Abstract

Two recent approaches in software engineering are capturing the attention of the academic community as well as the industry: model driven engineering and software product lines. The software product line approach to software development is being put into practice from several years and the results seen very promising. However, the resources required to implement this approach are very significant and, as such, a wide adoption of the software product line approach is still not a reality. This is in some sort a result of the *heavy* methods that such an approach requires. These methods usually require the use of high level abstractions to capture domain knowledge which is commonly represented using models. These abstractions are usually *far* from the abstractions used to implement the software solutions. Also, the processes are more complex than the ones commonly used for developing single software systems, since they imply a domain engineering approach to develop reusable assets as well as an application engineering approach to develop final applications in the domain.

Model driven engineering approaches promise to promote the use of models as assets of development to the same *level* as code. Models should be treated as first class entities in software development, similarly to what happens with code. To achieve this goal, some technologies have been proposed, such as, metamodeling frameworks and model transformation languages. These technologies have the potential to support the automation of many software development tasks of *heavy* methods, particularly those that rely significantly on modeling activities and that use models as a source for producing software. This is the case for the majority of the software product line development methods.

This thesis is about supporting the development of software product lines by adopting a model driven engineering approach. For that, we start by analyzing major domain engineering and product line methods. We identify and discuss fundamental concepts for product line development. Resulting from that, a set of common and crucial activities for software product line development are identified and contextualized in a variant of software product line development that we call domain-specific platform approach. This thesis presents methodological approaches and techniques to support the referred activities using several *kinds* of models and a model driven engineering approach. We particularly show how concepts, such as commonality and variability, can be modeled at different levels of abstraction during the development activities. We present approaches on how to relate concepts that are present at different levels of abstraction or from different perspectives of the system. The approaches presented in this thesis are essentially focused on computation independent models since these are the kind of models that have less support at the moment. This results from their nature of being computation independent and, therefore, more difficult to *relate* to the computation concepts that are used in models that support the implementation of the software system being developed. Consequently, the proposals presented in this thesis are essentially related to the following *kinds* of models: use cases; feature; entity; activity; component; and class.

This thesis contributes to the foundations of the novel model driven approach to software product line development. The approaches presented in this thesis, when applied, enable the automation of previously manual based tasks, such as: feature modeling; the creation of the first logical architecture of a software product line; and multi-stage domain specific modeling.

The approaches presented in this thesis are illustrated through demonstration cases. These demonstration cases reflect experimentations of our approaches using problems described on other publications in the field or result from our own experience in applying the approaches in the context of projects developed at a software development company. We also discuss how the methodological approaches and techniques can be realized by using metamodeling frameworks and model transformation languages, particularly EMF and QVT.

Resumo

Métodos e Técnicas para o Desenvolvimento Baseado em Modelos de Linhas de Produtos de Software

Duas recentes aproximações no campo da engenharia de software estão a capturar a atenção da comunidade académica e da indústria: a engenharia de software baseada em modelos e as linhas de produtos de software. O modelo de desenvolvimento de software baseado em linhas de produtos é já adoptado na indústria há alguns anos e os resultados são bastante prometedores. No entanto, os recursos necessários para a implementação deste modelo de desenvolvimento de software são bastante significativos e, portanto, a adopção em larga escala desta abordagem ainda não é uma realidade. Isto resulta, certamente, dos métodos *pesados* que são necessários para essa abordagem. Estes métodos normalmente requerem o uso de abstracções de alto nível que permitem capturar conhecimento do domínio que é, vulgarmente, representado através de modelos. Estas abstrações encontram-se muitas vezes bastante distantes das abstrações usadas nas implementações das soluções de software. Para além disso, os processos são mais complexos que os vulgarmente usados no desenvolvimento de sistemas de software singulares (por oposição a linhas de produtos), uma vez que estes implicam dois sub-processos: *engenharia de domínio* para desenvolver artefactos reutlizáveis e *engenharia de aplicações* para desenvolver as aplicações finais do domínio.

A engenhria de software baseada em modelos promete *promover* a utilização de modelos como artefactos de desenvolvimento em igualdade com o código. Os modelos passam a ser tratados como entidades de *primeira classe* no desenvolvimento de software, à semelhança do código. Para se atingir este objectivo foram propostas algumas tecnologias, tais como, ferramentas de meta-modelação e linguagens de transformação de modelos. Estas tecnologias têm características que potenciam a automação de muitas tarefas de desenvolvimento de software usadas em métodos mais *pesados*, particularmente aqueles que dependem signifcativamente de actividades de modelação e que usam modelos como fonte para a produção de software. Este é o caso da maioria dos métodos de desenvolvimento de linhas de produtos de software.

O tema desta tese é o desenvolvimento de linhas de produtos de software através da adopção de uma aproximação baseada em modelos (*model driven*). Nesse contexto, começa-se por analisar diversos métodos relevantes de *engenharia de domínio* e desenvolvimento de linhas de produtos de software. Identificam-se e analisam-se conceitos fundamentais para o desenvolvimento de linha de produtos de software. Em resultado dessa análise, um conjunto de actividades comuns e cruciais para o desenvolvimento de linhas de produtos de software são identificadas e contextualizadas numa especialização do processo típico de desenvolvimento de linhas de produtos de software que designamos por aproximação ao desenvolvimento de linhas de produtos de software baseada em plataformas especificas de domínio. Esta tese apresenta aproximações metodológicas e técnicas para suportar as referidas actividades usando diversos *tipos* de modelos e uma proposta para a engenharia de linhas de produtos de software baseada em modelos. Em particular, mostra-se como conceitos, tais como as características comuns ou variáveis dos sistemas, podem ser modeladas em diferentes níveis de abstração e em diferentes perspectivas nas diversas actividades de desenvolvimento de software. As propostas apresentadas nesta tese são essencialmente focalizadas em modelos *independentes da computação*, uma vez que este tipo de modelos são os menos suportados actualmente pelos métodos de desenvolvimento de software. Isto resulta da sua natureza, ou seja, de representarem conceitos que não são directamente suportados pelos sistemas computacionais actuais e, como tal, é mais difícil relacionar estes conceitos com os conceitos usados na implementação das soluções de software. Em consequência, as propostas apresentadas nesta tese estão essencialmente relacionadas com os seguintes *tipos* de modelos: diagramas de

casos de utilização; diagramas de características; diagrams de entidades; diagramas de actividades; diagramas de componentes e diagramas de classes.

Esta tese contribui para as bases de uma nova aproximação ao desenvolvimento de linhas de produtos de software baseada em modelos. As propostas apresentadas nesta tese, quando aplicadas, permitem a automação de tarefas de desenvolvimento de linhas de produtos de software que são, no presente, fundamentalmente manuais, tais como: a modelação de características de linhas de produtos; a criação de arquitecturas lógicas de linhas de produtos; a modelação especifica de domínio em multíplas etapas ou sítios.

As propostas apresentadas nesta tese são ilustradas através de casos de estudo. Estes casos de estudo refletem os resultados de experiências na implementação das referidas propostas usando problemas descritos noutras publicações do mesmo campo científico ou resultam da própria experiência do autor deste documento na aplicação das propostas no contexto de projectos desenvolvidos numa empresa de desenvolvimento de soluções informáticas. Esta tese também descreve como as metodologias e técnicas propostas podem ser implementadas através de ferramentas de meta-modelação e linguagens de transformação de modelos tais como o EMF e o QVT.

Abbreviations

4SRS	4-Step Rule Set
ADL	Architecture Description Language
AOP	Aspect-Oriented Programming
ATL	Atlas Transformation Language
CASE	Computer Aid Software Engineering
CFRP	Conceptual Framework for Reuse Processes
CIM	Computation Independent Model
CWM	Common Warehouse Metamodel
DARE	Domain Analysis and Reuse Environment
DARTS	Design Approach for Real-Time Systems
DSL	Domain Specific Language
DSSA	Domain-Specific Software Architecture
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
ERP	Enterprise Resource Planning
FAST	Family-Oriented Abstraction, Specification, and Translation
FeatuRSEB	*Feature based evolution of* RSEB
FODA	Feature-Oriented Domain Analysis
FORM	Feature-Oriented Reuse Method
GME	Generic Modeling Environment
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment
IESE	Institute for Experimental Software Engineering
ISV	Independent Software Vendor
JET	Java Emitter Templates
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MoDeLine	Model Driven Development of Software Product Lines
MOF	Meta-Object Facility
OCA	Object Connection Architecture
OCL	Object Constraint Language
ODM	Organization Domain Modeling
OMG	Object Management Group
OOSE	Object-Oriented Software Engineering
PIM	Platform Independent Model
PLUS	Product Line UML-Based Software Engineering
PSM	Platform Specific Model
PuLSE	Product Line Software Engineering
QVT	Query/Views/Transformations
RSEB	Reuse-Driven Software Engineering Business
SEI	Software Engineering Institute
SI	System Integrator
SME	Small to Medium Enterprises
SPEM	Software Process Engineering Metamodel
SQL	Structured Query Language
STARS	Software Technology for Adaptable, Reliable Systems
SWEBOK	Software Engineering Body of Knowledge
UML	Unified Modeling Language

UML-F	UML Profile for Frameworks
USDP	Unified Software Development Process
XML	Extensible Markup Language

Preface

After five *long* years of very intensive work, my PhD thesis (*tese de doutoramento*) is finally finished. I owe some acknowledgements to many people.

First of all I would like to thank my supervisor, Ricardo J. Machado. We met at University of Minho, after being introduced by João Álvaro Carvalho. Ricardo has been a full-time supervisor and also a good friend. I had delayed my PhD from a long period of time, since I really wanted to do something scientifically interesting but also useful for myself. Ricardo has made this possible for me. Many thanks.

I am also grateful to João Álvaro Carvalho. I participated in his course of preparation for PhD at University of Minho. A lot of things I now know about scientific research I own to him. Thank you. Thank you also for introducing me to Ricardo.

I would like to thank University of Minho for accepting me as a PhD student. Special thanks to the colleagues of the SEMAG research group. This last five years were the best working years so far.

Instead of doing the traditional acknowledgements, I am going to present a resume of my professional and scientific path and, during that, make the proper acknowledgements.

My bachelor degree is in Informatics and from ISEP, a Portuguese polytechnic school. After that, I started working at I2S, a Portuguese software house specialized in software solutions for the insurance industry. I started my collaboration with I2S in 1990. In that period, I also owned a position to become teaching assistant at ISEP. I started teaching at ISEP in 1993. However, I continued to work at I2S as an external collaborator. I have also collaborated in several other industry projects before my PhD.

I thank Carlos Ramos and Zita Vale, the supervisors for my Msc dissertation at FEUP, University of Porto (1996). Although in a somewhat different field (workflow), they, as well as other teachers and colleagues from my Msc course, had a great influence on my scientific studies.

I officially started my PhD in November, 2002. At that moment, I was working at I2S and also teaching at ISEP. I started my PhD with a focus on domain-specific languages and runtime variability techniques. I think this research topic come out of the influence from my work at I2S and also from the classes I was teaching at ISEP (manly classes dealing with operating systems, object-oriented programming, and software components). At I2S, I had always worked in the innovation group. There, my team (José Timóteo, Sérgio Ribeiro, Nuno Ferreira, Paulo Sousa, and many others) and I, had developed a domain-specific platform for supporting the specification and realization of (mainly) insurance contracts. It was essentially developed in C and had some interfaces for COM and Java. The main goal was to support the specification of the contract domain rules by domain specialists and also to be able to introduce new types of contracts in the running system without disrupting its regular behavior. So, the first year of the PhD (2003) resulted in the elaboration of a technical report of the state-of-the-art about domain engineering with a great focus on variability realization techniques. Special thanks to Fernando Brito e Abreu from UNL for its thorough revision and comments on this report.

In 2004, I published my first PhD papers. These papers resulted essentially from the previous state-of-the-art and from my first ideas on what I should attack in the field of domain engineering. This was also the year when I really started to interact with my scientific field community. I had

five major interactions: a visit to the Interactive Software Development and Renovation research group at CWI, Netherlands (I really have to thank Joost Visser and Arie van Deursen for that); my participation at SEDES2004 (Coimbra, Portugal), the first Portuguese doctoral symposium in software engineering; the ICSR8 conference (Madrid, Spain), where I met some notable figures in the field, e.g., Dirk Muthig, Hassan Gomaa, Stan Jarzebeck, and Jan Bosh; the QUATIC2004 conference (Porto, Portugal) and, finally, my first OOPSLA at Vancouver, Canada (I won't give any comments on that; if you have already attended one of the OOPSLA conferences you know what I am talking about). Those interactions had a major impact on my view on scientific research in the field of software engineering. In 2004, I realized that I had a long way to go and that I needed to dive deeper.

In April 2005, I decided to suspend my collaboration with I2S since I realized that it was very difficult for me to take the dive I needed unless I suspended my responsibilities at the software company.

In April 2005, I started the long dive with a model driven method developed at University of Minho, by a team leaded by Ricardo J. Machado, called *4-Step Rule Set* (4SRS). In June of that year, I presented a first approach to include support for variability in the method at the MOMPES2005 workshop at ACSD2005 (Rennes, France). By that time, I really started to see my work focusing on model driven and variability. Another important theme was UML, since the 4SRS method was based on the UML notation.

