems" by presenting them at a level of abstraction at which a system's high-level design are easily comprehensible. Second, architectural descriptions support "reuse" at multiple levels. Third, an architectural description provides a blueprint for "construction" by indicating the major components and dependencies between them. Fourth, it can expose the dimensions along which a system is expected to "evolve" (by explicitly defining the "load-bearing walls"). Fifth, it provides a basis for the analysis of the system's dependency, consistency and conformance. Finally, considering a viable software architecture as a key milestone in an industrial software development process improves the "management" of the project, understanding of requirements, implementation strategies, and potential risks.

Software development organizations that use architecture as a fundamental part of their way of doing business often define an Architecture-Based Development (ABD) process. Bass and Kazman describe an architecture-based development process including elicitation of architectural requirements, design, documentation, analysis, realization and maintenance of software architectures [12]. The architecture, itself, is the major reusable asset, and it is a blueprint for all activities in the development life cycle.

Many architecture-centric analysis and design methods have been created in the last decade. Starting with Software Architecture Analysis Method (SAAM) [78], the fundamental ones are Architecture Tradeoff Analysis Method (ATAM) [79, 80], Quality Attribute Workshop (QAW) [10], Cost-Benefit Analysis Method (CBAM) [77], Active Reviews for Intermediate Designs (ARID) [39], and Attribute-Driven Design (ADD) [13]. These architecture-centric methods are scenario-driven; directed by operationalized quality attribute models; focus on documenting the rationale behind the decisions made; and involve stakeholders so that multiple views of quality are elicited, prioritized, and embodied in the architecture [81].

These architecture-centric methods can influence a wide variety of activities throughout the Software Development Life Cycle (SDLC). As these methods have taken place as standalone methods, Kazman et al. [81] link these methods with the SDLC, including all the steps of understanding of business needs and constraints, elicitation and collection of requirements, architecture design, detailed design, implementation, testing, deployment, and maintenance.

In [117], Shaw and Clements analyze two decades of software architecture research by examining the maturation of the software architecture research area by tracing the evolution of research questions and results through their maturation cycle. They show how early qualitative results set the stage for later precision, formality, and automation, how results have built up over time, and how the research results have moved into practice.

Further details, such as Architecture Description Languages (ADLs), quality attributes, etc., have been discussed in relevant sections of Chapter 4 during the presentation of reference architecture modeling approach.

2.7 Asset-Based Development

Asset-Based Development organizes the software-related investments, requirements, models, code, tests, and deployment scripts to be used for future software project activities [88]. The processes, standards, tools and assets are four major constituents of Asset-Based Development. The following asset definition has been quoted from Larsen [88]:

> An asset is a collection of artifacts that provides a solution to a problem. An asset has instructions on how it should be used and is reusable in one or more contexts, such as a development or a runtime context. It may also be extended and customized through variability points.

As it is clearly indicated, this broad definition of asset includes any piece of artifact (e.g. models, requirements, tests, plans, binaries, etc.), those that are not executable and those that are useful to personnel in different roles and those that are relevant to different points in the development life cycle. Typically, the life cycle of assets includes the following major workflows: asset identification, production, management and consumption. A model is first identified as a candidate asset and then produced into a reusable asset for a specified context. It is then reviewed, the version is updated, and it is published as part of asset management. Finally, the model is searched, browsed, reused, and rated as part of asset consumption.

Asset-Based Development can be considered as a sub-methodology in the software development process. Though being not a complete software development process, asset-based development is a set of processes, activities and standards that facilitate the reuse of assets. Asset-based development is architecture centric [112].

2.8 Software Product Lines

A *Software Product Line* (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfies 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 [38]. A product line's scope is a description of the products that constitute the product line or what the product line is capable of producing. Within that scope, the disciplined reuse of core assets, such as requirements, designs, test cases, and other software development artifacts greatly reduces the cost of development.

The key objectives of SPLs are to capitalize on commonality and manage variation thus reduce the time, effort, cost, and complexity of creating and maintaining a different product line of similar software systems. Therefore, with the disciplined reuse of core assets and commonalities, SPLs can address problems such as dissatisfaction with current project performance, reduce cost and schedule, decrease complexity of managing and maintaining product variants, and quickly respond to customer / marketplace demands.

The key component enabling the effective resolution of these problems is the use of a product line architecture that allows an organization to identify and reuse software artifacts for the efficient creation of products sharing some commonality, but varying in known and managed ways. The architecture, in a sense, is the glue that holds the product line together [136].

The operation of a product line involves core asset development and product development using the core assets, both under sponsorship of technical and organizational management. In this sense, a SPL requires three essential activities [103]:

- The Core Asset Development is the ongoing activity of developing core asset base of product line. Its outputs are the core assets used in the family of products, and a production plan that tells how to use or tailor the core assets to produce a product.
- The Product Development is the engineering activity of turning out products using the core assets as prescribed by the production plan.
- The Management is the activities of technical and organizational management, without which the product line eventually will collapse.

Core asset development and product development from the core assets can occur in either order: new products are built from core assets, or core assets are extracted from existing products. There is a strong feedback loop between the core assets and the products. Core assets are refreshed as new products are developed. Use of core assets is tracked, and the results are fed back to the core asset development activity. In addition, value of the core assets is realized through the products that are developed from them. As a result, core assets are made more generic by considering the potential new products in sight. There is a constant need for strong, visionary management to invest resources in developing and sustaining the core assets.

SPLs have gained a lot of attention [103] since they provide the effective reuse of software artifacts; benefit from the validated architecture that is being used by different products; enable to focus on the truly unique aspects of products; facilitate the software integration since working with components whose integration has already been tested; enable effective workforce management based on a proven production plan; and realization of software product quality. An SPL may depend on an asset base ranging from the architecture itself to software components, from the development tools to test cases and test plans.

Domain Engineering and *Application Engineering* are two key complementary processes that are performed in product line approaches [109, 130, 133]. The domain engineering (*design-for-reuse*) is to provide the reusable platform and core assets that are exploited during application engineering when assembling or customizing individual applications. It includes domain analysis, domain design and domain realization and testing. Complementarily, application engineering (*design-with-reuse*) is to develop individual products using the platform and core asset base established in domain engineering. It covers the application requirements engineering, application design, application realization and testing. Further details of software product line approach have been discussed in relevant sections of the text.

2.9 Feature-Based Approaches

Feature-based approaches has been used extensively in domain engineering of software product lines to capture the common and variable parts of a family of similar products [45, 64, 75]. The use of features in domain analysis helped by providing a common vocabulary among different stakeholders, identifying and expressing the variability/-commonality of different products.

Features are any prominent and distinctive concepts or characteristics that are visible to various stakeholders [93]. It can be a structural property, the components of the designed object, a configuration, a set of relationships, a behavior, a function, or a property of a behavior or of a function [25]. In other words, it is an elaboration or augmentation of an entity that introduces a new service, capability or relation [15].

A feature model should represent the requirements of a product, its behavior, the quality attributes, and constraints that need to be satisfied. The most common representation of features is with FODA-style feature diagrams [74]. The notation for feature diagrams has been depicted in Figure 3.9 in Section 3.4.

Feature-Oriented Domain Analysis (FODA) [74] has been proposed as a conceptual model to express the business domain in terms of features. Later, Feature-Oriented Reuse Method (FORM) [75] has extended FODA to the software design phase and prescribes how the feature model is used to develop domain architectures and components for reuse.

Lee et al. [93] describe the concepts and guidelines for feature modeling; it starts from domain planning and continues with feature identification, categorization, organization and finally refinement with practical guidance. However, unless applied cautiously, even such propositions might direct its practitioners to monolithic and complex feature models with crosscutting relations, which may then constrain reuse maximization [17, 27]. Further discussion of feature-oriented requirements engineering has been discussed in Section 3.4.

2.10 Software Factories

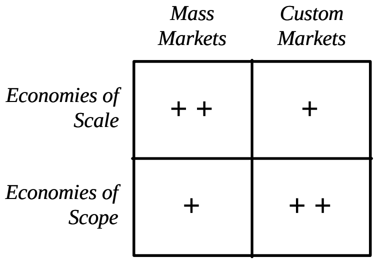
Mainly for past two decades, software industry has demanded personal productivity and now it turns its vision to the technologies that automate business processes. As the industry matures, businesses look for much richer functionalities and quicker response times. Accordingly, software industry should surpass the techniques that brought it to this point, and embrace the industrialization best practices achieved by manufacturing. These include product assembly from components, reducing labor-intensive tasks with automation, setting up the software product lines and supply chains, formalizing the interfaces, and standardizing architectures and processes. In short, such a vision is known to be the "software factory" approach.

The concept of SPL has been extended to define Software Factories that configure 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 [63].

2.10.1 Economies of Scale and Scope

In comparison to the other industries, the distinction between the economies of scale and economies of scope is critical. Economies of scale arise in the production of multiple implementations of a single design, while economies of scope arise in the production of multiple designs and their initial implementations [62]. Economies of scope arise by using the same styles, patterns and processes in development of multiple related designs, and again using the same languages, libraries and tools in development of their initial implementations [45]. Second distinction is the mass markets where the same product can be sold many times, and the custom markets where each product is unique [62].

In custom markets, economies of scope can be realized in software production, as in commercial construction, through systematic reuse, by using the same assets to develop multiple unique but similar products (similar to the construction of bridges or skyscrapers) [62]. The added-value of realizing economies of scale and scope in mass/custom custom markets has been depicted in Figure 2.1. Typical situation that software industry is facing is the realization of economies of scope in custom markets.



	Mass Markets	*Custom Markets*
Economies of Scale	+ +	+
Economies of Scope	+	+ +

Figure 2.1: Economies of scale and scope in mass and custom markets

2.10.2 Constituents of Software Factories

The vision of software factories requires the following steps to be undertaken in methods, tools and economics of software development [62]:

- *Development by Assembly*: Most of the development will be component assembly, involving customization, adaptation and extension.

- *Software Supply Chains*: Supply chains will emerge to create, package, consume and assemble the components specified using standard specification formats, developed by using standard techniques, methods, tools and processes.

- *Relationship Management*: Managing relationships between suppliers and consumers will increase the role of requirements engineering from the product delivery to warranty periods.

- *Domain Specific Assets*: Organizations with domain knowledge will be key players of software development and they can encapsulate and sell their knowledge as reusable assets. The tools will use abstractions and appropriate best practices encoded as domain specific languages, patterns and frameworks.

- *Mass Customization*: This is a long-term vision that can be realized after the wide adoption of software factories. Mass customization requires a value chain that integrates processes like customer relationship management, demand management, product definition, product design, product assembly and supply chain management.

- *Organizational Change*: The stakeholders will face with a new world at every phase of software development, acquisition and usage. The quicker the organizational change is the higher the benefits will appear.

The early projects for the industrialization of software development had come in late 80's like European [116], Japanese [2], or Brazilian [120] models. However, these attempts were too early to be successful without the help of contemporary research in systems modeling and software reuse.

The results of research on component orientation, generative approaches, and software product lines have guided the concept of Software Factories. The three axes of critical innovations for software factories have been depicted in Figure 2.2. The figure and discussion have been taken from [62].

Three axes are *Abstraction*, *Granularity* and *Specificity*. The critical innovations in abstraction are the development of model-driven techniques for model construction and transformation and abstraction provided by Domain Specific Languages (DSLs). In terms of granularity, specification of components with interface specifications (hidden implementations) and their composition [50] and later orchestration of coarse-grained Web services increased the size of software constructs used as vehicles of abstraction. Finally, specificity defines the scope of reuse. The value of abstraction increases with its specificity to some problem domain [70]. More specific abstractions can be used in

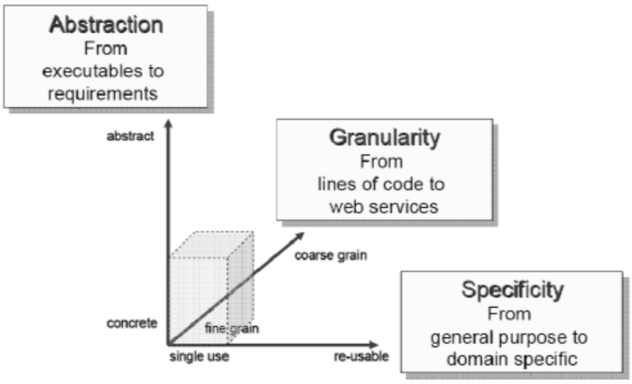


Figure 2.2: Three axes of critical innovations for software factories (from [62])

fewer products, but they contribute more to their development. On the other hand, more general abstractions can be used in more products, but they contribute less to their development. Higher levels of specificity allow more systematic reuse [62].

Software factories significantly increase the level of automation in application development. The approach recognizes that domain knowledge may exist at different maturity levels and thus a wide range of concepts, such as patterns, architectures, frameworks, components, aspects, and domain specific languages, etc., may be required for adequately packaging the knowledge as reusable assets. It does not use the Unified Modeling Language (UML), a general purpose modeling language which defines too many different and incompatible ways to describe abstractions, without defining enough semantics to make any of them usable for actual development [62].

Software factories make use of models based on highly tuned DSLs and XML as source artifacts, to capture life cycle metadata, and to support high fidelity model transformation, code generation and other forms of automation. DSLs are focused and specific to a domain and describes the concepts that a new framework offers.

In [63], a *Software Schema* is defined which describes the set of specifications that must be developed to produce a software product; and a software scheme corresponds to the product line scope. Schema is developed by the product line designer. *Software Template*, on the other hand, is the combination of a software schema for a product family, the processes for capturing and using the information it describes, and the tools used to automate that process. Within a software template, there exists patterns, models, frameworks and tools. Once a software template is loaded into an Integrated

Development Environment (IDE) then it is named as a software factory for producing members of the family.

Lenz and Wienands [94] present a software factory implementation in .NET environment and guides through a practical case study of building a Software Factory. The example covers major constituents of software factories: software product lines, architectural frameworks, model-driven development, and guidance in context.

Kulkarni et al. [84] propose a model-driven software factory for enterprise business application product family. They have enlarged their vision using multi-dimensional separation of concerns focusing on the organizational issues of software development. They reported that aspect-oriented restructuring has enabled them to organize the development team along two independent streams namely technology platform experts and design experts. Separation of design strategies has enabled leaner technology platform teams. However, they have employed a completely model-driven way where the approach proposed in this thesis study differs by being more architecture-driven.

Regio and Greenfield [113] report their experience gathered in designing and implementing a software factory for healthcare systems based on Health Level Seven (HL7) standard using the factory vision given in [62, 63]. Their experience in developing a factory for HL7 collaboration ports has shown that it is crucial to define better frameworks, tools and processes to specify the factory schema, manage factory configuration in a flexible and extensible way, and better understand how/when domain specific languages should be used. They also point out that toolkit support in software factory infrastructure will be helpful.

Frankel, in his idea paper [55], defines a Business Process Platform (BPP) on top of traditional technical software platform. This platform contains the frameworks of executable services and business process components. Users of the platform compose specialized applications that support custom services, business processes, and analytics. A further proposal was Business Process Factories that enable the definition of individual BPPs. However, the idea paper just elaborates the initial concepts and does not report any finding. In comparison to this study, the proposed model in the next chapter has similar abstractions with concrete baseline and results are investigated with case studies.

MDSoFa [87], standing for Model-Driven Software Factory, is a software factory environment based on the generative technique to produce languages, frameworks and tools in series. Four core technologies participate in the MDSoFa foundation. (i) Languages are described with MOF-level meta models, and a mapping notation allows expressing correspondence between languages, e.g. MOF to UML, DSL to DSL mappings. (ii) A rule-based language allows expressing patterns. (iii) A template-based language, for the rule implementation part, allows code expansion of template-based expressions with language, mapping, and rule information. (iv) In order to avoid monolithic production, production results are separated by concerns, e.g. separating model management from model checking concern.

Neema et al. [102] argue that it is essential to incorporate analytical techniques in software factories to assist with the architectural decisions. They have illustrated how architectural analysis can be done in a software factory setting, through an example. They have reported that analysis, especially quantitative analysis should be part of a software factory in order to validate architectural decisions in the design. The results of their study is in favor of the proposed approach here that incorporates the reference architecture modeling with the product line asset modeling in tandem.

While the ideas behind software factories are platform independent, Microsoft has been developing a family of Domain Specific Modeling (DSM) tools and processes that facilitate designing and building DSLs and serve as a foundation. They provide modeling environments (with tools, frameworks, patterns and processes) with the vision of constructing an integrated environment for software factories [41]. DSM mainly aims to raise the level of abstraction by specifying the solution in a design language that directly uses concepts from a specific problem domain. The final product is generated from these high-level specifications in a chosen programming language. The specification and generation are both domain specific. The variations can be managed at the model, generator or framework level [52].

An organization will only obtain the full benefit of reuse if a formal reuse program is employed and subject to quality control through formal planning and continuous improvement [114]. Matsumoto, in his talk on SPLC 2007 [98], reveals the management aspects of software factories, within the scope of his Toshiba experience, with particular emphasis on organizational management, process/project management, software engineering measurement and evaluation. He has reported that within 6 years after setting up the factory, the productivity increased more than 50% as per factory member and the reuse ratio increased from 13% to 48%. Furthermore, the number of faults has fallen from 7 to 0.2 as per K-SLOC (Kilo-Source-Line-of-Code).

Consequently, the concept of software factories has been investigated by several research groups. However, there has been no mutual understanding yet to generalize the establishment of Software Factories as the way manufacturing industry has been doing. Although the vision is not new and addressed by many researchers and industry experts, it still needs formal models and practical assistance for establishing them across different business domains.

CHAPTER 3

THE APPROACH: SOFTWARE FACTORY AUTOMATION

The main goal of this chapter is to present the high-level approach to the key research problem:

> How can we incorporate domain specific abstractions to improve systematic reuse of software assets on the basis of an industrialization model in order to enhance the software development productivity and product quality while reducing the per product development and maintenance costs?

This chapter proposes a software industrialization model, namely Software Factory Automation (SFA), based on Domain Specific Kit (DSK) conceptualization. Using the Domain Specific Kit abstraction, the reference architecture modeling and accompanied feature-based software asset modeling roadmaps have been charted.

The key idea is to specify asset models with more reusable Domain Specific Artifacts abstracted by DSKs (composed of a domain specific language, engine, and the toolset). This approach encapsulates correlated features (hence artifacts) within more cohesive asset models and composes them through a choreography engine, which is also driven by the same software asset meta-model used for DSKs. Defining domain specific languages under the supremacy of a meta-model enables the modeling and development of artifacts in isolation and facilitates the composition of Domain Specific Engines by means of a choreography engine.

3.1 Overview

Major constituents of the approach have been given in Figure 3.1, and briefly described as follows:

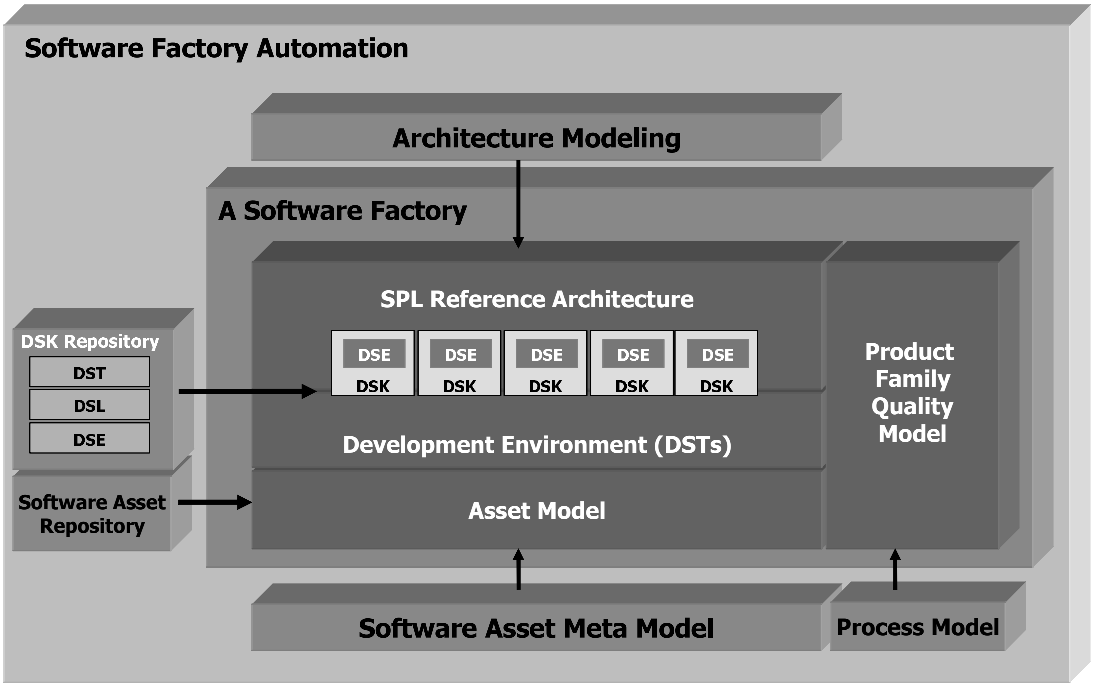

Software Factory Automation
Architecture Modeling
A Software Factory
SPL Reference Architecture
DSE
DSK
DSE
DSK
DSE
DSK
DSE
DSK
DSE
DSK
Product Family Quality Model
DSK Repository
DST
DSL
DSE
Development Environment (DSTs)
Software Asset Repository
Asset Model
Software Asset Meta Model
Process Model

Figure 3.1: The overview of the SFA approach

- **Domain Specific Kit**: The term Domain Specific Kit (DSK) denotes collectively a Domain Specific Language (DSL) to specify the artifacts, a Domain Specific Engine (DSE) to execute the artifacts, and a Domain Specific Toolset (DST) to develop and administer the artifacts of the domain. DSK is the core abstraction for the proposed model.

- **Reference Architecture Modeling**: This is the generic modeling roadmap to construct reference architecture for the target business domain. It employs the symmetric alignment technique that is explained in detail in Chapter 4, for modeling the SPL reference architecture, which correlates the quality targets and architectural aspects to running DSEs and choreography rules.

- **SPL Reference Architecture**: SPL reference architecture is the generalized architecture of a product family, and it defines the infrastructure common to end products and interfaces of components that will be included in the end products [56]. It is constructed using the reference architecture modeling technique.

- **Software Asset Meta Model**: Asset Meta Model (AMM) is an XML-based specification language to define a software product line and its asset model. It defines the global vocabulary, e.g. artifact types, variability points, choreography rules, context, etc., of product line modeling.

- **Asset Modeling Language**: Asset Modeling Language (AML) of a product family (i.e. distinct SPL) is derived from AMM and used to define the assets of the product line. An AML is particular to a distinct product family.

- **Software Asset Repository**: This is a global repository to store the reusable software assets. Later, an asset can be searched and reused in multiple product lines.

- **Development Environment (DSTs)** : Development Environment is the collection of dedicated Domain Specific Toolsets (DSTs) that are specific to development and administration of artifact types included in the domain.

- **Product Family Quality Model**: As stated in Section 2.8, software product line approach requires effective management activities for the asset development, product construction, and product line management. SFA approach provides a skeleton *Process Model* that can be customized to particular domains for different product families.

The rest of this chapter discusses the high-level properties and structures of the constituents of SFA approach. Before that, the main stakeholders are described in Table 3.1 to highlight the major roles.

Table 3.1: Main stakeholders in SFA approach

Stakeholder	**Responsibility**
SFA Engineer	Development of Software Factory Automation model.
SPL Engineer	Design of a specific SPL, DSKs, and its SFA-based asset model, as well as management of product line and its assets, which is in charge of product line management and core asset development compliant with general SPL model.
Product Engineer	Management of a specific product in an individual product line.
Product Line Staff	People responsible for all other product line activities including business domain modeling, configuration and release management, test management and asset tailoring for a specific product, etc.

3.2 The Concept of Domain Specific Kit (DSK)

Software Factory Automation is inspired by the way other industries have been realizing factory automation for decades. Industrial Factory Automation utilizes the concept of "Programmable Logic Controllers (PLCs)" to facilitate the production of domain specific artifacts in isolated units. PLCs may also take place in moving assembly lines to unify the production process. Factory automation in milk factories, for example, bridges diverse units of pasteurization, bottling, bottle tapping, and packaging through moving assembly lines, all designed by the use of PLCs. Bottle and bottle tap in this example are both domain specific artifacts that can be reused in the production of various milk products such as regular, skimmed or semi-skimmed, or even in bottling of straight or sparkling water, not just milk.

PLCs improve the reusability of domain specific artifacts with a consistent design in mind: PLC has a Programmable Processor (PP) to be programmed with a Computer Language (CL) through a Development Environment (DE). So does DSK abstraction of SFA model: the DSK has a Domain Specific Engine corresponding to PP, a Domain Specific Language corresponding to CL, and a Domain Specific Toolset corresponding to DE of PLC concept (See Figure 3.2). As the way PLCs are used for abstracting a wide range of functionalities like basic relay control or motion control, DSKs in SFA approach can be designed specifically to abstract certain things such as screen/report rendering or business rule execution in software factories.

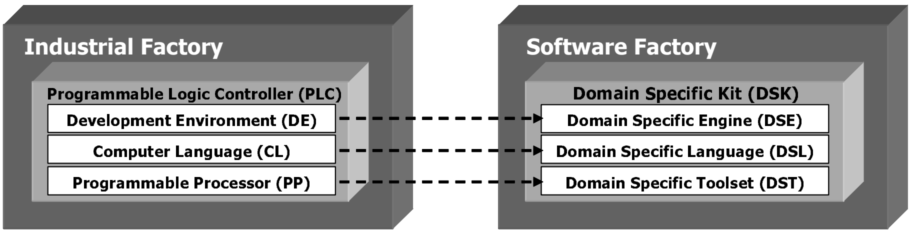

Industrial Factory
Programmable Logic Controller (PLC)
Development Environment (DE)
Computer Language (CL)
Programmable Processor (PP)
Software Factory
Domain Specific Kit (DSK)
Domain Specific Engine (DSE)
Domain Specific Language (DSL)
Domain Specific Toolset (DST)

Figure 3.2: Software Factory Automation and PLC analogy

DSK has further commonalities with PLC. Specifically, PLC is typically a Reduced Instruction Set Computer (RISC) based and contains a variable number of I/O ports. So does the DSK model. DSK has logical I/O ports to have seamless connection with each other for context propagation. DSLs are kept in higher-level abstractions so that the design and transformation can be easily accomplished as in the concept of RISC in PLCs. Moreover, DSE has inherent execution monitoring features in design as PP has extensions for Supervisory Control and Data Acquisition (SCADA) monitoring.

Basic terminology of SFA is given in Table 3.2, and the interrelation between the basic blocks are depicted in Figure 3.3.

Table 3.2: The terminology of DSK

Domain Specific Language (DSL)	A language dedicated to a particular domain or problem with appropriate built-in abstractions and notations
Domain Specific Engine (DSE)	An engine particularly designed and tailored to execute a specific DSL
Domain Specific Toolset (DST)	An environment to design, develop, and manage software artifacts of a specific DSL
Domain Specific Kit (DSK)	A composite of a Domain Specific Language (DSL), Engine (DSE) and a Toolset (DST)
Domain Specific Artifact (DSA)	An artifact that is expressed, developed, and executed by a DSL, DST, DSE, respectively
Domain Specific Artifact Type (DSAT)	A DSA type that a certain DSK can express, execute and facilitate the development

In Figure 3.3, DSL, DSE and DST collectively constitute the DSK. A DSK may be used to produce several artifact types (DSAT). Furthermore, artifacts of these types can be developed and contained within reusable software assets. In order to achieve that, DSK design aims to maximize the reuse of DSAs like screen and report layouts or certain business rules.

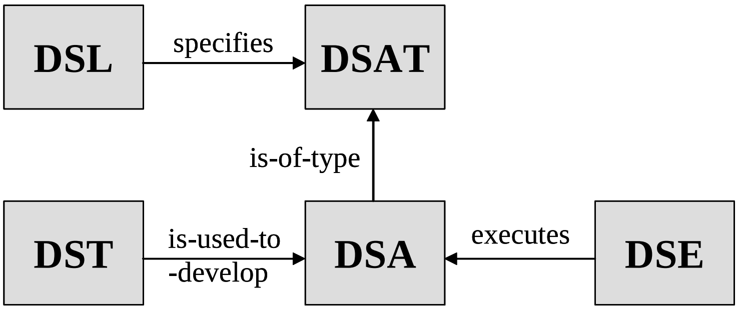


Figure 3.3: Conceptual model of DSK

3.2.1 Fundamentals of DSKs

Before giving some examples of DSKs, the fundamental properties have been stated as follows [7, 34]:

- A DSK provides higher level of abstraction for artifact definitions by means of Domain Specific Languages.

- A Domain Specific Engine of a DSK provides higher level of execution environments based on reference architectures.

- A set of Domain Specific Artifacts can be defined by using the DSL of a DSK.

- The artifacts are defined by declarative approaches.

- DSKs are lightweight and loosely coupled with each other; so their artifacts (DSAs) can be designed to be composed with others.

- DSAs are context-aware, so that they can communicate and be assembled using a common context.

- A DSE can take part in choreography via different protocols under the domain specific rules and constraints.

- DSKs are not particular to a product family, they can be reused across different product lines.

3.2.2 Examples of DSKs

In this subsection, several examples will be presented to elaborate the concept of DSK.

Relational Database Management Systems (RDBMS):

A tangible example to DSK abstraction is well-known Relational Data Base Management Systems (RDBMS) as shown in Figure 3.4.

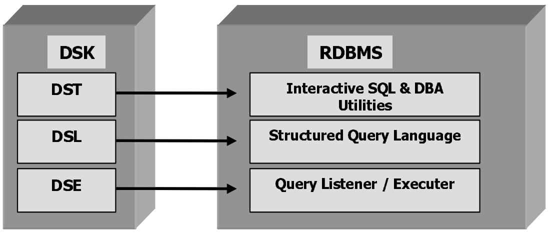

DSK
DST
DSL
DSE
RDBMS
Interactive SQL & DBA Utilities
Structured Query Language
Query Listener / Executer

Figure 3.4: RDBMS as a DSK

RDBMS is a business domain independent technology to be used without extensive programming efforts. SQL is generic enough to be a DSL for data base query language, interpreted and executed by a Query Engine as a Domain Specific Engine, and finally there are lots of tools for interactive SQL use and data base administration, as a Domain Specific Toolset. SQL queries, stored procedures, triggers and DML statements are basic Domain Specific Artifacts of RDBMS. Having such a capable structure, RDBMS is versatile enough to take part in choreography via different protocols such as ODBC, JDBC, ADO, DAO, etc.

Rich Internet Application (RIA) Framework:

Aurora RIA framework [8] is a business domain independent XML-based technology used for power screen design in Internet applications. A DSK abstraction of RIA framework has been depicted in Figure 3.5.

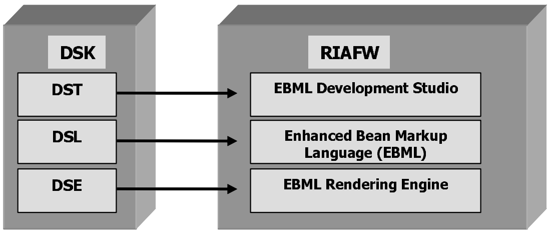


Figure 3.5: RIA framework as a DSK

EBML (Enhanced-Bean Markup Language), as a DSL, is a generic markup language to describe the structure of user screens and their behaviors. ERE (EBML Rendering Engine) is a Domain Specific Engine to interpret the EBML, which renders and manages the user screens. Finally, EDS (EBML Development Studio), as a DST, is a complete development and test tool for the user interface developers. Pages, regions, and popups are the artifact types that can be defined by EBML. RIA framework can take part in choreography via different protocols such as HTTP or SOAP.

Business Rules Management System (BRMS):

A BRMS enables the segregation of business rules from the application where they crosscut almost every tier from content to service [33]. Managing business rules on its own provides a clear separation of a crosscutting concern. RUMBA [33], for instance, provides a lightweight framework for dynamic integration of business rules with other business processes or business services using several architectural patterns. A DSK abstraction of RUMBA BRMS has been depicted in Figure 3.6.

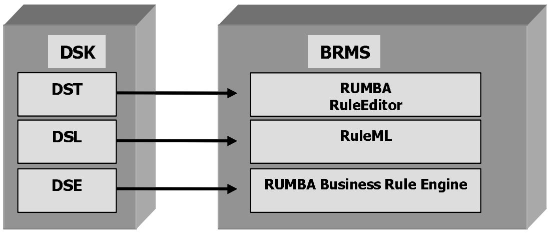


Figure 3.6: BRMS as a DSK

RuleML[1] is a kind of DSL to define business rules as independent artifacts. It has a RuleEditor as a DST, and a corresponding runtime engine (DSE) for rule execution. Rule and composite-rule are the artifact types. RUMBA runtime engine can take part in business choreography with API-based, service-based, or other type of interfaces.

The concept of "Domain Specific Kits" was first introduced by Griss and Wentzel within the context of "flexible software factories" [65]. Hybrid kits, defined in [65], consist of *(i) reusable components, (ii) carefully designed framework which captures the architecture or shape of a family of related products, and (iii) some form of "glue code" which could be either a typical language such as C or C++ or some problem-oriented language specific to the application domain.* They anticipate the use of both generative and compositional approaches. Consequently, they are very heavy-weighted structures, and they are more similar to today's reference architectures or reference models.

In this study, the concept of DSKs has been reshaped within the context of "Software Factory Automation (SFA)" idea introduced in [7]. The concept of DSKs here diverges from the Griss's definition; and the SFA model attributes a new content to the old term. The new definition is mainly different in the following respects:

- Kits are not specific to a product family.
- Kits can not contain architectures for family of products, instead they are combined to form a reference architecture of a product family.

[1] Rule Markup Initiative. http://www.ruleml.org/

- Artifacts of the kits are symmetrically composable; however the kits can make use of generative approaches internally.
- They make use of declarative choreography language.
- Kits, hence their artifacts, can be reused across different product lines.

It is worth to mention that [132] proposes similar usage of multiple DSLs in partial models within the context of Model-Driven Development. However, their primary motivation is to overcome the difficulties of manageability, readability and understandability of large models describing a complete application.

3.3 Software Factory Automation

Software Factory Automation (SFA) proposes a methodical approach to set up software product lines. Proposed modeling activities constitute the domain engineering phase of software product line setup. Involved activities are feature-oriented requirements engineering, SPL reference architecture modeling, and software asset modeling. The feature-oriented requirements engineering, corresponding to the domain analysis, yields Domain Feature Model (DFM) of the target domain. This is fed into the reference architecture modeling and feature-based software asset modeling phases. These two phases constitute the actual construction phases of a software product line. The overall construction process has been managed through a complete domain engineering life cycle model.

Based on the concept of DSK, as a fundamental building block, a detailed combined conceptual model of major modeling activities, has been presented in Figure 3.7. Although not explicitly shown in the figure, the functional requirements and business domain model (the boxes at the bottom) are expressed as feature diagrams. Therefore, SFA approach implies the use of *Feature-Oriented Requirements Engineering* approach (see Section 3.4). Feature-oriented domain analysis is applied first to discover functional and non-functional requirements of a product family. Hence, business domain is represented in terms of feature models (Domain Feature Model).

Reference Architecture Modeling correlates the architectural aspects and quality attributes of the problem domain to actual components and connectors of the solution domain. This method, known as “Symmetric Alignment” [32], assists the identification

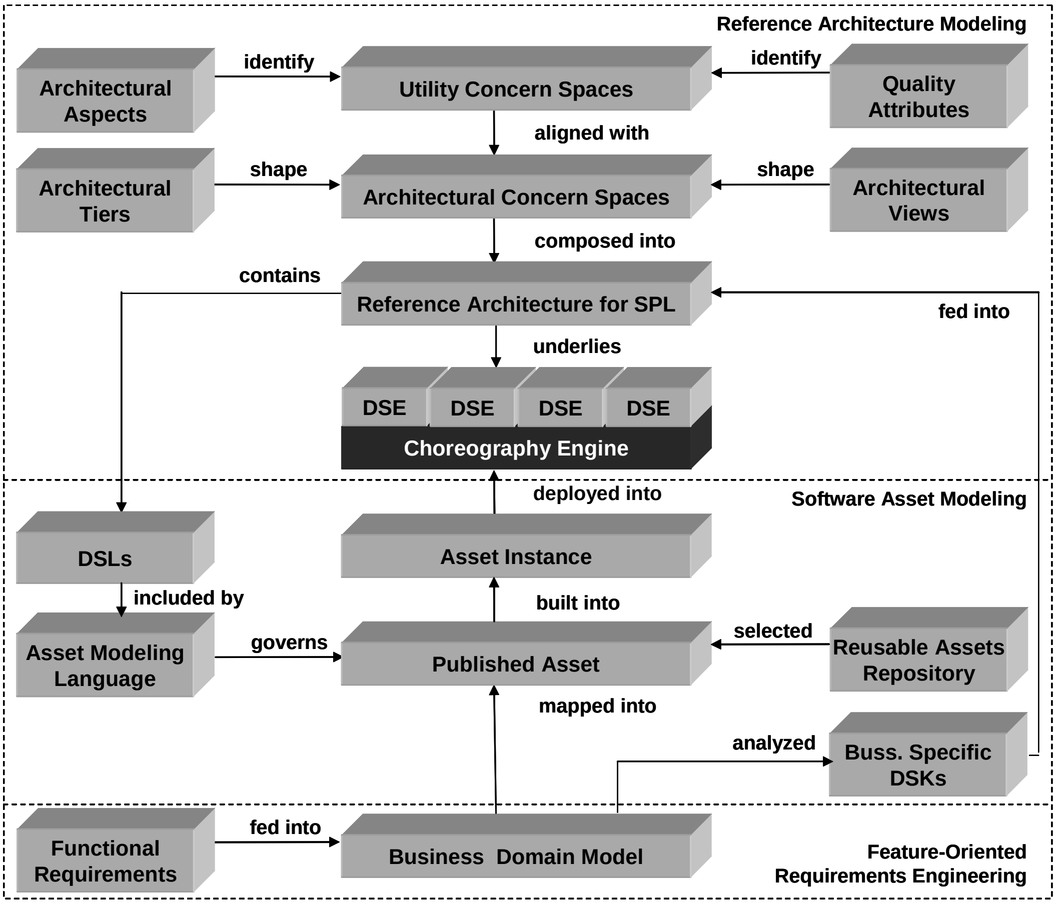


Figure 3.7: Detailed conceptual model of Software Factory Automation

of components (DSEs) and associated connectors (composition of DSEs) in structuring the SPL Reference Architecture. SFA facilitates the communication and coordination of DSEs through a choreography engine.