In July 2005, I attended the first summer school on generative and transformational techniques in software engineering, in Braga, Portugal. It was a major milestone on my PhD. I was then certain that metamodeling and model driven was the way to go in software engineering, particularly in the application areas of variability intensive systems (e.g., software product lines, software factories, and enterprise resource planning software).

During 2005, I have also done experimental software development with several metamodeling and model driven related tools, particularly GME, GReAT, OpenArchitectureWare, and ATL. During that year, I supervised an undergraduate final year project at ISEP regarding metamodeling with GME. Thanks to the student João Riqueza.

One year later, I had a paper accepted at SPLC2006 (Maryland, Baltimore). In that paper, I presented my findings in adopting UML use cases as CIM models for MDE approaches. I am very grateful for the comments of Czarnecki and Greenfield. I am also grateful for the revision of Tommi Manisto.

2006 was also the year when I started focusing my work on experimental tool support on the Eclipse platform and the modeling tools project, particularly EMF, GMF, and UML2. I have also worked with SmartQVT for model transformations and Feature Modeling from the University of Waterloo.

In March 2007, I had a paper accepted at the MOMPES2007 workshop at ETAPS2007 (Braga, Portugal). In that paper, I presented my findings in deriving the logical architecture of a system from UML use cases and activity diagrams. I thank Marcus Fontoura for his comments and suggestions on a draft of that paper.

In September 2007, I participated once again in SEDES (the second edition took place in Lisbon, Portugal, within QUATIC20007). This time with a paper that presented an overview of my PhD. That month, I had my second participation at the SPLC conference series (SPLC2007 took place at

Kyoto, Japan) with a paper that presented an approach to automate mapping between use cases and feature models.

During 2007, I supervised an undergraduate final year project at ISEP regarding metamodeling with EMF and GMF. Thanks to the student Abílio Pinto.

I wish to thank Luis Paupério and Anibal Oliveira, from I2S, who have supported my scientific activities. I also extend my acknowledgements to several other elements of the I2S *family*. They know who they are.

I wish to thank my coordinator from ISEP, Adriano Lhamas. He has always supported my teaching activities and endorsed my PhD work. This acknowledgement is extended to the Informatics Department and ISEP School. I also extend my acknowledgements to several other elements of the ISEP *family*, including my students. They know who they are.

A special thanks to all the unknown reviewers of my papers.

I specially wish to thank the members of the KTC *family*, a recent project that started during my PhD and that I had the privilege of being part of. They helped me putting my ideas in perspective with the real world.

Contents

List of Figures

List of Tables

1. Introduction

"One can envisage component systems and business models delineated so clearly that client personnel can themselves create the application systems"
Ivar Jacobson et al., in "Software Reuse"

This thesis is about two recent trends in the field of the software engineering discipline: software development based on models (as opposed to code) and the product line approach to software development. Therefore, this thesis presents contributions related to the research field of software engineering, particularly for model driven engineering of software product lines. This chapter presents the context for the research, the related knowledge areas, the research method, and the goals supported by the thesis contributions. In this chapter we also describe how the remainder of the document is structured.

1.1 Motivation

Software is widely spread and helps automate several activities. Software is present from low-cost cellular phones to cutting edge spacecrafts. It does not only control machines but also supports nuclear activities of entire organizations and governments. The world is greatly dependent on software and software has also become increasingly more complex. Nonetheless, if an observer from the outside of the software industry followed the way we develop software in the last thirty to thirty five years, he/she may detect only a small evolution. If we think of it in a detached way, we see that we still edit our programs with text editors and the syntax of our programs is similar to the way people programmed in C thirty years ago. Of course we know this is not totally true, because we now program in a different paradigm; we have integrated development environments that greatly enhance the features of the old text editors; and our programs run on top of software virtual machines. And, of course, we also have started to use analysis and design modeling tools. They help us to cope with the complexity of the software systems we develop. We build models of our software and we *try* to build our programs according to these models. The problem with this approach is that it is not simple to make the transition from the models to the code of the programs. Many times this is done manually or it is partly automated and, since we know that software, because of its characteristics, is prone to modification, it is almost a certainty that models and code will become unsynchronized. On another related perspective, we observe that the requirements for software projects are becoming more demanding and the rigor that stakeholders demand of other industries is gradually becoming also a reality in the software industry. One way to cope with such demands is to reuse as much already tested software. Recent practical approaches to reuse are software product lines and software factories.

The development of software systems requires knowledge from two main *sources*. One is of technical (computer) nature, i.e., programming in a specific language; manipulating xml documents; understanding a communication protocol. The other is usually of non-technical (i.e., non-computer) nature and relates to knowledge about the problem that the software system is supposed to attack. This latter knowledge is necessary to understand the *problem domain*, while the former is used to build a solution, i.e., it relates to the *solution domain*. Because usually

abstractions from those domains are so far apart it is very difficult to make accurate previsions about software projects. As a result, software projects are hard to manage, their costs may largely surpass budgets and the solution may not correspond entirely to the requirements [Johnson 1995].

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [Clements *et al.* 2002]. This implies one (or more) common domain(s) shared by the developed applications (products). Because applications share domains, it becomes possible to reuse software artifacts between applications and reduce the conceptual gap between the problem domain and the solution domain.

In a software product line approach the domain knowledge grows as each new application of the domain is developed. It is commonly accepted in the field that the initial investment in a product line approach can have return by the third developed application. Some authors, particularly Krueger, go further and defend that the adoption of a software product line can be beneficial from the first application developed if the approach is introduced incrementally [Krueger 2006]. Nonetheless, all well-known software product lines have been implemented in large organizations or have required significant consultant knowledge from software product line specialists. Examples of such organizations are, among others: Nokia; Philips; GM; Bosch; Hewlett-Packard; Boeing; and Ericsson[1]. Examples of software product line expert support organizations are SEI and IESE. Even if there are a few documented examples of software product lines in small to medium enterprises, one has to agree that the effort required to implement the necessary processes and methods may be out of reach for the majority of SMEs.

Recently, the software engineering community has assisted the appearance of several proposals, such as aspect-oriented programming [Kiczales *et al.* 1997], feature-oriented programming [Batory 2004] , domain-specific languages [Hudak 1998] and model driven engineering [D.C. Schmidt 2006]. Although diverse in nature, they all share the pretension of complementing or solving some limitations of the dominant object oriented paradigm. Some of these proposals, notably domain-specific modeling and model driven development are starting to capture the attention of the industry and major software development environments, like Eclipse and Visual Studio .Net are starting to support them. Also, large industry consortiums, such as OMG (Object Management Group) are supporting and promoting such proposals.

Model driven development is based on using models as the central artifact in software development as opposed to source code in traditional development. In these approaches, models are transformed into other models and, eventually, are transformed into code that can be executed. Theoretically, a model driven approach can be realized by standard modeling tools, such as UML modeling tools. All that is required is the possibility to automate transformation between models and models and code. In practice, it is common that modeling languages be adapted for the specific needs of the model driven method. For instance, a higher-level model could require adaptations to include information used to guide the transformation to lower-level model(s). In such a case, it is required that the modeling tool support some sort of metamodeling [D.C. Schmidt 2006]. Metamodeling consists essentially in *modeling the model*. If a model is like a language that can be used to represent a system, then a metamodel is the grammar of that language [Balasubramanian *et al.* 2006]. As a metaphor for metamodeling, we like to use the example of the puppet and the puppeteer: the puppeteer models the *world* of the puppet and, as such, a puppeteer is a metamodeler.

[1] See Product Line Hall of Fame available at http://www.sei.cmu.edu/productlines/plp_hof.html.

If we consider the puppeteer also as a puppet, then we see that the puppeteer can only model the *world* of the puppet according to the *rules* of its *world*. These rules are defined by the puppeteer of the puppeteer. In a metamodeling approach, this means that a model has a metamodel and, since a metamodel is also a model, it also has a metamodel. This can go on indefinitely. In practice, the Model Driven Architecture (MDA), which is a standard for model driven and metamodeling from OMG, only defines four layers of modeling [MDA 2003].

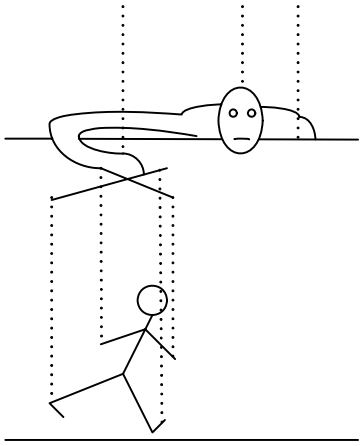

Figure 1: Metamodeling metaphor: *Puppeteer as Puppet*.

We particularly defend the model driven approach with metamodeling because it supports the adoption of domain-specific modeling languages and can be used at several levels of abstraction and in different components of a method. Also, with adequate tool support, this approach may automate the more cumbersome and demanding tasks of software engineering methods, like the methods used in software product lines.

1.2 Overview of Software Engineering Trends

In this section we present the most significant trends in software engineering that are related to our work.

Domain Engineering
Although definitions of concepts and terms may be volatile and a source of discussion, particularly in recent knowledge areas, we find it is preferable to have a bad definition than to have none at all. For domain engineering we will adopt the definition found in [Czarnecki 1998]:

"Domain Engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. reusable workproducts), as well as providing an adequate means for reusing these assets (i.e. retrieval, qualification, dissemination, adaptation, assembly, etc.) when building new systems."

By capturing the acquired domain knowledge in the form of reusable assets and by reusing these assets in the development of new products, organizations are able to deliver the new products in a shorter time and at a lower cost. According to the former definition, domain engineering is a systematic approach to achieving this goal.

Domain Engineering encompasses three main process components: Domain Analysis, Domain Design, and Domain Implementation.

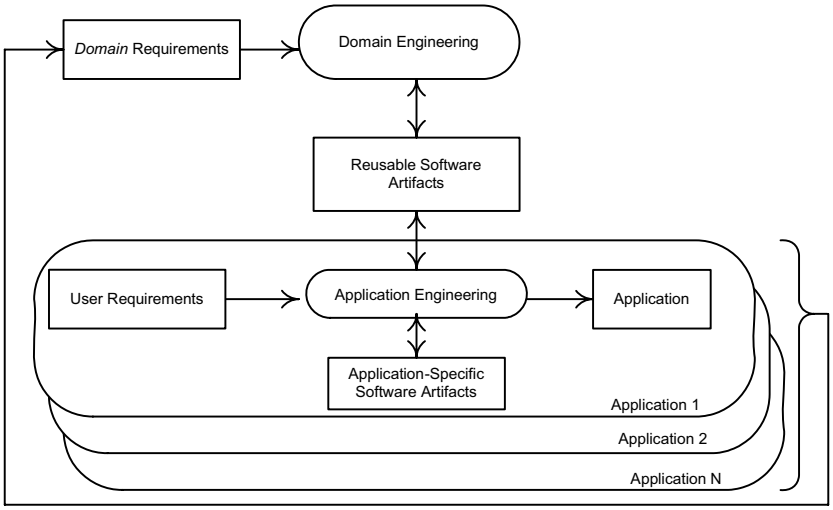

Figure 2: Domain Engineering and Application Engineering.

The major results of domain analysis are the domain scope, the domain model and the domain requirements. Domain design is concerned with the common architecture for the systems in the domain and the design of the reusable artifacts of the domain (for instance components and domain-specific languages). Domain implementation regards the realization of the designed artifacts. Figure 2 represents the relationship between domain engineering and application engineering.

As we can see from Figure 2, there is a parallelism between domain and application activities. Software artifacts that result from domain engineering can be reused in the development of applications of the domain. Software artifacts that are specific to an application do not go to the reusable artifacts repository. Applications of the domain can be used as a source for new domain requirements. This is the traditional view of domain engineering, where domain and application engineering are separated but cooperating activities. However, even for single system development a domain engineering approach is of great value, particularly if the application to be built is complex. Examples of adopting a domain approach to system design can be found in [Evans 2004]. The approach we present in this thesis also follows this direction.

Examples of domain engineering methodologies are Feature-Oriented Domain Analysis (FODA) [Kang *et al.* 1990], Draco [Neighbors 1984], FAST [Weiss 1998], and Organization Domain Modeling (ODM) [Simos *et al.* 1996].

Software Product Lines

Here, we adopt the following definition for software product line [Clements *et al.* 2002]:

"A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."

From this definition it becomes clear that a software product line involves domain engineering and also that it implies that the developed software systems are specific to a particular market segment or mission. It is an approach to software engineering that promotes reuse *within* the software development organization. It involves three essential activities: core asset development (domain engineering); product development (application engineering); and technical and organizational management.

In software product lines, applications share features, and as such, share common software artifacts. Applications are built by reusing these common artifacts in order to support common features and the specific or adaptable features are supported by variability mechanisms. This is very similar to the process depicted in Figure 2.

One central aspect of a software product line is the architecture. The product line architecture is common to all the products (applications of the product line) and needs only be instantiated for each one. The architecture is a blueprint for how each product is assembled from the components in the repository of artifacts. Some components may need to be tailored for a particular product using variability mechanisms. Some components will be specific to products. Product lines also include other concerns in the organization, such as marketing, management, and training. Software product lines are a global business approach to software development.