Complementarily, *Software Asset Modeling* starts with the outputs of feature-oriented domain analysis. Product line software asset modeling uses a feature-based approach to construct the software assets that are collections of domain specific artifacts with variability points. During asset modeling, identified features are mapped onto software assets. Product line assets are defined based on an asset meta-model and are tailored to assemble the products.

There is a two-way interaction between reference architecture modeling and software asset modeling. The former is required once at the beginning and the outputs are used in the product line asset modeling of that specific product family. However, the asset modeling may loop back to reference architecture modeling for extending the selected DSK set.

Collecting the product line requirements, determining the product line scope, establishing the product line platform (the reference architecture), defining the asset model, and identifying the product line core assets constitute the domain engineering model of SFA approach. The study here has been considered as a domain engineering model leveraged by the concept of DSK. A complete life cycle to bridge domain engineering and application engineering within SFA context is needed. However, defining the life cycle processes and an accompanied methodology are beyond the scope of this thesis and left as a future work.

The domain engineering life cycle has to define the technical and organizational management practices for a successful operation of the product line. Regardless of the Software Process Improvement (SPI) standard the organization has to comply with, such as International Standards Organization (ISO) Capability Maturity Model Integration (CMMI), Software Process Improvement and Capability Determination (SPICE) and Allied Quality Assurance Publications (AQAP), it is expected to be transparent to Software Factory Automation. In parallel with this vision, my colleagues at Cybersoft have been developing a separate SPI Hyperframe so-called “Lighthouse” [35] that achieves such a transparency. Lighthouse is an hyperframe for multi-model software process improvement which provides a collaboration and quality management system to improve software processes [35]. This is something like PLCs are configured in factories without affecting the ISO compliance of the actual manufacturing processes.

3.4 Feature-Oriented Requirements Engineering

Requirements Engineering (RE) process for SPL is crucial to understand, identify, and specify the requirements of potential members of the product family and requirements of the domain. Within the context of product lines, these activities are named as “domain analysis”. Domain analysis may be executed in multiple stages during product line setup [85]. Domain analysis forms the basis of domain engineering, which is the process of defining scope, commonality and variability of the product line and their realization in terms of core product line assets. Furthermore, common requirements for the members of product family are themselves valuable core assets that should be managed and maintained systematically. Common requirements make it inexpensive and easy to generate a complete requirements specification and product realization when a new product joins the product family.

An effective product line construction must capture current and future customer needs (product features), convert these needs into product line requirements, and employ these requirements to guide the design and realization of the product line core assets and products [103].

Kuloor and Eberlein [85] present a brief discussion of requirements engineering processes and techniques used in product line approaches. The study also reveals that most of the approaches lack in requirement engineering practices or leave the selection to the product line designer according to the needs of the target domain. The activities for elicitation, documentation, negotiation, validation/verification and management of requirements have to be adapted for product family development.

The main goal of this section is not to propose a requirement engineering or domain analysis model, but rather to set a viewpoint and a common starting point for the rest of the modeling activities. However, developing a requirements engineering model for SFA is one of the future research areas. Meanwhile, the understanding of the approach from requirements to reusable assets has been presented in Figure 3.8.

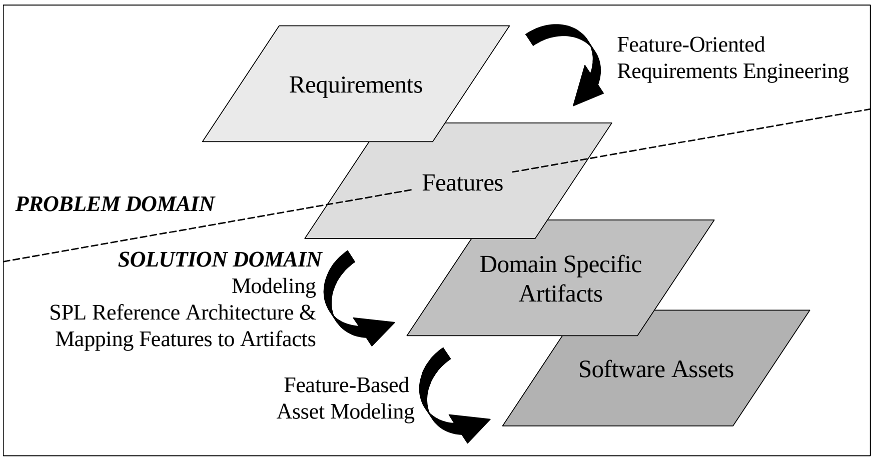


Figure 3.8: Transforming requirements to reusable software assets

Actually, this viewpoint of employing features in domain engineering is quite similar to that of [129] which presents a conceptual basis for feature engineering. The features are life cycle entities to bridge the problem and solution domains and they are meant to be logically modularizing the requirements. Turner et al. [129] argue that features can be used as first-class entities throughout the life cycle. However, the proposed approach here limits the use of features in understanding the problem domain and starting point of the solution domain.

Figure 3.8 implies that the requirement analysis captures the users' needs, relationships and constraints. The requirements are described as functional and non-functional properties; and functional ones can further be decomposed into business rules, business services, business actions, business flows, business constraints, etc. A feature may compose several of these requirements and a requirement may be satisfied by several features [129].

Various domain analysis techniques for modeling requirements have been proposed [74, 133, 54, 42, 109, 97], and it has been studied a lot both in academic and industrial environments. Lee et al. [93] present a broad discussion of domain analysis in general and feature-oriented domain analysis in particular.

Three primary reasons why feature-oriented domain analysis has been used extensively are as follows [93]:

1. It provides a common vocabulary and an effective communication medium among different stakeholders.

2. It is effective in identifying the variability and commonality among the products of the domain.

3. It can provide a basis for developing, parameterizing and configuring reusable assets of the domain.

In the light of this discussion, SFA anticipates the use of feature-oriented requirement engineering model for domain analysis. The proposed approach will be engaged in the feature modeling described in FORM [75]. The following definition has been quoted from [75]:

> "A feature model, including feature definitions and composition rules, describes a domain theory. It not only includes the standard terms/con-

> cepts and their definitions, it also describes how they are related structurally and compositionally."

In FORM, feature models include feature diagrams, composition rules, feature dictionary, and other issues and decisions; and they altogether represent the functional and non-functional requirements of the system.

The most common representation of features is with FODA-style feature diagrams [74]. A feature diagram is a treelike structure where each node represents a feature and each feature may be described by a set of sub-features represented as children nodes. Figure 3.9 depicts the notation to represent the features. They also support notations to distinguish between mandatory and optional features and to express simple constraints on the legal combinations of features. For the rest of this thesis, the feature diagrams will be presented using this notation.

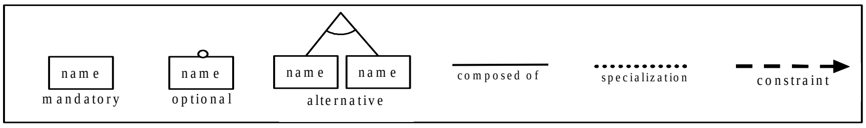


Figure 3.9: Notation of feature diagrams

Recently, feature-oriented approaches have been widely used in different phases of the requirements engineering. Ahn and Chong [1] propose a feature-oriented requirements tracing method consisting of requirements definition, feature modeling, feature prioritization, requirements linking, and traceability links evaluation. Zhang et al. [135] use a feature-oriented approach for analyzing requirement dependencies. Chen et al. [36] propose a semi-automatic approach to construct feature models based on requirements clustering, which automates the activities of feature identification, organization and variability modeling to a great extent. With the automatic support of this approach, high-quality feature models can be constructed in a more effective way.

In a recent study, Mei et al. [100] proposed a feature-oriented approach for modeling and reusing requirements of software product lines. Feature-Oriented Domain Modeling Method (FODM) includes a concrete form of feature model, namely Domain Feature Model (DFM), and a modeling process. Their approach addresses three levels of software requirements: business requirements, user requirements, and function requirements. These levels identify three stakeholders as the sources of features: organization or customers, end users, and system developers, respectively. They have also included a quality analysis section in the DFM and a corresponding step in modeling process to identify and capture quality attributes of the domain.

In [61], González-Baixauli et al. propose a goal-oriented view of features by elaborating the relation among goals/softgoals, features and use cases for modeling requirements. Similar to our approach, they separate the features in two concepts: the goals that model the capability features (general functionality and operations as well as non-functional requirements), and the tasks that model all the other feature types (operating environment, domain technology, and implementation technique).

Streitferdt developed a Family-Oriented Requirements Engineering (FORE) method that extends feature modeling and integrates into a data model, capable of holding all the information acquired within the requirements engineering phase [125]. Dependencies within feature models and between features and further model elements can be modeled with FORE Feature Constraint Language (FCL). FORE proposes a development process for system families and enables the resulting requirement model to be available as XML-Schema (FORE-Data Model).

The output of feature-oriented requirements engineering is expected to reveal the feature models and their descriptions in the form of textual requirements lists. The outcome will provide the functional and non-functional features. [125] and [100] likely provide the expected outcome with different perspectives. In this study, Mei et al.'s terminology [100] will be used, and the domain requirements will be named as *Domain Feature Model (DFM)*. The content of DFM will be a combination of FORM [75] and FODM [100]. A DFM includes:

- feature diagrams,
- composition rules,
- feature dictionary,
- list of requirements (functional and non-functional),
- quality attributes of the domain,
- other issues and decisions.

DFM is fed into reference architecture modeling and software asset modeling activities of the SPL construction. These coupled modeling activities together constitute the domain design in SFA approach. It is envisaged that feature modeling activities and product line modeling activities can be iterative in problem and solution domains, respectively.

As noted earlier, the goal of Section 3.4 is not to propose a requirement engineering methodology, but to describe a minimum set of outcomes expected from this phase to successfully proceed with the next phases.

3.5 Reference Architecture Modeling with DSKs

The reference architecture for an SPL is a generalized architecture of a product family, and it defines the infrastructure common to end products and interfaces of components that will be included in the end products [56].

SFA approach brings a six-step generic modeling roadmap to construct reference architecture for the target domain. It employs the symmetric alignment technique for modeling the SPL reference architecture.

Architectural modeling identifies several concerns in problem domain and associates them with design decisions in solution domain. There is, however, no commonsense on how to localize problem domain concerns and relate them to the solution domain. The constructed approach identifies the problem domain in “utility concern spaces” by correlating the “architectural aspects” and “quality attributes‘”, and the solution domain in “architectural concern spaces” by correlating the “multi-views” and “multi-tiers” of architectures. The symmetric alignment is used here for another correlation of “utility concern spaces” and “architectural concern spaces” instead of mapping architectural concerns of problem domain to design decisions of solution domain. This alignment helps finding architectural components, connectors and properties in the solution domain; hence results in the identification of DSEs to be plugged into the choreography engine. The six-step roadmap has been discussed in Chapter 4.

3.5.1 The Role of Choreography Language and Engine

Architecture modeling is expected to end up with the architectural components (including DSEs), their connectors and the composition context. At the end, the golden principle of separation of concerns for different domains has been achieved. Each concern has been expressed and executed by different DSL and DSE combination like

PLCs are controlling certain concerns in industrial automation. Naturally, every separation should end up with a composition as well. For the composition of domain specific artifacts, expressed in DSL, SFA employs a choreography model (a language and an engine), which relies on SOA as a paradigm for managing resources, describing process steps, and capturing interactions between a service and its environment.

A choreography model describes a collaboration between a collection of services (artifacts of different DSKs in this case) in order to achieve a common goal [11]. It captures the interactions in which the participants engage to achieve this goal and identify the dependencies between these interactions. A choreography captures interactions from a global perspective, meaning that all participating services are treated equally. The interactions include control and data flow dependencies, message correlations, time constraints, transactional dependencies, etc. A choreography does not describe any internal action that occurs within a participating service that does not directly result in an externally visible effect, such as an internal computation or data transformation. A choreography encompasses all the interactions between the participating services that are relevant with respect to the choreography's goal [11].

In addition to the choreography view that is a global perspective, standards for service composition cover two additional viewpoints: first, a behavioral viewpoint captures interactions from the perspective of one of the participants and can be seen as consisting of communication actions performed by the participant. Secondly, the orchestration viewpoint deals with the description of the interactions in which a given service can engage with other services (communication actions), as well as the internal steps between these interactions. Both behavioral and orchestration models are encapsulated by the artifacts of the domain (DSATs) in SFA model. Further discussion of service viewpoints can be found in [48].

Several composition standardization proposals have been put forward over the past years (WSFL[2], XLang[3], BPML[4], WSCL[5], WSCI[6]). The preeminent ones are Web Services Choreography Description Language (WS-CDL) [76] and Web Services Business Process Execution Language (WS-BPEL 2.0) [72]. However, there are significant dif-

[2] Web Services Flow Language, www.ibm.com/developerworks/webservices/library/ws-ref4/

[3] XLang by Microsoft, msdn2.microsoft.com/en-us/library/ms935352.aspx

[4] Business Process Modeling Language, www.bpmi.org

[5] Web Services Conversation Language, www.w3.org/TR/wscl10/

[6] Web Service Choreography Interface, www.w3.org/TR/wsci/

ferences between these two standards. The fundamental one is the fact that WS-BPEL is an orchestration language that implies a centralized control mechanism where as WS-CDL is a choreography language in which the control is shared between domains. In addition, the former is an execution language while the latter is a description language. Therefore, SFA anticipates the use of a choreography language similar to WS-CDL (it will be called as CDL from now on).

In SFA approach, employment of CDL-like choreography language provides a set of critical advantages:

- It provides an unambiguous and type safe language for describing the sequence diagrams.
- The domain specific artifact can be validated statically and at runtime against a choreography description.
- It ensures effective interoperability of domains, which is guaranteed because the domain specific artifacts will have to conform to a common behavioral multi-party contract specified in the CDL.
- It reduces the cost of artifact implementation by ensuring conformance to expected behavior described in the CDL. This, in turn, can be used to guide and structure testing and so reduce the overall time to deployment of an artifact.
- It is possible to use the CDL through a validating design tool.
- It enables to describe multi-party contract (artifact interface) in terms of a global model.
- It is possible to develop and use additional CDL-based tools to generate artifacts' skeletons and test programs based on CDL descriptions.
- It might be possible to use additional CDL-based tools to provide runtime verification of artifacts against their expected behavior as defined in the CDL description.

A CDL specification aims at composing interoperable, peer-to-peer collaborations between any type of party regardless of the supporting platform or programming model used by the implementation of the hosting environment. In SFA approach, CDL defines the choreography of DSAs according to choreography's goal. Since CDL is not an execution language, it can be compiled into an executable language of particular requirements of a domain. DSEs are execution engines for DSAs and they are composed via a choreography engine. Choreography engine requires the separation of concerns across different DSEs and, thus, deferred encapsulation [63] can be achieved through plugging in and out any DSE as needed. This provides an execution model for collaborative and transactional business processes based on a transactional finite-state machine. This is a non-monolithic execution model, and it does not need every sort of detail to be specified at once.

The features mapped into specific DSLs are going to be executed by corresponding DSEs. Therefore, dynamic plugging and context-awareness of DSEs are crucial for the runtime execution model. The choreography engine enables communication and coordination among DSEs. It ensures context management, state coordination, communication, produce/consume messaging, nested processes, distributed transactions, and process-oriented exception handling. Identification of architectural properties during architectural modeling facilitates the definition of SPL contextual information, which contains the stateful/stateless information to connect individual DSEs.

The design and development of SFA choreography language and engine are beyond the scope of this thesis. It has been included here to highlight the role of choreography in SFA approach. However, it has been validated in Chapter 6 that the composition of domain specific artifacts is quite achievable even via a simple composition model based on Composite Application Framework (WS-CAF) [28].

3.6 Software Asset Modeling with DSKs

SFA software asset modeling uses the abstractions provided by Domain Specific Kits. It is the definition of software product line in terms of DSKs, their compositions, their dependencies, and the global contextual information. Asset model provides means to effectively manage the commonality of features and their variations.

The primary input of this step is the DFM produced in the feature-oriented requirements engineering step. DFM is the global view of requirements of a product family.

Five-step feature-based software asset modeling with DSKs, shown in Figure 3.10, maintains two different views: *External View* is specified by Asset Capability Model (ACM), and *Internal View* is represented by Asset Meta Model (AMM). The former is expressed as a feature diagram describing the capabilities of an asset, and the latter is expressed with Asset Modeling Language (AML) for representing domain specific artifacts, contextual properties, and variability points. The approach has been discussed fully in Chapter 5.

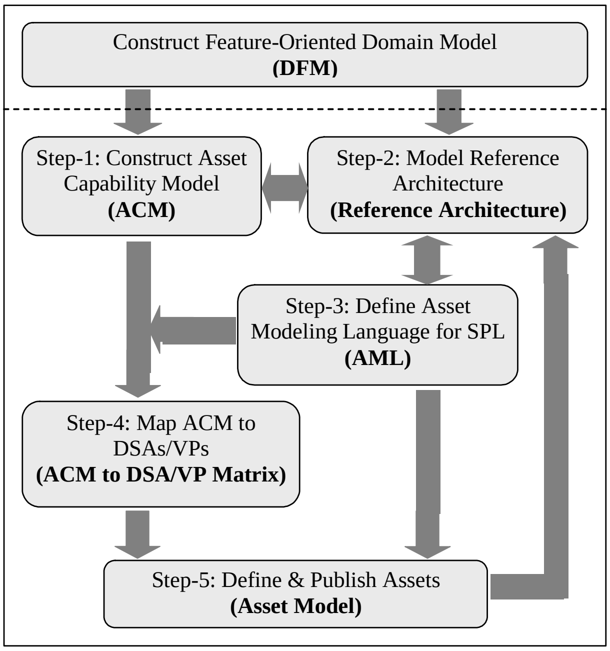

Construct Feature-Oriented Domain Model
(DFM)
Step-1: Construct Asset Capability Model
(ACM)
Step-2: Model Reference Architecture
(Reference Architecture)
Step-3: Define Asset Modeling Language for SPL
(AML)
Step-4: Map ACM to DSAs/VPs
(ACM to DSA/VP Matrix)
Step-5: Define & Publish Assets
(Asset Model)

Figure 3.10: Overview of asset modeling approach

The first step is the description of asset capabilities in terms of feature diagrams exposing the structural, functional and behavioral properties, and constraints on them. Actually, ACM is constructed from DFM by matching the feature model with respect to the core asset base and identifying the boundaries and features of new reusable assets. Step 2 employs the reference architecture modeling introduced in Chapter 4 and yields the SPL reference architecture for the target domain.

Asset Modeling Language (AML) for the product family is the main output of Step 3. AML defines the types of artifacts, variability mechanisms, context information, choreography rules, and other constraints of the target domain. An AML is instantiated for a product family from a meta model. The next step consists of the mapping of features to DSAs and Variability Points (VPs) under the governance of AML. Actually, AML defines domain specific types and variability mechanisms needed in a particular product family; and features of an asset (ACM) can be mapped to multiple DSAs/VPs. The mapping matrix is the output of this step. Finally, Step 5 defines and publishes the software assets using AML and DSA/VP mappings of the capability model. A published asset model describes the artifacts, provided variability points, accessible artifacts from the outer world, and external dependencies. Outputs of this step are the asset models for the product family.

There is an obvious iterative cycle between the reference architecture modeling and product line asset modeling. Thus, it has also been shown as a sub-step in asset modeling even the tasks are discussed in separate chapters. Chapter 4 and Chapter 5 describe the reference architecture and asset modeling, respectively.

3.7 How SFA fits the Vision of Software Factories

As it has been discussed in Section 2.10, there are several steps needed for the realization of the vision of software factories. Here, the proposed SFA model is discussed towards achieving software factories with respect to the following constituents:

- *Development by Assembly*: Most of the development is expected to be component assembly, involving customization, adaptation and extension. The model facilitates development by assembly with the following perspectives:

  (a) The basic unit is defined as a DSAT and artifacts are instances of DSATs composed to build products.

  (b) The DSKs are loosely-coupled and artifact-to-artifact composition model is symmetric.

  (c) DSLs and CDL are used to specify the DSAs and the choreography, respectively. This makes the composition and validation easy.

  (d) The choreography engine with plugged DSEs forms a common infrastructure which enables the product assembly.

  (e) The assets have variations which are explicitly defined in the model, so that the product developer can customize or adapt by using the facilities provided by the asset developer.

(f) The variability realization mechanisms can be enriched by the employment of new techniques in domain specific engines.

(g) The tools (DSTs) facilitates the development, identification and composition of artifacts during assembly.

- *Software Supply Chains*: Supply chains emerge to create, package, consume and assemble the components specified using standard specification formats, developed by using standard techniques, methods, tools and processes. The model enables setting up supply chains as follows:

  (a) Forming the supply chains requires an architectural alignment process. The proposed model has been leveraged by a model for construction of product line reference architecture.

  (b) The architecture is built on the domain specific abstractions provided by DSKs. Therefore, it does not reveal the inner working details of the common architecture.

  (c) Since DSLs are loosely-coupled and the composition is achieved declaratively, third party suppliers can easily come up with loosely-coupled designs and components.

  (d) In addition to the supply chains for reusable assets, supply chains for DSKs can be formed. These are specialized for the design, construction and support of specific DSKs.

  (e) Actually, some of the asset suppliers can be DSK suppliers on the way. As long as their domain knowledge matures, the business know-how can be packed as a DSK rather then asset.

  (f) Vendors may appear as “SFA Providers”. These organizations own their DSKs and their reusable core assets, and they can provide them to other organizations to setup the product lines.

- *Relationship Management*: Managing relationships between suppliers and consumers is facilitated by the proposed model as follows:

  (a) Managing the configurations based on customer demand can be achieved effectively by the declarative nature of the product development.

(b) Requirements changes after product delivery can be tailored using the provided variations of assets.

(c) DSTs also include the management tools within DSKs, hence they allow the product engineer to effectively manage the product configurations.

(d) The DSKs and reference architecture provide an abstraction so that quality of service is preserved during product configuration.

(e) Product feature management can be traced back to asset feature models (ACMs) so that the product options can be managed and configured accordingly.

- *Domain Specific Assets*: In order to leverage the domain specific assets, the proposed model suggests the following features:

  (a) DSKs and reusable assets are the primary means of encapsulating domain specific know-how.

  (b) The reference architecture that composes all DSEs is one of the most valuable assets.

  (c) The DSK providers are able to develop and later improve their languages (DSL) and toolset (DST) according to their domains, which improves the productivity.

  (d) Domain know-how will be stored and fetched in the form of DSKs and software assets.

- *Mass Customization*: The long-term vision of mass customization requires a value chain that integrates processes like customer relationship management, demand management, product definition, product design, product assembly and supply chain management. The proposed model improves the value chain as follows:

  (a) The model is especially focused on domain design issues of the product line approach, hence it improves the definition, design, and assembly of products by domain specific abstractions.

  (b) The available explicit VP mechanism improves the mass customization potential.

(c) The model supports the assets to be reusable in multiple product lines, hence it reduces the time-to-market and enhances the quality of products. (See Section 6.5)

(d) The model strongly enables the mass customization by letting business analysts to configure and build the end products of the product family. They can easily configure the product variations and the artifacts using the accompanied DSLs and by hard-wiring the asset variations declaratively.

- *Organizational Change*: The stakeholders will face a new world at every step of software development, acquisition and usage. The quicker the organizational change is higher the benefits will appear.

  (a) The model facilitates the separation of concerns, therefore companies have to change their development organizations based on this model. In other words, the model requires an "organizational alignment" with the reference architecture, DSKs, and reusable assets.

  (b) The development units have to be responsible from a well-defined unit of work with precisely described input and output sets.

  (c) They have to be organized as "competence centers". Actually this is very common in SPL approach where the team has to be at least in three main groups: core asset developers, product developers and product line managers. However, SFA pushes this further both in technical and business expertise areas. Such an organization is similar to the assembly line organizations in other industries, such as automobile manufacturing.

  (d) The process that they run can be recorded, measured and improved systematically.

  (e) The model also facilitates the exchange of development teams across different business domains (product lines), which, in turn, improves the productivity and decreases the organizational risks.

CHAPTER 4

REFERENCE ARCHITECTURE MODELING WITH DOMAIN SPECIFIC KITS

An architectural model of software is a model in which overall structure of an application is captured as the composition of interacting components [118]. Hence, software architects partition their applications due to several architectural concerns in a way that they can model, design, develop, test, and even maintain every part separately and supervise the development easily. However, there is no mutual understanding to localize such architectural concerns in the problem domain and a common roadmap to associate these problem domain concerns with the solution domain, and furthermore how domain specific approaches can help during this process.

Software architecture modeling is expected to relate the architectural aspects and quality targets to running components and connectors. However, this mapping is not trivial, and it may surprisingly end up with problem domain concerns tangled in and scattered through the solution domain. In the end, such crosscutting concerns may complicate the detailed design process as well.

This chapter proposes an architectural modeling approach, which has been published recently [32], first to localize these concerns in multiple concern spaces and then relate them from problem to solution domain. This approach identifies the problem domain in utility concern spaces by correlating[1] the "architectural aspects" and "qual-

[1] The term "correlation" is used here as "a synonym for association or the relationship between variables" rather than a statistical term as "a numeric measure of strength of linear relationship between two random variables".

ity attributes", and solution domain in "architectural concern spaces" by correlating the "architectural tiers" and "architectural views".

Main contribution of this approach is the mapping technique, symmetric[2] alignment, as the correlation of "utility concern spaces" and "architectural concern spaces" instead of asymmetrically relating architectural concerns of problem domain to design decisions of solution domain. Symmetry and asymmetry are used here as given by [67]. This alignment assists the determination of architectural components, connectors as well as the associated properties in the solution domain, and further helps to abstract these as Domain Specific Engines (DSEs). This chapter also includes an example application of the approach on the architectural modeling of a Web security framework.

Managing the concerns in architectural modeling is not easy particularly for identifying the risks, tradeoffs and sensitivity points [80]. Two points are vital for concern management: localization of architectural concerns and representation of these concerns in a specific solution. Similar to other modeling approaches, this perspective divides the software architectural modeling process into two: "identification of the problem domain" and "description of the solution domain". Software architecture modeling is quite complicated in that sense since both of the domains are not trivial to identify. There are no standard ways to associate them, either.

The modeling approach has been depicted in Figure 4.1. It localizes architectural concerns in multiple concern spaces for both problem and solution domains, and symmetrically aligns them as shown in the figure. The following sections will explain these six steps, respectively.

4.1 Identifying Quality Requirements

As discussed in the previous chapter, feature-oriented requirements engineering yields the Domain Feature Model (DFM) indicating business, user and function requirements. Hence, DFM is the input to reference architecture modeling for a product family. Starting point of this approach is the identification of quality requirements from the feature models so that architectural aspects can be extracted in the next step.

[2] In "symmetric" paradigm, all components are treated as first class and of identical structure, and nothing is more basic than any other's. However, in "asymmetric" paradigm, all kinds of heterogeneous structures are aligned into others to demonstrate a base model.

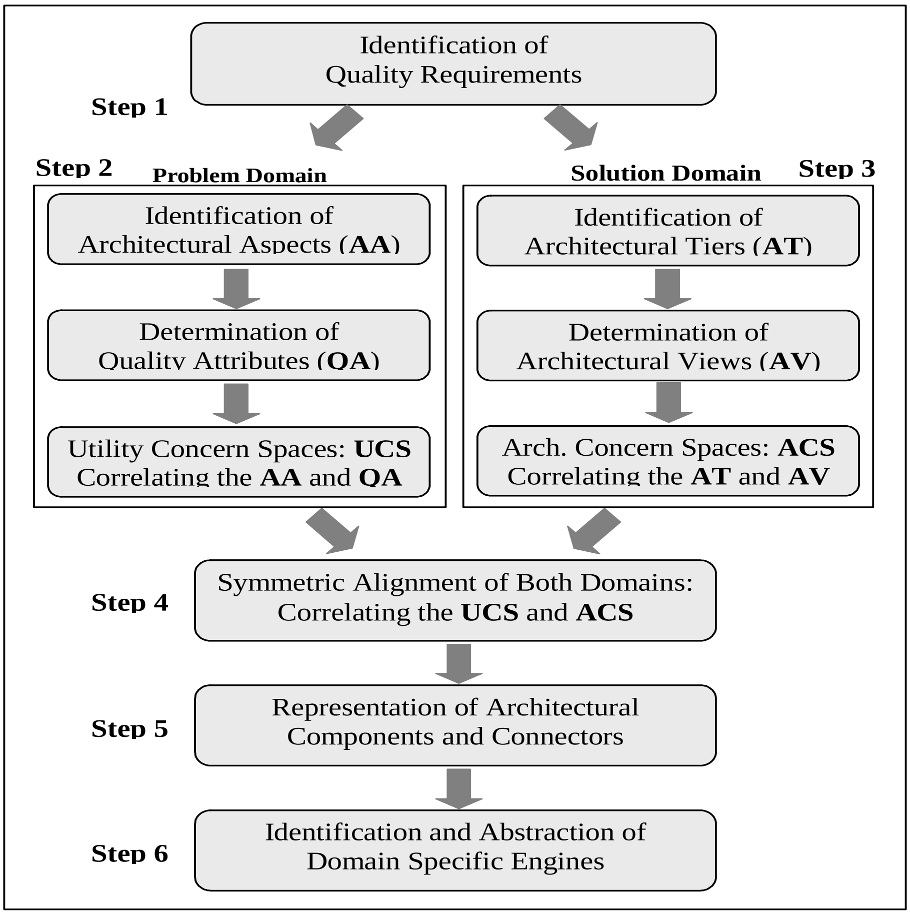

Identification of
Quality Requirements
Step 1
Step 2
Problem Domain
Solution Domain
Step 3
Identification of
Architectural Aspects (AA)
Determination of
Quality Attributes (QA)
Utility Concern Spaces: UCS
Correlating the AA and QA
Identification of
Architectural Tiers (AT)
Determination of
Architectural Views (AV)
Arch. Concern Spaces: ACS
Correlating the AT and AV
Step 4
Symmetric Alignment of Both Domains:
Correlating the UCS and ACS
Step 5
Representation of Architectural
Components and Connectors
Step 6
Identification and Abstraction of
Domain Specific Engines

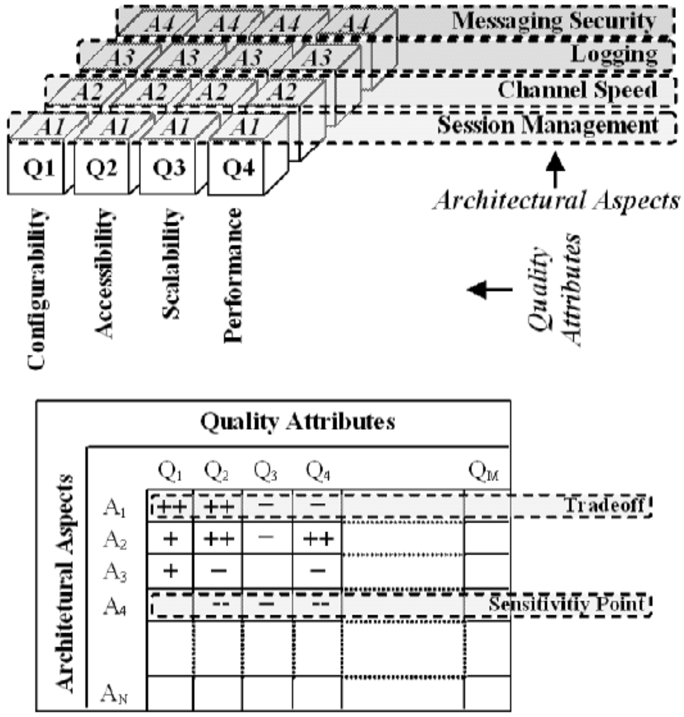
Figure 4.1: Reference architecture modeling approach

In order to collect the actual set of quality requirements during the construction of feature models, different methods can be used. However, what has been experienced is that scenario-based and stakeholder-oriented ones like [80] usually end up with more realistic and near to complete set of quality requirements than the others.

4.2 Identifying Problem Domain: Utility Concern Spaces

Identified set of quality requirements will form the problem domain concerns in multiple concern spaces called as *Utility Concern Spaces (UCS)*. UCS takes the name from ATAM [80] since it uses a specific technique to map the architectural aspects to quality attributes first in a hierarchical decision tree called as “utility” tree.

At this step, UCS is constructed by correlating the quality attributes taken from any quality model such as [69, 89], and the architectural aspects categorized by the architect. Consequently, the problem domain is modeled as a matrix presented in Figure 4.2 where the architectural aspects and derived quality attributes constitute the rows and columns of this matrix, respectively. Finally, every correlation set (the cubes of upper illustration in Figure 4.2) embodies an individual utility concern space.



	Q_1	Q_2	Q_3	Q_4		Q_M
A_1	++	++	−	−		
A_2	+	++	−	++		
A_3	+	−		−		
A_4		--	−	--		
A_N						

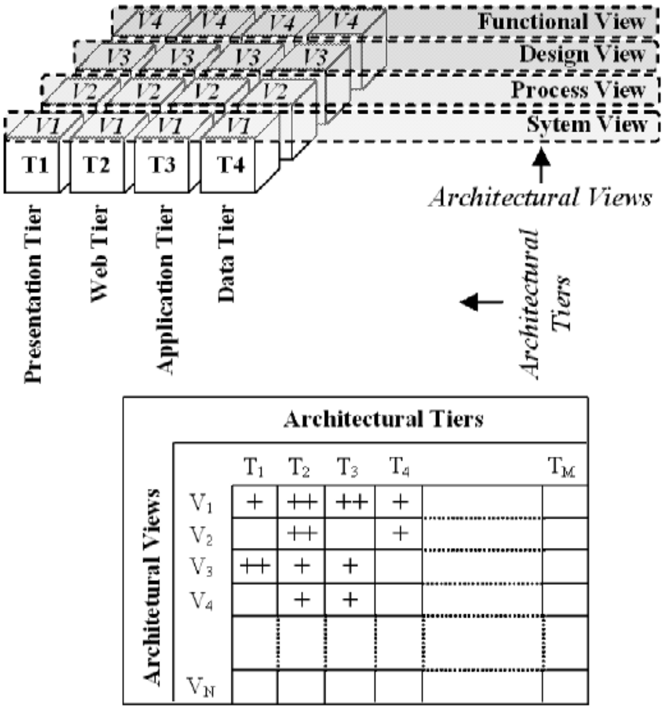
Figure 4.2: Utility Concern Spaces (UCS)

UCS is an NxM matrix to be formed by the architect as shown in Figure 4.2. Each cell is shown like "UA_4Q_2" where "U" represents the Utility Concern Space, "A_4" denotes the fourth architectural aspect; Messaging Security in the example, and "Q_2" shows the second quality attribute; Accessibility in this case. The unique UCS cell identifiers will be used to form the symmetric alignment matrix later in Step 4.

The symbols like "$--$" or "++" in Figure 4.2 denote the "correlation coefficient", i.e. strength, of mappings between architectural aspects and quality attributes. The lack of any symbol means there is no correlation. The "+" or "$-$" symbol denotes a positive or negative correlation, and "++" or "$--$" symbol shows the stronger correlations. The coefficients play an essential role for identifying the "sensitivity points" - parameter to which some quality attribute is highly related - and "tradeoffs" - factor that affects many quality attributes in opposite directions [80]. One can easily identify the sensitivity points in a UCS matrix if there are only plus or minus signs in an individual row. However, coexistence of both plus and minus signs in a single row means a basic or strong tradeoff due to correlation coefficients.

4.3 Describing Solution Domain: Architectural Concern Spaces

Akin to problem domain concerns, solution domain concerns are formed into *Architectural Concern Spaces (ACS)*, which is the correlation of architectural tiers and architectural views given by the matrix in Figure 4.3.



Architectural Views		T1	T2	T3	T4		TM
	V1	+	++	++	+		
	V2		++		+		
	V3	++	+	+			
	V4		+	+			
	VN						

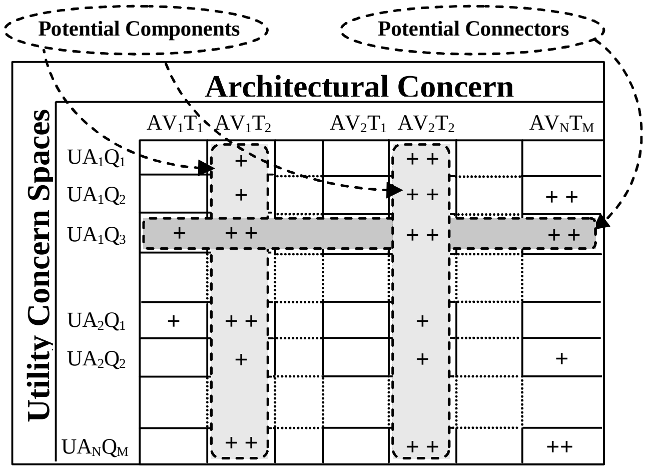
Figure 4.3: Architectural Concern Spaces (ACS)

Architectural tiers are well-known tiers of “N–tier” model, and architectural views are customized from Kruchten’s “4+1 View” [82] as follows: *Functional View* shows functions, system abstractions, and domain elements as components; whereas dependencies and data flows as connectors. *Design View* shows services, components, subsystems, aspects and the likes as components; but invokes, calls, joinpoints and queries as connectors. *Process View* represents processes, threads, workflows and the likes as components; whereas synchronization, control and data flows, triggers, and events as connectors. Eventually, *System View* shows servers and networking as components; but protocols and message queues as connectors.

16 distinct concern spaces are shown in this matrix, but there may be more or less as the number of tiers and views differs. For instance, an architectural design may require a further *Business Processing Tier* or a separate *Development View* with respect to different quality requirements identified at Step 1. Similar to UCS, the architectural concern spaces are shown as "AV_3T_1" where "A" represents the Architectural Concern Space, "V_3" denotes the third architectural view; Design View in this case, and "T_1" denotes the first architectural tier; Presentation Tier in the example. The unique ACS cell identifiers will be used again to form the symmetric alignment matrix later in Step 4. Figure 4.3 also includes the correlation coefficients like Figure 4.2 has, but note that ACS may have only positive correlations here.