Examples of product-line methods are PuLSE [Anastasopoulos *et al.* 2000] and PLUS [Gomaa 2005].

Software Factories

The term software factory was used to identify software development industrialization approaches that were developed in Japan since the fifties [Cusumano 1991]. Recently, Microsoft has reused the same term to describe a specific approach to software product line development. The definition of software factory according to [Greenfield *et al.* 2004]:

"A software factory is a software product line that configures extensible tools, processes, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypal product by adapting, assembling, and configuring framework-based components".

Software factories seek to achieve the same level of reuse of other industries by adopting similar approaches of automation. As such, the basis for software factories is essentially the same of software product lines. However, they introduce the concept of factory schema, which defines the artifacts and the assets used to build them. A software factory schema is a directed graph whose nodes are viewpoints and whose edges are computable relationships between viewpoints called mappings. According to these definitions, software factories are well suited for model driven development, since viewpoints could be realized by models and the mappings by transformations between the models that implement the viewpoints. With software factories it becomes more clear the advantage of adopting model driven approaches for the development of software product lines.

Regarding product lines, software factories also promote software automation and reuse at an inter-organizational scope, for instance, for the realization of software supply chains. In this context, for instance, software product development can be outsourced by sending a software factory template to an off shore SI (System Integrator). It is also possible that an ISV (Independent Software Vendor) develops a software factory not for in-house development of software products but for the development of products in its customers. These are some of the possibilities of realizing a software factory approach. If we do not take a restrictive view, we can say that some

packaged enterprise applications, such as ERPs [ERP 2007], already fall in the definition of software factory.

Domain Specific Languages

A domain specific language (DSL) regards a programming language which syntax and semantics are specialized for a particular application domain or type of problem [Hudak 1998; Thibault 1998]. Domain specific languages have been around for quite some time. Two well-known DSLs are Structured Query Language (SQL) [SQL 2003] and Hyper Text Markup Language (HTML) [HTML 2007]. The former is a DSL for the domain of querying and manipulating relational databases and the later a DSL for the domain of constructing hyper linked digital documents.

DSLs are very well suited for specifying specific perspectives or aspects of a system. In fact, it is current the use of several DSLs for the specification of software systems. We already mentioned two examples. Other common example is the use o XML [XML 2007] files for the configuration of systems. Usually DSLs are also one of the possible results of domain engineering, since they can be used to specify the variable parts of a system. One possible usage of such a language is to glue together (and possible adapt) the reusable software components of a product line in order to build a specific application of the domain.

Similarly to DSLs, models represent a perspective of a system and, as such, we can say that metamodels define domain specific languages and models are valid sentences of such domain specific languages.

As we will see next, there is a natural convergence between the previously presented approaches and model driven engineering. Next, we will briefly present the concept of model driven architecture, an OMG initiative in the context of model driven engineering.

Model Driven Architecture

Model Driven Architecture (MDA) is an OMG standard that aims at promoting a new way to develop software in which models are the central piece. According to the MDA specification, this approach is model driven because it *"provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification"* [MDA 2003].

One of the principles of MDA is the separation of domain knowledge and platform knowledge. As a result of such principle, there are three types of models: Computation independent models (CIM) represent business processes and entities; Platform Independent Models (PIM) resolves functional requirements without platform specifics; and Platform Specific Models (PSM) represent a solution for a specific platform and includes functional and non-functional requirements. Figure 3 presents and overview of MDA.

Basically, with MDA it is possible to: (1) specify the environment of the system and the requirements for the system (CIM); (2) specify a system independently of the platform that supports it (PIM); (3) specify platforms; (4) choosing a particular platform for the system; and (5) transform the system specification into one for a particular platform (PSM).

The Model Driven Architecture is based on the following OMG technologies and standards: (1) Meta Object Facility (MOF) [MOF 2006]: Meta-modeling language and interchange (XMI [XMI 2007]); (2) Unified Modeling Language (UML) [UML 2005]: A standard modeling language, instance of the MOF model; (3) Common Warehouse Metamodel (CWM) [CWM 2007]: Modeling languages for data warehouse applications; (4) Object Constraint Language (OCL) [OCL

2006]: Expression language, extends the expressive power of UML and MOF; (5) Query/Views/Tranformations (QVT) [QVT 2005]: A transformation definition language. Also for queries and views of models; and (6) Software Process Engineering Metamodel (SPEM) [SPEM 2005]: Metamodel and UML profile used to describe a concrete software development process.

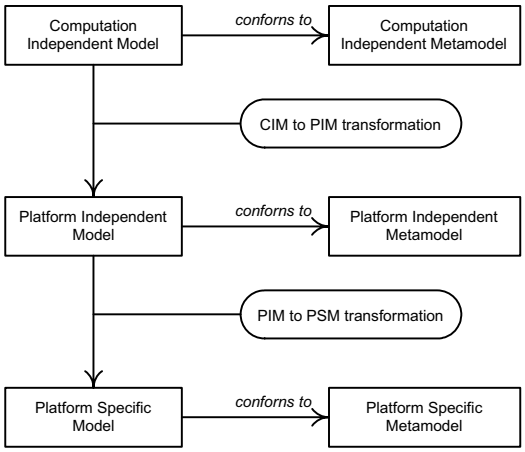

Figure 3: Model Driven Architecture.

Model Driven Engineering

As we have seen, MDA is the OMG view of what should be a model driven approach to software development. Model Driven Engineering (MDE) is a more broader vision, encompassing many popular current research trends related to generative and transformational techniques in software engineering, system engineering and data engineering [Bézivin 2005]. In this approach, the whole software life cycle is seen as a process of model production, refinement and integration. As such, source code is just another form of models.

This is not so different from MDA. We could say that the great difference is that MDE is not constraint by the OMG technologies and standards.

Applying an MDE approach requires, at least, two engineering levels: one related to metamodeling and the other related to modeling. If we think of it, the metamodeling part implies domain engineering, because the result will be a domain specific platform that will be used to support the modeling of specific applications. When we use the term domain specific platform, we intend to include, for instance, object-oriented frameworks, template languages, and modeling environments, which are based on the domain engineering process and are used to support the modeling of applications. This is a way of software development that is very different than the usual way, in which developers only work at one level, using tools for developing applications. In an MDE approach, developers also have to work at the tool level, i.e., the metamodeling level. As we will see throughout this thesis, MDE is based essentially on two new software development activities: metamodeling and model transformation.

In this thesis we use the term *model driven engineering* and *model driven development* interchangeably.

1.3 Research Goals

As the reader already has been able to notice, this thesis is about two main topics: model driven engineering and software product lines. In this thesis, we adopt a broader interpretation for software product lines that includes systems such as software factories as well as enterprise applications, such as ERPs. Although there are model driven approaches applied to such types of systems, they are essentially related to platform independent and platform specific models [Childs *et al.* 2006]. On the other end, in this thesis, the focus is on computation independent and platform independent models. The other context of this thesis is the adoption of a domain-specific platform [Braganca *et al.* 2004; Czarnecki *et al.* 2006] approach using model driven engineering, which is depicted in Figure 4.

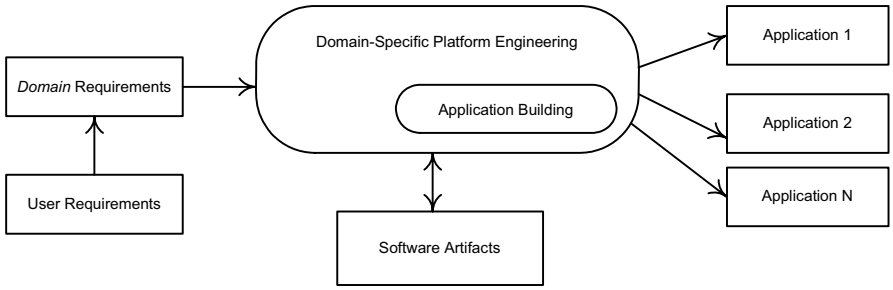

Figure 4: Software product line development using a domain-specific platform approach.

The main research objective of this thesis can be described by the following statement:

The primary objective of this thesis is to provide a set of methodological approaches and techniques that effectively enables the widely adoption of model driven methods for the development of software product lines.

In the context of this main objective, the following goals are defined:

Goal 1: Propose an approach for the modeling of commonality and variability in the context of computation independent models.

Goal 2: Propose an approach to support the derivation of platform independent architecture models based on computation independent models.

Goal 3: Propose an approach to support the tracing between analysis and design model elements.

Goal 4: Propose technical approaches for tool supporting of the goals 1 through 3.

Each of these goals topics resulted in approaches that were described in research papers. These papers were all peer-reviewed anonymously. They were also publicly presented and discussed by researchers specialized in the scientific field.

We propose to accomplish the research goals of this thesis by adopting model driven engineering techniques, and we particularly focus on metamodeling and transformation techniques. This objective is not pursued as a theoretical study. Instead, we take a pragmatic viewpoint in which existing theories, methods, tools and techniques can be combined to support our goal. We also take this approach because our research is in the field of software engineering where, in

contrast to other fields of computer science, the human factor is of the most importance. In this context, it is not sufficient to have the best technical solution; the process and all involved resources/persons must also be taken into account. In the next section we discuss our research approach.

1.4 Research Approach

As well as a model has to conform to a metamodel, research in software engineering has to *conform* to a research method. In this section, we briefly present the research method used in this thesis.

Software engineering is a discipline of the computer science with particularities when compared to other disciplines such as theory of computation or programming languages and compilers. According to the SWEBOK (Software Engineering Body of Knowledge), software engineering encompasses knowledge, tools, and methods for defining software requirements, and performing software design, software construction, software testing, and software maintenance tasks [SWEBOK 2007]. It also states that software engineering is related to other disciplines such as management, project management and software ergonomics. These are disciplines which are related to social sciences such as economics, management, anthropology and sociology. The human/social factor makes software engineering a particular discipline of an applied science (computer science). Figure 5 presents the context for software engineering research. Not all solutions in software engineering are solely technical. Software practitioners adopt tools and methods of software engineering that they will use in organizations, within teams, and for the purpose of developing software products such as information systems that will, most probably, interact heavily with persons.

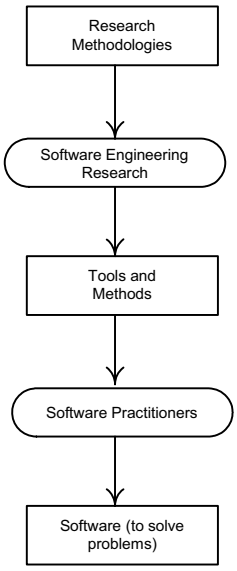

Figure 5: The context of Software Engineering Research.

This thesis follows the classification for software engineering research methods proposed in [Adrion 1993]: Scientific, Engineering, Empirical, and Analytical. Of these four research methods, and according to the same author, the empirical method is the most appropriated for software

engineering research. The empirical method is based on the application of the proposed model to case studies (in some contexts also called *demonstration cases*) in order to measure and analyze the results and, eventually, repeat the process. In contrast, the analytical method does not force the use of case studies; the results can be derived. The scientific method is the *traditional* research method that is based on the observation of the real world, and as such, is more tailored to natural sciences. In the engineering method, existing solutions are observed and better solutions proposed and developed. The new solutions are measured, analyzed and evaluated and the process is repeated if needed.

Although the empirical method is the most suited for software engineering research it is also a very demanding method since it requires the application of the proposed models to case studies in order to measure the results. In the context of the work of this thesis, it was not possible to fully develop the case studies that the method would require and, consequently, we do not have quantitative evaluations of our work. As such, we decided to use qualitative measures of our proposals. These measures resulted essentially from our own experience in several software engineering projects in the form of discussions and observations with practitioners. These qualitative assessments of our proposals are usually presented in the form of case studies that reflect real cases but are simplified in order to facilitate their description in research papers. We have also used another form of evaluation of our work that consisted in the analysis and comparison of other solutions to the same problems, which is a validation more common to the engineering research method.

Regarding validation approaches used by software engineering research works, Mary Shaw identifies the following types of validation [Shaw 2003]: Analysis, Evaluation, Experience, Example, and Persuasion. Shaw states that it is essential to select a form of validation that is appropriate for the type of research result and the method used to obtain the result. She also states that a simple example derived from a practical system may play a major role in validating a new type of development method.

1.5 Overview

The structure of this thesis partially covers the activities depicted in Figure 6. This figure presents the major activities regarding model driven development of software product lines. The figure only represents the major concerns of this thesis, it does not show all the activities involved in developing software product lines and it also does not show all the data and control flows and, as such, all the dependencies between activities.

The remainder of this thesis is structured in five chapters and three appendixes.

The contents of chapter 2 to 5 are partially based on research papers. All the research papers were peer reviewed. These chapters are as much as possible self-contained. As such, the reader is free to select the topic of interest without much concern regarding a required reading flow. Our only suggestion is the reading of Chapter 2 previously to chapters 3, 4 and 5, particularly if the reader is not familiar with the research field of domain engineering and software product lines. Each chapter covers specific topics depicted in Figure 6. To help the reader, we highlight the topics covered by the text at specific location of the chapters. Chapter 2 is an exception since it covers all the topics presented in Figure 6.

Chapter 2 presents the state-of-the-art of the research fields, particularly for domain engineering. We discuss several relevant domain engineering methods and compare them

according to three perspectives: variability identification; variability representation; and variability implementation. The major concepts of the field are also presented and related.

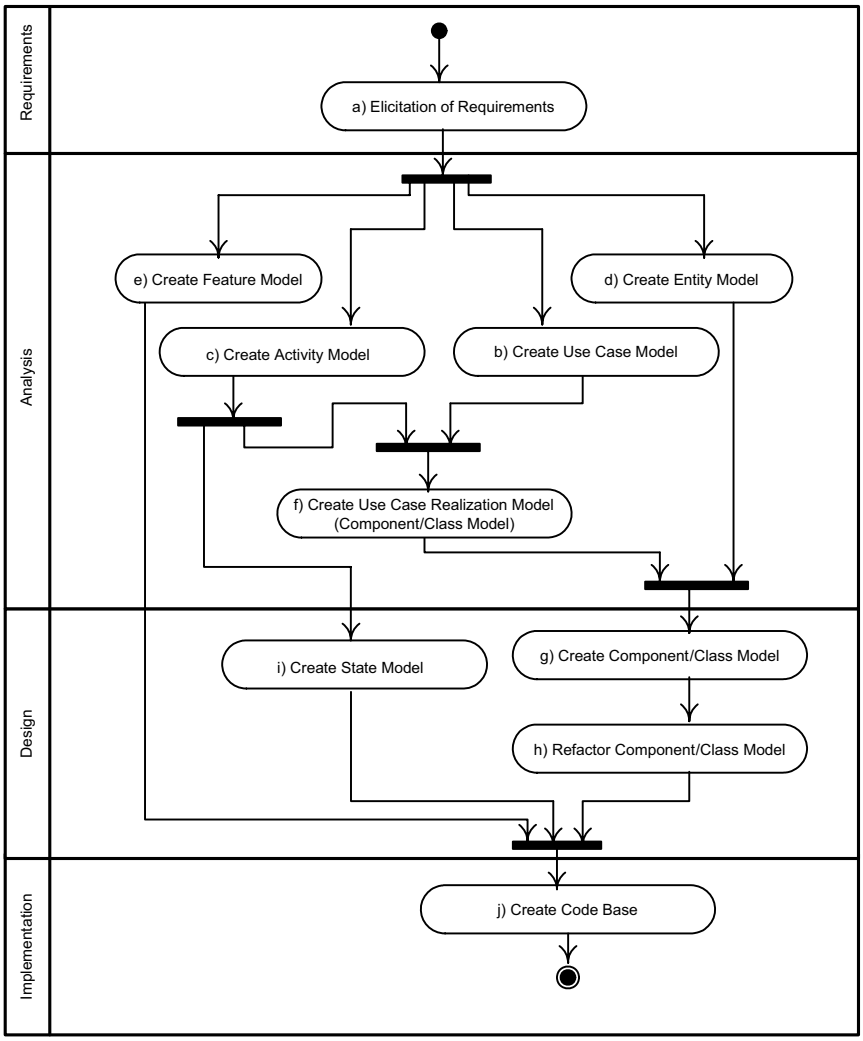

Figure 6: Excerpt of activities for model driven development of software product lines.

Chapter 3 is concerned with methodological proposals for model driven development of software product lines. It covers the analysis and design phases in a transversal way, i.e., it is not specific to any activity depicted in Figure 6. Part of it describes an approach to extend an existing model driven method, called 4SRS (4-Step Rule Set) [Machado *et al.* 2005], to explicitly handle variability. The second half of this chapter presents a proposal to support the transformation of analysis models into architectural models. It also delineates some approaches to detail the first logical architecture of a system by integrating design patterns in the proposed approach. The proposals discussed in the second half of Chapter 3 constitute the major components of MoDeLine (Model Driven Development of Software Product Lines), a model driven method for the development of software product lines that started from our experience in adapting 4SRS.

Chapter 4 is concerned with metamodeling and modeling issues. The first half of the chapter describes a proposal to adapt the UML 2.0 metamodel in a way that effectively enables the adoption of use case diagrams in model driven approaches aimed at the development of software

product lines. The second half of the chapter describes a proposal to extend a UML profile for the design of frameworks and product lines called UML-F so that it includes requirements and analysis diagrams.

Chapter 5 is dedicated essentially to model transformations. The first half of the chapter presents a proposal of mappings between use cases and feature diagrams. It also presents how these mappings can be supported by the QVT operational language and the SmartQVT tool [SmartQVT 2007]. The second half of the chapter presents a proposal to support multi-staged software development in the context of model driven and software product lines. This is one of the scenarios of usage for software factories.

Chapter 6 is dedicated to conclusions. In this chapter we analyze the research goals and how they have been covered by our work. We also present future work and open issues.

Appendixes A, B and C present some details about experimental implementations regarding the proposals presented in this thesis.

1.6 References

[Adrion 1993] Adrion, W. R., "Research methodology in software engineering," Dagstuhl Workshop on Future Directions in Software Engineering, 1993.

[Anastasopoulos *et al.* 2000] Anastasopoulos, M., J. Bayer, O. Flege and C. Gacek, "A Process for Product Line Architecture Creation and Evaluation - PuLSE-DSSA - Version 2.0," IESE 038.00/E, 2000.

[Balasubramanian *et al.* 2006] Balasubramanian, K., A. Gokhale, G. Karsai, J. Sztipanovits and S. Neema, "Developing Applications Using Model-driven Design Environments," *IEEE Computer*, vol. 39, 2006.

[Batory 2004] Batory, D., "Feature-Oriented Programming and the AHEAD Tool Suite," International Conference on Software Engineering, Edinburgh, Scotland, UK, 2004.

[Bézivin 2005] Bézivin, J., "Model Driven Engineering: Principles, Scope, Deployment and Applicability," GTTSE 2005, Braga, Portugal, 2005.

[Braganca *et al.* 2004] Braganca, A. and R. J. Machado, "Run-time Feature Realization based on Domain-Specific Platforms," ICSR8 (Poster Presentation), Madrid, 2004.

[Childs *et al.* 2006] Childs, A., J. Greenwald, G. Jung, M. Hoosier and J. Hatcliff, "CALM and Cadena: Metamodeling for Component-Based Product-Line Development," *IEEE Computer*, vol. 39, pp. 42-50, 2006.

[Clements *et al.* 2002] Clements, P. and L. Northrop, *Software Product Lines - Practices and Patterns*: Addison Wesley, 2002.

[Cusumano 1991] Cusumano, M. A., *Japan's Software Factories*. Oxford: Oxford University Press, 1991.

[CWM 2007] OMG, "Common Warehouse Metamodel Specification v1.0 (ad/01-02-01)," Available at http://www.omg.org, 2007.

[Czarnecki 1998] Czarnecki, K., "Generative Programming Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models," in *Department of Computer Science and Automation*: Technical University of Ilmenau, 1998.

[Czarnecki *et al.* 2006] Czarnecki, K., M. Antkiewicz and C. H. P. Kim, "Multi-level Customization in Application Engineering," *Communications of the ACM*, vol. 49, 2006.

[D.C. Schmidt 2006] D.C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, pp. 25-31, 2006.

[ERP 2007] Wikipedia, "ERP: Enterprise Resource Planning," Available at http://en.wikipedia.org/wiki/Enterprise_resource_planning, 2007.

[Evans 2004] Evans, E., *Domain-Driven Design - Tackling Complexity in the Heart of Software*: Addison Wesley, 2004.

[Gomaa 2005] Gomaa, H., *Designing Software Product Lines with UML*: Addison Wesley, 2005.

[Greenfield *et al.* 2004] Greenfield, J., K. Short, S. Cook and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*: Wiley, 2004.

[HTML 2007] W3C, "HTML: Hyper Text Markup Language," Available at http://www.w3c.org, 2007.

[Hudak 1998] Hudak, P., "Modular Domain Specific Languages and Tools," Fifth International Conference on Software Reuse, Victoria, Canada, 1998.

[Johnson 1995] Johnson, J., "CHAOS: The dollar drain of IT project failures," *Application Development Trends*, pp. 41-47, 1995.

[Kang *et al.* 1990] Kang, K. C., S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study Technical Report," Software Engineering Institute, Carnegie Mellon University CMU/SEI-90-TR-21, 1990.

[Kiczales *et al.* 1997] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin, "Aspect-Oriented Programming," European Conference on Object-Oriented Programming (ECOOP), Finland, 1997.

[Krueger 2006] Krueger, C. W., "New Methods in Software Product Line Development," 10th International Software Product Line Conference SPLC 2006, Baltimore, Maryland, USA, 2006.

[Machado *et al.* 2005] Machado, R. J., J. M. Fernandes, P. Monteiro and H. Rodrigues, "On the Transformation of UML Models for Service-Oriented Software," ECBS International Conference and Workshop on the Engineering of Computer Based Systems, Greenbelt, Maryland, 2005.

[MDA 2003] OMG, "Model Driven Architecture Guide Version 1.0.1," Available at http://www.omg.org, 2007.

[MOF 2006] OMG, "Meta Object Facility (MOF) 2.0 Core Specification (formal/06-01-01)," Available at http://www.omg.org, 2006.

[Neighbors 1984] Neighbors, J. M., "The Draco approach to constructing software from reusable components," *IEEE Transactions on Software Engineering*, vol. 10, pp. 64-74, 1984.

[OCL 2006] OMG, "Object Constraint Language Specification v2.0 Final Adopted Specification (formal/06-05-01)," Available at http://www.omg.org, 2006.

[QVT 2005] OMG, "MOF QVT Final Adopted Specification (ptc/05-11-01)," Available at http://www.omg.org, 2005.

[Shaw 2003] Shaw, M., "Writing Good Software Engineering Research Papers," 25th International Conference on Software Engineering, 2003.

[Simos *et al.* 1996] Simos, M., D. Creps, C. Klinger, L. Levine and D. Allemang, "Organization Domain Modeling (ODM) Guidebook, Version 2.0," Informal Technical Report for STARS STARS-VC-A025/001/00, 1996.

[SmartQVT 2007] France Telecom, "SmartQVT - Open Source Transformation Tool Implementing the MOF 2.0 QVT-Operational Language," Available at http://smartqvt.elibel.tm.fr/, 2007.

[SPEM 2005] OMG, "Software Process Engineering Metamodel Specification v1.0 (formal/05-01-06)," Available at http://www.omg.org, 2007.

[SQL 2003] ISO, "Information technology -- Database languages -- SQL -- Part 2: Foundation (SQL/Foundation)," Available at http://www.iso.org, 2003.

[SWEBOK 2007] IEEE, "Guide to the Software Engineering Body of Knowledge," Available at http://www.swebok.org, 2007.

[Thibault 1998] Thibault, S., "Domain Specific Languages: Conception, Implementation and Application." Paris: Université de Rennes 1, 1998.

[UML 2005] OMG, "Unified Modeling Language Version 2.0: Superstructure (formal/05-07-04)," Available at http://www.omg.org, 2005.

[Weiss 1998] Weiss, D. M., "Commonality Analysis: A Systematic Process for Defining Families," Second International Workshop on Development and Evolution of Software Architectures for Product Families, 1998.

[XMI 2007] OMG, "MOF 2.0/XMI Mapping Specification, v2.1 (formal/05-09-01)," Available at http://www.omg.org, 2007.

[XML 2007] W3C, "XML: Extensible Markup Language," Available at http://www.w3c.org, 2007.

2. Related Work

"Leave no Stone Unturned"
Euripides, in "Heraclidae"

This chapter presents the state-of-the-art of the research fields, particularly for domain engineering. We start by introducing and contextualize domain engineering. We then analyze eight software development methods that can be classified as domain engineering methods. The major concepts used in these methods are then discussed and a comparison is made on how each method deals with the following topics: variability identification; variability representation; and variability implementation.

2.1 Introduction

Domain engineering is the basis for software reuse, particularly in the case of software product lines. Domain engineering can enable an effective reuse of several types of artifacts in a given domain ranging from code to analysis models. Scoping the problem space to a given domain implies that the possible solution space is limited. Therefore, it becomes more simply to reuse parts of the solution space in several applications (that are also part of the domain solution space). In order to reuse these common parts in several applications, it is very probable that adaptations, or variations, will be needed (in these common parts) to meet the requirements of the different applications of the domain. Domain engineering focuses on finding common and variable parts of a domain in order to support reuse in that domain. This chapter presents an overview of domain engineering methodologies and related work. The major focus is on how to identify, represent and implement variability in a domain.

Providing a domain engineering state-of-the-art is an overwhelming task. We could start with the work of Parnas on program families [Parnas 1976] and with the work of Neighbors, to our knowledge, the first explicit domain engineering methodology [Neighbors 1980]. If we want to go even further, we can say that domain engineering appears also in the work of Dijkstra on structured programming [Dijkstra 1969], where he already speaks of step-wise program composition and program families. The more recent works are, naturally, on more specific sub-topics of the domain engineering field of knowledge. One of the most referenced works in the product-line area is [Kang *et al.* 1990] with the introduction of feature diagrams. Regarding methodologies, Gomaa discusses the adoption of UML 2.0 for software product line development [Gomaa 2005]. IESE has produced a lot of industrial experience reports on software product lines [Anastasopoulos *et al.* 2000]. An overview of the practical application of domain engineering in product lines can be found on [Clements *et al.* 2002] and in software factories on [Greenfield *et al.* 2004].