4.4 Symmetric Alignment of Both Domains

Symmetric alignment is a join step after structuring the utility and architectural concern spaces separately. Precisely speaking, symmetric alignment is correlating the UCS and ACS in another matrix as shown in Figure 4.4.



	AV₁T₁	AV₁T₂		AV₂T₁	AV₂T₂		AV_NT_M
UA₁Q₁		+			+ +		
UA₁Q₂		+			+ +		+ +
UA₁Q₃	+	+ +			+ +		+ +
UA₂Q₁	+	+ +			+		
UA₂Q₂		+			+		+
UA_NQ_M		+ +			+ +		++

Figure 4.4: Symmetric alignment matrix

Symmetric alignment matrix has the rows formed by UCS cells given in Figure 4.2 and the columns formed by ACS cells given in Figure 4.3. Besides, every cell of this alignment matrix identifies the place (ACS) where a UCS should be related with solution elements. In fact, this association is not an exhaustive one-to-one pairing. The prioritization of UCS and ACS pairing is directed by the correlation coefficients of ACS matrix in a way that the ACS cells with strong correlation coefficients (having "++" symbols like AV_1T_2 in Figure Figure 4.3) must be attempted first and then the others containing normal coefficients (having the "+" symbols like AV_3T_2). The cells without any correlation coefficient in ACS matrix should not be attempted for pairing with UCS cells. For example, the column AV_2T_1 in Figure 4.4 is left blank since the AV_2T_1 cell in Figure 4.3 does not contain any correlation coefficient.

The major contribution of this matrix is its guidance in identifying the architectural components, connectors and properties. The resulting matrix enables the separation of both problem and solution domain concerns, thus the architectural decisions can be taken in isolated regions of architectural concern spaces.

Potential components can be realized by analyzing the columns of this symmetric alignment matrix. If a column contains many coefficients, then it means that several UCS are tangled in an ACS, and such concerns may be better abstracted with separate architectural components. For instance, AV_1T_2 column of Figure 4.4 signals a potential architectural component since this column has strong mappings to several UCS.

The row-wise distribution of correlation coefficients signals a potential architectural connector. In fact, such a distribution means that a single UCS has solution elements scattered through many ACS that may require a connection between architectural components. For example, Figure 4.4 signals a potential connector resulted by the scattering of UA_1Q_3 through different ACS, and it may probably bridge the architectural components identified under AV_1T_2 and AV_2T_2 columns.

Architectural properties can also be identified from the intersection of UCS and ACS. For example, the potential component under AV_1T_2 column of Figure 4.4 is expected to expose the properties like "clustering of secure Web servers" since this ACS is correlating the System View and Web Tier, and intersected by UA_1Q_3 row as a UCS that correlates the Session Management and Scalability.

The final piece of information, expected from reference architecture modeling, is the set of contextual information (variables), which contains the stateful/stateless information to connect architectural components (may be individual DSEs). By analyzing the properties of components and connectors, a common context has been formed for communication and coordination of these individual components and DSEs.

4.5 Representing Components and Connectors

After identifying the components, connectors, properties and context variables, an architectural model should represent the complete architecture in a structured form.

In order to describe solution domain of architecture modeling, Architecture Description Languages (ADL) and other similar representation techniques have been introduced. Several ADLs have been designed so far with varying core concepts [86]. For example, Rapide language is an ADL built on the notion of partial ordered sets [96], and Unicon makes a smooth transition to code with a very generous type mechanism [118]. Wright allows the architectural styles to be formalized to check the consistency and completeness of architectures [5], whereas Acme can be seen rather as an interchange format between other languages and tools [58]. Koala is an industrial ADL used to develop consumer products [131]. Even people argue that UML lacks ADL features; it includes some informal ADL characteristics [86, 90].

This approach will not have a preference for such a representation and the architect is free to select any one from the existing approaches. For the rest of this thesis, the architectural drawings are provided with Software Architecture Analysis Method (SAAM) [78] architectural notation which has been highlighted in Figure 4.5.

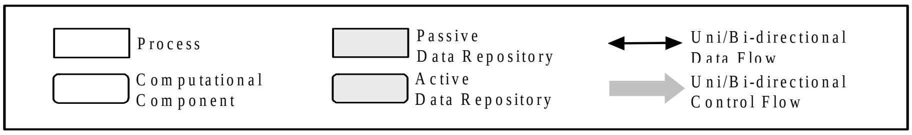


Figure 4.5: Architectural notation (from SAAM [78])

4.6 Identifying Domain Specific Engines (DSEs)

The final step in modeling reference architecture is the identification of Domain Specific Engines (DSEs) where appropriate. Devising DSEs is critical in increasing the level of abstractions of components that exist in the architecture. The tightly coupled components can be unified so that the number of connectors is reduced, set of interrelated components can be controlled by one DSL, and accompanied toolset can be developed accordingly.

There are several hints on DSE identification: first one is to determine the tightly coupled components and try to unify them. Second, analyze the artifact types of components and try to relate those that are similar to each other and encapsulate them with a DSE if this provides a further reduction in number of architectural elements. Third, analyze dependency of architectural components to each other or to a common context and try to introduce a DSE to configure and manage such a dependency through a DSL. At the end, identifying a DSE facilitates reducing architectural complexity.

The identified DSEs must comply with the characteristics of DSKs stated in Section 3.2. The DSEs will be composed through a composition model. The interaction model of composition relies on SOA as a paradigm for managing resources, describing process steps, and capturing interactions between an artifact and its environment. The artifacts can be composed via unified interfaces and they can be declaratively specified by a choreography language as stated in Section 3.5.1.

DSKs naturally introduce the use of corresponding domain specific artifacts within a reference architecture, and the applications developed in this reference architecture will be assembled using these domain specific artifacts. For instance, consider this step ending up with the selection of a Business Process Engine (BPE), a Business Rule Engine (BRE) and a Business Service Engine (BSE) in the reference architecture, then the applications of this domain will be built using the following artifact types: *processes*, *rules/composite_rules*, and *services*, respectively.

From the viewpoint of artifact composition, there is no significance whether DSEs reside on a single tier or crosscut multiple tiers. In the business rule segregation paper [33], it has been shown that business rules crosscut the process management and they are orthogonal to architectural tiers. Therefore a DSE for rule execution, in this case, abstracts the related architectural concerns and provides a declarative environment for management of business rules across many tiers.

4.7 Case Study: Web Security Framework

A simple example to demonstrate the applicability of the proposed model has been discussed here. This real-life example demonstrates the approach on the design of a Web security framework of a mission critical e-finance application for Central Registry Agency (CRA[3]) of Turkey. Furthermore, in Chapter 6, the reference architecture for two example product families in parallel to their asset modeling has been presented for validation purposes.

Step–1: Identifying Quality Requirements

CRA has a mission to dematerialize and register via available electronic records, capital market instruments, and rights attached thereon with respect to issuers, intermediary institutions, and right owners, check the integrity of actual records kept by member groups. The particular security needs that have arisen in the requirements engineering phase of security framework for CRA duties have been listed in Table 4.1. Accompanied with this list of requirements, the requirements engineering puts forward the DFM partially depicted in Figure 4.6.

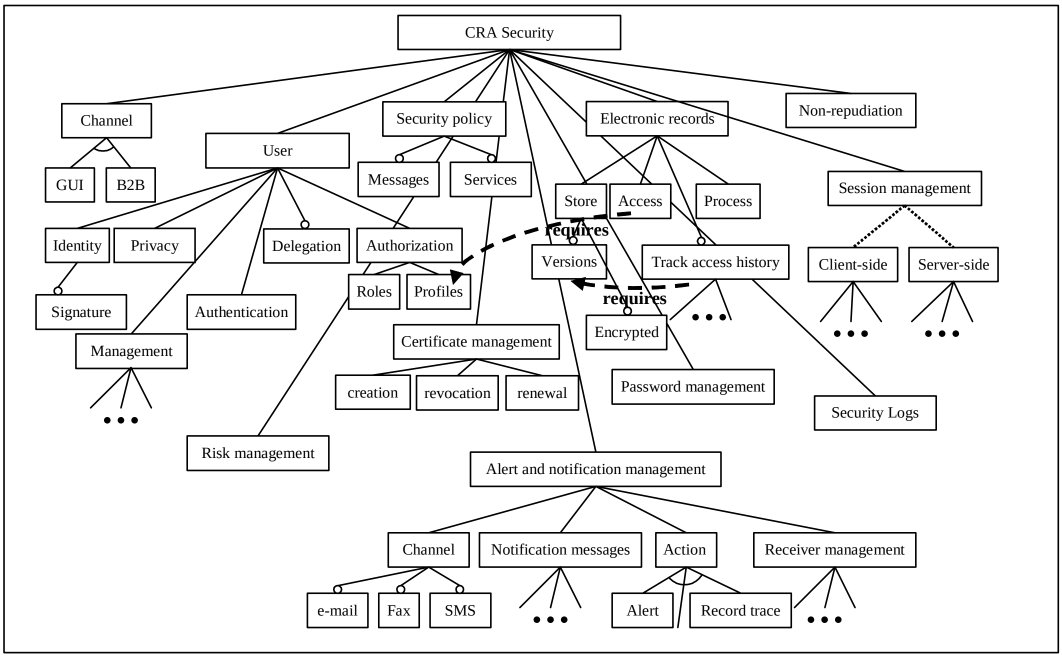


Figure 4.6: Feature diagram of the CRA security (partial)

[3] http://www.mkk.com.tr/

Table 4.1: (Step–1) Identified security requirements

R_1	The system has to preserve privacy in registering capital market instruments and rights.
R_2	Security management is critical in maintaining integrity of the records among different members (stakeholders).
R_3	The system must ensure confidentiality of records in line with the applicable regulatory provisions.
R_4	Network and distributed systems have to be secured.
R_5	The infrastructures have to be flexible enough to accommodate new security policies.
R_6	The personal data have to be secured.
R_7	The business processes and data confidentiality have to be guaranteed.
R_8	The client-side identity and session management are critical for user privacy preserving.
R_9	The user session management associated with smartcards needs clients to be able to keep required identification of the user transactions and to track all user activities.
R_{10}	The users are required to be authenticated first.
R_{11}	The users are allowed to access various resources based on user profiles and roles.
R_{12}	The authorization needs to take possible delegations into account.
R_{13}	The user can not later deny his/her transactions (non-repudiation). It ensures that a transferred message has been sent and received by the parties claiming to do so, which is a "legal" obligation for financial applications.
R_{14}	The system has to provide a secure location for storing and processing the system records.
R_{15}	The system has to protect electronic records from unauthorized access and undesired execution.
R_{16}	Some of the services require the user to be authenticated at service-level.
R_{17}	Security-based record update requires the management of older versions of records and keeping the record history with privileges.
R_{18}	All system-level events are required to be integrated with various alert mechanisms such as e-mail, SMS, and the likes.
R_{19}	Secure messaging mandates the use of encryption during message transition and security assertions like digital signatures to be part of the messages.
R_{20}	Requirement (R_{19}) applies to B2B integration as well.
R_{21}	B2B integration extends business processes across organizations, which results in multiplication of security concerns at both ends.
R_{22}	Certificate management (certificate creation, revocation, renewal) has to be provided.
R_{23}	Password management (password generation, change, revocation, validation, password complexity, encrypted storage) is required.
R_{24}	Detailed logging of all system actions (success/fail cases for database, application, network actions) is mandatory.

Step–2: Identifying Problem Domain

At the end of feature-oriented requirements engineering phase, a DFM including the list of requirements and feature models has been constructed as given in Table 4.1 and in Figure 4.6. For this domain it revealed 24 requirements and a corresponding feature model. Note that irrelevant details of this example have been omitted from the feature model for simplicity.

Next step in the architecture modeling is the identification of problem domain which starts with the classification of architectural aspects. The architectural aspects are distilled from the feature models and requirements list by elicitation of features, feature groups, feature dependencies, and feature interrelations. However, there is no straightforward method in identification of architectural aspects from the requirements since non-functional requirements are usually crosscut the functional requirements and they are scattered. At the end of this sub-step we obtain a candidate set of architectural aspects which has been listed in Table 4.2.

Table 4.2: (Step–2.1) Taxonomy of architectural aspects

Presentation Aspects	
A_1	Client-side Identity Management
A_2	Client-side Session Management
A_3	Session Management
A_4	Authentication
A_5	Authorization and Delegation
A_6	User Profiles and Roles
Application and Data Security Aspects	
A_7	Service-level Authorization
A_8	Data Security (Protection of Electronic Records)
A_9	Security-Based Record Update
A_{10}	Non-Repudiation
A_{11}	Security Alerts
Integration And Communication Aspects	
A_{12}	Messaging Security
A_{13}	Securing B2B Integration
A_{14}	Encryption
Administrative Aspects	
A_{15}	Logging
A_{16}	Certificate Management
A_{17}	Password Management
A_{18}	Security Risk Management

In order to clarify the discussion, several architectural aspects are presented on the feature diagram. Figure 4.7 is based on the feature diagram given in Figure 4.6 and some of the architectural aspects have been marked.

As it can be observed from Figure 4.7, there are different cases: Client-side Session Management (A_2) is scattered over several features such as GUI accessibility, client-side session requirements, and user identity/privacy features. Non-repudiation (A_{10}) has one-to-one correspondence with a feature node. Data Security (Protection of Electronic Records) (A_8) and Security-Based Record Update (A_9) are both tangled and scattered over several feature groups. Certificate Management (A_{16}), on the other hand, coincides with a feature subtree on its own. The association rules of features to architectural aspects will be investigated in further case studies.

After the categorization of security aspects, the next step is the identification of

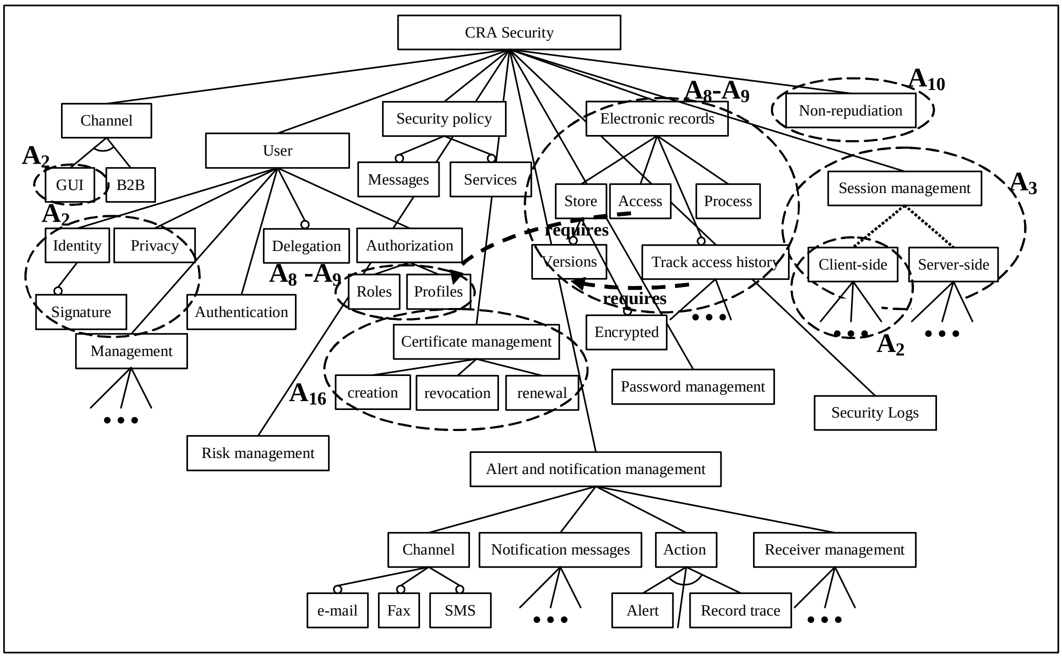


Figure 4.7: Example architectural aspects on CRA security feature diagram

quality attributes. The quality model has been adapted from the comprehensive list and precise classification in [89]. Table 4.3 contains the identified quality attributes for CRA case. Again this has been distilled from the requirements of the domain.

Table 4.3: (Step–2.2) List of quality attributes for security

Q_1	Usability	Q_{14}	Interoperability
Q_2	Accessibility	Q_{15}	Openness
Q_3	Operability	Q_{16}	Heterogeneity
Q_4	Simplicity	Q_{17}	Auditability
Q_5	Portability	Q_{18}	Traceability
Q_6	Performance	Q_{19}	Analyzability
Q_7	Scalability	Q_{20}	Configurability
Q_8	Latency/Responsiveness	Q_{21}	Distributeability
Q_9	Transaction Throughput	Q_{22}	Availability
Q_{10}	Modifiability	Q_{23}	Confidentiality
Q_{11}	Upgradeability	Q_{24}	Integrity
Q_{12}	Data/Version Consistency	Q_{25}	Maintainability
Q_{13}	Composability	Q_{26}	Reliability

The final step in identification of problem domain is the determination of correlation between architectural aspects ($A*$) and quality attributes ($Q*$). Table 4.4 shows some of the correlations since the complete matrix in Figure 4.2 cannot be fully visualized and discussed here due to space limitations (it is a 18x26 matrix in this example). Instead, the matrix grouped by the correlations has been presented in Table 4.4.

Table 4.4: (Step–2.3) UCS for Web security framework modeling (partial)

Strong Positive (++)	A_1	UA_1Q_{23}
	A_2	UA_2Q_{13}, UA_2Q_{14}
	A_3	UA_3Q_8, UA_3Q_{22}
	A_4	UA_4Q_2, UA_4Q_{23}
	A_{10}	$UA_{10}Q_{23}$
	A_{12}	UA_12_Q2
	A_{14}	$UA_{14}Q_{10}$
	A_{15}	$UA_{15}Q_{18}$, $UA_{15}Q_{19}$, $UA_{15}Q_{20}$
	A_{16}	$UA_{16}Q_2$, $UA_{16}Q_{14}$, $UA_{16}Q_{20}$
	A_{17}	$UA_{17}Q_{26}$
Positive (+)	A_1	UA_1Q_{17}
	A_2	UA_2Q_{11}, UA_2Q_{18}, UA_2Q_{20}
	A_3	UA_3Q_2
	A_4	UA_3Q_{17}
	A_{10}	$UA_{10}Q_{26}$
	A_{11}	$UA_{11}Q_{20}$
	A_{14}	$UA_{14}Q_{13}$
	A_{18}	$UA_{18}Q_{18}$, $UA_{18}Q_{19}$
Negative (−)	A_2	UA_2Q_3, UA_2Q_4
	A_3	UA_3Q_3, UA_3Q_7
	A_4	UA_4Q_1
	A_{11}	$UA_{11}Q_3$
Strong Negative (−−)	A_2	UA_2Q_8, UA_2Q_{21}
	A_{17}	$UA_{17}Q_4$

As discussed in Section 4.2, UCS matrix reveals the tradeoffs and sensitivity points. Considering the Table 4.4 (representing the partial UCS matrix), if any architectural aspect exists only in positive and strong positive groups or only in negative and strong negative groups, then it is a sensitivity point. An example to sensitivity point is the Security Risk Management (A_{18}) since it has only positive correlation with Traceability (Q_{18}) and Analyzability (Q_{19}).

Another example for sensitivity point is the Client-side Identity Management (A_1) that must provide Confidentiality (Q_{23}) and Auditability (Q_{11}). However, it has no negative correlation with any of the quality attributes. Hence, A_1 is relatively easy to decide as compared to A_2 that has both positive and negative correlations.

Conversely, if an aspect exists within both negative and positive groups, it indicates a tradeoff. A simple tradeoff example is as follows: Password Management (A_{17}) has a strong positive correlation with Reliability (Q_{26}), whereas it has a strong negative correlation with Simplicity (Q_4).

A further complicated tradeoff exists for Client-side Session Management (A_2). It has a strong positive correlation with Q_{14} and Q_{13}; positive correlation with Q_{11}, Q_{18}, and Q_{20}; whereas it has negative correlation with Q_3 and Q_4, and even a strong negative correlation with Q_8 and Q_{21}. The verbal expression of this variety of correlations is: "Client-side Session Management must be composable and interoperable with smartcards, and it must be upgradeable, configurable and traceable. Nevertheless, it should not create any latency and distributeability problems as well as it should be simple and operable."

Step–3: Describing Solution Domain

For structuring the solution domain, a 4-tier model with Presentation, Web, Application and Data Tiers has been employed. The architectural views will be the basic four architectural views introduced in Section 4.3: Functional, Design, Process and System Views. Based on the quality requirements, the correlation matrix of ACS is presented in Table 4.5.

Table 4.5: (Step–3) ACS for Web security framework modeling

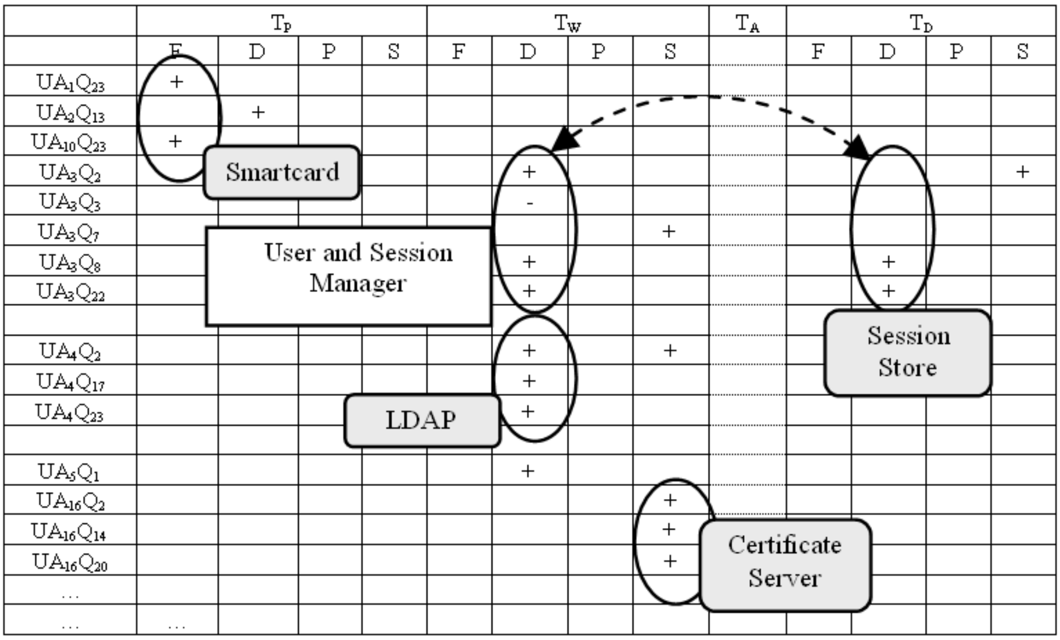
ACS (AV_XT_Y)	Presentation Tier (T_P)	Web Tier (T_W)	Application Tier (T_A)	Data Tier (T_D)
Functional View(V_F)	++	+	++	
Design View(V_D)	++	++	++	++
Process View(V_P)	+	+	+	+
System View(V_S)	+	++		+

Based on Table 4.5, AV_ST_A and AV_FT_D do not need to be considered for the symmetric alignment since there is no correlation between these tiers and views from the security perspective of CRA application.

Step–4: Symmetric Alignment of Concern Spaces

After identifying UCS and ACS, symmetric alignment of these concern spaces results in an alignment matrix given partially in Table 4.6. This matrix depicts some of the resulting component identifications and their connectors. The Application tier and corresponding views are eliminated from the figure just to simplify the presentation.

Table 4.6: (Step–4) Symmetric alignment of UCS and ACS


Smartcard
User and Session Manager
LDAP
Session Store
Certificate Server

	T_P				T_W				T_A	T_D			
	F	D	P	S	F	D	P	S		F	D	P	S
UA_1Q_{23}	+												
UA_2Q_{13}		+											
$UA_{10}Q_{23}$	+												
UA_3Q_2						+							+
UA_3Q_3						-							
UA_3Q_7								+					
UA_3Q_8						+					+		
UA_3Q_{22}						+					+		
UA_4Q_2						+		+					
UA_4Q_{17}						+							
UA_4Q_{23}						+							
UA_5Q_1						+							
$UA_{16}Q_2$								+					
$UA_{16}Q_{14}$								+					
$UA_{16}Q_{20}$								+					
…													
…	…												

Table 4.6 guides the identification of architectural components and connectors. For example, Client-side Identity Management (UA_1Q_{23}) and Non-repudiation ($UA_{10}Q_{23}$) indicate a component in presentation tier, which is realized with a "Smartcard" controlled by the Client-side Session Manager given in Figure 4.8.

For Session Management (A_3) aspect, UCS (UA_3Q*) are aligned to AV_DT_W and AV_DT_D. Such an alignment signals dedicated components for "User and Session Manager" together with "Session Store", respectively. Furthermore, alignment of UA_3Q_8 and UA_3Q_{22} to both AV_DT_W and AV_ST_D reveals that there should be a connector between the Session Manager and Session Store. Alignment of UA_3Q_2 to both AV_DT_W and AV_ST_D indicates that system components at Data Tier should be accessible through the Web Tier. The alignment of UA_3Q_7 to AV_ST_W shows that some precautions might be taken during the design so that Session Management should not decrease the scalability of the Web Tier. Hence, this points to an architectural property.

Authentication (A4) requires a dedicated component (LDAP) integrated with the User and Session Manager. Similarly, the alignment of $UA_{16}Q_2$, $UA_{16}Q_{14}$ and $UA_{16}Q_{20}$ to AV_ST_W points out a dedicated component, i.e. Certificate Server, to be integrated with the User and Session Manager at Web Tier.

UA_5Q_1 signals the presence of a component and decision is to introduce "User/Session Manager" for that. Actually, it signals another connector between the User and Role Management services in Application Tier, but it is not apparent in the partial matrix given in Table 4.6.

Step–5: Representing Components and Connectors

As stated in Section 4.5, the resulting architectural model can be broadly represented with any architecture description techniques. For simplicity, Figure 4.8 depicts merely the structural representation of the architecture by using the lexicon given in Figure 4.5.

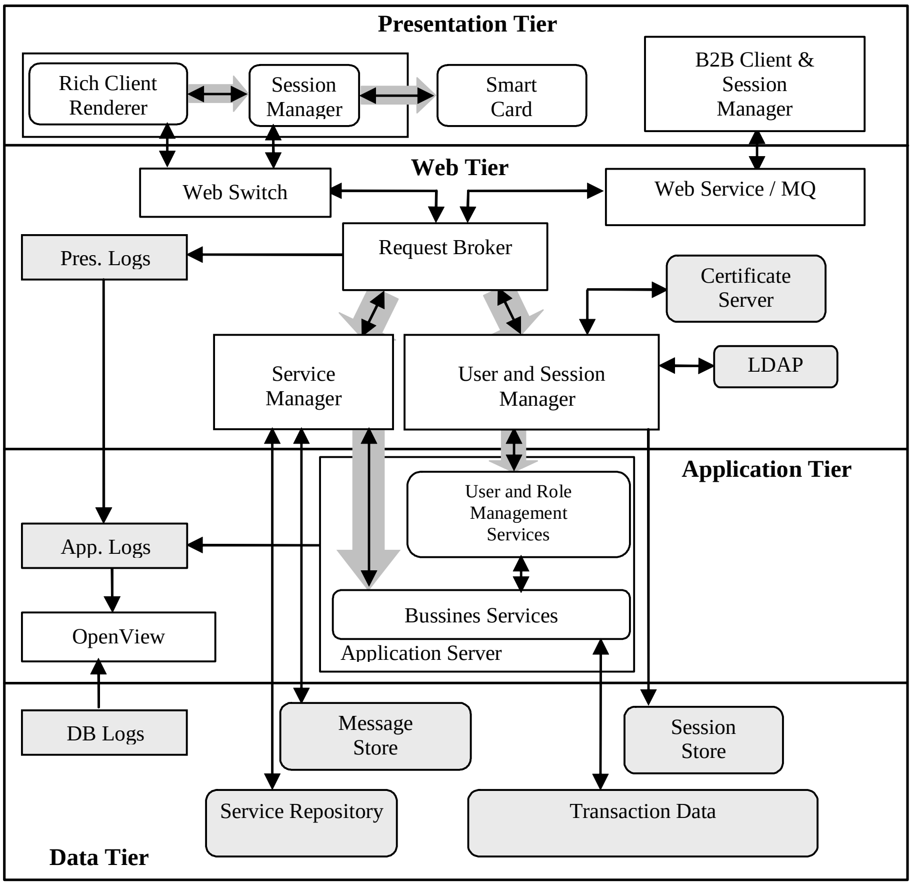


Figure 4.8: The Web security framework model

Step–6: Identifying DSEs

DSE identification is critical since it isolates/encapsulates some of the components and corresponding connectors in the solution domain. Hence this step simplifies the resulting architecture in the number of components and connectors.

In the scope of this example, four simple DSKs have identified. In other words, the corresponding domain specific engines are embedded into the choreography platform of the architecture. They are *Client-Side Rich Internet Application (RIA) framework*, *Business Service Engine*, *Logger* and *Global User Management*. The Figure 4.9 depicts these DSEs in place.

The first one is expressed in EBML (Enhance Bean Markup Language), rendered by ERE (EBML Rendering Engine), and designed via EDS (EBML Development Studio). For the second one, Business Service Engine is used for executing the business services, which are expressed using XML (DSL) and stored in a repository, there is a runtime engine (DSE) to execute services and they are developed using "Service Editor and Eclipse" (DST). For the third DSK, Log4j[4]-based configuration XML is the language (DSL), editors and management console form the toolset (DST), and contained logic for traceability and monitoring of logs constitute the engine (DSE). Similarly, for *User Management*, an appropriate DSL can be designed to configure users, policies, authorization models, and other settings, with appropriate toolset (DST) and runtime facilities (DSE).

4.8 Remarks

Software architecture has gained a vast impetus for improving the quality of software applications; hence architectural modeling has become an area of intense research [57]. An extensive study has been carried out to reveal the evolution of software architectures in line with achieving software quality [126]. Supportively, the quality attributes of software have been classified [14, 44, 69, 89, 110], architectural concerns have been identified with respect to conceptual [6, 8, 13, 118] and semantic [5] models as well as viewpoint and stakeholder-based reasoning [73, 78, 80, 86].

The concerns affecting architectural quality can be modeled in many ways, one of which is the individual mapping of problem domain concerns to the solution domain,

[4] http://logging.apache.org/log4j/

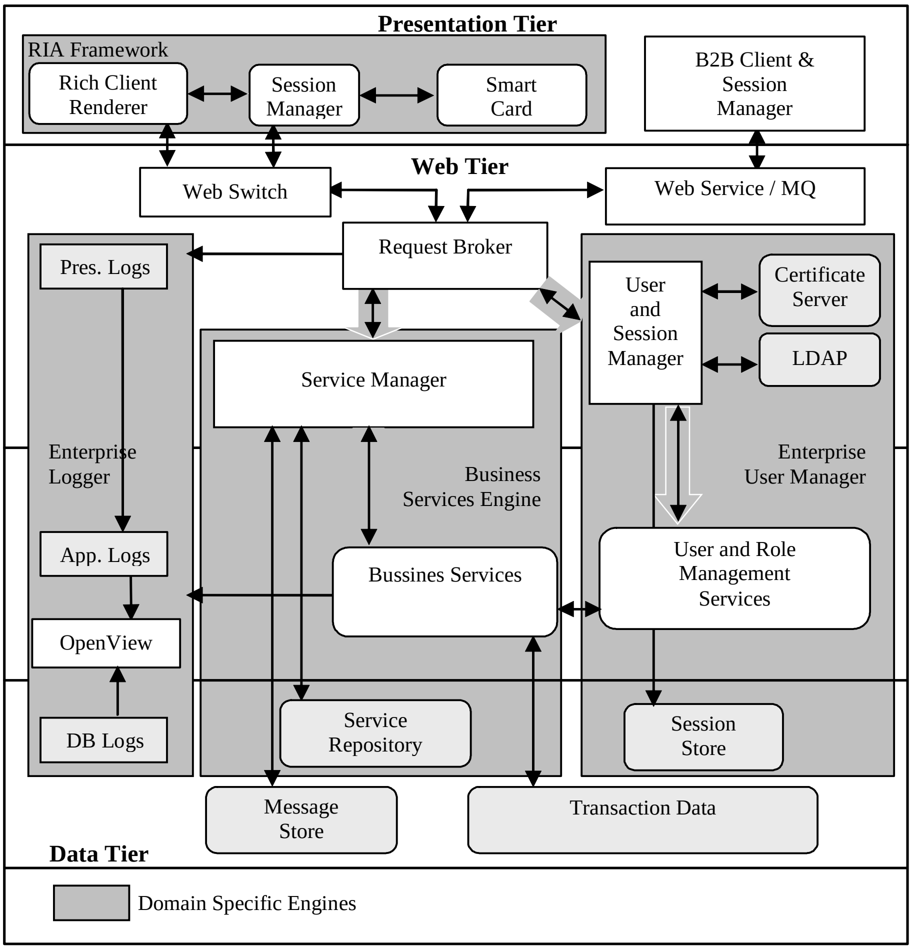


Figure 4.9: The Web security framework model with DSE abstractions

but such a mapping usually creates tangled and scattered concerns in the solution domain. An example to this in a Web-based security framework may be the tangling of authentication and authorization at Web tier as well as scattering of the identity management at Web, Application and Data tiers. In order to separate such crosscutting concerns, there are some approaches [68, 82, 90] with a multi-view perspective, but they primarily help in modeling the solution domain. The problem domain needs a similar structure so that architectures can be modeled with multi-dimensional separation of concerns [73, 105, 127].

Analyzability of the architectural models is another significant concern addressed by some of the methods. Software Architecture Analysis Method (SAAM) uses scenarios to evaluate quality properties of architecture, being biased towards evaluating maintainability [78]. It has some extensions on different tracks like SAAMCS for complex scenarios [86] and ATAM for architecture trade-off analysis [79, 80]. ATAM requires the business drivers and quality attributes to be specified as well as detailed architectural descriptions to be made available.

FORM [75] uses feature models for defining parameterized reference architectures and reusable components. The feature diagrams are layered in four layers: Capability Layer, Operating Environment Layer, Domain Technology Layer, and Implementation Technique Layer. Furthermore, feature models contain feature diagrams, composition rules, feature dictionary and other issues/decisions. The method semi-formally defines a mapping of feature models into subsystem, process and module models of a reference architecture. It depends more on the domain engineer's skills, creativity and experience to construct high quality domain products. The proposed approach differs from FORM in two respects: first, by capturing and isolating architectural aspects and quality attributes from the domain feature model; second, by feeding functional features into asset modeling activity to construct reusable assets.

In [95], a mapping from feature models to architecture models has been proposed. However, the approach has major weaknesses by mapping features directly to architectural components since the features may be tangled and scattered, and an architectural aspect may crosscut different features at the same time. Similarly, Sochos et al. propose a mapping of feature models to architectures [121, 122]. The features models are manipulated with a series of transformation rules, and later features are implemented by architectural components and their interactions are mapped to architectural connectors (as component interfaces). In addition to the drawbacks on crosscutting and tangling concerns, the latter study needs the formal semantics of transformation rules and the effective means of resolving feature interactions.

Regarding the architectural modeling, [49] states that "a future work is required to develop systematic ways of bridging quality requirements of software systems to their architecture; unsolved problem is how to take architectural concepts better to analyze software systems for quality attributes in a systematic way". The proposed approach given here is an attempt to solve this problem with the symmetric alignment and by utilizing Domain Specific Engines.

One final note about the resulting software architecture is that it employs a symmetric composition model in which all components are treated as first-class, co-equal building-blocks of identical structure, and in which no component's model is more basic than any other's. It does not make any distinction on Domain Specific Artifacts (DSAs) of different DSKs. Artifact-to-artifact composition is driven by the choreography rules of asset models. Harrison et al. make a good overview of symmetric and asymmetric paradigms for software composition [67]. From the point of reuse, they analyze that the use of symmetrically organized paradigms is a vehicle for promoting a reusable components industry, and relationship symmetry is essential to any kind of reuse. Their result also strengthens the usability of the proposed reference architecture modeling approach especially within the context of improving reuse.

The tool support for the technique presented in this chapter has already been started. The first prototype of the tool has been developed as a graduation project for Master's Degree without thesis [4]. Several screenshots of the prototype have been presented in Appendix-B. The tool provides a central repository for architectural modeling and information sharing, which also enables an XML export.

CHAPTER 5

SOFTWARE ASSET MODELING WITH DOMAIN SPECIFIC KITS

This chapter presents the proposed feature-based software asset modeling approach in detail. The asset modeling has been leveraged by Domain Specific Kits (a language, an engine, and a toolset) of the Software Factory Automation (SFA) concept introduced in Chapter 3.

This modeling approach uses the abstractions provided by Domain Specific Kits to improve the commonality of features, and provides means to effectively manage the variations of them by exploiting a meta-model. Using features in software modeling is not new, however, encapsulating them in individual asset models with domain specific abstractions looks more attractive since this approach ends up with more loosely coupled assets. The proposed approach creates more cohesive asset models by encapsulating the feature commonality within an asset. It further facilitates variability management with composition of Domain Specific Artifacts through the choreography engine of SFA reference architecture.

5.1 Feature-Based Asset Modeling Approach

Feature-based software asset modeling with DSKs maintains two different views of assets:

- *External View* is specified by Asset Capability Model (ACM).
- *Internal View* is represented by Asset Meta Model (AMM).

The former is expressed as a feature diagram describing the capabilities of an asset, and the latter is expressed with Asset Modeling Language (AML) for representing domain specific artifacts, contextual properties, and variability points.

Figure 5.1 depicts the software asset modeling approach in five steps that are expounded in the sequel. The primary input of this step is the DFM, which is the global view of requirements of product family; and it has been produced in the feature-oriented requirements engineering step.

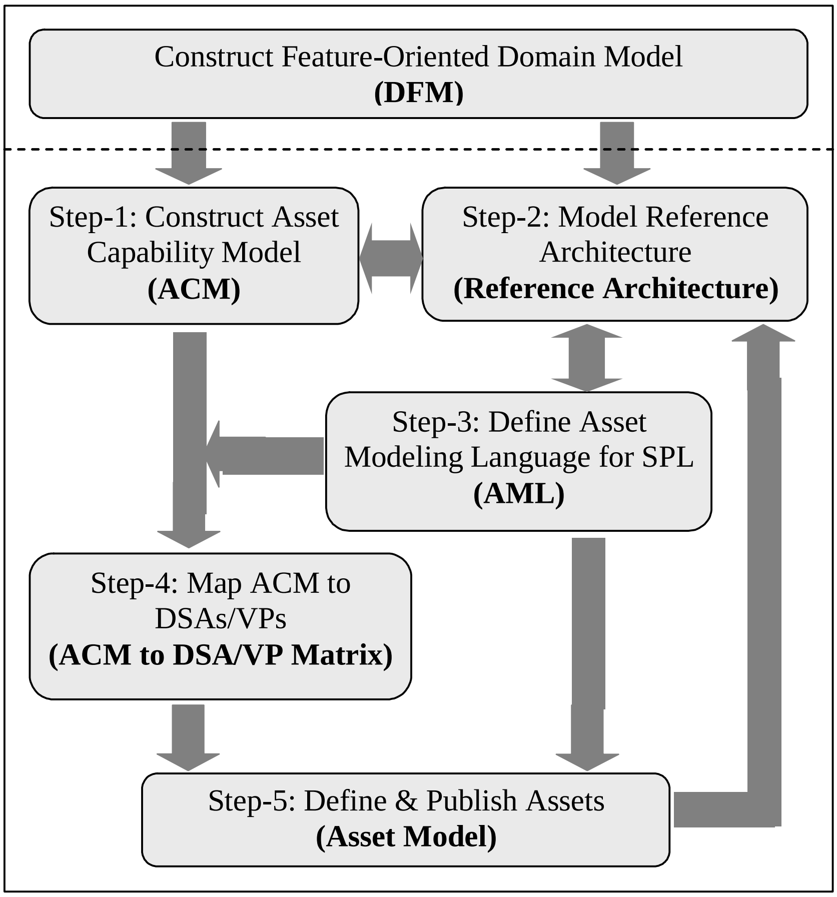

Construct Feature-Oriented Domain Model
(DFM)
Step-1: Construct Asset Capability Model
(ACM)
Step-2: Model Reference Architecture
(Reference Architecture)
Step-3: Define Asset Modeling Language for SPL
(AML)
Step-4: Map ACM to DSAs/VPs
(ACM to DSA/VP Matrix)
Step-5: Define & Publish Assets
(Asset Model)

Figure 5.1: Asset modeling approach

Step–1: Construct ACM.

The first step describes asset capabilities in terms of feature diagrams exposing the structural, functional and behavioral properties, and constraints on them. The requirements of business domain (DFM) has been partitioned into reusable assets, and feature model of each asset is described as Asset Capability Model (ACM), which is the principal output of this step. There is a bi-directional interaction between reference architecture modeling and asset modeling because the identifying DSKs and reusable assets may require several iterations. The output of this step is fed into Step 4 for creating the matrix to map ACM to domain specific artifacts and variability points. Section 5.2 explains the ACM in detail.

Step–2: Model Reference Architecture.

Step 2 is the SPL reference architecture modeling with DSKs as described in Chapter 4.

Modeling the reference architecture with symmetric alignment also yields the contextual information to connect DSEs, which is needed for the interaction and coordination of these DSEs with a standard composition model. Thus, defining DSLs under the supremacy of asset meta-model enables the modeling of artifacts in isolation and facilitates the composition of DSEs within a choreography engine.

In summary, Step 2 models the SPL reference architecture with a set of DSEs, contextual information, constraints for DSL composition, and the accompanying runtime model. In order to illustrate the ideas in this chapter, a high level architecture has been presented in Figure 5.2. It depicts the snapshot of the reference architecture taken from the actual core banking product line of Cybersoft, which underlies the running example in the text. The DSEs of the reference architecture and associated DSATs are described as follows:

- *ERE*: EBML Rendering Engine is used for rendering the rich content on clients. EBML (Enhanced Bean Markup Language) [6, 8] is the DSL for specifying DSATs such as *page*, *region*, and *popup*.
- *LRE*: Listing and Reporting Engine is used to render and print documents. DSAT is *report*.
- *BPE*: Business Process Engine is used for executing business process. DSAT is *process*.
- *BSE*: Business Service Engine is used for executing the business services. DSAT is *service*.
- *BRE*: Business Rule Engine (RUMBA) [33] is used for business rule segregation and execution of other artifacts. DSATs are *rule* and *composite-rule*.

- *PME*: Persistence Management Engine is used for Object-to-Relational (O2R)-mapping of persistent entities. DSAT is *pom (persistent object model).*

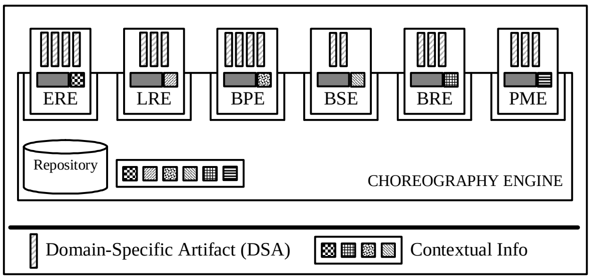


Figure 5.2: SPL reference architecture excerpt

Step–3: Define AML.

Asset Modeling Language (AML) for the target product family is the main output of this step. AML defines the types of artifacts, variability mechanisms, context information, choreography rules, and other constraints of the target domain. An AML is instantiated from Asset Meta Model (AMM) for a product family. Section 5.3 explains the AMM and AML in detail.

Step–4: Map ACM to DSA/VP.

The next step consists of the mapping of features to Domain Specific Artifacts (DSAs) and Variability Points (VPs) under the governance of AML. Actually, AML defines DSATs and variability mechanisms needed in a particular product family; and features of an asset (ACM) can be mapped to multiple DSAs/VPs. The mapping matrix is the output of this step, which will be explained in detail in Section 5.2.

Step–5: Define and Publish Asset Models.

The last step defines and publishes the software assets based on an AML and DSA/VP mappings of the capability model. A published asset model describes all the artifacts, provided variability points, public artifacts, and external (artifact) dependencies. Outputs of this step are the asset models for the product family. This step may loop back to reference architecture modeling because assets are likely to be encapsulated later as DSKs while their maturity improves. Section 5.5 explains the details of asset definitions.

5.2 Asset Capability Model (ACM)

The proposed software asset modeling depends on a conceptual model that treats a software asset as a "set of features and variations". As stated in Section 2.9, a feature can be a structural property, the components of the designed object, a configuration, a set of relationships, a behavior, a function, or a property of a behavior or of a function [25]. Highly complex entities and relationships in software can be synthesized by composing generic and reusable features [15].

A software asset can be modeled using a feature model which is a description of the relevant characteristics of some entity of interest; during domain design they are mapped into artifacts and variability points. Using the feature analysis technique [75] in domain analysis, the proposed approach intends to capture the "capability features" that describe a distinct service, operation, function or structure together with its non-functional properties such as expected response time or scalability concerns; and model them as "Asset Capability Model" (ACM).

Feature diagrams are exploited to model problem domain with the stabilized domain terminology that the users and developers use to communicate their ideas, needs and problems [75]. Hence, they best fit with the domain analysis. However, solution domain is usually expressed in terms of a set of artifacts and variability points to realize such features. Accordingly, Step 4 in the proposed approach builds asset models by mapping features to set of DSAs/VPs in domain design. Thus, the proposed modeling approach has been entitled as "feature-based" rather than "feature-oriented" since features will not be treated as first-class entities right after they have been mapped to DSAs/VPs. Features form the "external view" of assets whereas DSAs/VPs represent the "internal view" within the context of the proposed approach.

Figure 5.3 shows a feature diagram of "Document Manager" asset from the core banking SPL of Cybersoft to be used as a running example throughout this chapter. "Document Manager" has to support three alternative document store types that can be selected by the product developer: File System, Database or XML. It should facilitate several document types; basic actions for document lifecycle management like save, update, retrieve; document category and metadata management; search with options (e.g. keyword or full-text); and several other advanced features.

After having the feature model, the next step is mapping features to DSAs/VPs in

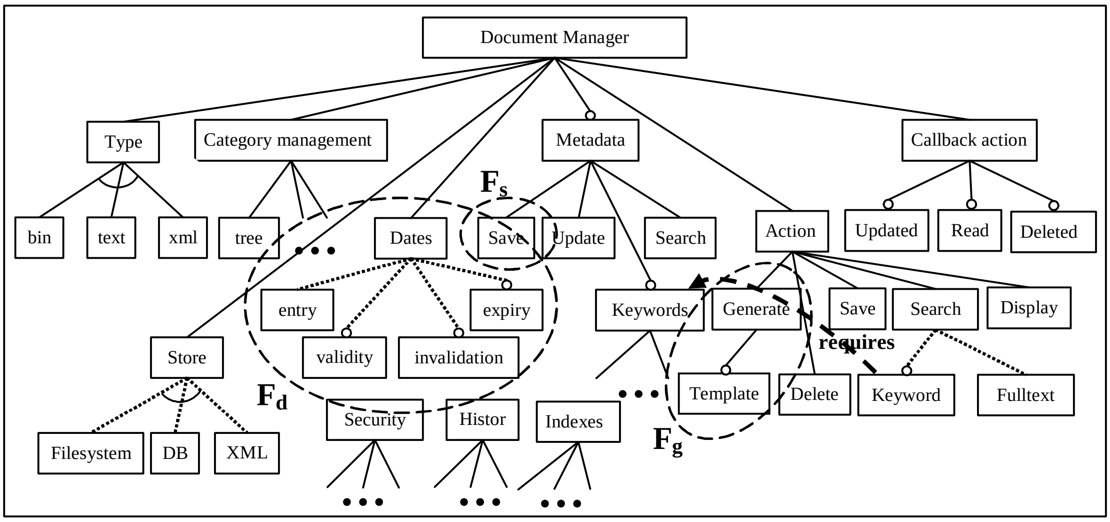


Figure 5.3: ACM for "Document Manager" (partial)

Step–4. A feature may be mapped to multiple DSAs/VPs to indicate that it can be realized throughout the development of that particular set of DSAs/VPs.

Given an asset $[S]$, with a set of features $[F_s]$, a Domain Specific Artifact $[A]$, a Domain Specific Engine $[DSE]$, and a Variability Point $[VP]$; then this software asset can be expressed as follows in the proposed modeling approach:

$$S = \sum_{i=1}^{n} F_i \; where \; F_i \in F_s \tag{5.1}$$

$$F_i = \sum_{j=1}^{t} A_{ij} + \sum_{k=1}^{m} VP_{ik} \; where \; A_{ij} \in DSE_j \; where \; F_i \in F_s \tag{5.2}$$

For example;

$F_1 = \{A_{11}, A_{12}, A_{14}\}$ in domains $\{DSE_1, DSE_2, DSE_4\}$

$F_2 = \{A_{21}, A_{23}\} + \{VP_{21}\}$ in domains $\{DSE_1, DSE_3\}$

Then asset $[S]$ with features $\{F_1, F_2\}$ can be realized using a set of artifacts from the domains $\{DSE_1, DSE_2, DSE_3, DSE_4\}$ with the following DSAs and VPs:

$S = \{F_1, F_2\}$

$S = \{\{A_{11}, A_{21}\}, \{A_{12}\}, \{A_{23}\}, \{A_{14}\}, \{VP_{21}\}\}$

The mapping yields a matrix constructed by putting DSKs in columns and features in rows. VPs are identified in a separate column at the end (see Table 5.1). The cells will be labeled with the names of the artifacts and the name of the variability points if there is a mapping, otherwise it will be left empty. At the end of mapping, each column has the list of artifacts for that particular DSK and list of VPs that asset should support.

Based on the ACM of “Document Manager” in Figure 5.3 and the reference architecture mentioned in Section 5.1, the partial mapping matrix has been presented in Table 5.1. Several feature mappings have been listed below to exemplify the mapping.

Table 5.1: ACM and ACM-to-DSA/VP matrix for “Document Manager” (partial)

Feature	ERE (EBML)	BSE (Service)	BPE (Process)	BRE (Rule)	PME (Persistent Entity)	Variability Point
Doc type					Document POM	vp-doctype (parameter)
Doc store		Action services				vp-store (conf property)
Doc category		Action services			Category POM	
Doc dates					Document POM	vp-dates (parameter)
Save document		save _document		isAuthorized	Document POM	
Retrieve document	Display document region	retrieve _document		isAuthorized	Document POM	
Display document	Display screen	retrieve _document		isAuthorized	Document POM	
Search document	Document search screen	search _document			Document POM, Metadata POM, Keyword POM	vp-search (parameter) variant=keyword requires vp-keyword.
Update document	Approval screens	update _document	Update Approval Process	Approval Rules	Document POM	
Generate document		generate _document			DocumentPOM	vp-template (parameter)
Attach Keywords		Action services			Keyword POM	vp-keyword (conf property)
On document deletion						vp-deleted (artifact substitution)
...						

- “Save document (Fs)” can be realized using a service, a rule for authorization, and a persistent entity (pom) for document information, that is,

$$F_s = \{save\_document, isAuthorized, DocumentPOM\}$$
$$\text{in domains } \{BSE, BRE, PME\}$$

- “Optional document date (Fd)” can be supported by mapping to a pom to store the dates and a VP to turn on/off date tracking, that is,

$$F_d = \{DocumentPOM\} + \{vp - dates\}$$
$$\text{in domain } \{PME\}$$

- “Generate document (Fg)” can be realized using a generation service, a pom for the generated document, and a VP for document template, that is,

$$F_g = \{generate\_document, DocumentPOM\} + \{vp - template\}$$
$$\text{in domains } \{BSE, PME\}$$

Feature-to-artifact mapping can not be guided with generic rules that are independent from the domain, since the selection of DSKs, hence artifacts, is domain specific. Mapping semantics may depend on the feature classes and artifact types for a domain; later mapping rules for a domain can be defined and can be applied to identify artifacts during the design of assets.

5.3 Software Asset Meta Model (AMM)

A software asset model for a product family is not just a collection of domain specific artifacts. An asset definition needs more to compose the internal and external artifacts, to define contextual information, and to manage variability. Conceptually, an asset is a composition of artifacts specified by using different DSLs.

As stated in Chapter 4, SFA architectural modeling identifies the set of DSEs and associated DSLs with the composability rules under a choreography engine which constitutes the SPL reference architecture. Compliant with this architecture, SPL needs a modeling language to define the assets for product assembly. The Asset Modeling Language (AML) of a product family (i.e. distinct SPL) is derived from an XML-based meta-model, namely the Asset Meta Model (AMM).

Figure 5.4 presents the relationship between meta-models of software assets. The AML for an SPL is an instance of AMM; assets are defined using AML of a product family; and finally, executable assets are instantiations of these assets assembled in different products.

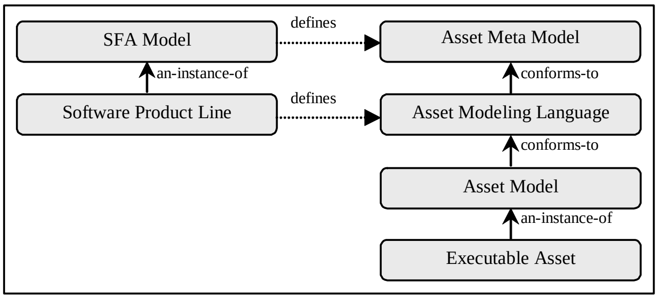


Figure 5.4: Asset meta modeling levels

Table 5.2 presents an analogy of the metamodeling approach with Meta Object Facility (MOF[1]) and Unified Modeling Language (UML). It can be summarized as follows: AMM is a meta-meta-model to define a meta-model (AML) which is used to define assets of a product family. Defining the domain specific artifact types together with their dependencies, choreography rules, and variations constitute an AML for an SPL. AMM also provides proper means to define variability points associated with artifact types and their realizations.

Table 5.2: Analogy between AMM and MOF

Meta Model Level	**Object-Oriented Modeling**	**Software Asset Modeling**
Meta-Meta-Model	Meta Object Facility (MOF)	Asset Meta Model (AMM)
Meta-Model	Unified Modeling Language (UML)	Asset Modeling Language (AML)
Model	Class Model	Asset Model
Instance	Runtime Objects	Executable Assets

AML enables to define reusable software assets based on the artifacts (abstractions) provided by individual DSKs. The common asset definition at a meta-level (AMM) enables the design and cross-utilization of reusable software assets across multiple product lines. Figure 5.5 shows an overview of AMM.

The conceptual building blocks of AMM are described below. The excerpts in Figures 5.6, 5.9 and 5.10 are taken from the core banking asset model. Note that the AML definition is coupled with the reference architecture given in Section 5.1.

- *Domain Specific Artifact Type (DSAT)*: Identifies the type of artifacts that can be built by using a DSK. A specific asset may contain artifacts of these types. For instance, a software asset in the running example may contain DSATs such as *page* described in EBML, *rule* described in RuleML, or *process* described in JPDL (see Figure 5.6).

- *Domain Specific Kit*: An artifact type is bound to a particular DSK. In other words, the artifacts are expressed using a specific DSL. Furthermore, there is an accompanying toolset in the domain specific development environment and a runtime engine to be plugged into choreography engine (see Figure 5.6).

  For instance, if business rule segregation is needed for the product family, then:

[1] http://www.omg.org/technology/documents/formal/mof.htm

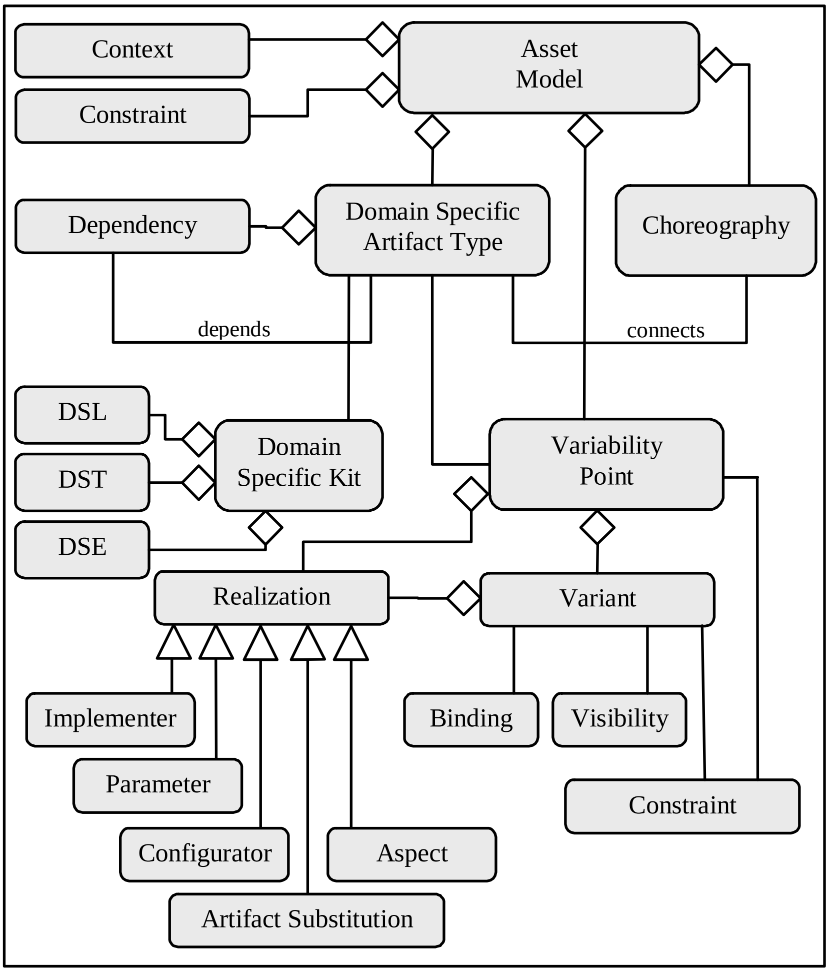


Figure 5.5: Software Asset Meta Model (AMM)

DSL is the rule specification language such as the RuleML,

DSE is a rule inference engine such as the RBE in RUMBA [33] framework,

DST is a related rule definition editor such as the RUMBA RuleEditor.

- *Context*: The contextual information that an asset model depends on comes from the reference architecture and asset modeling activities. The context includes all variables to be shared by the DSEs through a global namespace. This approach has already been followed in many domains, such as banking, ERP, and e-government, for several years as a fundamental mechanism in the Service-Oriented Architecture and the associated Enterprise Service Bus of Cybersoft [6, 8].

```xml
<domain-specific-artifact-types>
   <domain-specific-artifact-type name="page">
      <domain-specific-kit>
         <domain-specific-language name="ebml"/>
         <domain-specific-engine name="ere"/>
         <domain-specific-tool name="eds"/>
      </domain-specific-kit>
      <dependency>
         <reference-type name="process"/>
         <reference-type name="service"/>
         <reference-type name="rule"/>
         <reference-type name="region"/>
         <reference-type name="popup"/>
      </dependency>
   </domain-specific-artifact-type>
   <domain-specific-artifact-type name="region".../>
   <domain-specific-artifact-type name="popup".../>
   <domain-specific-artifact-type name="rule".../>
   <domain-specific-artifact-type name="service">
      <domain-specific-kit>
         <domain-specific-language name="servicexml"/>
         <domain-specific-engine name="bse"/>
         <domain-specific-tool name="ServiceEditor"/>
      </domain-specific-kit>
      <dependency>
         <reference-type name="process"/>
         <reference-type name="rule"/>
         <reference-type name="composite-rule"/>
         <reference-type name="pom"/>
      </dependency>
   </domain-specific-artifact-type>
   <domain-specific-artifact-type name="process".../>
   ...
</domain-specific-artifact-types>
```

Figure 5.6: Defining DSATs and DSKs in AML

The context has not only architectural variables such as `"session_identifier"`, `"session_context"`, `"trans_id"`, `"security_attributes"` or `"user_id"`, but also business domain specific variables like `"institution_code"`, `"branch_code"` or `"customer_id"` of banking. Both architectural and business domain definitions are product line specific extensions, and resulting AML describes them (see Figure 5.7).

- *Constraint*: The product family level constraint type definitions that will be applicable to all assets are also defined by the AML. For instance, there are three constraint types defined in example, and one of them is mandatory indicating that an asset has to declare its context dependency (see Figure 5.8). The others indicate the required assets and excluded assets, however assets might ignore

```
<context>
   <var id="session_identifier"/>
   <var id="session_context">
      <bag id="session_bag">
         <var id="logintime"/>
         <var id="timeout"/>
         ...
      </bag>
   </var>
   <var id="trans_id"/>
   <var id="institution_code"/>
   <var id="branch_code"/>
   <var id="customer_id"/>
   ...
</context>
```

Figure 5.7: Defining context in AML

these definitions.

This part of AMM might be closely related with the choreography language and engine in the future to indicate some directives to the choreographer. Using directives on assembly, usage, deployment and runtime, the choreographer can coordinate the interactions of artifacts.

```
<constraints>
   <constraint-type optional="true">requires</constraint-type>
   <constraint-type optional="true">excludes</constraint-type>
   <constraint-type optional="false">context</constraint-type>
   ...
</constraints>
```

Figure 5.8: Defining constraints in AML

- *Choreography*: The Choreography block in AML defines the terms and conditions for DSAT interactions (see Figure 5.9). A choreography definition indicates that two DSATs may communicate with each other, and parameters like link type, connection type and communication protocol are also specified within this block. For instance, *page* calls (references) a *service*, and also aggregates a *region* as part of the *page* (see Figure 5.9 for the definition of this example).

  The choreography block in this version of AML is in parallel with the composition model described in Section 6.2.2 which is based on Composite Application Framework (WS-CAF) [28].

```
<choreography>
   <interaction source-dsat="page" dest-dsat="service"
         link-type="call" comm-model="sync" .../>
   <interaction source-type="page" dest-type="region"
         link-type="aggregate" comm-model="sync" .../>
   <interaction source-dsat="process" dest-dsat="service"
         link-type="call" comm-model="sync" .../>
   ...
</choreography>
```

Figure 5.9: Defining choreography rules in AML

- *Dependency*: This block indicates that a DSAT may depend on another one. Using this definition, one can elucidate the usage rules of artifact types and their interrelation. For example, "page may access service" needs to be defined here as a dependency so that related asset instances can be constrained during design time, and choreographer can behave accordingly during runtime (see Figure 5.6 for dependency definitions).

- *Variability Point (VP)*: VPs are the locations in an asset that might have a parameter provided or customized by the product developer. Definition of a VP includes variable items, constraints, variants, visibility, binding properties, etc. (See Figure 5.9) A VP at the AML-level is associated with an artifact type, and it indicates that an artifact type can have a variable part that is usually a structural variation. Later, it is left to the asset provider whether a particular artifact needs to support this variation or not. The details of both AML-level and asset-level VP definitions can be found in Section 5.4.

Once these blocks are instantiated based on AMM, the resulting AML will be specialized for an SPL. All reusable coarse-grained assets to take place in a product family can be defined using this AML. Note that different product lines imply the use of different AMLs (modeling languages).

Within the scope of an SPL, software assets are defined using the AML associated with that product family where products are assembled by using these reusable software assets. Managing the variability of assets is of utmost importance for the successful assembly of products out of a core asset base. In the next section, the details of variability point definitions and realizations in the proposed model will be discussed.

5.4 Managing Variability in Software Assets

Management of variations is the key discriminator between conventional software engineering and software product line engineering [83, 109]. Variability points enable the development of products by reusing predefined and adjustable artifacts. The built-in support to define variability points in the proposed approach has been adapted from the Orthogonal Variability Model (OVM) [109]. Variability point definitions identify the following details:

- *Variability Point (VP):* It identifies the name of VP, associated artifact type (or artifact), variants, binding time, visibility and type of realization.
- *Variant:* This determines the list of available alternatives. There might be some constraints attached to each alternative. A VP may have no variants if its variability depends on a parameter value provided by asset user (product developer).
- *Binding:* Binding time of variant to a variability point can be set by the product developer.
- *Visibility:* A variability point may be external (visible to customers) or internal.
- *Constraint:* The constraints on VPs, such as "requires" or "excludes", are added as a part of the asset definition so that the product developer can decide on which VPs to use during product design. Constraints can also be attached to VP variants as they may express a limitation on variant value.
- *Realization:* Realization is associated with a VP or a variant. The asset developer has to decide on the realization mechanism and provide appropriate support accordingly to instantiate the realization mechanism.

The proposed asset modeling approach provides the following realization mechanisms for VPs:

- *Artifact substitution:* This is the most common variability mechanism in the proposed model since DSAs are definitely loosely coupled with each other and the composition mechanism relies on SOA for interaction of domains. The asset developer simply leaves a slot for the product developer to attach another artifact.

The service-to-service or page-to-region substitutions are typical examples of this mechanism.

- *Implementers (Plug-in):* Using proper techniques, such as the bytecode engineering facilities provided by RUMBA [6], the product developer can supply some part of an artifact (e.g. service) even at runtime.

- *Parameters:* These are the code-level parameters that product developer can easily feed during the execution of some artifacts. For example, Business Services Engine may execute services with user supplied parameters.

- *Configurators:* They are actually a predefined set of configuration properties that an asset supports. Configurators are set during asset instantiation by the product developer at the beginning of product development. For example, "Document Manager" can be instantiated with one of three store types, such as DB, XML, or Filesystem.

- *Aspects:* Dynamic composability of basic aspects can be supported at DSK-level. Then, the artifacts of that DSK can benefit from runtime variability mechanism. For example, a rule expression can be configured in RUMBA rule inference engine [33].

An asset developer has to implement variation mechanisms that support the optional and alternative features in ACM. Further variation mechanisms and their basic properties are reported in [9].

A realization mechanism implies different binding times of variability [109]. Different binding times can be specified throughout the asset and product development life cycle: design time, development time, build time, installation time, startup time, and runtime. Not all variability mechanisms can be applied at every point. Binding time selection is critical, since it restricts the applicable variability mechanisms. Deciding an early binding time results in less variability, and late binding brings more complexity. Therefore a careful analysis of the binding flexibility is important, because shift of binding times leads often to change of mechanism [22].

Three VP definitions at AML-level for the running example of this chapter are presented in Figure 5.10. First, a service body can be changed at runtime that is realized by an implementer (Line 2 in in Figure 5.10). Second, datasource of a persistent entity, pom, can be configured at design time by a configurator (static property) (Line 5 in in Figure 5.10). Finally, the report has a VP to select the presentation format during runtime, which is passed as a parameter to report generation (Line 8 in in Figure 5.10). Alternatives are `pdf` and `html` as shown in Figure 5.10.

```
1  <variability-points>
2     <variability-point artifact-type="service"
3        vp-name="body" binding="runtime" visibility="external"
4        vp-key="body" realization="implementer"/>
5     <variability-point artifact-type="pom"
6        vp-name="datasource" binding="designtime" visibility="external"
7        vp-key="datasource" realization="configurator"/>
8     <variability-point artifact-type="report"
9        vp-name="format" binding="runtime"
10       visibility="external" vp-key="format"
11       realization="parameter">
12       <variants>
13          <val>html</val>
14          <val>pdf</val>
15       </variants>
16    </variability-point>
17    ...
18 </variability-points>
```

Figure 5.10: Defining variability points in AML

It is worth to point out that there are two levels of VP definitions: “AML-level” and “asset-level”. A VP is associated with an artifact type or artifact if it is at the AML-level or asset-level, respectively. Realizations at both levels are limited by the mechanisms provided by DSKs and binding time constraints.

5.5 Defining and Publishing Software Assets

The last step in asset modeling process is the definition and publishing of assets. Once DSAs and required VPs are modeled, an asset provider is going to provide the artifacts within that asset, such as a *page* design in EBML or a *process* description in JPDL. Artifacts are developed using the associated DSTs. Assets are defined by using the AML of an SPL, where asset definition includes the followings:

- *Artifacts*: This is the list of artifacts contained in an asset.
- *Public Artifacts*: These are the artifacts that are available to other assets so that they can reference and use them.

- *External Artifacts*: These are the artifacts that an asset needs to use from other assets.

- *Variability Points*: They can be employed to vary products during the product development.

An asset definition for the running example ("Document Manager") has been depicted in Figure 5.11, 5.12 and 5.13. Figure 5.11 presents the definition of artifacts contained in "Document Manager" asset. Public artifacts and external artifacts from other assets, e.g. `GetUserInfo` is provided by "UserManager" asset, have been depicted in Figure 5.12. Note that each artifact has a `<uses>` tag to express its dependency on other DSAs, and `<supports>` tag to indicate that their behavior can be managed by variants of a VP.

```
<asset name="document-manager" asset-meta-model="Banking">
   <artifacts>
      <page name="display_document">
         <uses type="service">retrieve_document</uses>
      </page>
      <region name="display_document_region">
         <uses type="service">retrieve_document</uses>
      </region>
      <service name="save_document">
         <uses type="rule">isAuthorized</uses>
         <uses type="pom">documentPOM</uses>
         <supports-vp name="vp-doctype"/>
      </service>
      <service name="retrieve_document">
         <uses type="rule">isAuthorized</uses>
         <uses type="pom">documentPOM</uses>
      </service>
      <service name="generate_doc">
         <supports-vp name="vp-template"/>
      </service>
      <rule name="isAuthorized">
         <uses type="service">getUserInfo</uses>
      </rule>
      <rule name="isAuthorizedforDeletion"/>
      <pom name="documentPOM">
         <supports-vp name="vp-doctype"/>
      </pom>
      <pom name="metadataPOM"/>
      <pom name="keywordPOM"/>
      ...
   </artifacts>
```

Figure 5.11: Excerpt from "Document Manager" (artifacts)

```
<public-artifacts>
   <page name="display_document"/>
   <region name="display_document_region"/>
   <service name="save_document" vp-key="doctype"/>
   <service name="generate_doc" vp-key="template-name"/>
   <service name="search_document" vp-key="option"/>
   <rule name="isAuthorized"/>
   ...
</public-artifacts>
<external-artifacts>
   <service>getUserInfo</service>
   ...
</external-artifacts>
```

Figure 5.12: Excerpt from “Document Manager” (public and external artifacts)

VP definitions are shown in Figure 5.13. Each identified VP in mapping has been defined with its parameters and variants. The `<vp-key>` tag here enables the product developer to use the asset variability. Note that they are also indicated in `<public-artifacts>` definitions.

The reusability of software assets directly depends on the use of common DSLs for many SPLs. An asset can be reused across different product lines only if DSKs required by the asset specification exist in AMM definition. In other words, such assets can be reused if dependent artifact types are available. The complete version of “Document Manager” asset has been used in two distinct product families, in banking and ERP domains. For the former, it is used to generate and store the statements of transactions; and for the latter it is used to generate and save invoice/receipt documents.

5.6 Using Software Assets

This section aims to explain briefly the use of assets during product assembly without going into the details of choreography. In order to illustrate better, another asset, “Alert and Notification Manager” (ANM), has been modeled by using the same approach. ANM manages end user notifications as their requests or jobs are completed successfully or abnormally. It also signals alerts through several channels if some rules are satisfied or a timeout occurs, etc.

The instantiation of assets are carried out by means of an XML configuration file. Using a very simplified instantiation script, the example in Figure 5.14 shows how these two assets can be linked to provide an alert when a document is deleted.

```
<variability-points>
   <variability-point name="vp-doctype"
      binding="runtime" visibility="external"
      realization="parameter" vp-key="doctype">
      <variants>
         <val>bin</val>
         <val>text</val>
         <val>xml</val>
      </variants>
   </variability-point>
   <variability-point name="vp-store"
      binding="designtime" visibility="external"
      realization="configurator" vp-key="store">
      <variants>
         <val>filesystem</val>
         <val>db</val>
         <val>xml</val>
      </variants>
   </variability-point>
   <variability-point name="vp-search"
      binding="runtime" visibility="external"
      realization="parameter" vp-key="option">
      <variants>
         <val requires-vp="vp-keyword">keyword</val>
         <val>fulltext</val>
      </variants>
   </variability-point>
   <variability-point name="vp-template"
      binding="runtime" visibility="external"
      realization="parameter"
      vp-key="template-name"> // no variant
   </variability-point>
   <variability-point name="vp-updated" .../>
   <variability-point name="vp-deleted" .../>
   ...
</variability-points>
```

Figure 5.13: Excerpt from “Document Manager” (variability points)

The anm_alert and anm_trace services of ANM use the settings in <messages> and <receivers> tags in the Configurator. The onDelete key of VP (vp-deleted) in “Document Manager” is subscribed to anm_alert service (see Line 7 in Figure 5.14). When a document is deleted, the document manager triggers anm_alert to run. The artifacts communicate through a common context which is also used to manage the global state and parameter passing during choreography.

5.7 Remarks

Feature orientation has been researched widely for managing the requirements and variability in product family development [18, 47, 91], and feature models have been

```
<sfa-init main="PhD2007">
   <asset name="document-manager">
      <configurators>
         <store>filesystem</store>
         <keyword>on</keyword>
         <onUpdate>anm_trace</onUpdate>
         <onDelete>anm_alert</onDelete>
         ...
      </configurators>
   </asset>

   <asset name="anm">
      <configurators>
         <channel>mail</channel>
         <messages>...</messages>
         <receivers>...</receivers>
         ...
      </configurators>
   </asset>
</sfa-init>
```

Figure 5.14: Instantiating software assets

used in conjunction with Object-Oriented Programming (OOP) [15], Aspect-Oriented Programming (AOP) [92], and Generative Programming (GP) [45] models. The proposed model, in this respect, differs from the previous models by employing domain specific abstractions for the realization of features.

The model presented here does not prescribe any of the mentioned programming models; rather DSKs try to abstract them as much as possible. This approach separates asset concerns by mapping features to DSAs, and later composes them using the SPL reference architecture. Hence, it generates more cohesive asset models to improve the asset reusability by reducing the interdependencies.

FORM [75] extends FODA [74] for modeling the reference architecture and makes use of object-oriented engineering with the feature categorization at different layers. However, the crosscutting relations at multiple feature models may dramatically increase complexity as the number of features grows [17]. Mapping the features into Domain Specific Artifacts aims to reduce such complexities, but requires effective mechanisms for realizing variability during composition. Built-in realization mechanisms (configurators, implementers and aspects with runtime bindings) for VPs enable the variability management with deferred encapsulations, which is also needed in ambient applications [60, 91].

The use of Domain Specific Languages decreases the number of DSAs since DSLs define artifact types in higher abstractions compared to OO and AO models. Though the design of DSLs is non-trivial, the design process can benefit from the SPL approach itself [134]. Besides, conceptual model of AMM guides the design of DSLs for supporting the capabilities (e.g. context, constraints, variability points, etc.) needed to model reusable artifacts which take part in asset models.

The proposed model also facilitates the reusability of software assets on multiple product lines as long as the common DSKs and contextual constraints are valid for them. Asset models and the artifacts can be reused; and this eliminates the redundancies and possibility of inconsistencies in feature models if they are used in multiple product lines [27].

SFA model is similar to XML-based feature modeling of [29] in terms of defining a product family and assets using meta-models. However, while [29] relies on XML-based generative technologies, SFA puts explicit focus on variability management as the first-level aspect of the model.

The asset models can be stored and searched in reusable asset libraries such as Reusable Asset Specification (RAS) [112] repositories. RAS describes the structure and nature of assets with their classification, solution, usage and related assets. It requires tool support to search, locate and decide which asset to use in product development. DSTs in the proposed model are also responsible for such issues. As assets are kept in domain specific repositories, DSTs may empower the process of searching, locating and deciding in asset repositories.

The asset terminology of the proposed model is similar to Larsen's study [88] in Section 2.7. Although his asset definitions are broad and general, Larsen adopts "models" as reusable assets and follows Model-Driven Development and uses ABD to complement it. In Larsen's study, UML is used to specify components and systems; MDA [99] specifies model organization for business-driven component architectures; and RAS [112] packages patterns, components, and other artifacts as assets to leverage the business.

CHAPTER 6

EXPERIMENTATION AND VALIDATION

The evaluation of the study from different viewpoints has been discussed here with a series of validation efforts. For the validation purposes, there are two case studies modeled using the proposed approach. The results have been validated with respect to the original problem definition and research questions introduced in Chapter 1.

After presenting the product line models for Investment Banking (INV) and Financial Gateways (FGW), the reusability of software assets and Domain Specific Kits will be discussed with the data collected from the case studies. Then the achieved quality improvements and evaluation of basic requirements of software factory will be discussed; later comparison of the approach with other product line approaches will be carried out. Finally, it will be briefly discussed how the concept of DSKs has been applied to design of another domain, legacy migration to service-oriented environment.

The cases introduced in this chapter have been partly designed and developed by the proposed approach in real life. Based on the benefits exploited so far, they are planned to be fully transformed in the near future.