Software Engineering
The development of software systems is still a very hard and difficult engineering process. In fact, the main aim of software engineering, according to Fritz Bauer is "The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines" [Naur *et al.* 1969]. To achieve these goals, Pressman states that

software engineering encompasses a set of three key elements: methods, tools and procedures [Pressman 2004]. In the context of these elements, various software engineering paradigms have been proposed and used. Examples are the waterfall model [Royce 1970], the spiral model [Boehm 1988a], or Rational Unified Process (RUP) [RUP 2004].

All these paradigms aim at provide sound engineering principles. Even if many of these paradigms have been widely adopted, it is still very hard to, make accurate predictions of a software project delivery date, for instance. If the duration of the project is not accurate then the project is not economically feasible. In order to maintain the economically feasibility of the project, normally the product outcome will have less functionalities, be less reliable or less efficient. It may even be economically worst because of maintenance cost that came out of the poor reliability and efficiency of the product. We have seen various documented reports of such difficulties in the software industry since the CHAOS report [Johnson 1995].

Reuse and Software Product Lines

Recently, more pragmatic approaches, like Extreme Programming [Beck 1999], have been proposed. Several of the authors of such approaches have founded the *Manifesto for Agile Software Development* [Beck *et al.* 2001]. One such pragmatic approach is based on the intuitive concept of reuse. The reuse approach is based on building new software systems reusing already existing and proved artifacts. With this approach software engineering projects become more predictable. Particularly, predictions of costs and delivery dates become more accurate. Software reliability can also improve because of the reuse of already tested and proved artifacts.

In the past, reuse has been adopted in the industry with relative success. Examples are the use of class libraries like wxWindows [wxWindows 2007]. These all have the benefit of providing the programmer the possibility of code reuse that deals with programming needs, like implementing graphical windowing systems or data containers structures such arrays and lists. However, these are all reuse of software of generic nature. The advantages of software reuse can be much more if exploited in specific domains.

Product families and product lines aim at promoting reusability within a given set of software products [Bosch 2000]. Software product lines have achieved substantial adoption by the software industry. The adoption of product line software development approaches has enabled a wide variety of companies to substantially decrease the cost of software development, maintenance, and time to market and increase the quality of their software products [Bosch 2002].

To accomplish reusability among various software products, there must be common characteristics among them. Normally, this means that the various software products must share the same domain. Therefore, an organization that has built several software systems in a domain also has acquired very good knowledge of such a domain. This knowledge can be used when building new software systems in the same domain. A fundamental technical requirement for achieving successful software reuse is the systematic discovery and exploitation of commonality across related software systems [Prieto-Diaz 1990]. A software product line approach involves domain engineering to build common assets, application engineering to reuse these assets when building new products, and technical and organizational management to apply the approach.

Commonality and Variability

By capturing the acquired domain knowledge in the form of reusable assets and by reusing these assets in the development of new products, organizations will be able to deliver the new products in a shorter time and at a lower cost [Czarnecki 1998].

So, we can say that reuse has to do with finding commonalities among software systems within a domain. Nonetheless, to build diverse software systems within a domain we also need to specify variability. Domain engineering focuses on supporting systematic and large-scale reuse by capturing both the commonalities and the variability of systems within a domain to improve the efficiency of development and maintenance of those systems. As such, variability is one of the key aspects of domain engineering.

Figure 7 depicts the life cycles of domain engineering and application engineering based on documentation available at the Software Engineering Institute (SEI) [SEI 2007b] of Carnegie Mellon University.

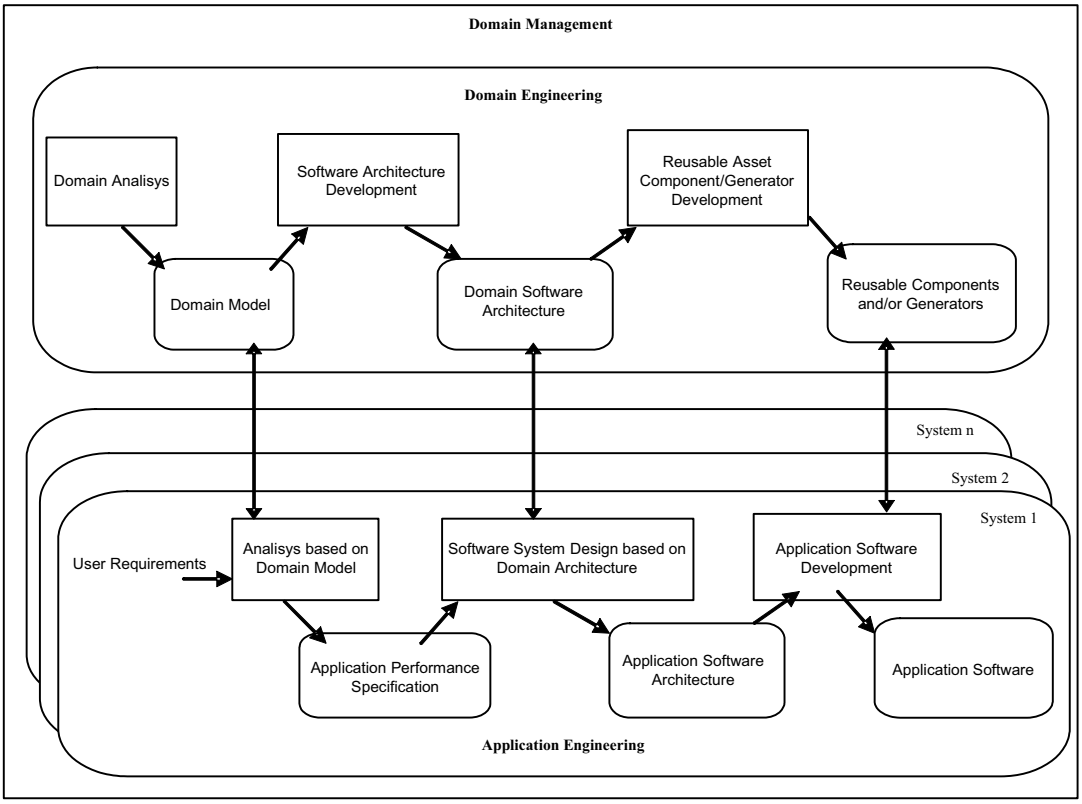

Figure 7: Domain engineering vs. application engineering (based on [SEI 2007b]).

In the figure it is clear that domain engineering has to do with *engineering for reuse* and application engineering with *engineering with reuse*. Based on that figure, it is also clear that the application engineer must reuse artifacts from domain engineering to instantiate a new application in the domain. This new application will have common functionalities with others in the domain but will also have differences that make it a particular instance of that domain. This means that the notion of variability and the methods, techniques and technology used to achieve variability are one of the most important issues in domain engineering.

Variability in software is achieved fundamentally by the following techniques [Svahnberg *et al.* 2000; Gurp 2003]:

- *Inheritance*, is used when the variation point is a method that needs to be implemented for every application, or when an application needs to extend a type with additional functionality.

- *Extensions and extension points*, are used when parts of a component can be extended with additional behavior, selected from a set of variations from a variation point.
- *Parameterization, templates and macros*, are used when unbound parameters or macros expressions can be inserted in the code and later instantiated with the actual parameter or by expanding the macro.
- *Configuration and module interconnected languages*, are used to select appropriate files and fill in some of the unbound parameters to connect modules and components to each other. By configuration is meant the process in which source code is selected from a code repository and put together to form a particular product. Module interconnection languages are one way of describing configurations.
- *Generation of derived components*, is adopted when there is a higher level language that can be used for a particular task, which is then used to create the actual component.

There are also more recent techniques and methods that came from the academia but have still limited adoption in the industry, such as: aspect-oriented programming [Kiczales *et al.* 1997], subject-oriented programming [Ossher *et al.* 1994]; frame technology [Bassett 1997]; feature-oriented programming [Batory 2004]; and generative programming [Czarnecki 1998].

Domain Engineering

In domain engineering, the term *domain* is used to denote or group a set of systems or functional areas, within systems, that exhibit similar functionality [SEI 2007a]. Domain engineering is the foundation for the product line software development approaches [Foreman 1996].

One can say that domain engineering started with the work of Dijkstra regarding *structured programming* and the notion of *programming for reuse* [Dijkstra 1969; Czarnecki 1998].

The next major reference is the work of Parnas on program families [Parnas 1976]. Parnas stated why one should study program families instead of individual programs. He also stated that a set of programs is considered a family when it is the case that in order to study this set, it is necessary to study the common properties among the elements of the set first, and then study the properties of the individual family members. He also stated that in a program family one should first study the commonalities (common features) and then the variability (diverse features) of each program.

The work of Neighbors is also of major importance. He introduced the first domain engineering methodology, named Draco, in his PhD [Neighbors 1980]. In his thesis he argues that many software systems are very similar and so should be built out of reusable software components. He also states that for reuse to be successful it is necessary to reuse not only code, but also analysis and design artifacts. Neighbors states that “the concept of domain analysis is introduced to describe the activity of identifying the objects and operations of a class of similar systems in a particular problem domain”.

There are not so many well-documented domain engineering methodologies. There are also less documented applied case studies. The Software Engineering Institute of Carnegie Mellon University is one of the exceptions. In fact, in 1990, SEI published a technical report regarding Feature-Oriented Domain Analysis [Kang *et al.* 1990]. To our knowledge this is the first method which claims it self to be a domain engineering methodology. One major advantage of this methodology regarding others is that it has much public available documentation. Another advantage is that much of this documentation regards applied cases.

In domain engineering, the domain products represent the common functionality and architecture of applications in a domain. These are generic and should be reused in the development of new systems in the domain. The generic nature of the domain model implies that there is variability in the possible implementations of applications (systems) in the domain. The development of new systems in the domain requires refinements in domain products so that the specificity of new system can be achieved. When we add specificity we are removing the variability of the domain model, i.e., we are selecting one of the possible choices of implementation. The process of removing generality - or adding specificity - in order to build a new system needs some mechanism to implement variability.

As already mentioned, the domain products are artifacts that can be reused in building new systems in the domain. These artifacts can be abstractions of functionalities or designs (i.e., architecture) to be reused in the development of new systems in the domain. Product frameworks and product lines are based on the reuse of such abstractions [Bosch 2000]. Thus, one can say that domain engineering should be used in the development of product lines and product frameworks.

In a domain there are common parts that represent invariants of the domain. If we are talking about product frameworks or product lines we can say that these common parts are software components that implement invariant functionalities (abstractions). In order to differentiate between diverse products in a product line or different applications from a framework we need a variability mechanism. On the modeling phase there is also the need to represent variability. Some usual design concepts that can represent variability are:

- *Aggregation/decomposition and generalization/specialization*. These are modeling concepts very familiar to object-oriented programmers. With aggregation, grouping several abstractions creates a new abstraction. Decomposition is the inverse of aggregation. When we decompose one abstraction into its components we are refining that abstraction.
- *Generalization/specialization*. When we create one abstraction by using the commonalities between abstractions we are generalizing. With generalization abstractions loose specificity. Specialization is the inverse of generalization. An abstraction is specialized when we add features to the abstraction. Specialization is also a refinement.
- *Parameterization*. Parameterization is a technique in which software is adapted/configured by substituting the values of the parameters in the software.

These design concepts are heavily used in the original *GoF* (*Gang of Four*) design patterns [Gamma *et al.* 1995]. Some authors defend that some design patterns can be considered as small product lines [Pree *et al.* 2002]. Patterns can be considered a micro-architectural view of a system. Most work in patterns is from a component or design reuse perspective. Software patterns are widely used in the development community because they normally originate from best practices [Fowler 2002].

There are several fields/disciplines that are related to domain engineering. As we saw, architecture modeling is a very important activity in domain engineering. One area of particular interest is how to describe a software architecture [Garlan *et al.* 1994]. Architecture Description Languages (ADLs) are used to describe the components, connectors, and information about their interactions that compose a system. There are several languages and tools that can be classified as ADLs [Medvidovic 1997]. This is also a very active field of research that is very strong related to domain engineering. For instance, the concept of software architecture is central to the Domain-Specific Software Architecture (DSSA) method [Hayes-Roth 1994]. We can see DSSA as an application of the concept of software architecture in a domain. In fact, one of the outputs of domain engineering is a software architecture for the domain.

The Object Connection Architecture (OCA) was presented in [Peterson *et al.* 1994] as a method that uses the outputs of Feature-Oriented Domain Analysis to build a generic design for the domain. This generic design encompasses software components that conform to the software architecture model proposed for structuring software systems in OCA.

There are also other methods and techniques that can relate to domain engineering because of the focus they put on reuse. Examples are the OOram method [Reenskaug *et al.* 1996] and the work on software frameworks [Fayad *et al.* 1999].