6.1 Defining the Scope of Example Domains

Determining the scope of product line is an essential activity. The product line scope is a description of the products that will constitute the product line or that the product line is capable of including [103]. At its simplest, scope may consist of an enumerated list of product names. More typically, the things that the products all have in common and the ways in which they vary from one another are described. A scope definition might include features or operations they provide, performance or other quality attributes they exhibit, platforms on which they run, and so on.

Setting the scope and setting the product line requirements seem like similar activities. Clements makes a clear statement of the distinction between product line scope and requirements [37]. A completely precise scope is, in fact, a requirements specification for the product line. In practice, however, scoping and requirements engineering are done by different people, stop at different points, and are used for different purposes by different stakeholders.

The main goal here is neither to define the formal scope of the example domains nor to specify the complete requirements. An overview of the domains is explained to give a brief understanding of the cases.

6.1.1 Investment Banking (INV)

The first case study is the product family for investment banking. The scope of the product line is defined by listing the members of the family and determining the common and variable features of the products. Members of the product family have been determined as follows (the acronyms in parenthesis indicate the product code valid for the rest of the discussion in this chapter):

- *Fixed Income Securities (FIS):* This product supports all buy/sell, repo/reverse repo, security transfer, auction, physical delivery, and money market preparations of fixed income securities. The supported security types are Bonds, Foreign or Local T-Bills, Eurobonds, and other commercial papers.

- *Mutual Funds (FND):* A mutual fund is a form of collective investment that pools money from many investors (customers) and invests their money in stocks, bonds, short-term money market instruments, and/or other securities. This product is used to manage the customer orders, buy/sell, settlement, and transfer operations of mutual funds issued by local or foreign financial institutions. Informed or uninformed Mutual Funds can be of any currency units. This product supports the mutual fund operations from the viewpoint of a retail banking.

- *Equities (EQT):* This product supports all order management, buy/sell and transfer operations, stock management, public offerings, capital increases, custody operations, and on-demand credit management. All domestic and foreign equities are within the scope of this product.

- *Derivatives Exchange (DEX):* This product is used to manage the forward and options operations of customers in the derivatives exchange market. Contract management, order processing, and guarantees are basic process groups needed.

- *Fund Management (FDM):* Fund management product supports the life cycle of mutual funds from the viewpoint of fund managers. In a mutual fund, the fund manager trades the fund's underlying securities, realizing capital gains or losses, and collects the dividend or interest income, and calculates the value of a share. The investment proceeds are then passed along to the individual investors. It provides a back-end system for asset/stock operations and plan executions and ongoing monitoring of investments.

- *Portfolio Management (PRT):* This is similar to the fund management; but the money is not pooled as in the case of mutual funds but executed on behalf of the investor as a private banking service. Both fund and portfolio management products have similar back-end operations.

After identifying the members of the product family, common and variable requirements of the product line have been determined as follows:

(1) All the products have a set of definitions that is the basis of common operations.

(2) All products are required to be integrated and co-work with existing third party customer, accounts and accounting products of the customers.

(3) All products depend on a common organization and authorization module.

(4) Price bargaining between Central Treasury Office and branches or agent offices is needed for all FIS operations.

(5) Flexibility in pricing policy is required based on the channel, customer type, etc.

(6) End-of-day process is very common for all type of financial products, therefore the products need to provide a batch/scheduled job management environment.

(7) Rediscount is a fundamental process during the end-of-day operations in financial systems. Rediscount accounting with parametric formulas is required.

(8) All products need a blacklist module to manage and prevent those customers that exist in the blacklists.

(9) Typical financial operations require approval in the form of Maker-Checker or multi-step approvals. They also need dynamic validation and check services.

(10) The operations need common controls like time limitations, amount limitations based on user roles, the input document requirements, etc.

(11) The deductions (commission, tax, expense, etc.) of operations need to be managed dynamically, and they can be flexible based on branch, instrument type, customer group, etc.

(12) The output documents are required to be saved in printable formats.

(13) All products need a common logging mechanism.

(14) Each product has its own core business functionality, set of operations, and user requirements.

(15) Products may utilize a common asset definition with common market closing price, common statements of account, unique overall portfolio report, common stock reports, and common operation reports.

(16) Common asset definition enables common primitive asset management operations: asset transaction definitions, stock and balance structures, different asset costing methods (detail, average), different asset stock decreasing methods (FIFO, LIFO, Min tax, Min cost, Max cost), taxation methods, and closing the day.

(17) Common asset definitions and common primitive operations are required to be tuned for different instruments and for different installations.

6.1.2 Financial Gateways (FGW)

In defining the scope of the second case study, the same approach will be followed. The scope is defined by listing the members of the family and determining the common and variable features of the products. The construction of this product family has been partly supported by Tübitak TEYDEB (OCTOPODA Project, ProjectNo: 3060543). An overview of the product line has been depicted in Figure 6.1. The model used here is a simplified version of the original product line.

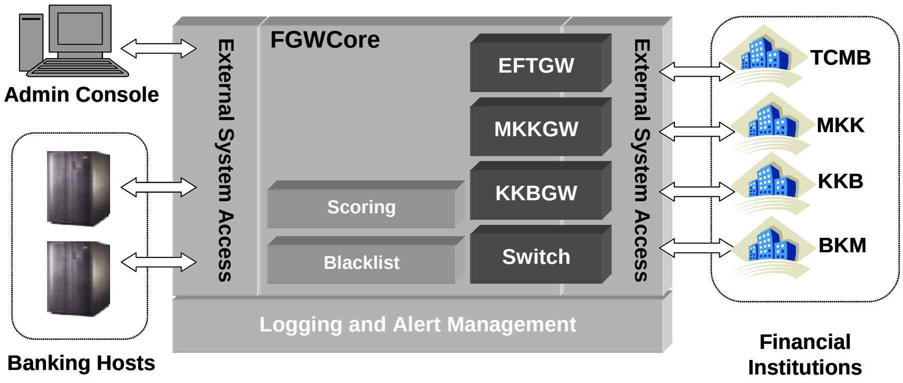


Figure 6.1: OCTOPODA financial gateways product family overview

Members of the product family have been determined as follows (the acronyms in parenthesis indicate the product code valid for the rest of the discussion in this chapter):

- *EFT Gateway (EFT):* This is a gateway to Central Bank of the Republic of Turkey (TCMB[1]) for electronic fund transfer services. TCMB integration is achieved through a custom protocol (Host Link Protocol – HLP). The product is required to support EFT operations at the same time. Effective means of logging and alerting are critical, while integration with the value-added services such as blacklist management and scoring is a distinguishing feature of the product.

- *CRA Gateway (CRA):* This is a gateway to Central Registry Agency of Turkey (CRA[2]) for electronic registration of securities. This gateway requires the use of either Web service or message queue based integration. The message format is same for both cases, and signature-based security policy is applied. All messages must be signed by the users and saved by the system. The requests are processed asynchronously, therefore collecting further notifications is mandatory. Again,

[1] http://www.tcmb.gov.tr/

[2] http://www.mkk.com.tr/

integration with logging, alerting, and blacklist management is a distinguishing feature.

- *Credit Bureau Gateway (CRB):* This is a gateway to Credit Bureau of Turkey (KKB[3]) for accessing to a comprehensive picture of consumer, and such information is used in the assessment of consumer credit worthiness and related credit granting purposes by the banks and other consumer finance institutions. The system returns all information extracted from the database for the applicant (consumer) being searched, enabling the credit risk of all parties to be assessed. The primary challenge of the gateway is KKB host systems are running on mainframe and accessed via mainframe specific protocols. Integrating with the local caching mechanism is a distinguishing feature for this product.

After identifying the members of the product family, common and variable requirements of the product line have been determined as follows:

(1) All products need a common infrastructure for service-based integration, user authorization and session management.

(2) A common admin console is required for ease-of-administration.

(3) All products need a common external access layer for integration to both internal and external hosts (B2B integration). This is required to be dynamically configurable, to support different communication protocols, and to give service in different modes such as synchronous, asynchronous, etc.

(4) A common security mechanism is required and on-demand customizations have to be supported additionally.

(5) Custom communication protocols may be built and integrated for some products (e.g. EFT requires the use of custom protocol – HLP)

(6) EFT and CRA gateways have to be integrated with blacklist management module to prevent the processing of incoming/outgoing messages from the individuals on blacklists.

[3] http://www.kkb.com.tr/

(7) All system logs including the operational and financial audit logs have to be recorded.

(8) Gateways may require the use of notification system for generating system alerts.

(9) The system alerts must be directed to SMS, e-mail, and other channels.

(10) Integrated log and alert system can produce rule-based alerts with “Event Mining” and “Event Sequence Matching” algorithms.

(11) Blacklists are coming from different sources with different formats and data models. Dynamic usage and matching rules of blacklists with different products are fundamental requirements (applying “Record Matching” and “Data Linkage” algorithms). Additional support for the application of mining algorithms on these lists will be an important value-add.

In the next section, the required DSKs and reference architectures of both INV and FGW domains have been discussed.

6.2 Reference Architectures

The discussion of reference architectures for case studies has been organized in two steps: first, the required common DSKs for the case studies have been described, then the reference architectures for INV and FGW domains are depicted.

6.2.1 Domain Specific Kits for Case Studies

The list of DSKs that are used in the examples and their descriptions are given in Table 6.1. Then, for each DSKs, the DSL description, the artifact types and descriptions, the development environment (DST), the runtime engine (DSE), and other details are defined in this subsection.

6.2.1.1 RIA Presentation Kit

Presentation tier is one of the most tedious and error-prone parts of the Web application development. HTML-based applications have become very popular so far because cost of deployment is low, architecture is relatively simple, and HTML is trivially easy

Table 6.1: List of DSKs used in the case studies

Name	Description/Purpose
RIA Presentation Kit	Business domain independent XML-based technology to be used for power screen design in Internet applications [8].
Reporting Kit	Business domain independent XML-based technology to be used for report content generation, rendering and presentation in Internet applications.
Business Services Kit	A lightweight kit for development, publishing, administration of business services with a registry, repository, meta-model and policy management services [8].
RUMBA Business Rules Kit	Business domain independent Aspect-Oriented kit for business rules segregation where all aspects, facts, rules and rule-sets can be defined and managed dynamically by means of a GUI console [33].
BPM Kit	A jBPM [71]-based kit for business process management (BPM) providing design, development and execution of business processes.
Persistence (POM) Kit	An XML-based Object-to-Relational (O2R) mapping kit for defining, deploying and executing SQL queries by mapping to Plain Old Java Objects (POJOs) [8].
Batch Processing Kit	A special purpose kit for defining, scheduling and execution of batch jobs with enterprise-class features, such as transactions and clustering.

to learn and use. But the benefits of being Web-based outweighed the loss of significant user interface (UI) functionality since Web (HTML) has originally been designed for publishing. Consequently, certain application domains do not fit to the limited capabilities of HTML. As a reflex to the limitations of HTML, *Rich Internet Application (RIA)* concept introduced the client-side rendering approach that can present very dense, responsive, and graphically rich user interfaces [104, 66, 101]. It combines best of the desktop, Web, and communications [51].

EBML (Enhance-Bean Markup Language) [6, 8] has been derived from User Interface Markup Language (UIML[4]) [108], which is an open standard user interface description language in XML. The motivation of UIML is to facilitate better tools for creation of user interfaces that work on any platform available today. From the point

[4] UIML, http://www.uiml.org/

of multi-tier Web architecture, UIML only describes the presentation layer. The original UIML specification is not optimized for fast rendering, its markup requirements are high, and it has little help on advanced UI widgets like tabbed panes, in-cell editable grids, tree tables (explorer like screens) and popup querying controls. Therefore, the original UIML concept has been tailored for Web presentation, empowered with pluggable widget technology, enriched with advanced screen widgets, and the resulting markup language has been named as Enhanced Bean Markup Language (EBML).

EBML is capable of expressing reusable screen regions, defining sanity checks as well as arithmetic and logical expressions, executing local and remote method calls, versioning and caching structural parts separately, dealing with static reference data dynamically, managing the client context, and supporting localization and internationalization without extra client-side coding [8].

A generic Java applet has been implemented as the rendering engine, called as EBML Rendering Engine (ERE). Similar to Web Browsers rendering HTML content, ERE is a generic program that can render different screens as far as they are expressed in terms of EBML, thus it can thoroughly provide the client/server UI functionality to end users. By separating screen layouts (including reusable regional parts) from the actual data and method calls, an effective client side caching is possible.

Rendering capabilities have been conceptualized as a DSK in Table 6.2. EBML (Enhance-Bean Markup Language), as a DSL, is a generic markup language to describe the structure of user screens and their behavior. ERE (EBML Rendering Engine) is a Domain Specific Engine to interpret the EBML, which both renders and manages the user screens. Finally, EDS (EBML Development Studio), as a DST, is a complete development and test tool for the user interface developers. *Page*, *Region*, and *Popup* are the artifact types that can be defined by EBML. Excerpts from a sample EBML file, a screenshot of EDS, and an example screen rendered by ERE have been included in Appendix C.

Table 6.2: RIA Presentation Kit

DSL	EBML (Enhance-Bean Markup Language)
DST	EDS (EBML Development Studio)
DSE	ERE (EBML Rendering Engine)
DSAT	*Page*, *Region*, and *Popup*

6.2.1.2 Reporting Kit

Enterprise Web applications require effective reporting and listing capabilities. Generating and rendering reports, supporting several reporting formats (PDF, Word, XML, HTML, etc.), and the print management are some of the key requirements.

The Reporting Kit is based on JasperReports[5] which is a powerful open source reporting tool that has the ability to deliver rich content onto the screen, to the printer or into PDF, HTML, XLS, CSV and XML files. It has been written in Java and can be used in a variety of Java enabled applications, including Java EE or Web applications, to generate the dynamic content. Main goal of JasperReport is to help creating page-oriented ready-to-print documents.

The reports are described as XML files (JRXML) which is later uploaded, compiled and deployed to the report server. The structure of JRXML is provided as a Document Type Definition (DTD) file supplied with the JasperReports engine. There is an open source visual report builder/designer for JasperReports, iReport[6] and there is also a built-in Swing viewer. iReport allows users to visually edit complex reports with charts, images, and subreports. The data to print can be retrieved through several ways including multiple JDBC connections, TableModels, JavaBeans, XML, Multidimensional Expressions (MDX), EJBQL[7], Hibernate[8], etc. In the example architecture, the data has been served with an included Content Manager.

Table 6.3 summarizes the reporting kit. JRXML, as a DSL, is a generic markup language to describe the structure of reports. JasperReports is a Domain Specific Engine to process report definitions as well as to generate and print reports. Finally, iReport and Swing-based viewer, as a DST, is a complete development and test tool for the report developers. *Report* is the artifact type that can be defined by JRXML. An excerpt from JRXML, a screenshot of iReport, and an example report have been included in Appendix C.

6.2.1.3 Business Services Kit

Service is a piece of software that can be reused across the enterprise, in the context of

[5] http://jasperforge.org/sf/projects/jasperreports

[6] http://jasperforge.org/sf/projects/ireport

[7] EJB QL, http://java.sun.com/developer/technicalArticles/ebeans/ejb20/index.html

[8] www.hibernate.org

Table 6.3: Reporting Kit

DSL	JRXML
DST	iReport and Swing-based viewer
DSE	JasperReports
DSAT	*Report*

many business processes or subprocesses, consisting of an interface, implementation, contract, and data [16]. A contract consists of the global constraints that component will maintain (invariants); the constraints that need to be met by the client (preconditions); and the constraints that component promises to establish in return (postconditions) [8].

A clear separation of business logic from the presentation, business rules and persistence results in improved abstraction, and hence, reuse of the services. The component-based approach enables partitioning the business logic into proper components (set of artifacts) in such a way both to maximize intra-component relations (coherency) and minimize inter-component interactions (coupling) [124].

Business Services Kit is a lightweight framework for development, publishing, and administration of business services with a registry, repository, meta-model and policy management services [8]. As summarized in Table 6.4, services are expressed as XML definitions (ServiceXML) and they are also registered into a repository through a database schema. Service Executor Runtime is an engine to invoke services. The services can be defined and developed using a Service Editor and Eclipse IDE. *Service* is the artifact type that can be defined by XML, and it is connected to a piece of executable code and accessible via different protocols. An example ServiceXML definition and a screenshot of Service Editor have been included in Appendix C.

Table 6.4: Business Services Kit

DSL	ServiceXML
DST	Service Editor and Eclipse IDE
DSE	Service Services Engine
DSAT	*Service*

6.2.1.4 BPM Kit

Business Process Management (BPM) is an emerging technology that organizes the flow of business processes in terms of workflows, rules, and other business entities for improving the efficiency of processes as they are defined, executed, managed and changed. In this respect, a business process is defined as inclusive and dynamically coordinated set of collaborative/transactional activities [119].

There are many different vendors and BPM languages, such as BPEL, BPELJ, BPML, ebXML's BPSS, WSCI and WfMC's XPDL. An open source business process model, JBoss's jBPM [71], has been selected for the case study. jBPM enables flexibility by supporting multiple-process languages with the same scalable process engine platform. JBoss jBPM's pluggable architecture is extensible and customizable on every level, therefore it is very suitable for the DSK extension. Complying with DSK abstraction, jBPM has three subcomponents: a process engine, process monitor, and a process language.

Table 6.5 summarizes the BPM kit. jBPM Process Definition Language (JPDL), as a DSL, is an XML-based language to describe the business processes. jBPM has a process engine that keeps track of the states and variables of all active processes, and provides a communication infrastructure that forwards tasks to appropriate process, user or application. GPD (Graphical Process Designer), as a DST, is a complete business process design environment. *Process* is the artifact type that can be defined by JPDL. An excerpt from an example JDPL process definition, a screenshot of GPD and a snapshot of a process flow have been included in Appendix C.

Table 6.5: Business Process Management Kit

DSL	JPDL (jBPM Process Definition Language)
DST	GPD (Graphical Process Designer)
DSE	jBPM
DSAT	*Process*

6.2.1.5 RUMBA Business Rules Kit

Almost at every tier of enterprise application, business rules crosscut several parts of process management such as workflows, task assignments, and business transactions.

Managing business rules on its own hence improves the dynamism of processes in the sense of modeling, implementing, executing, and even maintenance [33].

In [33], a taxonomy has been presented for the separation of business rules crosscutting the BPM. It has specified the process management as an orthogonal model to architectural tiers of enterprise applications and classified the business rules according to this orthogonal model. Accordingly, business rules can be classified based on content, orchestration, workflow, operation, task, transaction, service, and domain. Hence, segregation and management of business rules in isolation enables a better treatment of complexity, criticality, frequency of change, order of execution, type of access, and responsibility issues.

Table 6.6 summarizes the business rule management kit. RuleML, as a DSL, is a generic markup language to describe the business rules. RUMBA is a Domain Specific Engine to execute the business rules. Finally, there is a set of management screens to define and deploy business rules. *Rule* and *composite-rule* are the artifact types that can be defined by RuleML. A screenshot of RUMBA RuleEditor have been included in Appendix C.

Table 6.6: RUMBA Business Rules Kit

DSL	RuleML
DST	RUMBA Design Environment
DSE	RUMBA Runtime
DSAT	*Rule* and *Composite-Rule*

6.2.1.6 Persistence (POM) Kit

Persistence, as a fundamental mechanism for an enterprise application, has been abstracted a long time ago with Object-to-Relational (O2R) Mapping frameworks. These frameworks have significantly reduced the amount of code to access a relational database, and they have also supported object caching and object-oriented idioms. Several O2R mapping frameworks have been proposed and they have been in use in many enterprise applications, even the same framework is available in both Java or .NET environments. Therefore, there may be several DSKs defined based on

Hibernate[9], iBatis[10], and many others. The choice in the example is to build a DSK based on an in-house developed persistence mechanism, POM [8].

As summarized in Table 6.7, PomXML (Persistent Object Model XML), as a DSL, is a generic markup language to describe the SQL mappings of persistent entities. The PomXML enables mapping tables, stored procedures, functions, views and queries to Plain Old Java Objects (POJOs); it is used to generate the database scripts, associated DDL schema and related interface classes automatically based on these definitions. These mappings and generations are performed by means of POM Studio (GUI environment to manage POM definitions), as a DST. POM has a runtime engine to serve the relational queries, and it has been extended to support object caching. *Pom* is the artifact type that can be defined by PomXML. An excerpt from an example PomXML definition and a screenshot of POM Studio have been included in Appendix C.

Table 6.7: Persistence Kit

DSL	PomXML
DST	POM Studio
DSE	POM Runtime
DSAT	*POM (Persistent Object Model)*

6.2.1.7 Batch Processing Kit

Batch processing, as an integral part of an enterprise application, is the execution of a series of *jobs* on a computer without human intervention whenever they are scheduled or triggered by an event. All input data is preselected through scripts or command-line parameters. Typical core banking system has end-of-day jobs for execution of the operations such as credit pay backs, rediscounts, creating balance sheets, etc.

A special purpose kit has been developed for scheduling and execution of batch jobs. Batch Processing Kit has been developed based on Quartz[11] job scheduling system that can be integrated with or used along side virtually any Java EE application. It is used to create complex schedules for executing jobs whose tasks are defined as *services*.

As summarized in Table 6.8, a repository of batch jobs has been set with their

[9] www.hibernate.org

[10] http://ibatis.apache.org/

[11] http://www.opensymphony.com/quartz/

parameters, scheduling rules, execution dependencies, etc. This information has been defined using an XML file. job.xml, as a DSL, is a generic markup language to describe batch or scheduled jobs. Job Management Console, as a DST, is a Web application for registering, management and monitoring of batch jobs. Quartz Job Scheduler is an engine to execute batch jobs with the provided parameters at scheduled times. *Job* is the artifact type that can be defined by XML. An example batch job definition and a screenshot of job management have been included in Appendix C.

Table 6.8: Batch Processing Kit

DSL	Job.xml
DST	Job Managemenet Console
DSE	Quartz Enterprise Job Scheduler
DSAT	*Job*

6.2.2 Reference Architectures of The Product Lines

Product line reference architecture of Investment Banking (INV) domain has been constructed with the roadmap presented in Chapter 4. Details of the construction are omitted for the compactness of discussion. Figure 6.2 depicts a simplified version of the reference architecture. The choreographer in this drawing has been added to the SAAM lexicon (presented in Figure 4.5). The architecture of INV domain utilizes all seven DSKs defined in Section 6.2.1.

Similarly, the reference architecture of FGW product line has been depicted in Figure 6.3. The FGW domain utilizes only four of the DSKs given in previous section: RIA Presentation, Reporting Kit, Business Services Kit, Persistence (POM) Kit. Due to the nature of FGW domain, some other architectural elements for external connections have been employed in this architecture.

The choreographer model in these case studies has been designed in compliance with the *Web Services Composite Application Framework (WS-CAF)* [28]. WS-CAF is divided into three parts:

1. Web Service Context (WS-CTX) is a lightweight framework for simple context management that ensures all Web services participating in an activity share a common context and can exchange information about a common outcome.

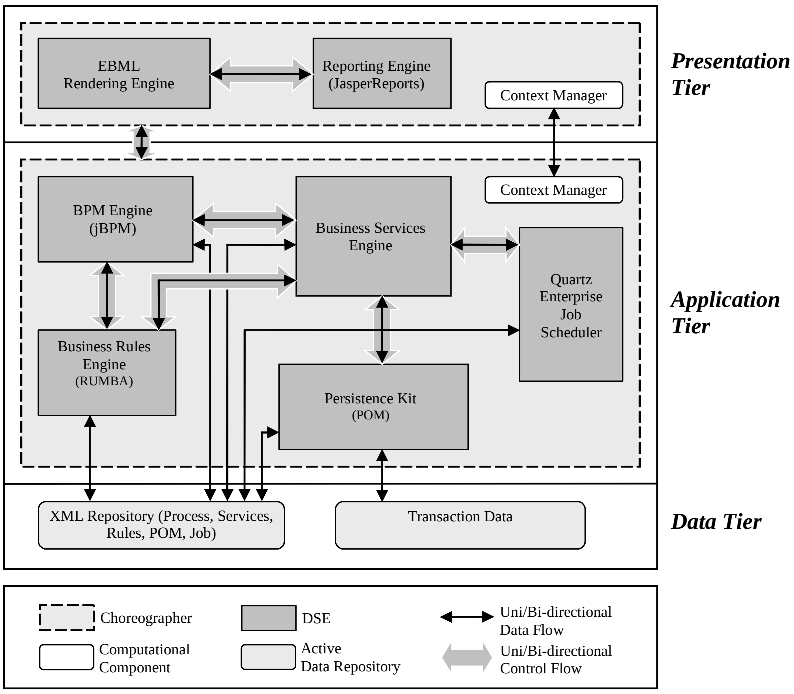


Figure 6.2: Reference architecture of INV product line (simplified)

2. Web Service Coordination Framework (WS-CF) builds on WS-CTX and provides a sharable mechanism to manage context augmentation and lifecycle.

3. Web Services Transaction Management (WS-TXM) builds on WS-CF and defines three distinct transaction protocols for interoperability across existing transaction managers, for long running compensations, and for asynchronous business process flows.

In compliant with this general approach, the choreographer handles the context management, coordination, and transaction management of all parties (DSEs). The context includes the following information:

- session identifier and context,
- security attributes,
- transaction identifier and context,
- client identifier,
- business domain specific identifiers, such as "branch_code" or "customer_id" for investment banking.

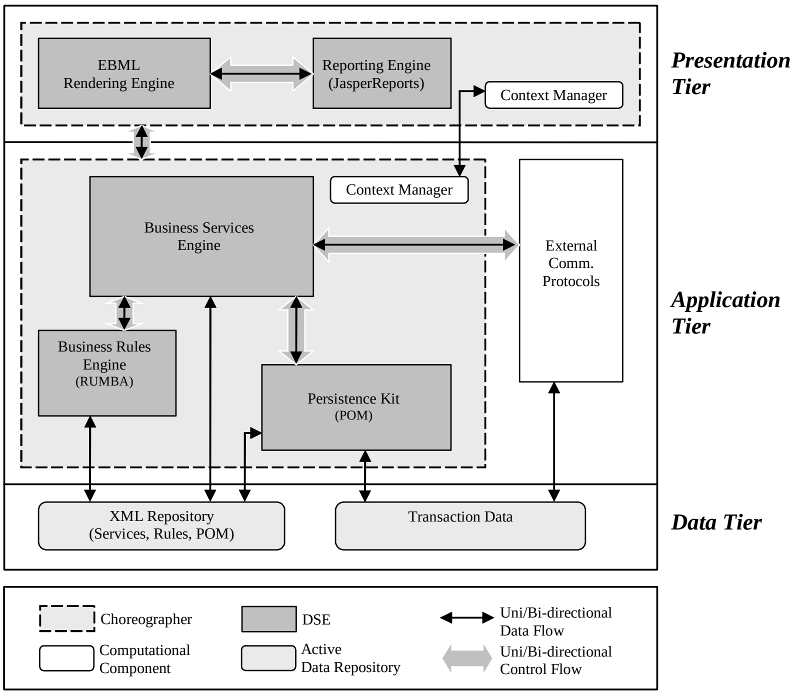


Figure 6.3: Reference architecture of FGW product line (simplified)

The interaction model of composition relies on SOA as a paradigm for managing resources, describing process steps, and capturing interactions between an artifact and its environment. Therefore, a uniform service (*Call*) has been designed with uniform interfaces. The *Call* has a uniform interface, called as *Bag*, which is a lightweight hierarchical container with efficient impose/expose mechanisms and fast-access methods. Each asset provides a known set of artifacts in terms of "named calls" together with specified input and output sets. Name of the call as well as its input and output parameters are all predefined and controlled by the choreographer. (See Line 15 and 25 at Figure C.2 in Appendix C.)

Figure 6.4 shows all artifact types and their dependencies. The dependency graph points out that "the artifact type A depends on (or calls) artifact B if there is an arc between A and B". For instance, a *process* may call a *service*, a *rule* or a *composite-rule*; and it may be called from a *service*, a *page* or a *region*. Since FGW reference architecture utilizes only four of the DSKs (RIA Presentation, Reporting Kit, Business Services Kit, Persistence (POM) Kit), it has a reduced set of artifact types (only those that are marked with "*").

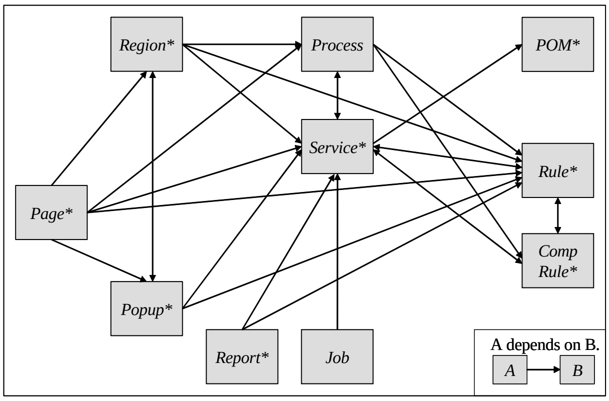


Figure 6.4: Dependency of artifact types in case studies

6.3 Asset Models

The results of asset modeling for case studies have been presented here. The individual steps of asset modeling have been omitted in order to highlight the results of the case studies. The discussion of reusable assets of INV and FGW domains has been presented and their brief descriptions are given in Appendix D.

Based on the reference architectures of domains, SFA approach builds a modeling language to define the assets for product assembly. An AML example with similar DSKs has already been given in Chapter 5, which is similar to the Asset Modeling Languages (AMLs) of INV and FGW product lines. AML defines the domain specific artifact types and their dependencies given in Figure 6.4.

Following the roadmap given in Chapter 5, assets of the product families have been modeled, and a consolidated asset table has been given in Table 6.9. Table is structured as follows: the first column is the name of the asset, the second column shows the scope of asset utilization, the next group of columns under INV product family indicate whether an asset has been used in that INV product, and similarly the final group of columns under FGW product family indicate whether an asset has been used in that FGW product. The cell is marked with ($\surd$) if the asset is being used by that product.

In this section, the reuse ratio and reuse scope, as defined in [52], will be investigated. It has been stated in [52] that the success of reuse can be measured by primarily two factors:

Table 6.9: Asset utilization within and cross product families

		INV Products						FGW Products		
Assets		FIS	FND	EQT	DEX	FDM	PRT	EFT	CRA	CRB
Customer Core	⋆					√	√	√	√	√
Customer Advanced	●	√	√	√	√					
Blacklist Manager	⋆	√		√				√	√	
Document Manager	⋆	√	√	√	√	√	√	√		
Account Manager	⋆	√	√	√	√	√	√	√	√	
Deduction	⋆	√	√	√	√	√	√	√		
Accounting Gateway	⋆	√	√	√	√			√		
Accounting	●					√	√			
Administration	⋆	√	√	√	√	√	√	√	√	√
Ext. System Data Transfer	⋆	√		√		√	√		√	√
Alert and Notification Man.	⋆	√	√	√	√	√	√	√	√	
Repo	○	√								
Fixed Income Common	○	√								
Fixed Income Trade	○	√								
BPP	○	√								
Asset Delivery	○	√								
Auction	●	√					√			
Asset Lending	○	√								
DEX Operations	○				√					
Equity Common Operations	○			√						
Order Management	○			√						
Credit	○			√						
Capital Increase	○			√						
Public Offering	○			√						
Mutual Fund Buy/Sell Ops.	○		√							
Fund Transfer	●		√			√				
Fund Man. Backend	○					√				
Portfolio Man. Backend	○						√			
Asset/Stock Invest Core	●	√	√	√	√	√	√			
Cash Invest Core	●	√	√				√			
Asset Transfer	●	√	√							
FGW Core	⋆					√	√	√	√	√
FGW Communication	●								√	√
EFT Messaging (HLP)	○							√		
EFT Operations	⋆					√	√	√		
KKB KRS	○									√
KKB LKS	○									√
CRA Electronic Registry	⋆		√	√		√	√		√	
CRA Core Operations	○								√	

- Reuse scope for a reusable component
- Reuse ratio in the target application

Satisfying these two measures at the same time is not trivial. Large reuse scope is achieved by those reusable components that provide relatively low level of functionality (e.g. libraries). Improving the level of functionality decreases the reuse scope, but at the same time increases the reuse ratio of the final product. The latter case is common for the product line approaches (with coarse-grained assets) but only in the limited scope of product family. Therefore, increasing the reuse scope beyond the boundaries of a single product family while keeping the reuse ratio high is critical.

The asset utilization column in Table 6.9 indicates the reuse scope of assets with the following symbols: "$\star$" indicates that an asset is being used in two product families, "$\bullet$" indicates that an asset is being used within at least two products of a family, whereas "$\circ$" indicates that an asset is being used only in a single product of a family.

There are 39 assets used to build INV and FGW product lines. The distribution of assets according to reuse scope is as follows: there are 12 assets used in two product families (31%), 8 assets used within at least two products in a family (20%), and 19 assets used only in a single product of a family (49%). These results show that half of the assets are reused in at least two products and large reuse scope beyond the product lines has been achieved with the proposed modeling approaches.

On the other hand, the reuse ratio for each product can be calculated as follows (call this Product as P): A_r/A_P where

A_r: the number of assets used in P and in at least one more product, and

A_P: the total number of assets used in building the product P.

For example, 19 assets have been used in building the FIS product (A_P is 19), 13 of 19 assets that are used in FIS and in at least one more product (A_r is 13), the reuse ratio for FIS is $13/19 = 68\%$.

The reuse ratio for all products varies between 68% and 92%. The reuse ratio for individual products decreases if product families are considered independently. For instance, EFT product uses many assets that are also reused in INV product line, therefore the reuse ratio of EFT is 81% if both product families are taken into consideration. On the other hand, the value is 55% if it is calculated only in the context of FGW product line. In any case, the reuse ratio is above the 50%.

There several are factors that have to be noted for the right interpretation of these results:

- Those products that share a large number of common business functionality, such as DEX, FND, FDM, have better reuse ratios since they only differ in their core business flows and functionality.

- Two assets, "Customer Core" and "Customer Advance", differ in their reusability. Although this information is not available in the table, "Customer Core" does not depend on BPM and Batch Processing Kits, therefore it is reusable in both product lines. However, "Customer Advance" is dependent on BPM Kit for advance customer approval workflows this is only usable in INV product line, not in FGW.

- Some of the domain assets (first 11 of them) are the end product of banking projects, which have been carried out for several years. Therefore, they have been designed from scratch and this is one of the factors that has improved the reuse ratio.

- The domain knowledge obtained before designing these domains has been realized by senior business analysts. That is another critical factor in achieving those high reuse ratios.

As stated in [27], if multiple product lines share many common features and variations, developing and maintaining the common artifacts become a critical requirement. The reusability of assets in multiple product lines is one of the unique outcome of the proposed approach. Since the asset modeling languages are derived from the same meta-model and they depend on the same DSK set, these provide the expected benefit of utilization of software assets across multiple product lines. Furthermore, keeping a single copy of capability features of assets (ACMs) and maintaining the variability points within these assets eliminate the redundancies and possibility of inconsistencies when assets are used in multiple product lines independently.

6.4 Reusability of Domain Specific Kits

This section presents that DSKs can be reused across multiple product lines. For this purpose, several product lines that have been developed at Cybersoft are introduced and their DSK utilization has been depicted in Table 6.10.

The following product lines have been analyzed in this discussion:

- *Core Banking System (BNK)*: Core banking system is a complex product family with many products sharing many features, but having many variations at the same time. A typical core banking application includes products ranging from corporate banking applications to consumer products, from accounting system to payment systems. Each of these product sets can be considered as a separate product lines, but they have been included as a single product family.

- *Investment Banking (INV)*: This product family has already been discussed in Section 6.1.1.

- *Financial Gateways (FGW)*: This product family has already been discussed in Section 6.1.2.

- *Enterprise Resource Planning (ERP)*: This product family has been developed for ERP installations in public sector organizations. It has all major ERP products such as stock management, purchasing, accounting, finance, human resources, fixed assets, materials management, etc.

- *Tax Automation (TAX)*: Even this is a single product, several DSKs from other domains have been used in the architecture of this system.

- *Insurance (INS)*: Insurance application has been designed and developed using some of the DSKs already in use.

Within the context of these product lines, several new DSKs are introduced in addition to the ones that have been discussed in Section 6.2.1. These new DSKs are as follows:

- *Data Access Layer (DAL):* This is actually an alternative kit for persistence, which has a service-based interface for RDBMS access with many enterprise features, such as clustering RDBMS instances, table and query virtualization for very large volume of result sets, etc.

- *Document Management System (DMS):* This is a document management system to track and store electronic documents and images of paper documents. This has a logical central store organization and many enterprise level integration options.

- *Authorization and Authentication Server (CSAAS):* CSAAS provides a central control for the collection of user ID authentication and authorization processes, which is isolated from the applications. It is designed to allow integration by multiple applications via service-based interfaces, and enables the use of rule-based policy definitions. It comprises both RAD (Resource Access Decision) and RBAC (Role Based Access Control) features.

Table 6.10 shows the DSK usage in different product lines. It is the product line (domain) requirement that determines whether a particular DSK is used in that domain. For instance, DAL and POM are alternative persistence kits. DAL, which is a CORBA-based persistence kit, has been employed in taxation system since the back-end RDBMS server consolidation is mandatory for that domain and it is particularly needed to restrict the data access with some additional features. All other product lines have utilized POM as a persistence kit.

Table 6.10: Reusability of DSKs across multiple SPLs

	Product Families					
DSKs	**BNK**	**INV**	**FGW**	**ERP**	**TAX**	**INS**
RIA Presentation Kit	√	√	√	√	√	√
Reporting Kit	√	√	√	√	√	
Business Services Kit	√	√	√	√	√	√
BPM Kit	√	√		√		
RUMBA Business Rules Kit		√				√
Persistence (POM) Kit	√	√	√	√		√
Batch Processing Kit	√	√				
DAL Kit					√	
DMS Kit	√				√	
CSAAS Kit				√	√	

RIA Presentation and Business Services Kits have been reused in all product families. This actually makes some of the business domain independent core assets, such as document, alert manager, organization, etc., reusable across these product lines since presentation and business implementations are based on the same domain specific abstraction.

On the other hand, some of the product families, given in Table 6.10, make use of other non-listed architectural components for similar facilities. For instance, only the core banking and investment banking products use Batch Processing Kit for batch and scheduled operations, however other domains have their particular batch job management facilities.

At first glance, this also seems to be facilitated by frameworks and libraries. However, the frameworks are language dependent and usually integrate the components via composition mechanisms of the underlying language. The language independent composition infrastructure, such as CORBA, does not provide a required level of abstraction and addresses this issue in the object level. Java EE application server component model, on the other hand, addresses the composition issues through Java language and other Java EE services. In SFA approach, DSK provides a language independent abstraction for development of domain specific artifact and employs a fully declarative composition model. This also enables incorporation of business specific DSKs into the model which extends the benefits and improves the reuse of coarse-grained components as DSKs.

6.5 Quality Improvements

In this section, the proposed approach has been explored with respect to the observed quality improvements. The discussion has been pursued along with those quality attributes affected by the approach.

- *Reusability*: Improving the reusability of domain know-how is one of the key motivations of SFA approach. The reuse has been increased in two respects: the first is achieved by using DSKs as the main building block, which increases the reuse of domain specific abstractions that can be reused even across multiple product lines. Secondly, the modeling of software assets provides the reusability within the product line and across product lines. These have been discussed with examples in Section 6.3 and Section 6.4.

- *Traceability*: The SFA approach provides traceability from one end (requirements) to the other end (reusable assets). Requirements expressed as feature models are mapped to domain specific artifacts and later they are bound to software assets. Feature models are pivoting points as they are linking requirements of the problem domain to artifacts of the solution domain. An answer to typical scenario: “how the effect of change in requirements can be determined” is easy to detect. The requirements are expressed as feature models, and they are, in turn, linked to the artifacts and variability points. The dependency of artifacts and variability points clearly identifies the scope of change in solution domain.

- *Testability*: The testability issues have been collected as follows:

  (a) Managing the dependency of artifacts and their composition via CDL can guide and structure testing, and thus, reduce the overall time to deploy an artifact.

(b) The scope of regression testing can easily be determined through full traceability.

(c) It is possible to develop and use additional test programs based on CDL descriptions.

(d) DSEs can be engineered in such a way to run automated test runs.

- *Productivity*: The productivity effect of the model has been determined in the following points:

  (a) Building applications by developing the domain specific artifacts and composing them by means of a choreography language increases the developer productivity considerably. The time-to-develop an artifact is relatively low with respect to developing the same artifact with a low level programming language.

  (b) This also enables the static validation of artifacts against a choreography description, which, in turn, lowers the defect ratio and improves the quality.

  (c) The cost of implementing artifacts can be reduced since conformance to expected behavior described in the CDL is ensured.

  (d) It might be possible to develop and use CDL-based tools to generate artifacts skeletons. This will prevent the developers dealing with configurational or structural issues, hence guide their efforts to actual development of artifacts.

  (e) Productivity during development of artifacts is improved since DSTs are tailored to do it more effectively. In addition, developing the artifacts in isolation provides a concentrated effort that improves the productivity.

  (f) Through the use of DSLs, developers need not to know the inner working details of infrastructural issues, which will reduce the training time.

  (g) Separation of concerns during the development by means of DSKs enables setting up separate teams that use specialized DSTs.

- *Maintainability*: Maintainability of applications are improved since pieces of software (artifacts) can be managed separately. The declarative nature of DSLs brings the ease-of-change. This altogether reduces the maintenance time of software assets. Moreover, flexibility of products is improved with first-hand support of variability points and alternative realization mechanisms. On the other hand, it requires effective configuration and release management practices.

- *Performance*: The performance can be improved since Domain Specific Engines can utilize the native capabilities or they can make use of particular domain specific techniques internally. On the other hand, the performance might decrease since the applications are built using the assembly of artifacts as compared to developing everything in a single language. Similarly the choreography, context management and propagation, and other composition overheads increase the runtime costs.

- *Reliability*: Reliability of the products is naturally improved since DSKs and assets are reused in multiple products and even in multiple product families, so that they have better chance to improve their maturity through reusability.

It has to be noted here that the above discussion is based on subjective evaluation during case studies. The quantitative analysis of empirical data from further case studies is needed for better investigation of the quality attributes.

6.6 Comparison with Major Product Line Approaches

In order to compare the proposed approach with other product line engineering approaches, the comparison model, adapted from [52], has been used.

In [52], set of comparison criteria has been determined based on the lessons learned from the product line community ($C_1 - C_3$) and additional requirements that must be satisfied for larger scope of reuse ($C_4 - C_7$). The criteria are as follows:

C_1: *Rely on an abstract and stable description of the problem to solve.*

C_2: *Identify explicitly the variations in terms of features, not in terms of a solution.*

C_3: *Reuse coarse-grain, high-functionality components.*

C_4: *Allow abstract architecture evolution.*

C_5: *Variation mechanisms must be improved.*

C_6: *A high-level mapping between the abstract architecture and the components.*

C_7: *Reuse components developed elsewhere.*

Major product line approaches to be used in comparison are described here very briefly. The discussion of the approaches and their evaluation with respect to the set of criteria given above is already available in [52]. Therefore the details will not be discussed here. The product line approaches are as follows:

- *Domain Specific Language (DSL)*: This approach can be highlighted as designing and generating a high-level language where domain specific concepts are promoted as first class entities. A DSL is designed to abstract the common architecture of the product family, and it enables to represent the concepts of the application domain. A program, written in that language, is compiled into an executable code.
- *Generative Programming (GP)*: Generative programming focuses on automating the creation of product families by generating a family member from a specification written in a high-level language (DSLs). DSLs are designed after the feature model has been constructed and strongly emphasizes the variation control.
- *Model Driven Engineering (MDE)*: MDE is a top-down approach and is primarily focused on using meta-models to capture domain specificity. MDE applies a series of transformations from platform independent models into platform specific models. The approach has already been briefly described in Section 2.4.
- *Domain Specific Modeling (DSM)*: DSM is a combination of DSL and GP approaches. It has three elements: a DSL, a generator and a framework. Domain concepts are expressed in DSL, the domain variations can be expressed either in a DSL, in a generator, or they may be defined and implemented in a framework. The approach has already been briefly described in Section 2.10.

Table 6.11 has been adapted from [52] which rates product line approaches with respect to the criteria set (the rates are expressed as "$\star$").

Evaluation of the SFA approach with respect to the criteria is as follows:

C_1: *Rely on an abstract and stable description of the problem to solve*: SFA anticipates the use of feature-oriented domain model to start building reference architecture and asset models. This is actually a description of the problem domain. Therefore, this criterion has been fully satisfied.

Table 6.11: Comparison of the approaches

	C_1	C_2	C_3	C_4	C_5	C_6	C_7
DSL	⋆⋆⋆		⋆⋆⋆				
GP	⋆	⋆⋆	⋆⋆⋆		⋆⋆		
MDE	⋆⋆⋆	⋆	⋆			(⋆⋆⋆)	
DSM	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆				
SFA	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆	⋆⋆⋆	⋆⋆	⋆⋆⋆

C_2: *Identify explicitly the variations in terms of features, not in terms of a solution*: The proposed model manages the variations with feature models in problem domain, and maps them to the artifacts and variability points in the solution domain. Therefore, this criterion has been fully satisfied.

C_3: *Reuse coarse-grain, high-functionality components*: The assets, as sets of features, are coarse-grained, and they include the domain specific artifacts with a set of variability points. The assets are reusable within a product family, and they can also be reused in other product lines even they are not specifically designed for that domain. This criterion has been fully satisfied, and no other approach provides support for reuse in multiple product lines.

C_4: *Allow abstract architecture evolution*: Although the reference architecture modeling and asset modeling proceed by feedbacking each other, they both rely on a feature-based domain model. Therefore, they are not quite tolerant in changing the requirements and scope of product lines.

However, such a change does not require the product line setup to start from scratch. As reference architecture modeling proceeds by extracting the architectural aspects and quality attributes from feature-oriented domain model, it brings another layer of separation. Furthermore, using DSKs might probably hide some of those changes, in other words, the impact of change is limited to a single DSK. As a result, this criterion is partially satisfied.

C_5: *Variation mechanisms must be improved*: The asset modeling approach in SFA enables to define variations and their realization mechanisms. The variability realization mechanisms can be expanded since their implementations are dependent on DSEs. As new DSEs enable alternative realization mechanisms, the assets, hence products, can utilize these variation mechanisms. This criterion has been fully satisfied.

C_6: *A high-level mapping between the abstract architecture and the components*: This requirement indicates that there should be no direct relationship between a feature and underlying implementation. In SFA approach, DSKs provide the building blocks of solution space, artifact types and variability mechanisms that the feature models are mapped to. There is a clear separation of concerns since artifact types are executed by different DSEs and composed via CDL. The high-level mapping has been achieved. However, although the mapping is domain specific, the formal mapping rules need more research. Therefore, this is partially supported.

C_7: *Reuse components developed elsewhere*: This requirement can be evaluated in two respects: first, the components that are core assets of other product lines can be reused as long as their dependencies are satisfied. Second, the DSK encapsulation can help to introduce other components and services as new DSKs and they can be configured by a suitable DSL and be used within another product line. (See next section how DSKs can be utilized for this purpose.) This criterion has been fully satisfied.

6.7 DSKs in Migration to Service-Oriented Computing

This section covers the application of the DSK concept to another field: legacy migration to service orientation. This section has been mainly excerpted from two recent publications [30, 31].

6.7.1 Migration Strategy

Converting legacy applications to services allows systems to remain largely unchanged while exposing functionality to a large number of clients through well-defined service interfaces. Migrating a legacy system to SOA, e.g. wrapped as Web Services, may be relatively straightforward. However, characteristics of legacy systems like platform, language, architecture, and the target SOA may unexpectedly complicate the task. This is particularly the case during migration to highly demanding SOA enforcing the rich content rendering, service composition and mashups.

There exist some proposals for SOA migration and each approaches to the problem mainly from different viewpoints though they have commonalities. The alternatives have been presented and systematically compared in [30].

In [30], we have proposed a six-step mashup migration strategy depicted in Figure 6.5. The strategy addresses both behavioral and architectural aspects of the migration. The first two activities are for the modeling of target enterprise business (MODEL) and the analysis of legacy systems and infrastructure (ANALYZE). These activities lead to two main steps: (MAP & IDENTIFY) maps model requirements to legacy components and services identifications; and (DESIGN) models mashup server architecture with Domain Specific Engines (DSEs), which abstracts legacy components.

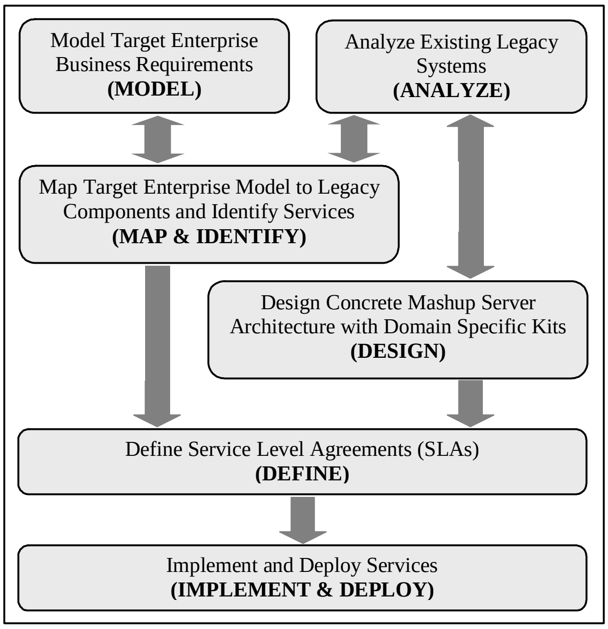

Model Target Enterprise Business Requirements (MODEL)
Analyze Existing Legacy Systems (ANALYZE)
Map Target Enterprise Model to Legacy Components and Identify Services (MAP & IDENTIFY)
Design Concrete Mashup Server Architecture with Domain Specific Kits (DESIGN)
Define Service Level Agreements (SLAs) (DEFINE)
Implement and Deploy Services (IMPLEMENT & DEPLOY)

Figure 6.5: A roadmap for migration to service-oriented computing

Both the mapping and architectural design activities might cause a loopback to MODEL and ANALYZE activities to re-reconsider some of the decisions and improvements. As a result of these major activities, target system service dependency graph has been constructed and mashup server concrete architecture has been designed. Defining the Service Level Agreements (SLAs), including non-functional and contextual properties, is the next step (DEFINE) that will obviously be followed by implementation and deployment activities (IMPLEMENT & DEPLOY).

6.7.2 The Role of DSKs in Migration Strategy

The migration strategy, itself, is not the particular reason to include this work here. Rather, it has been leveraged by the specific pluggable DSEs that can be utilized as abstraction of different service sources in mashup server architecture. Each DSE may require varying management activities depending on the service source. They also maintain the contextual information and they are configured using the DSLs. Defining DSLs under the governance of a meta-model enables the exposition of service from different sources with varying attributes.

The reference architecture model for the mashup server has been depicted in Figure 6.6, which highlights the generic reference architecture that enables plugging the specific DSEs for different systems (service sources). The reference architecture depends on a meta-model for specifying DSLs for different service sources, a common service repository, and a policy for managing contextual information. The reference architecture enables the integration of another mashup server using a particular DSE (see DSE_5 in Figure 6.6).

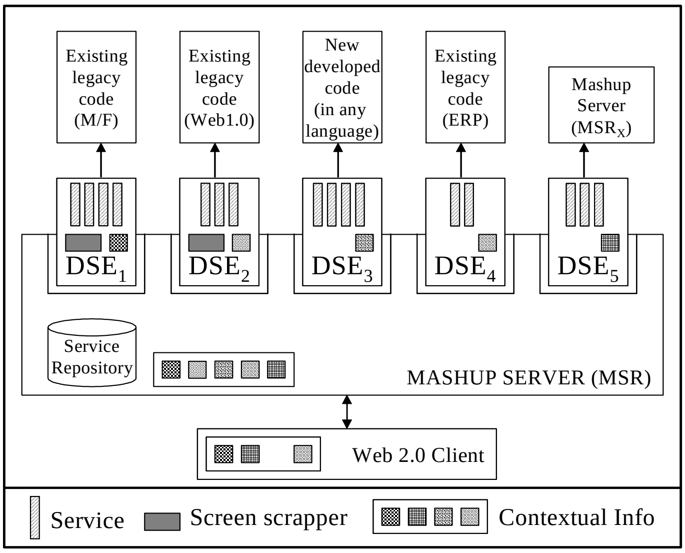


Figure 6.6: Mashup reference architecture with DSEs

The DSEs can wrap different legacy systems. Consider, for instance, a Web 1.0 system without an API. It heavily mixes presentation with content and makes it hard to sift out meaningful data from the rest of the elements used for formatting, spacing, decoration or site navigation. In such a situation, the DSE can employ “Screen/Web scraping” techniques by analyzing the page structure and wrapping out the relevant records. In some cases the task is even more complex than that: the data can be scattered over more pages. Then, triggering of a GET/POST request may be needed to get the input page for the extraction or authorization that might be required to navigate to the page of interest. The situation may be even more complex if there are workflows running in the provider system and tight security policies are applied.

In case the legacy code is provided by a mainframe, the screen scrapping techniques are essential to simulate the working of the mainframe terminals. Using screen scrapping, the data would be retrieved from the host and also posted onto the host. Mashed up services are accessed through Web 2.0 clients supported with AJAX, Flash, Flex, JavaScript, and other XML-based rendering [8] technologies. Even smarter clients can be used to access several mashup servers from a single client empowered with mashup choreography abilities.

6.7.3 Experimenting the Migration Strategy

The example to demonstrate the mashup reference model and migration strategy comes from the black list management in financial applications: mashing up financial gateways and black list management (BLM) sources. The BLM sources include Central Bank lists, credit history lists, capital market black lists, internal black lists of the bank, etc. These services are accessed via different ways: (a) through a standalone legacy program accessed via the user screens, (b) through gateway accessing mainframe, (c) through MQ-based access, (d) through existing AS/400 based core banking solution. The mashup problem is as follows:

i. “Provide a third party service to the banks so that they can access these varying sources via a single interface.”

ii. “Enrich this service with value-added services such as combining the black list records with Ministry of Finance records accessed through a Web service.”

iii. “Enable bank to associate their customer information while querying black lists on demand.”

Initial modeling of the problem revealed that the domain specific approach is effective in absorbing variations in service sources. The design of Domain Specific Engines can hide all the dirty tricks and low-level details of external accesses. The modeled DSEs are as follows:

- **DSE-CB**: Modeled to access Central Bank list with the old legacy program. This engine must do screen scrapping to initiate a query and extract information from the display screen.
- **DSE-MF**: Modeled to retrieve/post data from/onto the mainframe.
- **DSE-MQ**: Modeled to access message-based system with message queues.
- **DSE-AS400**: Modeled to retrieve/post data from/onto the AS/400. This engine must also do screen scrapping and manage complex screen flow logic.
- **DSE-WS**: Modeled to encapsulate Web service access.

The approach can utilize the existing legacy application, such as Central Bank lists, mainframe lists, and AS/400 screens, without any change at the code level. Furthermore, it is relatively easy to integrate Ministry of Finance querying with a particular DSE. “Querying black list” service can support advanced functionality such as record matching among all black lists and data linkage algorithms. Those new services can be implemented using any technology of choice during refactoring.

The initial investigation showed that the model is quite tolerant to variations in service sources and can help to reuse existing legacy components compared to our earlier implementations where such external links are managed at the code level.

6.8 Remarks

As a concluding remark to this chapter, the original research questions introduced in Chapter 1 have been revisited, and they are linked to validation efforts in this chapter as follows:

Q1. *Can domain specific know-how be abstracted and reused across different business domains?*

Yes, domain specific know-how has been abstracted and encapsulated into Domain Specific Kits (DSKs), and they have been used as a basic building block to express the domain specific types and artifacts. This has been validated through defining a set of DSKs for different purposes and by utilizing them across different product families. (See Section 3.2, Section 6.2.1, Section 6.4, and Appendix C.)

Q2. *Can such abstractions increase the reusability of software assets?*

Yes, this has a positive effect on reuse scope and reuse rates of software assets, while DSKs, itself, are reusable across multiple product lines. By using the proposed reference architecture and asset modeling activities, two product families have been modeled. The reuse scope and reuse rates in these families have been investigated for validation purposes. It has also been examined and validated that the assets can be reused not only within a product family but also in multiple product families. (See Chapter 5, Section 6.3, and Section 6.4.)

Q3. *What should be the content of reusable assets for increased reuse scope?*

The reusable software assets in the proposed model includes Domain Specific Artifacts, their dependencies, variability mechanisms, and contextual information. The proposed asset modeling approach reveals software assets that are reusable in multiple product lines. (See Chapter 5, Section 6.3, and Section 6.4.)

Q4. *How do those domain specific abstractions be employed and helpful in modeling the reference architecture of a product family?*

A reference architecture modeling approach has been constructed based on the separation of concerns both in problem and solution domains. The concept of DSK has been successfully utilized in product line reference architecture. This has been validated by constructing reference architectures of two domains. (See Chapter 4 and Section 6.2.)

Q5. *Can we construct a roadmap for setting up software product lines for different domains out of a reference model?*

Yes, the proposed coupled modeling activities, i.e. reference architecture and asset modeling, constitute the domain design phase of a software product line. Software Factory Automation has been proposed as an industrialization model and it has been used experimentally to model two product families (INV and FGW). (See Section 6.1, Section 6.2, and Section 6.3.)

CHAPTER 7

SUMMARY AND CONCLUSIONS

7.1 Summary

This study proposes an industrialization model, *Software Factory Automation*, for establishing software product lines. Major contributions of this thesis are the conceptualization of *Domain Specific Kits (DSKs)* and a *Feature-Based Domain Design Model* based on DSKs for software product lines. The concept of DSK has been inspired by the way other industries have been successfully realizing factory automation for decades. DSKs, as fundamental building blocks, have been deeply elaborated with their characteristic properties and with several examples.

The constructed domain design model has two major activities: first, building the product line reference architecture using DSK abstraction; and second, constructing the reusable asset model again based on DSK concept. Both activities depend on outputs of feature-oriented analysis of product line domain. The outcome of these coupled modeling activities is the "reference architecture" and "asset model" of the product line.

Reference architecture modeling approach has been based on symmetric alignment of problem and solution domains structured in multiple concern spaces. Representing both domains in multiple concern spaces and aligning two symmetric matrices in another matrix assist identifying the sensitivity and tradeoff points in problem domain as well as the components, connectors and properties in solution domain. This architectural modeling approach has also been exemplified on modeling of a Web security framework, which revealed the practical cases how components and connectors, later DSKs, can be derived from the symmetric alignment matrix.

On the other hand, the asset modeling approach uses the abstractions provided by DSKs to encapsulate the commonality of features, and provides means to effectively manage the variations of them by exploiting a meta-model. Using features in software modeling is not new, however, encapsulating them in individual asset models with domain specific abstractions looks more attractive since this approach ends up with more loosely coupled assets. The proposed approach creates more cohesive asset models by encapsulating the feature commonality within an asset. It further facilitates variability management with composition of Domain Specific Artifacts through the choreography engine of SFA reference architecture.

The approach has been validated by constructing product lines for two different product families. The reusability of DSKs and software assets within and across these product lines has been discussed. Finally, the constructed model has been evaluated in terms of quality improvements, and it has been compared with other software product line engineering approaches.

7.2 Conclusions

The presented SFA approach has been conceived to bring the focus *on essential difficulties* in software development. For this purpose, the SFA model relies on two things: higher level of –domain specific– abstractions providing separation of concerns and high level composition of these abstractions. Therefore, the technology or platform is not the main focus of SFA approach, which has been usually the case for previous approaches [23].

The SFA model has been built on DSK abstraction, and the value proposition behind the fundamental DSK abstraction is to provide abstract artifacts expressed using a DSL that are efficiently developed using a specialized toolset (DST) and executed efficiently by regarding engine (DSE). Different DSKs address different set of concerns, which provide the segregation of responsibilities not only in technical infrastructure but also in the development team organization as well. As a result, very specialized teams with different concerns can be formed with increased productivity. The loose coupling and symmetric composition of artifacts eliminates the inter-dependency of teams, and enables the development of product by assembly of Domain Specific Artifacts.

Another important characteristic of DSKs is that they can be generically reused across multiple product lines since they are abstract enough and loosely coupled. This also brings the reusability of their artifacts in the form of coarse-grained assets in multiple product lines. This has been shown through a case study including two different product families in Chapter 6.

The proposed approach has been evaluated with respect to the vision of software factories. The most fundamental property of the model is the effective incorporation of domain specific assets into software development. The model provides development by assembly by the declarative nature of DSL and CDL, and by providing explicit variability points in software assets. SFA facilitates setting up software supply chains by providing a common architectural baseline and suppliers of coarse-grained assets and DSKs. The model provides the flexibility of product options in satisfying the diversity of customer requirements.

The SFA approach contributes to the long term vision of mass customization of software, but it is too early to say that it has been fully satisfied. However, it provides improvement especially on product definition, design and assembly for mass customization.

The employment of SFA approach brings serious organizational changes to software development teams. The “organizational alignment” based on separation of concerns has to be taken, and there will be a new model at every step of the software development, acquisition and usage for all stakeholders. This is both positive and negative side of the model. The organizational change is not easy, but quicker the organizational change is higher the benefits will be. The “competence centers” in technical and business expertise areas will be similar to the assembly line organizations in other industries, such as automobile manufacturing. Finally, the team performance can independently be recorded, measured and improved.

Besides improving reusability, which is the major proposal of the model, the proposed model achieves better in a number of other quality attributes. The reuse has been increased by using Domain Specific Kits as the main building block and by modeling software assets on domain specific abstractions. The constructed roadmap from requirements engineering to product development maintains end-to-end traceability.

In terms of testability, the model benefits from the traceability of the requirements and declarative nature of product development; independent development of artifacts further improves the testability of the artifacts. Traceability and testability, together, improve the maintainability of the products, and maintainability also benefits from the variability management support in asset meta-models.

Another fundamental contribution of the approach is the productivity gain. The separation of concerns at large scale simplifies the development of artifacts since they usually address a single or a limited set of concerns. Time to develop, test and deploy an artifact is significantly reduced, since toolset is specialized for a single artifact type. CDL-based composition of artifacts enables the validation and conformance to expected behavior; and tool support can be provided to generate artifact skeletons for better productivity.

However, nothing has only positive sides; there are several drawbacks as well. One of the drawbacks of the proposed model is the possible performance loss due to the declarative development and execution model with independent engines. In addition, there is an extra overhead during execution due to choreography, context management and propagation, etc. This demands higher performance from choreography engine and DSEs as well as the development of alternative execution models. Besides, this triggers another drawback that the design of choreography engine gets more complex.

Another difficulty of the model is that it tries to define a meta-model and an XML syntax simultaneously to form asset models. As a result, there is no distinction between syntactic and semantic aspects. Therefore, it might be more convenient to develop a modeling notation rather then XML syntax later. Artifacts produced by DSLs and composed via CDL need a formal ground so that they can be checked with formal models.

An important aspect of the software asset model is that they can be later constructed as Domain Specific Kits with appropriate abstractions. This will further raise the abstraction and enable the construction of business specific DSKs. In order this to happen, a methodical study of DSLs and construction of a type system are needed.

Finally, the proposed model has the potential to provide global “SFA Providers”. These organizations will have their own DSKs and reusable core assets, so that they can provide them to other organizations aiming to setup their specialized product lines.

7.3 Future Work

While the study represents a significant improvement in automating the software development and providing reuse, it also opens a number of important further long term or short term research areas suggested by the results of this dissertation.

The SFA concept has six fundamental blocks, where three of them, namely the concept of DSKs, reference architecture modeling with DSKs, and software asset modeling with DSKs, have been studied here. In order to complete the vision of SFA approach, other three major long term research areas are as follows:

- *Feature-Oriented Requirements Engineering*: SFA anticipates the use of feature-oriented requirements engineering model for domain analysis. The features are life cycle entities to bridge the problem and solution domains, and they are meant to be logically modularizing the domain requirements and expressed as a *Domain Feature Model (DFM)*. Although it has been set a global viewpoint in Section 3.4, a complete feature-oriented requirements engineering process has to be devised for SFA approach.

- *Domain Engineering Life Cycle Management*: A complete domain engineering life cycle management process has to be defined for the technical and organizational practices needed for a successful operation of product lines. Such a life cycle needs to address the organizational management, process/project management, software engineering measurement and evaluation, with formal planning and continuous improvement.

- *Design of Choreography Language and Engine*: As SFA employs the principle of separation of concerns by utilizing the DSKs, a choreography model for domain specific artifacts has been described. Such a model describes collaboration between artifacts (executed by different DSEs) in order to achieve a common goal, and it captures the interactions and the dependencies between these interactions. It has been discussed in the text that the composition of artifacts is quite achievable even via simple composition model, however, employing a declarative choreography language for this purpose has many advantages (see Section 3.5.1). Besides, the design of choreography engine will certainly highlight more details on the syntax and semantics of DSL.

In addition to these long-term research areas, there are several short term but equally important research areas as well:

- Although the reference architecture modeling approach identifies many architectural issues systematically, some steps do still need systematic approaches in decision-making. One basic example is the determination of correlation coefficients in architectural modeling. Statistical analysis on a set of applications may create a catalog of useful coefficients to further guide the software architects.
- During reference architecture modeling, the exploration and formalization of rules for identification of components, connectors, and later DSEs from alignment matrix will be quite helpful. In order to achieve this goal, the modeling has to be applied on several case studies for the extraction of rules.
- In this dissertation, proposed approach has been investigated with a set of case studies. Further case studies are needed to improve the understanding of the approach, since this will enable us to extend the meta-model and modeling language for defining software assets as well.
- Automated tool support for software asset modeling with integrated repositories is a fundamental step in achieving the software factory vision.
- Defining an ontology of software assets can be quite helpful in development of asset modeling tools and repositories. Such an ontology can also be helpful in asset selection and checking the consistency of compositions.
- The guidance or recommendation on asset selection can be explored using the External Views (ACM) of software assets as they are represented with feature models. The reasoning approaches on feature models can be utilized and automated support can be provided in partitioning/mapping DFM into ACMs and selection of existing assets.
- A methodical approach for the design of DSLs, an exploration of a type system for DSATs including the higher-order types for business specific DSKs and their composition will further increase the level of abstraction since it will facilitate the encapsulation of software assets as business specific DSKs.

REFERENCES

[1] S. Ahn and K. Chong. A feature-oriented requirements tracing method: A study of cost-benefit analysis. In *ICHIT'06: Proceedings of the 2006 International Conference on Hybrid Information Technology*, pages 611–616, Washington, DC, USA, 2006. IEEE Computer Society.

[2] N. Akima and F. Ooi. Industrializing software development: A Japanese approach. *IEEE Software*, 6(2):13–21, 1989.

[3] Z. Aktas and S. Cetin. We envisage the next big thing. In *Integrated Design and Process Technology, IDPT-2006, Society for Design and Process Science*, San Diego, CA, USA, 06 2006.

[4] O. Alic. Development of a tool for architectural modeling with symmetric alignment of multiple concern spaces. M.Sc. graduation project report, Middle East Technical University, Department of Computer Engineering, June 2007.

[5] R. J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, School of Computer Science, 1997.

[6] N. I. Altintas and S. Cetin. Integrating a software product line with rule-based business process modeling. In D. Draheim and G. Weber, editors, *Trends in Enterprise Application Architecture, VLDB Workshop, TEAA 2005, Trondheim, Norway, Revised Selected Papers*, volume 3888 of *LNCS*, pages 15–28. Springer, 2006.

[7] N. I. Altintas, S. Cetin, and A. H. Dogru. Industrializing software development: The "Factory Automation" way. In D. Draheim and G. Weber, editors, *Trends in Enterprise Application Architecture, TEAA 2006, Berlin, Germany, Revised Selected Papers*, volume 4473 of *LNCS*, pages 54–68. Springer, 2006.

[8] N. I. Altintas, M. Surav, O. Keskin, and S. Cetin. Aurora software product line. In *Turkish Software Architecture Workshop, Ankara, September 2005*, 2005.

[9] F. Bachman and P. C. Clements. Variability in software product lines. Technical Report CMU/SEI-2005-TR-012, Carnegie Mellon University, Software Engineering Institute, 2005.

[10] M. R. Barbacci, R. Ellison, A. J. Lattanze, J. A. Stafford, C. B. Weinstock, and W. G. Wood. Quality attribute workshops, third edition. Technical Report CMU/SEI-2003-TR-016, Carnegie Mellon University, Software Engineering Institute, 2003.

[11] A. Barros, M. Dumas, and P. Oaks. A critical overview of the web services Choreography Description Language (WS-CDL), BPTrends Newsletter, www.bptrends.com, Volume 3, Number 3, March 2005.

[12] L. Bass and R. Kazman. Architecture-based development. Technical Report CMU/SEI-1999-TR-007, Carnegie Mellon University, Software Engineering Institute, 1999.

[13] L. J. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition.* Addison-Wesley Professional, 2003.

[14] L. J. Bass, M. Klein, and F. Bachmann. Quality attribute design primitives and the attribute driven design method. In *PFE'01: Revised Papers from the* 4^{th} *International Workshop on Software Product-Family Engineering*, pages 169–186, London, UK, 2002. Springer-Verlag.

[15] D. Batory. Feature oriented programming for product-lines. In *European Conference on Object-Oriented Programming 2006*, France, 2006.

[16] G. K. Behara and S. Inaganti. Approach to service management in SOA space, Wipro White Paper, www.bptrends.com, February 2007.

[17] K. Berg, J. Bishop, and D. Muthig. Tracing software product line variability: from problem to solution space. In *SAICSIT'05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 182–191. SAICSIT, 2005.

[18] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Sci. Comput. Program.*, 53(3):333–352, 2004.

[19] B. Boehm. Economic analysis of software technology investments. In T. Gulledge and W. Hutzler, editors, *Analytical Methods in Software Engineering Economics.* Springer-Verlag, 1993.

[20] B. Boehm. Managing software productivity and reuse. *IEEE Computer*, 32(9):111–113, 1999.

[21] G. Booch. *Object-oriented analysis and design with applications (2nd ed.).* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

[22] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink, and Klaus Pohl. Variability issues in software product lines. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 13–21, London, UK, 2002. Springer-Verlag.

[23] F. P. Brooks. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[24] A. W. Brown. *Large-Scale Component-Based Development.* Prentice Hall, 2000.

[25] D. C. Brown. Functional, behavioral and structural features. In *KIC5:* 5^{th} *IFIP WG5.2 Workshop on Knowledge Intensive CAD.*, Malta, 2002.

[26] A. Bucchiarone and S. Gnesi. A survey on service composition languages and models. In *Proceedings of the First International Workshop on Web Services Modeling and Testing (WsMaTe'06), Palermo, Italy*, 2006.

[27] S. Bühne, K. Lauenroth, and K. Pohl. Why is it not sufficient to model requiements variability with feature models? In *AURE04*, pages 5–12, Japan, 2004.

[28] D. Bunting, M. Chapman, and et. al. Web services composite application framework (WC-CAF) ver1.0 [online], www.arjuna.com/library/specs/ws_caf_1-0/ws-caf-primer.pdf, 28 July 2003.

[29] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger. Xml-based feature modeling. In *Software Reuse: Methods, Techniques and Tools*, volume 3107 of *LNCS*, pages 101–114. Springer, 2004.

[30] S. Cetin, N. I. Altintas, H. Oguztuzun, A. H. Dogru, O. Tufekci, and S. Suloglu. Legacy migration to service-oriented computing with mashups. In *ICSEA'07: Proceedings of the International Conference on Software Engineering Advances.* IEEE Computer Society, 2007.

[31] S. Cetin, N. I. Altintas, H. Oguztuzun, A. H. Dogru, O. Tufekci, and S. Suloglu. A mashup-based strategy for migration to service-oriented computing. In *ICPS'07: IEEE International Conference on Pervasive Services, July 15 - 20, 2007, Istanbul, Turkey*, pages 169–172. IEEE Computer Society, 2007.

[32] S. Cetin, N. I. Altintas, and C. Sener. An architectural modeling approach with symmetric alignment of multiple concern spaces. In *ICSEA'06: Proceedings of the International Conference on Software Engineering Advances*, page 48, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[33] S. Cetin, N. I. Altintas, and R. Solmaz. Business rules segregation for dynamic process management with an aspect-oriented framework. In *Business Process Management International Workshops, Vienna, Austria, September 4-7, 2006, Proceedings*, volume 4103 of *LNCS*, pages 193–204. Springer, 2006.

[34] S. Cetin, N. I. Altintas, and O. Tufekci. Improving model reuse with domain specific kits. In *International Workshop on Model Reuse Strategies, MoRSe 2006, Warsaw, Poland, Oct. 17, 2006*, 2006.

[35] S. Cetin, O. Tufekci, E. Karakoc, and B. Buyukkagnici. Lighthouse: An experimental hyperframe for multi-model software process improvement. In *EuroSPI2 Conference*, pages 62–73, 2006.

[36] K. Chen, W. Zhang, H. Zhao, and H. Mei. An approach to constructing feature models based on requirements clustering. In *RE'05: Proceedings of the* 13^{th} *IEEE International Conference on Requirements Engineering (RE'05)*, pages 31–40, Washington, DC, USA, 2005. IEEE Computer Society.

[37] P. Clements. What's the difference between product line scope and product line requirements? Technical Report news@sei interactive, Second Quarter 2003, Carnegie Mellon University, Software Engineering Institute, 2003.

[38] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[39] P. C. Clements. Active reviews for intermediate designs. Technical Report CMU/SEI-2000-TN-009, Carnegie Mellon University, Software Engineering Institute, 2000.

[40] Peter Coad and Edward Yourdon. *Object-oriented analysis (2nd ed.).* Yourdon Press, Upper Saddle River, NJ, USA, 1991.

[41] S. Cook. Domain-specific modeling. *Microsoft Architect Journal*, October 2006.

[42] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998.

[43] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. In *ICSEA'06: Proceedings of the International Conference on Software Engineering Advances*, page 44, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[44] I. Crnkovic, M. Larsson, and O. Preiss. Concerning predictability in dependable component-based systems: Classification of quality attributes. In R. de Lemos, C. Gacek, and A. B. Romanovsky, editors, *WADS*, volume 3549 of *LNCS*, pages 257–278. Springer, 2004.

[45] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[46] K. Czarnecki and C. H. P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, San Diego, CA, USA, 2005.

[47] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg. Feature models are views on ontologies. In *SPLC'06: Proceedings of the* 10^{th} *International on Software Product Line Conference*, pages 41–51. IEEE Computer Society, 2006.

[48] R. M. Dijkman and M. Dumas. Service-oriented design: A multi-viewpoint approach. *Int. J. Cooperative Inf. Syst.*, 13(4):337–368, 2004.

[49] L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *IEEE Trans. Softw. Eng.*, 28(7):638–653, 2002.

[50] A. H. Dogru and M. M. Tanik. A process model for component-oriented software engineering. *IEEE Software*, 20(2):34–41, 2003.

[51] J. Duhl. The business impact of rich internet applications, IDC White Paper, 2003.

[52] J. Estublier and G. Vega. Reuse and variability in large software applications. In *ESEC/FSE-13: Proceedings of the* 10^{th} *European Software Engineering Conference held jointly with* 13^{th} *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 316–325, New York, NY, USA, 2005. ACM Press.

[53] R. G. Fichman and C. F. Kemerer. Object technology and reuse: Lessons from early adopters. *Computer*, 30(10):47–59, 1997.

[54] W. B. Frakes, R. Prieto Díaz, and C. J. Fox. Dare: Domain analysis and reuse environment. *Ann. Software Eng.*, 5:125–141, 1998.

[55] D. S. Frankel. Business process platforms and software factories. In *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, San Diego, CA, USA, 2005.

[56] B. P. Gallagher. Using the architecture tradeoff analysis method to evaluate a reference architecture: A case study. Technical Report CMU/SEI-2000-TN-007, Carnegie Mellon University, Software Engineering Institute, 2000.

[57] D. Garlan. Software architecture: a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[58] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[59] R. Gitzel and A. Korthaus. The role of metamodeling in model-driven development. In *Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI2004) , 19-21 July, 2004, Orlando, USA*, July 2004.

[60] H. Gomaa and M. Saleh. Feature driven dynamic customization of software product lines. In *Reuse of Off-the-Shelf Components, 9^{th} International Conference on Software Reuse, ICSR 2006, Turin, Italy, June 12-15, 2006, Proceedings*, volume 4039 of *LNCS*, pages 58–72. Springer, 2006.