Next, we will briefly present some domain engineering methods. Some of them are explicitly aimed at software product lines. Our goal is not to present an exhaustive list of methods but one that is representative of the evolution of the methods in this field and that took a major role in this thesis to frame the research efforts described in the next chapters of this document.

2.2 Draco

Neighbors introduced the first domain engineering methodology, named Draco, in his PhD [Neighbors 1980].

Draco uses domain-specific languages, prettyprinters, source-to-source transformations and software components. Draco is based on the assumption that we can specify a program in a high level domain, i.e., the problem domain, and then transform that program successively into other domains, until we get a program in an executable domain. When we achieve this stage we have a solution to the initial problem in an executable format. We can say that Draco is a transformational system. In that perspective it has a lot of communalities with the model driven approach to software development.

The method is based on the definition of several domains with the respective domain-specific language and transformations between the domains. The *initial* domain is the domain of the problem or the business domain. A program in the domain is specified using this domain-specific language.

One can say that the Draco system was a precursor of domain engineering methodologies. It is also accurate to say that Neighbors work seems to have influenced almost every methodology or technology in the field of domain engineering. Other works based on transformation are Intentional Programming [Simonyi 1995], GenVoca [Batory *et al.* 1992], and Generative Programming [Czarnecki 1998].

The method is composed of basically three activities: determine domains of interest; research the domain; and construct a software system.

Determine domains of interest
The determination of the domains of interest is based on the goals of the organization and on the identification of areas where there is a demand for many similar systems. We can say that this corresponds to scoping the domain. The output of this activity is a problem domain in which the organization is interested in producing software. This result is used as input to the research domain activity.

Research the Domain

The research domain activity is based on: analyze the domain; construct a domain; test a domain; and add a domain to the library of domains. The result of domain analysis should be, at least, a domain analysis report. If the report has enough detail, then it can be used as a source to the activity of domain construction. The analysis of a domain is based on: information about the domain; experience with building systems in the domain; and the library of domain analysis reports and already available Draco domains (since new domains should be constructed in terms of domains already known to Draco). According to the author of Draco, the construction of a domain is a craftsman activity. Figure 8 presents the details of the research domain activity.

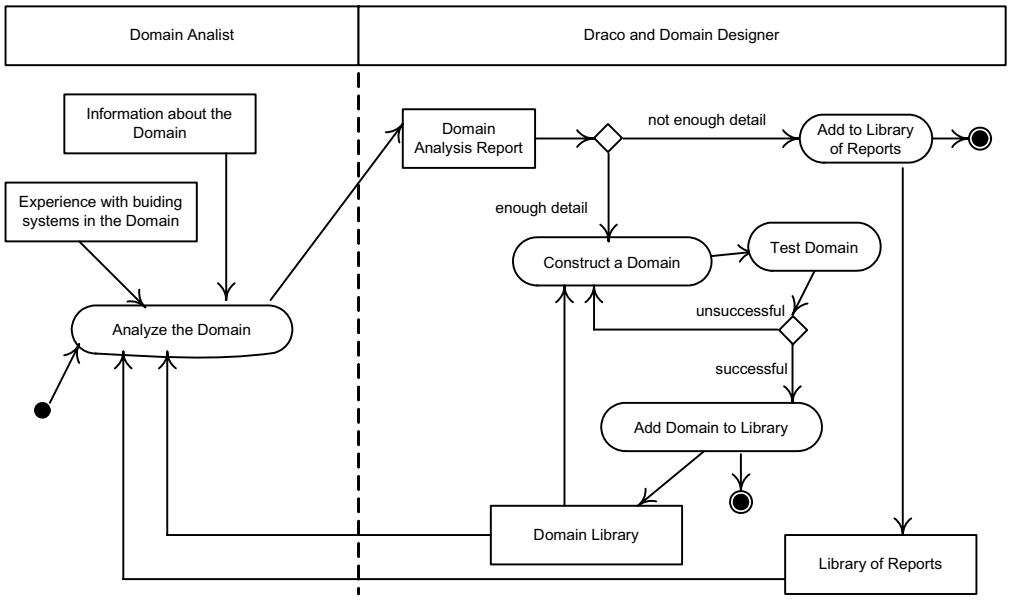

Figure 8: Draco activity of research a domain (based on [Neighbors 1980]).

Construct a Software System

The domain construction activity of Draco is based on the central concept of domain languages and transformations between them. As such, the final outputs of domain engineering in Draco are always domain-specific languages and tools. Because of that, the initial analysis phase of the method is very oriented towards identifying language constructs. The syntax of the domain language is designed (external form) and also a suitable internal form of the language (similar to a parse tree). A prettyprinter is created which can generate the external form of the language based on the internal form. Source-to-source transformations are designed which provide a mechanism to specify refinements in the internal representation language of the domains. Finally, the components of the domain are also designed. These components represent objects and operations in the domain. The components of a domain relate the internal form of a domain to the internal form of other domains.

At the application engineering level, applications can be built with or without the Draco system. Even if applications are built outside Draco, the knowledge that results from the process can be an input for the domain analyst. One can say that a major input of knowledge about the domain is the actual organization experience in building applications in the domain. The Draco method suggests that there can be other sources of information such as documents about the domain. The method also states that after the identification of the objects and operations of the

domain, the domain designer can specify the syntax of the domain language. What the method doesn't state is how to do this. Draco also doesn't specify clearly how to identify the objects and operations of the domain. These objects and operations represent the commonalities of a domain. The variability in Draco is the way we can combine these operations and objects. Since these combinations are only limited be the grammar of the language, the results of the domain analysis identify a very wide scope of variability, i.e., all the possible programs we can build with the domain language.

2.3 Feature-Oriented Domain Analysis (FODA)

The primary goal of the Feature-Oriented Domain Analysis method [Kang *et al.* 1990] is to provide a basis for understand and communicate about the problem space addressed by software in a domain. In order to achieve this goal, the method is based on the examination and study of a class of related software systems and the common underlying theory. The result should be a reference model that describes the class of software systems. The method also proposes a set of architectural approaches for the implementation of new systems. This means Feature-Oriented Domain Analysis, as the name implies, is focused on analysis of the domain, i.e., the analysis and representation of the problem space.

Since the central focus of the method is domain analysis, it encompasses basically three phases:

- Context Analysis: defining the extent of a domain for analysis;
- Domain Modeling: describing the problems within the domain that are addressed by the software;
- Architecture Modeling: creating the software architecture(s) that implements a solution to the problems in the domain.

Each of the phases of the domain analysis method is composed of several activities. The results of these activities are documents that describe domain knowledge. These documents define the scope of the domain, describe the problems solved by software in the domain and describe architectures that can implement solutions.

As we can see, the method has one phase for architecture modeling, which could mean it also addresses the creation of software solutions. As we will see later, this is not entirely true. In fact, the original method is very vague on how to evolve from the problem representation into the solution space.

The method also defines the possible roles of participants in the domain analysis process: end user, domain expert, domain analyst, requirements analyst and software designer. These roles can be further classified by their 'relation' with the method. The end user and the domain expert are sources to the method. The domain analyst is a producer. Requirement analyst, software engineer and the end user are consumers. Figure 9 represents the phases and products of Feature-Oriented Domain Analysis.

One very important aspect is the roles people, or systems, play. If we refer again to Figure 9, we can imagine a requirements analyst and a software designer using the products of a domain analysis when implementing a new system in the domain. In this scenario, we can also imagine a domain analyst using the feedback from the implementation of new systems to further extend and evolve domain analysis.

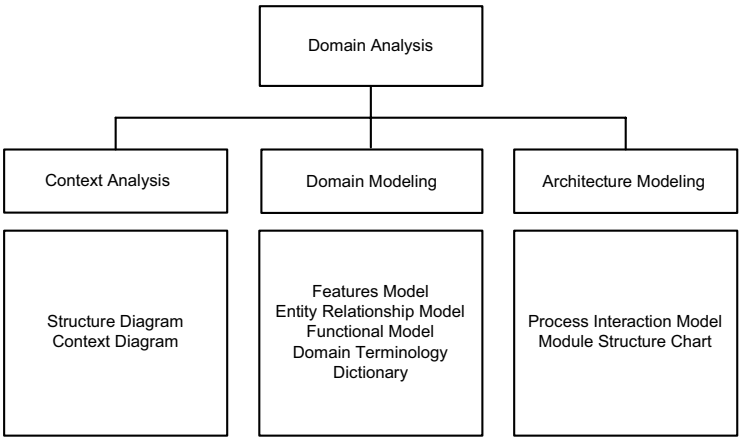

Figure 9: Phases and products of Feature Oriented Domain Analysis (based on [Kang *et al.* 1990]).

Features are the core concept of Feature-Oriented Domain Analysis. We can think of a feature as a characteristic of a concept. The method uses features to represent characteristics of concepts of the domain. Some of these features are invariant in the domain. Others can vary and there may be rules in the selection and composition of features in a domain. The concept of features in domain modeling is very used because the terminology adopted is very close to the end user and the domain experts. As such, domain models that use features become of easy understanding.

Context Analysis

The objective of the initial phase of the method is the definition of the scope of a domain in terms of the probability that the domain will give usable domain products. The relationships between the domain and the external elements are evaluated. The degree of variability of the domain is also evaluated. The availability of domain sources (experts, documentation, etc.) is also used to scope the domain.

As depicted in Figure 9 the documentation resulting from context analysis is the structure diagram and the context diagram. The structure diagram is used to show the relations between the domain and other domains. This type of diagram includes higher, lower and peer level domains regarding the domain in study. Higher domains are domains that include the domain.

The context diagram is a top-level data-flow diagram of the interfaces the domain has with other domains or entities. The particularity of this data-flow diagram is that the variability of the data-flows across the domain boundary must be indicated. If the variations are due to different features of the applications in the domain, this fact must be described. Because features are only introduced in the phase of domain modeling, this means that context analysis and domain modeling may be done in parallel. Entities that appear in the context analysis must be described.

Domain experts, end-users, documents and applications of the domain are all sources of knowledge that the domain analyst can use in the context analysis. The FODA method suggests some guidelines in order to scope the domain. However, no precise process is indicated. The feature model construction can start in parallel with the domain scope, so that initial identified variability can be represented using feature models. A common model can be constructed by classifying specifics of the contexts into general categories so that each context can be defined as an instantiation of the common model.

The authors of the method also suggest that applications used for the scoping of the domain be described using context and structure diagrams in order to validate the boundary of the domain. This should also be done using, at least, one application not included in the analysis.

Domain modeling

The method uses aggregation and generalization to capture the commonalities of the applications in the domain in terms of abstractions. Refinements are used to capture the differences between applications. Parameterization is used to specify the context of the refinements. As such, one can say that the result of the method is a group of abstractions of a domain and a series of refinements of each abstraction with parameterization. When a new refinement is introduced in the domain, the context in which the refinement is made must be defined in terms of parameters. Parameterization is the technique used by Feature-Oriented Domain Analysis to select the refinements of the domain abstractions. With the refinements, new applications in the domain can be specified.

Features and feature modeling are extensively used in the FODA method to model variability. The method uses *features* to model (parameterize) the capabilities of applications from the end-user perspective. In the domain modeling phase the domain analyst uses the information sources and the other products of the context analysis to support the creation of a domain model.

Feature analysis allows the domain analyst to capture the diverse capabilities of the applications in the domain according to the end-users. This is a very productive analysis tool since it models the problem space from the end-user's viewpoint. Thus, feature diagrams normally do not include technical capabilities. The viewpoint of the user is normally centered in the services or functionalities provided by applications and operating environments in witch they run. For instance, programming technical features should not appear in a feature diagram.

Figure 10 presents an example of a feature diagram. A feature diagram looks like an inverted tree. The structure of the relationships between features is represented by the connectors and visual indicators that can be used in the diagram. For instance, in Figure 10, air conditioning is an optional feature, as denoted by the circle in the end of the feature line.

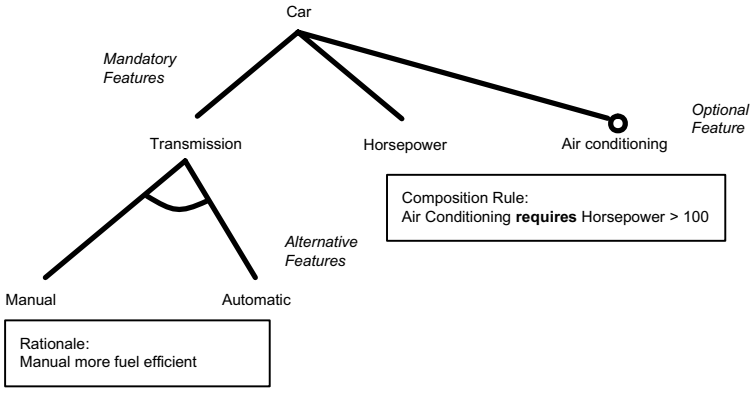

Figure 10: Possible feature diagram for a car (based on [Kang *et al.* 1990]).

There can be rules regarding the combination of features that cannot be specified only by visual indicators. In this case, the method uses what is called composition rules. In Figure 10 there is a composition rule that says the air conditioning feature, when present, requires that the feature

horsepower have a value greater than 100. Composition rules are used to define the semantics existing between features that are not expressed visually in the diagram.