[61] B. González-Baixauli, M. A. Laguna, and Y. Crespo. Product line requirements based on goals, features and use cases. In *International Workshop on Requirements Reuse in System Family Engineering (IWREQFAM)*, pages 4–7, 2004.

[62] J. Greenfield and K. Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *OOPSLA'03: Companion of the* 18^{th} *annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 16–27. ACM Press, 2003.

[63] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[64] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the RSEB. In *ICSR'98: Proceedings of the* 5^{th} *International Conference on Software Reuse*, page 76. IEEE Computer Society, 1998.

[65] M. L. Griss and K. Wentzel. Hybrid domain specific kits for a flexible software factory. In *Proceedings of the Ann. ACM Symp. Applied Computing*, pages 47–52, 1994.

[66] W. Grosso. Laszlo: An open source framework for rich internet applications, http://today.java.net/pub/a/today/2005/03/22/laszlo.html, 2005.

[67] W. H. Harrison, H. L. Ossher, and P. L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Research Report RC22685 (W0212-147), IBM, December 2002.

[68] IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std 1471, 2000.

[69] ISO/IEC 9126-1:2001, Software Engineering – Product Quality, Part 1: Quality Model, 2001.

[70] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[71] JBoss jBPM [online], http://www.jboss.com/products/jbpm, 2007.

[72] D. Jordan, J. Evdemon, and et. al. Web services business process execution language version 2.0, OASIS Standard, 11 April 2007.

[73] M. M. Kandé and A. Strohmeier. On the role of multi-dimensional separation of concerns in software architecture. In P. Tarr, L. Bergmans, M. Griss, and H. Ossher, editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October 2000.

[74] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Software Engineering Institute, 1990.

[75] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[76] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web services choreography description language version 1.0, W3C Working Draft, 17 December 2004.

[77] R. Kazman, J. Asundi, and M. Klein. Quantifying the costs and benefits of architectural decisions. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 297–306, Washington, DC, USA, 2001. IEEE Computer Society.

[78] R. Kazman, L. J. Bass, M. Webb, and G. D. Abowd. SAAM: A method for analyzing the properties of software architectures. In *Proceedings of the* 16th *International Conference on Software Engineering, May 16-21, 1994, Sorrento, Italy.*, pages 81–90. IEEE Computer Society / ACM Press, 1994.

[79] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In 4th *International Conference on Engineering of Complex Computer System, ICECCS'98*, page 0068. IEEE Computer Society, 1998.

[80] R. Kazman, M. Klein, and P. Clements. ATAM: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Carnegie Mellon University, Software Engineering Institute, 2000.

[81] R. Kazman, R. L. Nord, and M. Klein. A life-cycle view of architecture analysis and design methods. Technical Report CMU/SEI-2003-TN-026, Carnegie Mellon University, Software Engineering Institute, 2003.

[82] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[83] C. W. Krueger. Practical strategies and techniques for adopting software product lines. In *Workshop on Industrial Experience with Product Line Approaches*, 2002.

[84] V. Kulkarni and S. Reddy. Enterprise business application product line as a model driven software factory. In *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, San Diego, CA, USA, 2005.

[85] C. Kuloor and A. Eberlein. Aspect-oriented requirements engineering for software product lines. In 10^{th} *IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003), 7-10 April 2003, Huntsville, AL, USA*, page 98. IEEE Computer Society, 2003.

[86] R. Land. An architectural approach to software evolution and integration. Licentiate thesis, Mälardalen University Press, September 2003.

[87] B. Langlois and D. Exertier. MDSoFa: a Model-Driven Software Factory. In *Proceedings of the International Workshop on MDSD at OOPSLA 2004, October 25, 2004*, 2004.

[88] G. Larsen. Model-driven development: Assets and reuse. *IBM Systems Journal*, 45(3):541–553, 2006.

[89] M. Larsson. *Predicting quality attributes in component-based software systems.* PhD thesis, Department of Computer Science and Engineering, Mälardalen University, 2004.

[90] N. Lassing, D. Rijsenbrij, and H. van Vliet. Using UML in architecture-level modifiability analysis. In *ICSE 2001 Workshop on Describing Software Architecture with UML*, pages 41–46. IEEE Computer Society Press, 2001.

[91] J. Lee and D. Muthig. Feature-oriented variability management in product line engineering. *Commun. ACM*, 49(12):55–59, 2006.

[92] K. Lee, K. C. Kang, M. Kim, and S. Park. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In *SPLC'06: Proceedings of the* 10^{th} *International on Software Product Line Conference*, pages 103–112. IEEE Computer Society, 2006.

[93] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In *ICSR-7: Proceedings of the* 7^{th} *International Conference on Software Reuse*, pages 62–77, London, UK, 2002. Springer-Verlag.

[94] G. Lenz and C. Wienands. *Practical Software Factories in .NET.* Apress, Berkeley, CA, USA, 2006.

[95] D. Liu and H. Mei. Mapping requirements to software architecture by feature-orientation. In *The Second International Workshop on Software Requirements and Architectures, STRAW'03 at ICSE'03, Portland, OR*, 2003.

[96] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Trans. Software Eng.*, 21(4):336–355, 1995.

[97] C. Lung, J. E. Urban, and G. T. Mackulak. Analogy-based domain analysis approach to software reuse. *Requir. Eng.*, 12(1):1–22, 2006.

[98] Y. Matsumoto. Essence of toshiba software factory. In *SPLC'07: Proceedings of the* 11^{th} *International on Software Product Line Conference – Invited Talk.* IEEE Computer Society, 2007.

[99] Model Driven Architecture, OMG, http://www.omg.org/mda, 2007.

[100] H. Mei, W. Zhang, and F. Gu. A feature oriented approach to modeling and reusing requirements of software product lines. In *Proceedings of the* 27^{th} *Annual International Computer Software and Applications Conference (COMPSAC'03).* IEEE Computer Society, 2003.

[101] K. Mullet. The essence of effective rich internet applications, Macromedia White Paper, 2003.

[102] S. Neema, J. Scott, and G. Karsai. Architecture analysis in software factories. In *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, San Diego, CA, USA, 2005.

[103] L. M. Northrop. A framework for software product line practice, v4.2. Technical Report http://www.sei.cmu.edu/productlines/framework.html, Carnegie Mellon University, Software Engineering Institute, 2007.

[104] C. O'Rourke. A look at rich internet applications. *Oracle Magazine*, 2004.

[105] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development.* Kluwer, 2000.

[106] C. M. Pancake. The promise and the cost of object technology: a five-year forecast. *Commun. ACM*, 38(10):32–49, 1995.

[107] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Commun. ACM*, 46(10):24–28, 2003.

[108] C. Phanouriou. *UIML: A Device-Independent User Interface Markup Language.* PhD thesis, Virginia Polytechnic Institute and State University, Computer Science Department, 2000.

[109] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, September 2005.

[110] O. Preiss, J. Wong, and A. Wegmann. On quality attribute based software engineering. In 27th *EUROMICRO Conference 2001: A Net Odyssey, 4-6 September 2001, Warsaw, Poland*, pages 114–120. IEEE Computer Society, 2001.

[111] PuLSE: Product line software engineering, Fraunhofer, fogo.iese.fraunhofer.de/pulse/.

[112] OMG RAS Specification, v2.2, www.omg.org/technology/documents/formal/ras.htm.

[113] M. Regio and J. Greenfield. Designing and implementing an hl7 software factory. In *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, San Diego, CA, USA, 2005.

[114] M. A. Rothenberger, K. J. Dooley, U. R. Kulkarni, and N. Nada. Strategies for software reuse: A principal component analysis of reuse practices. *IEEE Transactions on Software Engineering*, 29(9):825–837, 2003.

[115] M. A. Rothenberger and J. C. Hershauer. A software reuse measure: monitoring an enterprise-level model driven development process. *Information and Management*, 35(5):283–293, 1999.

[116] W. Schöfer and H. Weber. European software factory plan–the esf profile. In *Modern software engineering, foundations and current perspectives*, pages 613–637. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[117] M. Shaw and P. Clements. The golden age of software architecture. *IEEE Software*, 23(2):31–39, 2006.

[118] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[119] H. Smith and P. Fingar. *Business Process Management (BPM): The Third Wave*. Meghan-Kiffer Press, 2003.

[120] F. G. Sobrinho and M. D. Ferraretto. Software plant: the Brazilian software consortium. In *ACM'87: Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*, pages 235–243, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[121] P. Sochos, I. Philippow, and M. Riebisch. Feature-oriented development of software product lines: Mapping feature models to the architecture. In M. Weske and P. Liggesmeyer, editors, 5th *Intl. Conf. on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a NetworkedWorld, Germany*, volume 3263 of *LNCS*, pages 138–152. Springer, 2004.

[122] P. Sochos, M. Riebisch, and I. Philippow. The feature-architecture mapping (FArM) method for feature-oriented development of software product lines. In *ECBS'06: Proceedings of the* 13th *Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, pages 308–318, Washington, DC, USA, 2006. IEEE Computer Society.

[123] A. Z. Spector. The steady path to services-oriented computing. In *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, Keynote Talk, 2006.

[124] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[125] D. Streitferdt. *Family-Oriented Requirements Engineering.* PhD thesis, Technical University Ilmenau, 2004.

[126] M. Svahnberg. *Supporting Software Architecture Evolution.* PhD thesis, Blekinge Institute of Technology, 2003.

[127] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering ICSE'99*, pages 107–119, 1999.

[128] O. Tufekci, S. Cetin, and N. I. Altintas. How to process [business] processes. In *Integrated Design and Process Technology, IDPT-2006, Society for Design and Process Science*, San Diego, CA, USA, June 2006.

[129] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *J. Syst. Softw.*, 49(1):3–15, 1999.

[130] F. van der Linden. Software product families in Europe: The Esaps & Café projects. *IEEE Softw.*, 19(4):41–49, 2002.

[131] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[132] J. B. Warmer and A. G. Kleppe. Building a flexible software factory using partial domain specific models. In *Sixth OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon, USA*, pages 15–22, Jyvaskyla, October 2006.

[133] D. M. Weiss and C. T. R. Lai. *Software product-line engineering: a family-based software development process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[134] U. Zdun. Concepts for model-driven design and evolution of domain-specific languages. In *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, San Diego, CA, USA, 2005.

[135] W. Zhang, H. Mei, and H. Zhao. A feature-oriented approach to modeling requirements dependencies. In *RE'05: Proceedings of the* 13th *IEEE International Conference on Requirements Engineering (RE'05)*, pages 273–284, Washington, DC, USA, 2005. IEEE Computer Society.

[136] D. Zubrow and G. Chastek. Measures for software product lines. Technical Report CMU/SEI-2003-TN-031, Carnegie Mellon University, Software Engineering Institute, 2003.

APPENDIX A

GLOSSARY

Application Engineering

Application engineering is to develop individual products using the platform and core asset base established in domain engineering (*design-with-reuse*). (See Section 2.8.)

Architecture-Based Development (ABD)

A software architecture acts as a blueprint for all activities in the development life cycle. An ABD process includes elicitation of architectural requirements, design, documentation, analysis, realization and maintenance of software architectures. (See Section 2.6.)

Architectural Concern Spaces (ACS)

In reference architecture modeling, the solution domain multiple concern spaces that are formed by correlating architectural tiers and architectural views. (See Section 4.3.)

Asset-Based Development

Asset-Based Development is a set of processes, activities and standards that facilitate the reuse of assets. Asset-Based Development can be considered as a sub-methodology in the software development process. (See Section 2.7.)

Asset Capability Model (ACM)

The capabilities of software assets expressed as feature diagrams that describe a distinct service, operation, function or structure together with its non-functional properties such as expected response time or scalability concerns. (See Section 5.2.)

Business Process Management (BPM)

An emerging technology that organizes the flow of business processes in terms of workflows, rules, and other business entities for improving the efficiency of processes as they are defined, executed, managed and changed. (See Section 2.5 and Section 6.2.1.4.)

Component-Based Development (CBD)

Developing software systems by an assembly of components that are already developed and prepared for integration and in general considered as black boxes. This aims the development of components as reusable entities as well as the maintenance and upgrading of systems by customizing and replacing their components. (See Section 2.3.)

Domain Engineering

Domain engineering is to provide the reusable platform and core assets that are exploited during application engineering when assembling or customizing individual applications (*design-for-reuse*). (See Section 2.8.)

Domain Feature Model (DFM)

A model expressing the functional and non-functional requirements of product line. A DFM includes feature diagrams, composition rules, feature dictionary, list of requirements, quality attributes of the domain, other issues and decisions. (See Section 3.4.)

Domain Specific Language (DSL)

A language dedicated to a particular domain or problem with appropriate built-in abstractions and notations. (See Section 3.2.)

Domain Specific Engine (DSE)

An engine particularly designed and tailored to execute a specific DSL. (See Section 3.2.)

Domain Specific Toolset (DST)

An environment to design, develop, and manage software artifacts of a specific DSL. (See Section 3.2.)

Domain Specific Kit (DSK)

A composite of a DSL, DSE and DST. (See Section 3.2.)

Domain Specific Artifact (DSA)

An artifact that is expressed, developed, and executed by a DSL, DST, DSE respectively. (See Section 3.2.)

Domain Specific Artifact Type (DSAT)

A DSA type that a certain DSK can express, execute and facilitate the development. (See Section 3.2.)

Model-Driven Development (MDD)

A model-centric software engineering approach which aims at improving the quality and lifespan of software artifacts by focusing on models instead of code [59]. (See Section 2.4.)

Orthogonal Variability Model (OVM)

A model that defines the variability of a software product line. It relates the variability to other software development models. (See Section 5.4.)

Service-Oriented Computing (SOC)

A new computing paradigm that takes services as basic elements. SOC relies on Service-Oriented Architecture (SOA) when constituting the service model. (See Section 2.5.)

Software Asset

A collection of artifacts with variations that provides a solution to a problem. An asset has instructions on how it should be used and is reusable in one or more contexts, such as a development or a runtime context. It may also be extended and customized through variability points. (See Section 2.7 and Section 5.2.)

Software Factory Automation (SFA)

A methodical approach to set up software product lines starting from domain analysis to domain design using DSKs. (See Section 3.3.)

Software Factory

Software Factory [63] is the fundamental model that can be viewed as a comprehensive and integrative approach to generative software development, with the goal of automating product development in the context of software product line engineering [46].

Software Product Line (SPL)

A set of software-intensive systems sharing a common, managed set of features that satisfies 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 [38]. (See Section 2.8.)

SPL Reference Architecture

The generalized architecture of a product family that defines the infrastructure common to end products and interfaces of components that will be included in the end products [56]. (See Chapter 4.)

Utility Concern Spaces (UCS)

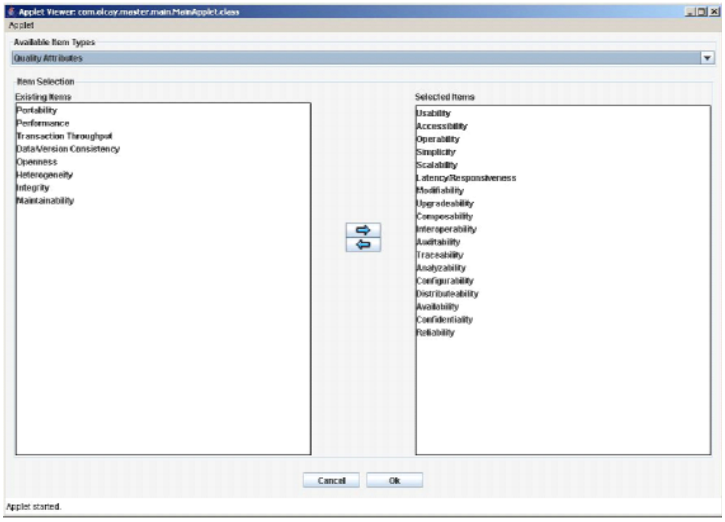
In reference architecture modeling, the problem domain multiple concern spaces that are constructed by correlating the quality attributes and the architectural aspects. (See Section 4.2.)

APPENDIX B

ABOUT RAMTool

RAMTool is a simple tool to support architectural modeling discussed in Chapter 4. Since the UCS and ACS may be too many, the symmetric alignment matrix can be huge and the analysis of the matrix may be too complicated. Therefore an effective tool during architecture design is unavoidable. This appendix includes several screenshots from the prototype tool [4].

Figure B.1 is used to define the primitives required for UCS and ACS construction. The "Item Type" in the screen can be any of the following: Quality Attributes, Architectural Aspects, Architectural Views and Architectural Tiers.



Available Item Types: Quality Attributes

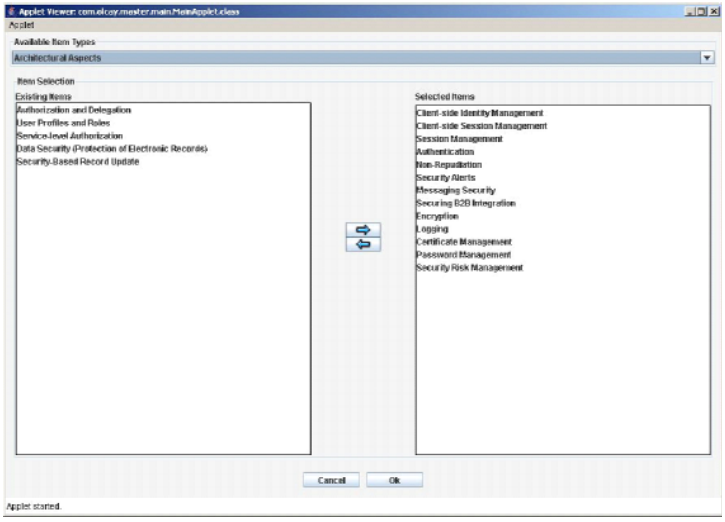
Item Selection

Existing Items
- Portability
- Performance
- Transaction Throughput
- Data/Version Consistency
- Openness
- Heterogeneity
- Integrity
- Maintainability

Selected Items
- Usability
- Accessibility
- Operability
- Simplicity
- Scalability
- Latency/Responsiveness
- Modifiability
- Upgradeability
- Composability
- Interoperability
- Auditability
- Traceability
- Analyzability
- Configurability
- Distributeability
- Availability
- Confidentiality
- Reliability

Figure B.1: RAMTool – Definition and selection of quality attributes

The screenshot of architectural aspect definition/selection has been depicted in Figure B.2. It is possible to select from set of aspects available in the repository.



Available Item Types: Architectural Aspects

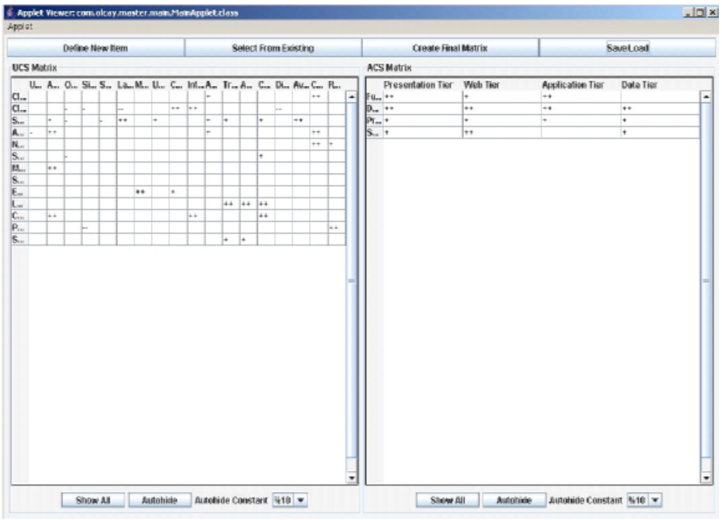
Item Selection

Existing Items:
- Authorization and Delegation
- User Profiles and Roles
- Service-level Authorization
- Data Security (Protection of Electronic Records)
- Security-Based Record Update

Selected Items:
- Client-side Identity Management
- Client-side Session Management
- Session Management
- Authentication
- Non-Repudiation
- Security Alerts
- Messaging Security
- Securing B2B Integration
- Encryption
- Logging
- Certificate Management
- Password Management
- Security Risk Management

Figure B.2: RAMTool – Definition and selection of architectural aspects

The UCS and ACS matrices are constructed by the tool (See Figure B.3). Initially, the values of the cells are "". The user can change the correlations by selecting values from a combobox "++", "+", "0", "−" and "−−".



UCS Matrix

	U...	A...	O...	Si...	S...	La...	M...	U...	C...	Inf...	A...	Tr...	A...	C...	Di...	Av...	C...	R...
Cl...											+						++	
Cl...			-	-		--			++	++					--			
S...		+	-		-	++		+			+	+		+		++		
A...	-	++									+						++	
N...																	++	+
S...			-											+				
M...		++																
S...																		
E...							++		+									
L...												++	++	++				
C...		++								++				++				
P...				--														++
S...												+	+					

Autohide Constant %10

ACS Matrix

	Presentation Tier	Web Tier	Application Tier	Data Tier
Fu...	++	+	++	
D...	++	++	++	++
Pr...	+	+	+	+
S...	+	++		+

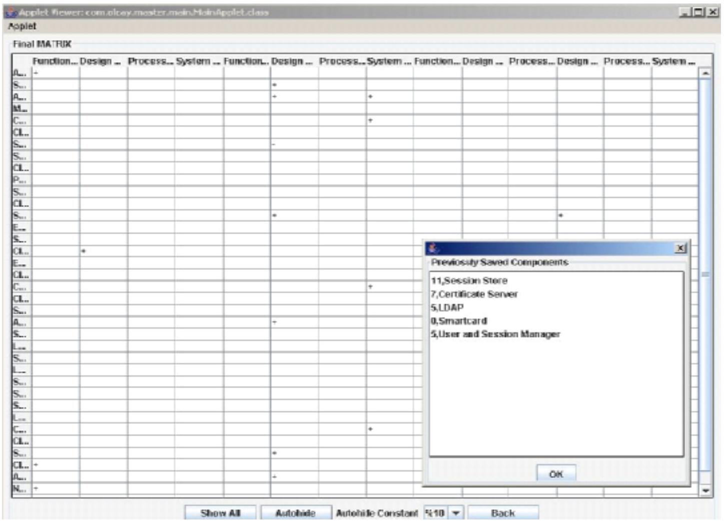
Autohide Constant %10

Figure B.3: RAMTool – UCS and ACS matrices

"Autohide" option in all screens provides a filtering facility of sparse rows/columns.

By Autohide button, user may filter out the sparse rows/columns, the sparsity of rows/columns are decided against the threshold selected from the combobox.

In Figure B.4, symmetric alignment matrix is formed and user first records the correlations and later identified components and connectors.



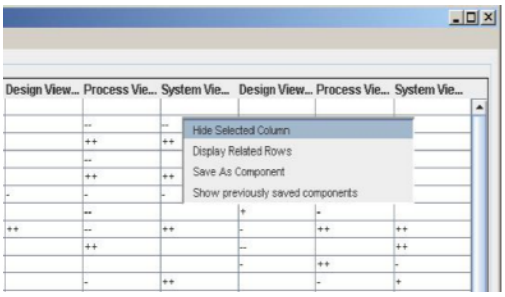
	Function...	Design ...	Process...	System ...	Function...	Design ...	Process...	System ...	Function...	Design ...	Process...	Design ...	Process...	System ...
A...	+													
S...						+								
A...						+		+						
M...														
C...								+						
Cl...														
S...						-								
S...														
Cl...														
P...														
S...														
Cl...														
S...						+						+		
E...														
S...														
Cl...		+												
E...														
Cl...														
C...								+						
Cl...														
S...														
A...						-								
S...														
L...														
S...														
L...														
S...														
S...														
S...														
L...														
C...								+						
Cl...														
S...						+								
Cl...	-													
A...						-								
N...	-													

Previously Saved Components

11,Session Store
7,Certificate Server
5,LDAP
0,Smartcard
5,User and Session Manager

Figure B.4: RAMTool – Symmetric alignment matrix

It is also possible to perform column-wise operations during architectural analysis. The system facilitates the user to mark the irrelevant rows or to filter the correlated rows of a column (See Figure B.5.



Design View...	Process Vie...	System Vie...	Design View...	Process Vie...	System Vie...
	--	--			
	++	++			
	--				
	++	++			
-	-	-			
	--		+	-	
++	--	++	-	++	++
	++		--		++
			-	++	-
	-	++		-	+

Figure B.5: RAMTool – Operations on symmetric alignment matrix

The tool provides a central repository for architectural modeling and information sharing and also enables an XML export. Web-based screens enable the sharing of information by several users.

This prototype can be extended in several ways:

1. It may identify the potential architectural elements (components and connectors) and possibly guide the user.

2. It may give guidance to the user for prioritizing the exploration of huge sparse matrices.

3. Support for annotation on the alignment matrix will improve the productivity during modeling activity since it spans over a time frame.

4. Integration with ADL tools will be helpful to exchange information.

5. It is possible to construct a case-driven repository so that it can learn from modeling different cases on-the-fly.

APPENDIX C

EXAMPLES FROM DSKs USED IN CASE STUDIES

This appendix includes examples from the Domain Specific Kits that have been used in the case studies and examples in the text. For each of the DSK, one or more excerpts from its language, a screenshot of its development environment and a runtime snapshot of one of its artifacts have been presented.

RIA Presentation Kit

- Excerpts from an EBML file (Figure C.1 and Figure C.2). Figure C.1 shows the structure part of EBML, and Figure C.2 shows the event, rule and data sections.
- A screenshot of EDS (Figure C.3).
- An example screen rendered by ERE (Figure C.4).

Reporting Kit

- A sample JRXML file for report format specification (Figure C.5).
- A sample DSXML file for report content generation (Figure C.6).
- iReport report designer screenshot (Figure C.7).
- A snapshot of a generated report (Figure C.8).

Business Services Kit

- A screenshot of Service Editor (Figure C.9).
- An example ServiceXML definition (Figure C.10).

BPM Kit

- An example JDPL process definition (Figure C.11).
- A screenshot of GPD (Graphical Process Designer) (Figure C.12).
- A snapshot of a process flow (Figure C.13).

RUMBA Business Rules Kit

- A screenshot of RumbaEditor (Figure C.14).

Persistence (POM) Kit

- A screenshot of POM Studio (Figure C.15).
- An example PomXML definition (Figure C.16).

Batch Processing Kit

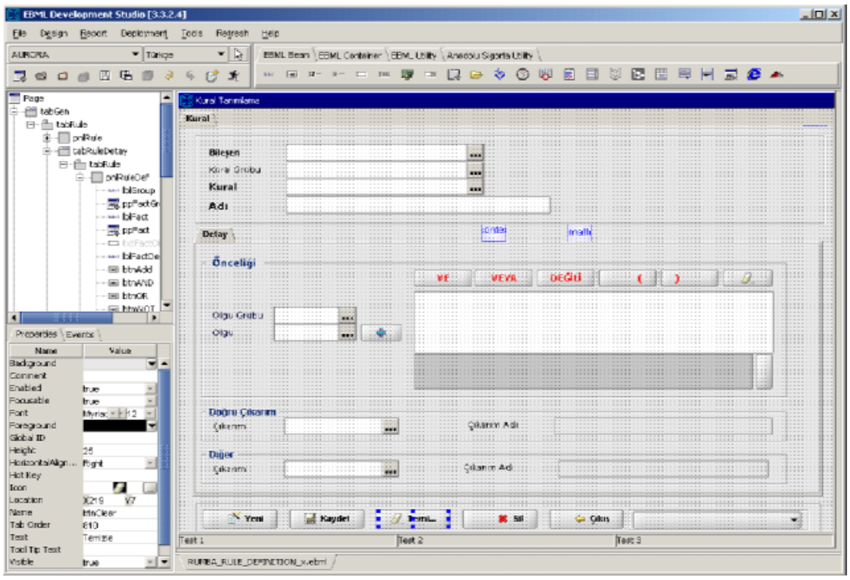
- An example batch job definition with job.xml (Figure C.17).
- A screenshot of Job Management Console (Figure C.18).

```
<ebml v="noversion" pid="VARLIKLARIM.ebml" size="800x620" type="0"
    techVersion="3.0.0" language="tr">
  <interface>
    <structure>
      <bean class="JCSPage" id="Page">
        <style>
          <p name="title">VARLIKLAR</p>
          <p name="pageSize">800,620</p>
        </style>
        <bean class="JCSPanel" id="Page.pnl_Musteri">
          <style>
            <p name="title">Musteri Sorgulama</p>
          </style>
          ...
          <bean class="JCSHandleButtonField" id="...hnd_MusteriNo">
            <style>
              <p name="minCharCount">0</p>
              <p name="popup">PP_INTERNETMUSTERI</p>
              <p name="inputs">...</p>
              <p name="outputs">...</p>
            </style>
          </bean>
      </bean>
      <bean class="JCSTabbedPane" id="Page.tab_Varliklar">
        <style>
          <p name="bounds">10,140,765,355</p>
          <p name="focusable">true</p>
        </style>
        <bean class="JCSTabPage" id="Page.tab_Varliklar.tab_Pasif">
          <style>
            <p name="bounds">2,25,760,327</p>
            <p name="tabTitle">Pasifler&Riskler</p>
          </style>
          <bean class="JCSTable" id="...table_Hesaplar">
            <style>
              <p name="reorderingAllowed">true</p>
              <p name="minimumRowCount">1</p>
              <p name="bounds">13,30,740,270</p>
              <p name="focusable">true</p>
              <p name="adapterInfo">
              <columns>
                <column id="MUSTERINO" type="TEXT_ADAPTER" ...>
                  <style>
                  ...
                  </style>
                </column>
        ...
        <bean class="JCSRegion" id="Page.rg_Genel">
          <style>
            <p name="regionName">RG_SBUTTONS</p>
            ...
          </style>
        </bean>
        ...
      </bean>
    </structure>
```

Figure C.1: An example EBML file (definition of structure)

```
    <events>
      <lC type="action" ref="ActPrint">
        <p m="setData" id="Page.JCSTextPrinter58796">
          <var id="vVarlikText"/>
        </p>
        <p m="print" id="Page.JCSTextPrinter58796"/>
      </lC>
      <lC type="action" ref="actSorgula">
        <p m="cleanup" id="Page.tab_Varliklar.tab_Pasif"/>
        <p m="cleanup" id="Page.tab_Varliklar.tab_Talimatlar"/>
        <p m="cleanup" id="Page.tab_Varliklar"/>
        <p m="setText" id="Page.pnl_Musteri.txt_Aciklama"/>
        <p m="setText" id="Page.pnl_Musteri.txt_MusteriAd"/>
        <p m="setText" id="Page.pnl_Musteri.txt_MusteriSoyad"/>
        <call service="SBIB_MUSTERI_BILGI_GETIR" status="0">
          <inputs>
            <p n="MUSTERINO" id="Page.pnl_Musteri.hnd_MusteriNo"/>
          </inputs>
          <outputs>
            <p n="AD" id="Page.pnl_Musteri.txt_MusteriAd"/>
            <p n="SOYAD" id="Page.pnl_Musteri.txt_MusteriSoyad"/>
            <p n="ALLINFO" id="Page.pnl_Musteri.txt_Aciklama"/>
          </outputs>
        </call>
        <call service="SBIB_GET_VARLIKLAR" status="0">
          <inputs>
            <p m="getText" n="MUSTERINO" id="hnd_MusteriNo"/>
          </inputs>
          <outputs>
            <p m="getText" n="ACIKLAMA" id="txt_Aciklama"/>
            <p m="getName" n="VARLIKTABLO" id="tbl_Hesaplar"/>
            <p n="TALIMATTABLO" id="tab_Talimatlar"/>
            <p m="getText" n="AD" id="txt_MusteriAd"/>
            <p m="getText" n="SOYAD" id="txt_MusteriSoyad"/>
          </outputs>
        </call>
      ...
    </events>
  </interface>
  <data>
    <var id="vMusteriNo"/>
    <var id="vHesapNo"/>
    <var id="vHesapTipi"/>
    ...
  </data>
  <ruleset>
    <rule id="VLHESAPRULE">
      <call rule="VHESAPRULE">
        <inputs>
          <p m="getText" n="VARLIKTIPI" id="...tbl_Hesaplar.VARLIKTIPI"/>
          <p>1</p>
        </inputs>
      </call>
    </rule>
    ...
  </ruleset>
</ebml>
```

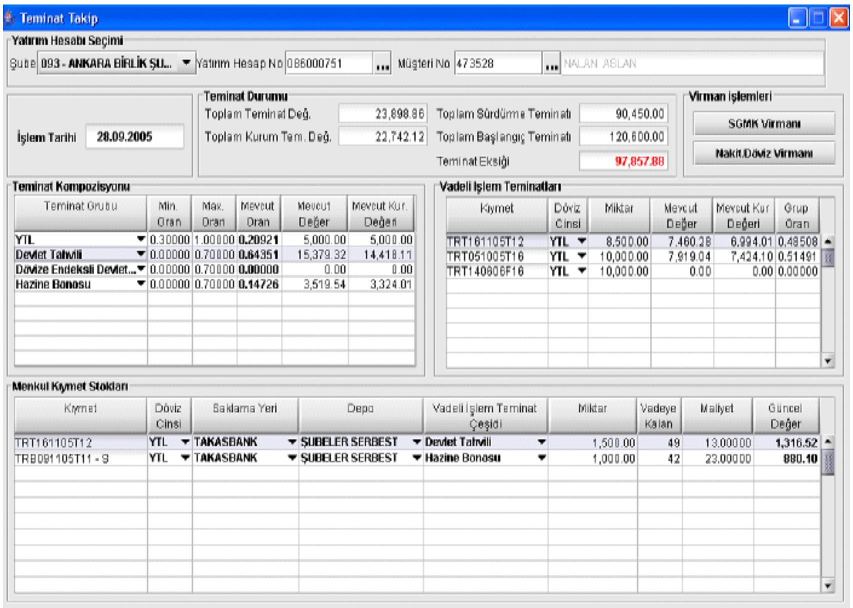
Figure C.2: An example EBML file (definition of events)



Properties Events

Name	Value
Background	
Comment	
Enabled	true
Focusable	true
Font	Myriad 12
Foreground	■
Global ID	
Height	25
HorizontalAlign...	Right
Hot Key	
Icon	
Location	X219 Y7
Name	btnClear
Tab Order	810
Text	Temizle
Tool Tip Text	
Visible	true

Figure C.3: A screenshot of EDS



Yatırım Hesabı Seçimi

Şube 093 - ANKARA BİRLİK ŞU... Yatırım Hesap No 086000751 Müşteri No 473528 NALAN ASLAN

İşlem Tarihi 28.09.2005

Teminat Durumu

Toplam Teminat Değ.	23,898.86	Toplam Sürdürme Teminatı	90,450.00
Toplam Kurum Tem. Değ.	22,742.12	Toplam Başlangıç Teminatı	120,600.00
		Teminat Eksiği	97,857.88

Virman İşlemleri

Teminat Kompozisyonu

Teminat Grubu	Min. Oran	Max. Oran	Mevcut Oran	Mevcut Değer	Mevcut Kur. Değeri
YTL	0.30000	1.00000	0.20921	5,000.00	5,000.00
Devlet Tahvili	0.00000	0.70000	0.64351	15,379.32	14,418.11
Dövize Endeksli Devlet...	0.00000	0.70000	0.00000	0.00	0.00
Hazine Bonosu	0.00000	0.70000	0.14726	3,519.54	3,324.01

Vadeli İşlem Teminatları

Kıymet	Döviz Cinsi	Miktar	Mevcut Değer	Mevcut Kur Değeri	Grup Oran
TRT161105T12	YTL	8,500.00	7,460.28	6,994.01	0.48508
TRT051005T16	YTL	10,000.00	7,919.04	7,424.10	0.51491
TRT140606F16	YTL	10,000.00	0.00	0.00	0.00000

Menkul Kıymet Stokları

Kıymet	Döviz Cinsi	Saklama Yeri	Depo	Vadeli İşlem Teminat Çeşidi	Miktar	Vadeye Kalan	Maliyet	Güncel Değer
TRT161105T12	YTL	TAKASBANK	ŞUBELER SERBEST	Devlet Tahvili	1,500.00	49	13.00000	1,316.52
TRB091105T11 - S	YTL	TAKASBANK	ŞUBELER SERBEST	Hazine Bonosu	1,000.00	42	23.00000	880.10

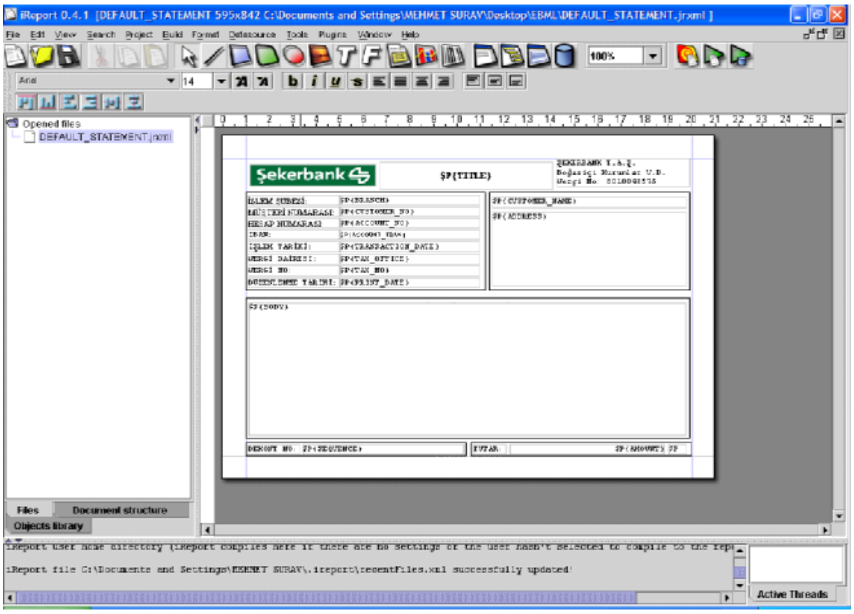
Figure C.4: A screen rendered by ERE

```
<!-- Created with iReport - A designer for JasperReports -->
<!DOCTYPE jasperReport PUBLIC "//JasperReports//DTD␣Report␣Design//EN"
  "http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">
<jasperReport name="DEFAULT_STATEMENT"
     columnCount="1" printOrder="Vertical" orientation="Portrait"
     pageWidth="595" pageHeight="842" columnWidth="535"
     columnSpacing="0" leftMargin="28" rightMargin="28"
     topMargin="28" bottomMargin="28" whenNoDataType="NoPages"
     isTitleNewPage="false" isSummaryNewPage="false">
  <property name="ireport.scriptlethandling" value="0" />
  <property name="ireport.encoding" value="UTF-8" />
  <import value="java.util.*" />
  <import value="net.sf.jasperreports.engine.*" />
  <import value="net.sf.jasperreports.engine.data.*" />
  <parameter name="TITLE" class="java.lang.String" ...>
    <defaultValueExpression>
      <![CDATA["DEKONT"]]></defaultValueExpression>
  </parameter>
  <parameter name="CUSTOMER_NO" class="java.lang.String" .../>
  <parameter name="ACCOUNT_NO" class="java.lang.String" .../>
  <parameter name="TRANSACTION_DATE" class="java.lang.String" .../>
  <parameter name="CURRENCY" class="java.lang.String" ...>
    <defaultValueExpression>
      <![CDATA["YTL"]]></defaultValueExpression>
  </parameter>
  ...
  <background>
    <band height="365"  isSplitAllowed="true" >
      <rectangle radius="0" >
        <reportElement mode="Transparent" x="0" y="193"
          width="537" height="149"
          forecolor="#000000" backcolor="#FFFFFF" key="rectangle-1"/>
        <graphicElement stretchType="NoStretch" pen="Thin" .../>
      </rectangle>
      <textField isStretchWithOverflow="true" ...>
        <reportElement mode="Opaque" x="162" y="4"
          width="209" height="60"
          forecolor="#000000" backcolor="#FFFFFF" key="textField-1"/>
        <box topBorder="None" topBorderColor="#000000" .../>
        <textElement textAlignment="Center" ...>
          <font fontName="SansSerif" pdfFontName="tahoma.ttf" .../>
        </textElement>
      <textFieldExpression ...>
        <![CDATA[$P{TITLE}]]></textFieldExpression>
      </textField>
      ...
    </band>
  </background>
  <title>...</title>
  <pageHeader>...</pageHeader>
  <columnHeader>...</columnHeader>
  <detail>...</detail>
  <columnFooter>...</columnFooter>
  <pageFooter>...</pageFooter>
  <summary>...</summary>
</jasperReport>
```

Figure C.5: A sample JRXML file

```
<report v="1.00" type="service" csvSupport="true">
      <service>UTL_PREPARE_DEFAULT_STATEMENT</service>
            <report-parameters>
                  <param>TITLE</param>
                  <param>SEQUENCE</param>
                  <param>AMOUNT</param>
                  <param>BODY</param>
                  <param>CUSTOMER_NAME</param>
                  <param>ADDRESS</param>
                  <param>BRANCH</param>
                  <param>CUSTOMER_NO</param>
                  <param>ACCOUNT_NO</param>
                  <param>TRANSACTION_DATE</param>
                  <param>TAX_OFFICE</param>
                  <param>TAX_NO</param>
                  <param>PRINT_DATE</param>
                  <param>CURRENCY</param>
            </report-parameters>
</report>
```

Figure C.6: A sample DSXML file


iReport 0.4.1 [DEFAULT_STATEMENT 595x842 C:\Documents and Settings\MEHMET SURAV\Desktop\EBML\DEFAULT_STATEMENT.jrxml]
File Edit View Search Project Build Format Datasource Tools Plugins Window Help
100%
Arial
14
Opened files
DEFAULT_STATEMENT.jrxml
Şekerbank
Files
Document structure
Objects library
Active Threads

$P{TITLE}

ŞEKERBANK T.A.Ş.
Boğaziçi Kurumlar V.D.
Vergi No: 8010048575

İŞLEM ŞUBESİ: $P{BRANCH}
MÜŞTERİ NUMARASI: $P{CUSTOMER_NO}
HESAP NUMARASI $P{ACCOUNT_NO}
IBAN: $P{ACCOUNT_IBAN}
İŞLEM TARİHİ: $P{TRANSACTION_DATE}
VERGİ DAİRESİ: $P{TAX_OFFICE}
VERGİ NO: $P{TAX_NO}
DÜZENLEME TARİHİ: $P{PRINT_DATE}

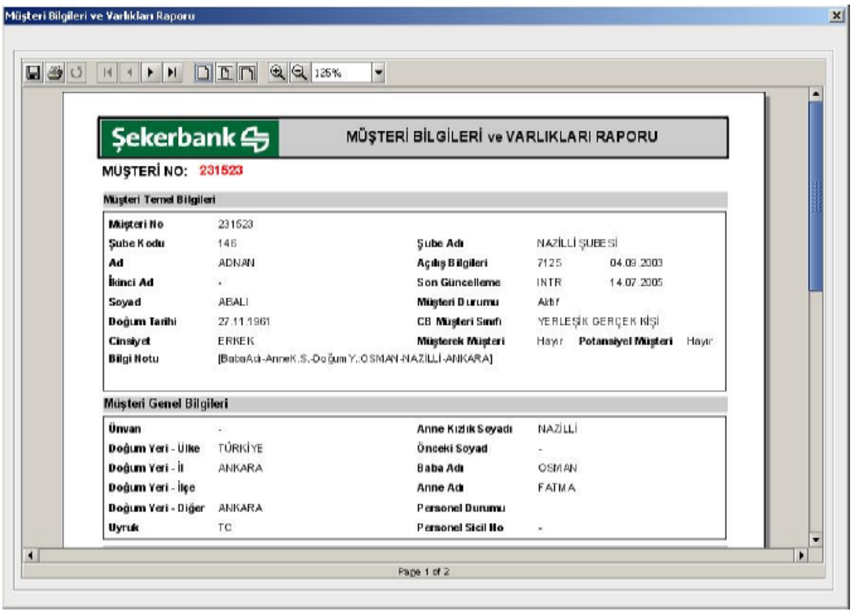
$P{CUSTOMER_NAME}
$P{ADDRESS}

$P{BODY}

DEKONT NO: $P{SEQUENCE}
TUTAR: $P{AMOUNT} $P

```
iReport user home directory (iReport compiles here if there are no settings or the user hasn't selected to compile to the rep
iReport file C:\Documents and Settings\MEHMET SURAV\.ireport\recentFiles.xml successfully updated!
```

Figure C.7: A screenshot of iReport



MÜŞTERİ BİLGİLERİ ve VARLIKLARI RAPORU

MÜŞTERİ NO: 231523

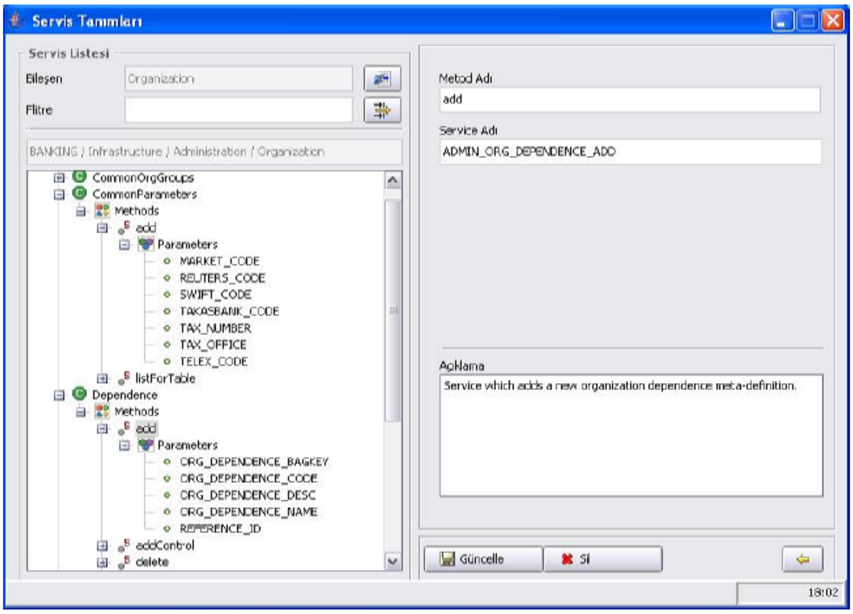
Müşteri Temel Bilgileri

Müşteri No	231523				
Şube Kodu	146	Şube Adı	NAZİLLİ ŞUBESİ		
Ad	ADNAN	Açılış Bilgileri	7125	04.09.2003	
İkinci Ad	-	Son Güncelleme	INTR	14.07.2005	
Soyad	ABALI	Müşteri Durumu	Aktif		
Doğum Tarihi	27.11.1961	CB Müşteri Sınıfı	YERLEŞİK GERÇEK KİŞİ		
Cinsiyet	ERKEK	Müşterek Müşteri	Hayır	Potansiyel Müşteri	Hayır
Bilgi Notu	[BabaAdı-AnneK.S.-DoğumY.:OSMAN-NAZİLLİ-ANKARA]				

Müşteri Genel Bilgileri

Ünvan	-	Anne Kızlık Soyadı	NAZİLLİ
Doğum Yeri - Ülke	TÜRKİYE	Önceki Soyad	-
Doğum Yeri - İl	ANKARA	Baba Adı	OSMAN
Doğum Yeri - İlçe		Anne Adı	FATMA
Doğum Yeri - Diğer	ANKARA	Personel Durumu	
Uyruk	TC	Personel Sicil No	-

Figure C.8: A snapshot of a generated report



Bileşen: Organization

Flitre:

BANKING / Infrastructure / Administration / Organization

- CommonOrgGroups
- CommonParameters
  - Methods
    - add
      - Parameters
        - MARKET_CODE
        - REUTERS_CODE
        - SWIFT_CODE
        - TAKASBANK_CODE
        - TAX_NUMBER
        - TAX_OFFICE
        - TELEX_CODE
    - listForTable
- Dependence
  - Methods
    - add
      - Parameters
        - ORG_DEPENDENCE_BAGKEY
        - ORG_DEPENDENCE_CODE
        - ORG_DEPENDENCE_DESC
        - ORG_DEPENDENCE_NAME
        - REFERENCE_ID
    - addControl
    - delete

Metod Adı: add

Service Adı: ADMIN_ORG_DEPENDENCE_ADD

Açıklama: Service which adds a new organization dependence meta-definition.

Figure C.9: A screenshot of Service Editor

```
<services>
   <service>
      <service-name>ADMIN_ORG_DEPENDENCE_ADD</service-name>
      <domain-name>INV</domain-name>
      <project-code>Infrastructure</project-code>
      <component-group>Administration</component-group>
      <component>Organization</component>
      <class-name>...administration.organization.Dependence</class-name>
      <class-method>add</class-method>
      <is-batch>false</is-batch>
      <description>Adds a new organization
            dependence meta-definition.</description>
      <parameters>
         <param name="ORG_DEPENDENCE_BAGKEY" data-type="String"
            io-type="input" is-column="" is-mandatory="true" />
         <param name="ORG_DEPENDENCE_CODE" data-type="String"
            io-type="input" is-column="" is-mandatory="true" />
         <param name="ORG_DEPENDENCE_DESC" data-type="String"
            io-type="input" is-column="" is-mandatory="true" />
         <param name="ORG_DEPENDENCE_NAME" data-type="String"
            io-type="input" is-column="" is-mandatory="true" />
      </parameters>
   </service>
   <service>
      <service-name>ADMIN_ORG_COMMON_PARAMETERS_ADD</service-name>
      <domain-name>INV</domain-name>
      <project-code>Infrastructure</project-code>
      <component-group>Administration</component-group>
      <component>Organization</component>
      <class-name>...organization.CommonParameters</class-name>
      <class-method>add</class-method>
      <is-batch>false</is-batch>
      <description>Adds common parameters to a new org.</description>
      <parameters>
         <param name="MARKET_CODE" data-type="String"
            io-type="input" is-column="" is-mandatory="true" />
         <param name="REUTERS_CODE" data-type="String"
            io-type="input" is-column="" is-mandatory="true" />
         <param name="SWIFT_CODE" data-type="String"
            io-type="input" is-column="" is-mandatory="true" />
         <param name="TAKASBANK_CODE" data-type="String"
            io-type="input" is-column="" is-mandatory="true" />
         <param name="TAX_NUMBER" data-type="String"
            io-type="input" is-column="" is-mandatory="true" />
         <param name="TAX_OFFICE" data-type="String"
            io-type="input" is-column="" is-mandatory="true" />
         <param name="TELEX_CODE" data-type="String"
         io-type="input" is-column="" is-mandatory="true" />
      </parameters>
   </service>
   ...
</services>
```

Figure C.10: An example service definition

```
<process-definition name="78000" label="MEROPSTalepGiris"
  processGroup="..." componentOid="..." xmlns="urn:jbpm.org:jpdl-3.1">
  <start-state name="Baslat">
    <transition name="initializeProcess"
        to="IslemKontrol"></transition>
  </start-state>
  <node name="IslemKontrol">
    <event type="node-enter">
      <check-all name="check-all"></check-all>
      <set-status process="20802" history="20802"/>
    </event>
    <transition name="OperasyonYoneticisineGonder"
        to="BaslatanKontrol"></transition>
  </node>
  <decision name="BaslatanKontrol">
    <handler class="MakerDecisionHandler">
      <makers>
        <maker>
          <channelCode>01</channelCode>
          <organizationType></organizationType>
          <organizationGroup></organizationGroup>
          ...
          <transitionRef>checkerControlDecision</transitionRef>
        </maker>
      </makers>
    </handler>
    <transition name="TalepOnayaGit" to="TalepOnay"></transition>
    <transition name="checkerControlDecision"
        to="checkerControlDecision"></transition>
  </decision>
  <task-node name="TalepOnay">
    <task name="TalepOnay">
      <assignment class="CSAssignmentHandler">
        <actors type="ANY">
          <actor>
            <actorCount>1</actorCount>
            <rule></rule>
            <organization>@</organization>
            <unit></unit>
            <profile>021</profile>
            <screen name="RG_MOP_CORE_REQUEST_DEFINITION"/>
          </actor>
        </actors>
      </assignment>
    </task>
    <transition name="Onayla" to="checkerControlDecision"></transition>
    <transition name="GeriGonder" to="TalepDuzeltme"></transition>
  </task-node>
  <decision name="checkerControlDecision">
    <handler class="CheckerControlDecisionHandler"/>
      <transition name="true" to="CheckerKontrolu"/>
      <transition name="false" to="MakerTalebiBaslat"/>
  </decision>
  ...
</process-definition>
```

Figure C.11: An example JDPL process definition

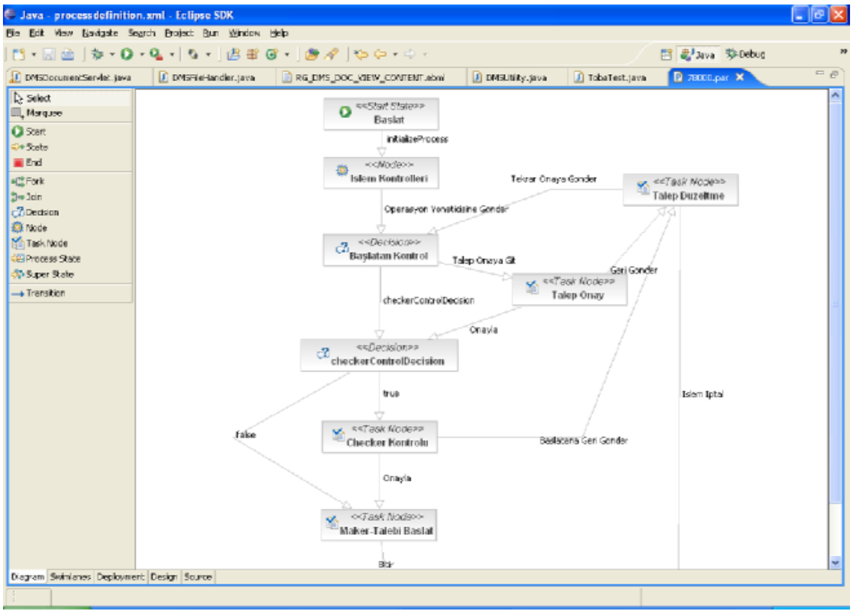


Figure C.12: A screenshot of GPD (Graphical Process Designer)

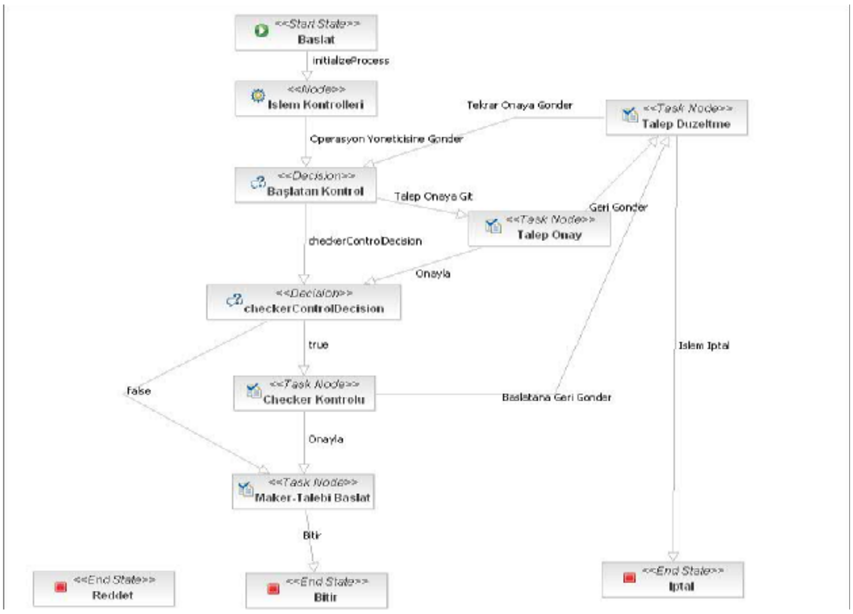
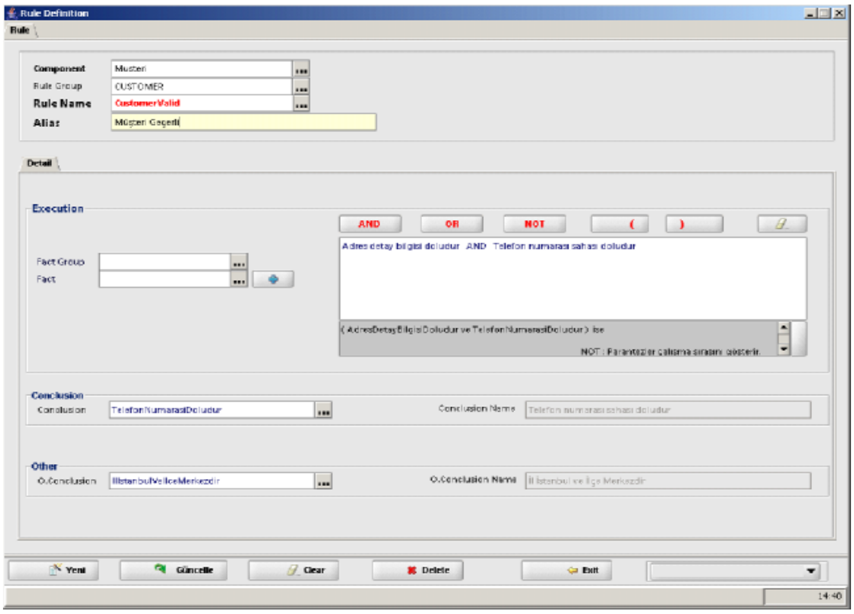


Figure C.13: A snapshot of a process flow



Component: Musteri
Rule Group: CUSTOMER
Rule Name: CustomerValid
Alias: Müşteri Geçerli

Fact Group:
Fact:

Adres detay bilgisi doludur AND Telefon numarası sahası doludur

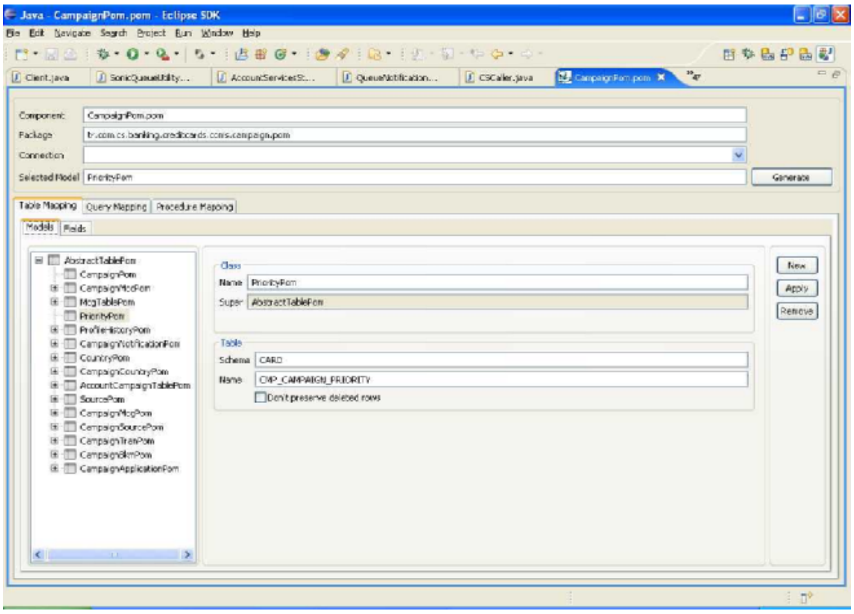
(AdresDetayBilgisiDoludur ve TelefonNumarasiDoludur) ise

NOT : Parantezler çalışma sırasını gösterir.

Conclusion: TelefonNumarasiDoludur
Conclusion Name: Telefon numarası sahası doludur

O.Conclusion: IlIstanbulVeIlceMerkezdir
O.Conclusion Name: Il İstanbul ve İlçe Merkezdir

Figure C.14: A snapshot of RUMBA Editor



Component: CampaignPom.pom
Package: tr.com.cs.banking.creditcards.ccms.campaign.pom
Connection:
Selected Model: PriorityPom

- AbstractTablePom
  - CampaignPom
  - CampaignMccPom
  - MccTablePom
  - PriorityPom
  - ProfileHistoryPom
  - CampaignNotificationPom
  - CountryPom
  - CampaignCountryPom
  - AccountCampaignTablePom
  - SourcePom
  - CampaignMccPom
  - CampaignSourcePom
  - CampaignTranPom
  - CampaignBinPom
  - CampaignApplicationPom

Name: PriorityPom
Super: AbstractTablePom

Schema: CARD
Name: CMP_CAMPAIGN_PRIORITY
□ Don't preserve deleted rows

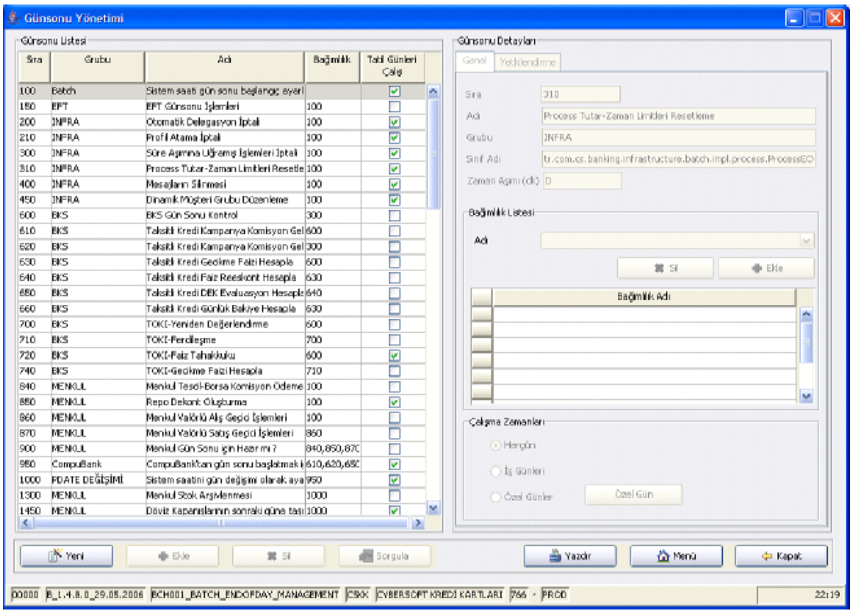
Figure C.15: A screenshot of PomStudio

```
<component name="CampaignPom.pom"
   package="tr.com.cs.banking.creditcards.ccms.campaign.pom">
  <connection driver="oracle.jdbc.driver.OracleDriver"
   jndiname="BankingDS" name="Banking␣Datasource" .../>
    <model dbSchema="" dbTable="" name="AbstractTablePom" >
      <field dbColumnConstraint="PRIMARYKEY" dbColumnName="OID"
            dbColumnPrecision="16" dbColumnScale="0"
            dbColumnType="CHAR" javaType="String" name="oID" />
      <field dbColumnName="STATUS" .../>
      <field dbColumnName="LASTUPDATED" .../>
      <model dbSchema="CARD"
            dbTable="CMP_CAMPAIGN_NOTIFICATION"
            name="CampaignNotificationPom" >
        <field dbColumnConstraint="NONE" dbColumnName="CAMPAIGN_OID"
         dbColumnPrecision="16" dbColumnScale="0"
         dbColumnType="CHAR" javaType="String" name="campaignOid" />
        <field dbColumnName="NOTIFICATION_TYPE" .../>
        <field dbColumnName="MESSAGE_CODE" .../>
        <field dbColumnName="NOTIFICATION_TIME" .../>
        <field dbColumnName="NOTIFICATION_CHANNEL" .../>
      </model>
      ...
    </model>
  <model dbSchema="" dbTable="" name="AbstractQueryPom" >
    <field dbColumnConstraint="PRIMARYKEY" />
    <field dbColumnConstraint="NOT␣NULL" />
    <field dbColumnConstraint="NOT␣NULL" />
    <model name="NotificationQueryPom" >
      <query>SELECT ’1’ AS LASTUPDATED, N.STATUS, ...
            FROM CARD.CMP_CAMPAIGN C,
            CARD.CMP_CAMPAIGN_NOTIFICATION N</query>
      <field dbColumnName="NOTIFICATION_TYPE"
            dbColumnType="VARCHAR" javaType="String"
            name="notificationType" />
      <field dbColumnName="MESSAGE_CODE"
            dbColumnType="VARCHAR" javaType="String"
            name="messageCode" />
      <field dbColumnName="NOTIFICATION_TIME" .../>
      <field dbColumnName="NOTIFICATION_CHANNEL" .../>
    </model>
    ...
  </model>
  <model dbSchema="" dbTable="" name="AbstractProcedurePom" ...>
  ...
  </model>
</component>
```

Figure C.16: An example PomXML definition

```
<batch-jobs>
   <job>
      <job-name>EFT Gateway Listener3</job-name>
      <job-group>EFT</job-group>
      <description>EFT Gateway Listener for
            branches ending with 3</description>
      <job-service-name>...batch.impl.eft.EFTReadL1L3</job-service-name>
      <is-durable>false</is-durable>
      <is-volatile>false</is-volatile>
      <is-stateful>false</is-stateful>
      <requires-recovery>false</requires-recovery>
      <recovery-service-name/>
      <job-param-data>
         <param name="READ_INDEX" value="3" data-type="String"
         is-column="" is-mandatory="true"/>
      </job-param-data>
      <is-manuel>false</is-manuel>
      <thread-count>1</thread-count>
   </job>
   ...
</batch-jobs>
```

Figure C.17: An example job definition



Sıra	Grubu	Adı	Bağımlılık	Tatil Günleri Çalış
100	Batch	Sistem saati gün sonu başlangıç ayarl		☑
150	EFT	EFT Günsonu İşlemleri	100	☐
200	INFRA	Otomatik Delegasyon İptal	100	☑
210	INFRA	Profil Atama İptal	100	☑
300	INFRA	Süre Aşımına Uğramış İşlemleri İptal	100	☑
310	INFRA	Process Tutar-Zaman Limitleri Resetle	100	☑
400	INFRA	Mesajların Silinmesi	100	☑
450	INFRA	Dinamik Müşteri Grubu Düzenleme	100	☑
600	BKS	BKS Gün Sonu Kontrol	300	☐
610	BKS	Taksitli Kredi Kampanya Komisyon Gel	600	☐
620	BKS	Taksitli Kredi Kampanya Komisyon Gel	300	☐
630	BKS	Taksitli Kredi Gecikme Faizi Hesapla	600	☐
640	BKS	Taksitli Kredi Faiz Reeskont Hesapla	630	☐
650	BKS	Taksitli Kredi DEK Evaluasyon Hesapla	640	☐
660	BKS	Taksitli Kredi Günlük Bakiye Hesapla	630	☐
700	BKS	TOKİ-Yeniden Değerlendirme	600	☐
710	BKS	TOKİ-Ferdileşme	700	☐
720	BKS	TOKİ-Faiz Tahakkuku	600	☑
740	BKS	TOKİ-Gecikme Faizi Hesapla	710	☐
840	MENKUL	Menkul Tescil-Borsa Komisyon Ödeme	100	☐
850	MENKUL	Repo Dekont Oluşturma	100	☑
860	MENKUL	Menkul Valörlü Alış Geçici İşlemleri	100	☐
870	MENKUL	Menkul Valörlü Satış Geçici İşlemleri	860	☐
900	MENKUL	Menkul Gün Sonu için Hazır mı ?	840,850,870	☐
950	CompuBank	CompuBank'tan gün sonu başlatmak i	610,620,650	☑
1000	PDATE DEĞİŞİMİ	Sistem saatini gün değişimi olarak aya	950	☑
1300	MENKUL	Menkul Stok Arşivlenmesi	1000	☐
1450	MENKUL	Döviz Kapanışlarının sonraki güne taşı	1000	☑

Sıra: 310

Adı: Process Tutar-Zaman Limitleri Resetleme

Grubu: INFRA

Sınıf Adı: tr.com.cs.banking.infrastructure.batch.impl.process.ProcessEO

Zaman Aşımı (dk): 0

Adı:

	Bağımlılık Adı

● Hergün

○ İş Günleri

○ Özel Günler

00000 | B_1.4.8.0_29.05.2006 | BCH001_BATCH_ENDOFDAY_MANAGEMENT | CSKK | CYBERSOFT KREDİ KARTLARI | 766 - PROD | 22:19

Figure C.18: A screenshot of Job Management Console

APPENDIX D

DESCRIPTION OF ASSETS IN CASE STUDIES

This appendix includes the brief description of software assets in INV and FGW product lines. Table D.2 and Table D.1 include the FGW and INV assets, respectively. The same assets may exist in both tables.

Table D.1: Description of INV product line assets

Assets	**Description**
Customer Core	Management of customer core information, it includes only personal and basic required information
Customer Advanced	Management of comprehensive information of customers. It supports approval workflows and several integrations.
Blacklist Manager	Management of blacklist pools (external or internal) and provides support for customer linkage
Document Manager	Central document manager (See Figure 5.3 and Table 5.1)
Account Manager	Management of customer accounts and assets
Deduction	Determination, calculation and recording of deductions of transactions
Accounting Gateway	Proxy component to access accounting system
Accounting	Accounting system (either this or the previous one can be active)
Administration	Core common infrastructure to manage organization and declarative authorization using profiles and ACLs
Ext. System Data Transfer	File transfer utilities for sending/receiving to/from external systems
Alert and Notification Man.	Unified alert and notification utility integrated with channels
Repo	Operations of repo/reverse repo
Fixed Income Common	Management of core definitions and common primitive operations
Fixed Income Trade	FIS Buy/Sell operations
BPP	BPP Operations
Asset Delivery	Physical delivery of assets
Auction	Customer auction operations
Asset Lending	Lending and tracking assets
DEX Operations	Customer Derivative operations
Equity Common Operations	Common core infrastructure and operations for equity management
Order Management	Equity order management
Credit	Equity trade with credit option
Capital Increase	Capital increase operations
Public Offering	Public offering operations
Mutual Fund Buy/Sell Ops.	Buy/Sell operations of mutual funds
Fund Transfer	Transfer operations of mutual funds
Fund Man. Backend	Fund management core business operations from the viewpoint of fund managers.
Portfolio Man. Backend	Portfolio management core business operations
Asset/Stock Invest Core	Common infrastructure for asset management and primitive operations including the common reporting
Cash Invest Core	Common cash operations
Asset Transfer	Common asset transfer operations

Table D.2: Description of FGW product line assets

Assets	**Description**
Customer Core	Management of customer core information, it includes only personal and basic required information
Customer Advanced	Management of comprehensive information of customers. It supports approval workflows and several integrations.
Blacklist Manager	Management of blacklist pools (external or internal) and provides support for customer linkage
Document Manager	Central document manager (See Figure 5.3 and Table 5.1)
Account Manager	Management of customer accounts and assets
Deduction	Determination, calculation and recording of deductions of transactions
Accounting Gateway	Proxy component to access accounting system
Accounting	Accounting system (either this or the previous one can be active)
Administration	Core common infrastructure to manage organization and declarative authorization using profiles and ACLs
Ext. System Data Transfer	File transfer utilities for sending/receiving to/from external systems
Alert and Notification Man.	Unified alert and notification utility integrated with channels
FGW Core	Common core utilities for financial gateways. Provides common client application and administration services for financial gateways
FGW Communication	Common communication infrastructure for financial gateways
EFT Messaging (HLP)	Management of EFT messaging and Host Link Protocol
EFT Operations	Management of EFT requests and operations
KKB KRS	Management of KRS (Credit Risk Search) messages
KKB LKS	Management of KRS (Limit Control System) messages
CRA Electronic Registry	Administration of electronic registry operations
CRA Core Operations	Common operations including extra administration utilities for CRA gateway

VITA

Personal Information

Surname, Name	: Altıntaş, Nesip İlker
Nationality	: Turkish (TC)
Date and Place of Birth	: 17 December 1970, Aydın
Marital Status	: Married, One child.
Phone	: +90 532 442 96 30
email	: ilker.altintas@cs.com.tr

Education

Degree	**Institution**	**Year of Graduation**
M.Sc.	METU Computer Engineering, Ankara	1995
B.S.	METU Computer Engineering, Ankara	1992
High School	Maltepe Military High School, İzmir	1988

Professional Experience

Years	**Place**	**Enrollment**
2000–Present	Cybersoft Information Technologies, Co.	Projects Coordinator Project Manager
1999–2000	MONAD Software and Consultancy, Co.	Senior Software Engineer and Chief Architect
1997–1998	METU Computer Engineering, Ankara	Senior Software Engineer
1992–1997	METU Computer Engineering, Ankara	Research Assistant

Foreign Languages

Fluent English

Awards and Honors

1. 1994 TUBITAK Husamettin Tugac Research Award ("Metu Object-Oriented DBMS" project).
2. 1992 Best Senior Project Award in Computer Engineering Department, METU. ("Turkish Word Processor and Speller" project).
3. 1992 High Honor Student in B.S. (Cum GPA: 3.75 / 4.00).
4. 1988 Turkish Physics Olympiads, Aegean Region, 2nd Degree.

Publications

1. N. I. Altintas, S. Cetin, and A. H. Dogru. Industrializing software development: The “Factory Automation” way. In D. Draheim and G. Weber, editors, Trends in Enterprise Application Architecture, TEAA 2006, Berlin, Germany, Revised Selected Papers, volume 4473 of LNCS, pages 54-68. Springer, 2007.
2. S. Cetin, N. I. Altintas, H. Oguztuzun, A. H. Dogru, O. Tufekci, and S. Suloglu. Legacy migration to service-oriented computing with mashups. In ICSEA’07: Proceedings of the International Conference on Software Engineering Advances. IEEE Computer Society, 2007.
3. S. Cetin, N. I. Altintas, H. Oguztuzun, A. H. Dogru, O. Tufekci, and S. Suloglu. A mashup-based strategy for migration to service-oriented computing. In ICPS’07: IEEE International Conference on Pervasive Services, July 15 - 20, 2007, Istanbul, Turkey, pages 169-172. IEEE Computer Society, 2007.
4. S. Cetin, N. I. Altintas, and C. Sener. An architectural modeling approach with symmetric alignment of multiple concern spaces. In ICSEA’06: Proceedings of the International Conference on Software Engineering Advances, page 48, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
5. S. Cetin, N. I. Altintas, and R. Solmaz. Business rules segregation for dynamic process management with an aspect-oriented framework. In Business Process Management International Workshops, Vienna, Austria, September 4-7, 2006, Proceedings, volume 4103 of LNCS, pages 193-204. Springer, 2006.
6. S. Cetin, N. I. Altintas, and O. Tufekci. Improving model reuse with domain specific kits. In International Workshop on Model Reuse Strategies, MoRSe 2006, Warsaw, Poland, Oct. 17, 2006, 2006.
7. O. Tufekci, S. Cetin, and N. I. Altintas. How to process [business] processes. In Integrated Design and Process Technology, IDPT-2006, Society for Design and Process Science, San Diego, CA, USA, June 2006.
8. N. I. Altintas and S. Cetin. Integrating a software product line with rule-based business process modeling. In D. Draheim and G. Weber, editors, Trends in Enterprise Application Architecture, VLDB Workshop, TEAA 2005, Trondheim, Norway, Revised Selected Papers, volume 3888 of LNCS, pages 15-28. Springer, 2006.
9. N. I. Altintas, M. Surav, O. Keskin, and S. Cetin. Aurora software product line. In Turkish Software Architecture Workshop, Ankara, September 2005, 2005.
10. N. I. Altintas and H. C. Bozsahin, Reducing the Order Bias in Incremental Learning, TAINN-VI, 6th Turkish Symposium on Artificial Intelligence and Neural Networks, June 1997, Ankara, Turkey.
11. Y. Ceken, I. Altintas, M. Altinel, H. Guven, and A. Dogac, Experiences in Design and Implementation of a Health Care Information System, in Proc. of Intl. ORACLE User Week, San Fransisco, November 1996.
12. A. Dogac, M. Altinel, C. Ozkan, I. Durusoy, and I. Altintas, Design and Implementation of an Object-Oriented DBMS Kernel, 10th Intl. Symposium on Computer and Information Systems, Izmir, November 1995.

13. A. Dogac, M. Altinel, C. Ozkan, B. Arpinar, I. Durusoy and I. Altintas, METU Object-Oriented DBMS Kernel, DEXA 1995, 6th Intl. Conference and Workshop on Database and Expert Systems Applications, September 4-8 1995, London, UK.
14. A. Dogac, C. Ozkan, B. Arpinar, C. Evrendilek, I. Altintas, et al. METU Object-Oriented DBMS, ACM-SIGMOD 1994. Prototype demo description. Minneapolis, USA.

Hobbies

Economics, Sports and Aquarium World.