Feature diagrams are a rich and useful tool regarding their expressiveness for documenting variability (mainly) from the end-user perspective. Features, and feature diagrams, are the basis for the specification of variation points in the final architecture of a system. They are also used to support the specification of the features to be included in particular applications of the product line. This process is usually called feature configuration.

Apart from the visual diagram features also have a textual description. The proposed form for describing features is presented in Figure 11.

```
Name: <standard feature name>
Synonyms: <name> [FROM <source name>]
Description: <textual description of the feature>
Consists Of <feature names> [ { optional | alternative } ]
Source: <information source>
Type: { compile-time | load-time | run-time }
Mutually Exclusive With: <feature names>]
Mandatory With: <feature names>]
```

Figure 11: Textual description of a feature (based on [Kang *et al.* 1990]).

Because features capture domain knowledge from the end-user perspective, it's very natural that most features are, in fact, capability features. Features related to capabilities can be further categorized into three areas [Myers 1988]:

- functional features: these are basically services that are provided by the applications;
- operational features: these are related to the operation of applications ;
- presentation features: these are related to what and how information is presented to end-users.

These are just the most common categories of features. It is possible that more categories of features exist in a given case. These new categories can be identified in the analysis of a domain. The method does not discard this possibility.

Apart from then end-user perspective there is the need to capture more precise domain knowledge from an implementation perspective. Feature-Oriented Domain Analysis does this using a kind of entity-relationship model. The purpose of this model is to represent the domain knowledge explicitly in terms of domain entities and their relationships, and to make them available for the derivation of objects and data definitions during the functional analysis and architecture modeling.

The entity-relationship model is based on Chen's method [Chen 1976] with the adoption of generalization and aggregation concepts from semantic data modeling that are used as predefined relationship types [McLeod 1978; Borgida *et al.* 1984].

Given the fact that the entity-relationship model contains domain knowledge from the implementation perspective it is the base for identifying and derive objects and components. The method makes no assumptions regarding implementation technology. Object-oriented programming or other methods and techniques can be used.

This functional model should be built after the feature and entity-relationship diagrams. Feature, entities and relationships can be used to support the activity of functional analysis. For instance, alternative features in the feature model may be used to identify generic functions. Alternative features are specializations of a more general feature, and the functionality corresponding to the general feature is defined as a generic function witch is inherited by the functions implementing the alternative features. In addition, the generalization/specialization relationships (i.e., *is-a* relationships) of the entity-relationship model can be used to identify generic objects and the functionality associated with the generic objects.

Architecture Modeling

As depicted in Figure 9, Feature-Oriented Domain Analysis also encompasses an architecture modeling phase. The focus of this phase is shifted to the design of solutions in the domain. In this context, the primary goal is to provide a base architecture to support the systems in the domain and the building of software components to be reused (when building these systems).

The architecture model is a high-level design of the applications in a domain. Therefore, the method focuses on identifying concurrent *processes* and domain-oriented common *modules*, and on allocating the features, functions, and data objects defined in the domain model to the processes and modules. The packaging of functions and objects into modules must be done considering the processing time of the features (e.g., compile-time, activation-time, and run-time) that each module implements.

One of the objectives of features is that they be used for the construction of software components. This has to do with moving from the problem space into the solution space. Implementation techniques must be used according to the analysis. The implementation techniques vary mainly according to the binding time of the feature. For instance, stable features with compile binding time can be build/packaged with pre-processor techniques or application generators and run-time features can be implemented as menu options.

Features have to be 'transformed' into software constructs that realize the variation points. There are different moments when this is possible. These moments are binding times, when the feature is realized in terms of software. The method describe three possible binding times for the realization of optional or alternative features:

- *Compile-time*: features that are decided when the system is built and do not change. This kind of features should be realized at compile-time of the system (package) for efficiency reasons.
- *Load-time*: features that are defined only at the beginning of the execution of the system. These features remain stable during the execution of the system. This usually originates what is called 'table-driven' software.
- *Run-time*: features that can change during the run-time of the system. The method gives as example menu-driven software. One example of such is a word processor that can have the auto spelling checker feature active or not.

According to Czarnecki these binding times are incomplete. In reality there may be other binding times, e.g. linking time or first call time (that is very important for just-in-time compilation) [Czarnecki 1998]. We can then generalize the binding time concept according to the specific times of the systems in the domain. For instance, there can be specific times like debugging time and testing time. It is possible also to conceive special times in the life cycle of applications like off-line time or emergency time.

Apart from the binding time (when to instantiate the feature or component) of a feature, there is also the problem of the binding local, i.e., the location of the feature (*where* to instantiate the feature or component). For this reason the concept of binding site was introduced to cover both situations [Simos *et al.* 1996].

The method proposes a layered approach for modeling the architecture of a product line. Figure 12 presents the architectural layers proposed by FODA. The architecture is defined at various levels of abstraction so that reuse can occur at the level appropriate for a given application.

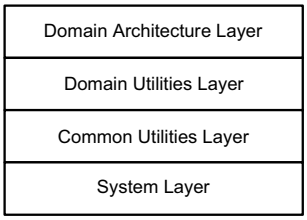

Figure 12: Architectural layers in Feature-Oriented Domain Analysis (based on [Kang *et al.* 1990]).

In terms of architecture, an application is a collection of programs (i.e., processes) that can be compiled separately and executed in parallel. These processes can be defined based on the functional analysis. Each process must be designed as a hierarchy of modules with the allocation of functions and data objects defined in the data-flow model. Then, domain-oriented common modules that can be used across the applications must be identified to increase the reusability.

At the top, the *domain architecture layer* is represented as a model showing the concurrent domain-processes and inter-connections between them. This model is called a process interaction model and is represented using the DARTS (Design Approach for Real-Time Systems) methodology [Gomaa 1984].

The *domain utilities layer* shows the packaging of functions and data objects into modules and the inter-connections between them. This is called module structure charts and is represented using the Structure Chart notations [Yourdon *et al.* 1978] following the DARTS methodology.

The *common utilities layer* contains modules that can be used across different domains. Normally programming aspects that are common to applications of diverse domains are realized in modules in this layer (examples are synchronization and communication aspects). Aspects that regard the operating system or the programming languages are part of the *system layer*.

As we saw, Feature-Oriented Domain Analysis presents some guidelines into what should be a domain architecture. Nevertheless, it doesn't present much information regarding the process of going from the problem space into the solution space. This issue is addressed in Feature-Oriented Reuse Method (FORM) [Kang *et al.* 1998], an evolution of the FODA method.

Feature-Oriented Reuse Method

Feature-Oriented Reuse Method (FORM) [Kang *et al.* 1998] is an evolution of Feature-Oriented Domain Analysis. In this evolution, features became the central concept of domain engineering and feature models are used not only in requirements engineering but also in the design phase (architectures) and in the building of software components. This means that FORM extends the adoption of features from the domain problem space into the decision and the solution space.

The designers use features to build architectures and the engineers use features to build applications in the domain. Figure 13 presents the mapping between features and artifacts/components in FORM.

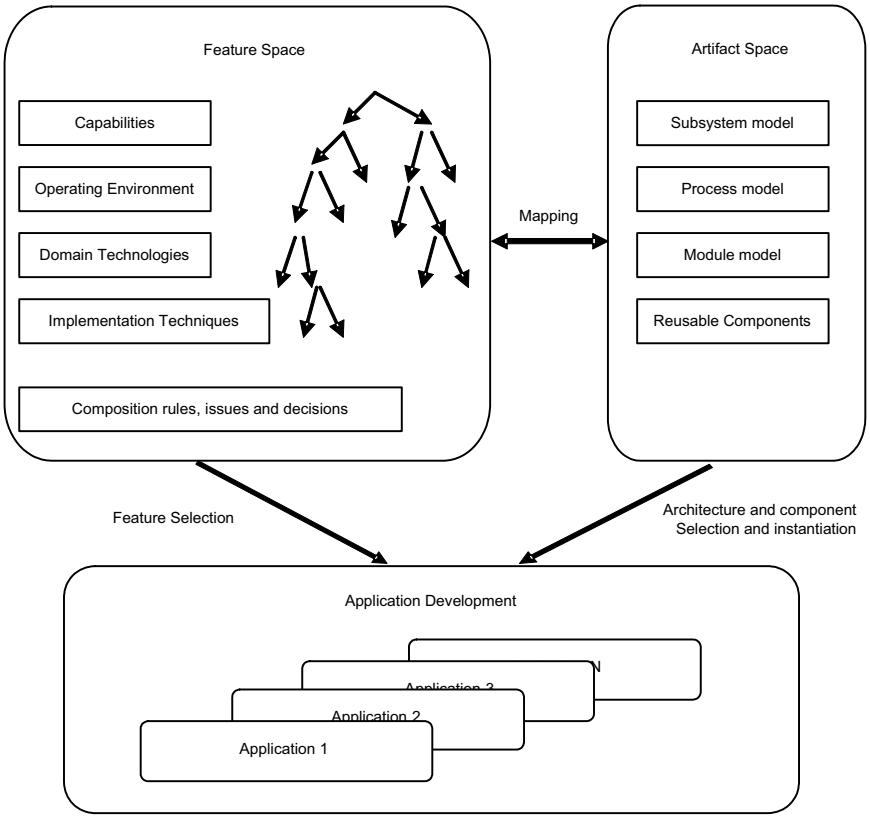

Figure 13: Overview of Feature-Oriented Reuse Method (based on [Kang *et al.* 1998]).

As with Feature-Oriented Domain Analysis, the FORM method also entails four major aspects related to features: capabilities, operating environment, domain technologies and implementation techniques. The major difference is that FORM focuses all the feature aspects and not only the capability features. By doing so, the method captures not only the features of the domain applications in terms of functionalities but also the features in terms of implementation details. We can imagine these different kinds of features as layers. There can be dependencies between features in different layers. For instance, the selection of one functional feature may imply the use of one specific implementation feature.

The idea beyond the focus on features in domain engineering is that they are used from the analysis phase into the architecture and component building phase. By doing so, in the application engineering process, the requirements phase can be done by selecting domain features of interest for the new application. This selection will guide the selection of the application architecture and the reuse of software components.

The adoption of features through all the engineering process raises one major problem: how to connect, or map, between the 'traditional' feature model used to describe the problem to a decision space and ultimately a solution space (components for reuse in application building)

As depicted in Figure 13, the artifact space has also layers. These layers represent reference architectures in the domain at different levels of abstraction. The method uses functional features mainly to identify required components, while non-functional (technical or implementation) features are used to partition components or to select type of connectors between components.

In FORM, modules are in fact the software components to be reused. Every process from the process model should be described in terms of modules. As modules reflect 'selection' of features at different levels, we can say that they normally satisfy a set of features. In fact, a module contains an abstract specification that satisfies features. Therefore, it is possible to have several concrete software components that match the specification of one module. The implementation of a module can be done in several ways according to diverse reuse strategies. For instance, there can be pre-coded components or parameterized template components of even skeleton code components that need to be completed.

Object Connection Architecture

The Object Connection Architecture (OCA) method is another method that we can say that complements FODA in the design phase [Peterson *et al.* 1994].

The OCA method is a proposal to realize concrete designs and software components based on domain models as the ones resulting from Feature-Oriented Domain Analysis. It also relies on the notion of subsystem as the first level of partition of a system. Subsystems interact with each other by means of *imports* and *exports*. We can say that imports and exports define the interface of the subsystems. Objects represent the behavior (and possible state) of real-world or virtual entities. Subsystems are composed of objects. A subsystem uses a *controller* to coordinate the activities of its composing objects. At a higher level, there is the notion of an *executive* that coordinates the subsystems. *Surrogates* are a concept that OCA uses to represent logical or physical devices that interface with the system. The concept of *signature* is used to represent the interface of OCA components (objects, subsystems and surrogates). To support a separation between an object and its actual implementation each object has a *manager*, which is a mediator between the object implementation and the clients of the object. The objective is to achieve a higher degree of independence from the implementation.

OCA describes the process of mapping from domain models to software components. Let's take for instance the case of the object concept. Objects are one of the more important parts of an OCA architecture because they represent the base functionality. They are the building blocks of subsystems which in turn compose the architecture of a system. As mentioned, objects represent entities, so they can be identified based on diagrams such the entity-relationship diagram of Feature-Oriented Domain Analysis. Objects that need to be represented are identified based on the selected features. For instance, if we don't select any feature related to an entity of the domain we don't need to define an object for that entity. According to OCA, one way to discover the operations of an object is from the possible features of the object identified in the feature diagram. Other source can be the functional model. The possible composition of features can also guide the object definition. For instance, all mandatory features (descendent from a selected feature) need to be implemented in the object. Alternative features can be supported with different implementations of the object.

2.4 Organization Domain Modeling (ODM)

Organization Domain Modeling (ODM) is another major domain engineering methodology [Simos *et al.* 1996]. To our knowledge, ODM is in structure similar to Feature-Oriented Domain Analysis

but its process is far more elaborated and detailed. As its name implies, the method is focused on the organizational aspects of domain engineering as opposed to more technological focus of other methods. As a result from this approach, the method is very detailed in terms of the process, activities and roles it prescribes for the adoption of domain engineering in an organization. The specific objectives of ODM are: to make the domain engineering process more systematic, formal, manageable and repeatable; to ground domain engineering projects in a specific organization context; to maximize use of legacy artifacts and knowledge; to reveal the hidden constraints embedded in legacy systems and artifacts; to encourage exploration of maximum variability within the domain; to provide effective strategies for selecting an intended scope of applicability for asset bases; and to support evolution of the asset base, and the scale-up of the technology to support new kinds of organizations, organized around domains rather than around systems or products.

According to the authors of ODM, domain engineering has two fundamental aspects that make it distinct with regard to single system engineering: (1) its scope regards multiple systems and, therefore, the aim is to model the space of solution alternatives for several applications of the domain; (2) its scope can be smaller then the scope of single systems. This may seem contradictory with the previous aspect but it is true if domain engineering is focused on specific functionality of applications of the domain as opposed to model the whole system.

The first aspect regards the scope for the market of the domain, while the second aspect regards the scope of feature coverage of the domain. These are key aspects of ODM that have influence in all the phases, activities and artifacts of the methodology.

Another focus of ODM, that justifies its name, is the recognition and assertion that domains are always socially situated or socially constructed. They are shaped by the context of multiple overlapping communities of use, development, maintenance, customization, and application. ODM explicitly models social and organizational context to ground the entire process.

Figure 14 presents the phases of ODM. ODM was projected to be adaptable so the method supports other activities that are not the core of domain engineering but that, eventually, organizations may require in their adoption of domain engineering. In ODM these are called supporting methods, and they can also be integrated in the ODM process. ODM is also a part of a broader context called *Software Technology for Adaptable, Reliable Systems* (STARS) [DARPA 1994] *Conceptual Framework for Reuse Processes* (CFRP) [Unisys 1993], a defense research program sponsored by the U.S. Department of Defense related to software reuse. For instance, CFRP includes application engineering while ODM is only focused on domain engineering.

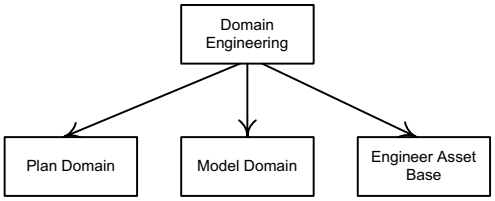

Figure 14: ODM process phases (based on [Simos *et al.* 1996]).

ODM can be classified as a very flexibly, adaptable and comprehensive domain engineering method. ODM does not prescribe any particular system modeling, engineering, or market analysis method. The user of ODM has to customize the method for its requirements, provide methods, and select appropriate notations and tools. This characteristic of the method also implies consequences. One of the major consequences is that the ODM process requires many resources that may be out of reach for some organizations, for instance, for small and medium enterprises (SME).

Plan Domain

As a result of its high concern with the organization context, ODM has a scoping phase that is very detailed when compared, for instance, with FODA. In ODM the scoping phase is called *plan domain*. Figure 15 presents the tasks for the *plan domain* phase. Basically, *plan domain* is composed of *set objectives*, *scope domain* and *define domain* sub-phases.

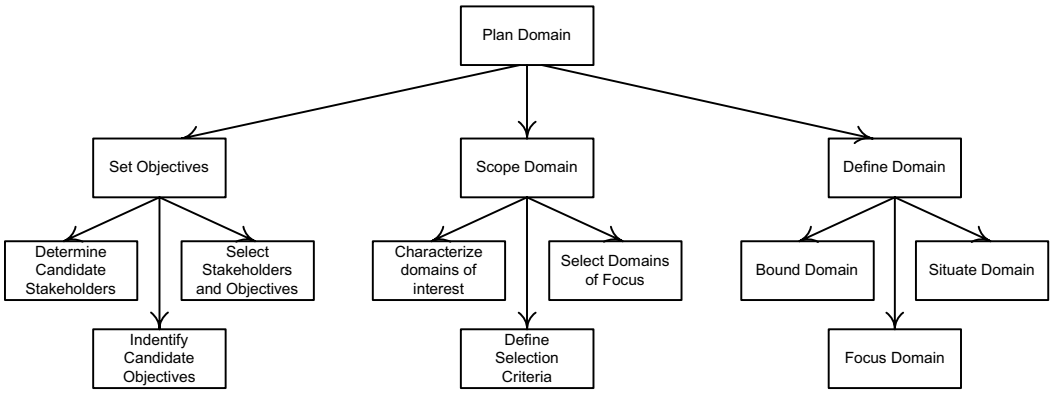

Figure 15: ODM plan domain phase (based on [Simos *et al.* 1996]).

The artifacts that result from the *plan domain* phase of ODM basically permit the identification of the domain and its boundaries. In the end of this phase it is possible to know what aspects of systems fall within the scope of the domain. This means identifying key interfaces from domain functionality to related system capabilities that are to be considered external or internal to the domain. This information permits an informal identification of common functionality of the domain as well as capabilities that are inside or outside of the domain. The information also includes the historical context of the domain, broader, narrower, and related domains.

The *plan domain* phase does not really identify variability. What are identified are the capabilities or functionalities that are within or outside the domain. It is only in the *model domain* phase that variability is identified and modeled. Figure 16 presents the tasks of the ODM *model domain* phase.

As Figure 16 depicts, a domain in ODM is described by a lexicon, concepts and features. In order to identify these elements data from the domain must be collected. This is the goal of the *acquire domain information* sub-phase. Similarly to other methods, ODM uses system artifacts and domain informants as sources for this task. Significant outputs of this task are:

- Features of Interest: Short, informal sentences describing features of a system or many systems to which attention was drawn during data elicitation.
- Domain Terms: Terms that have specific meanings for practitioners in the system settings studied, as deduced from the artifacts and informant data.

Model Domain

With the acquired information, a domain model is constructed based on three fundamental elements: *lexicon*, *concepts* and *features*. As such, the *lexicon*, the c*oncepts* and the *features* of the domain are the base to describe commonality and variability. In ODM, domain modeling can be thought of as defining a formal language for describing domain entities and behavior. Following the language analogy, lexicon terms provides the vocabulary for this language, concepts provide

the semantics, and features correspond to actual sentences or statements in the language. Unlike a natural language, however, the domain language produced in domain modeling creates a fixed repertoire of statements that can be made about domain entities.

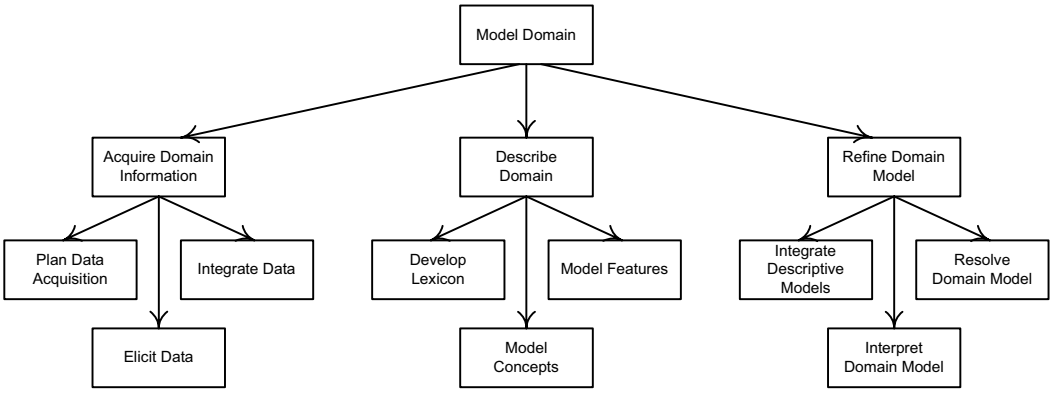

Figure 16: ODM model domain phase (based on [Simos *et al.* 1996]).

ODM adopts a broader definition for the term *concept*. Usually, in software engineering, the term *concept* is used for entities primarily within the operational environment for the systems within the domain. However, in ODM, the term *concept* is used to address elements in the development setting or any other domain setting. The criteria to choose to focus on something as a *concept* should reflect phenomena that display patterns of commonality and variability across domain exemplars significant for the modeling objectives. In order to facilitate the starting of concept modeling, ODM supports *concept starts sets* as a way to customize the method and help the modeler. A *concept starter set* provides an initial set of elements which are generally presumed to be a useful starting point in modeling certain kinds of domains.

In ODM, a *feature* is a difference observed by modelers among multiple exemplars of a concept of interest. Features should always be of interest to some domain stakeholders. As a result of this context of utilization, the concept of feature is broader in ODM as compared to FODA. Nonetheless, in both cases it is used to model variability. Another key difference is that ODM does not prescribe any specific modeling language while other methods like, for instance FODA, do. ODM also introduced the concept of *binding site* of a feature. Basically it represents the site where a feature is bound to a particular variant. This concept is significant because prior to ODM the most important factor for feature implementation was its *binding time*. With ODM, and the introduction of the *binding site* concept, methods start to explore other concepts for characterizing features.

The primary artifact that results from this phase of ODM is the *domain model*. This model describes the common and variant features of systems within the domain, and rationale for these variations. This model will be the base for selecting the range of variability to be supported by the asset base. The building of the asset base is presented next.

Engineer Asset Base

The last phase of ODM is where variability is implemented. Figure 17 presents the tasks that constitute the *engineer asset base* phase of ODM. In this last phase of ODM the goal is to scope, architect, and implement an asset base that supports a subset of the total range of variability encompassed by the *domain model*, a subset that addresses the domain-specific requirements of a specific set of customers.

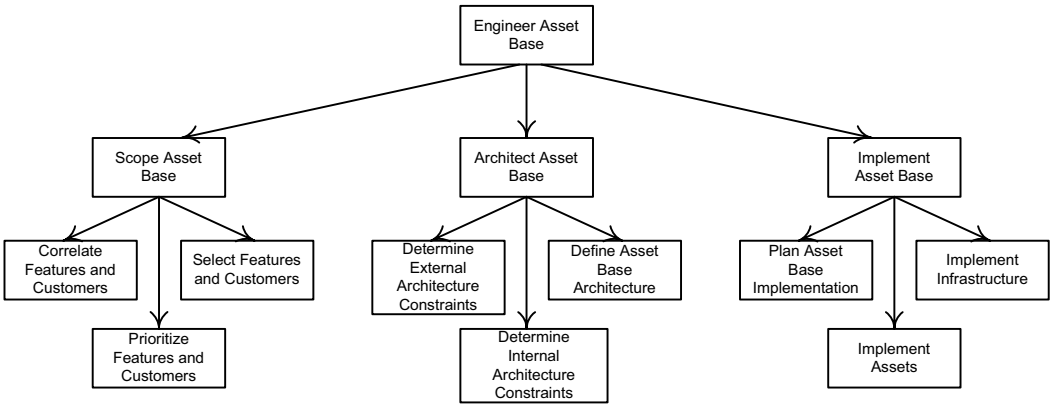

Figure 17: ODM engineer asset base phase (based on [Simos *et al.* 1996]).

2.5 Domain Analysis and Reuse Environment (DARE)

According to its authors, Domain Analysis and Reuse Environment (DARE) is a domain engineering method which aims to support automation and focuses on domain analysis to extract high level domain information from experts [Prieto-Díaz *et al.* 1995].

In DARE there is a clear analogy between the method activities and the scientific and engineering methods. According to its authors, domain analysts follow the scientific method, while software engineers follow the engineering method. This statement is based on the fact that the goal of domain engineering is to specify a class of problems, to propose a generic solution, to create an architecture that is representative of the generic solution, and to design reusable elements that will fit the architecture. In this perspective, proposing a domain architecture is an inductive process equivalent to postulating a theory, analyzing existing systems is equivalent to conducting observations, abstracting the architecture from existing systems is equivalent to doing experiments, and refining the architecture is equivalent to testing the theory. On the other end, the goal of a software engineer is to match a given problem to an instance of a generic solution in a domain, to specify the solution, and to design and create the product.

DARE uses the concept of *domain book* to capture the information resulting from domain analysis. This information results from domain experts, domain documents, and code from systems in the domain. Once completed, the domain book provides a detailed specification of the domain, including the generic architecture for the domain and domain specific reusable components.

One of the major advantages that DARE authors state about the method is its comprehensive prescriptive strategy for domain analysis, and its tool support. Figure 18 presents an overview of the DARE method. We observe that a postulated domain architecture is basically the result of a top-down process, that is, by induction from domain experts experience. On the other end, we see that the bottom-up process is deductive and based on faceted conceptual clustering [Prieto-Diaz 1991]. This bottom-up approach is systematic and repeatable and, as such, can be automated. This possibility of automating a significant part of the domain engineer method is a significant difference from other methods that do not explicitly aim at this goal.

As depicted in Figure 18, the DARE method uses two approaches for domain analysis. One is top-down, significantly based on the domain expert's knowledge and highly creative. The other is bottom-up, based on domain documentation and code, and can be automated to a significant extent.

The automation is based on a technique used in library science for deriving faceted classification schemes for special collections [Prieto-Diaz 1991]. The two approaches are then combined to fill a domain book. The domain book is used in DARE to structure and represent the outputs of the domain analysis process.

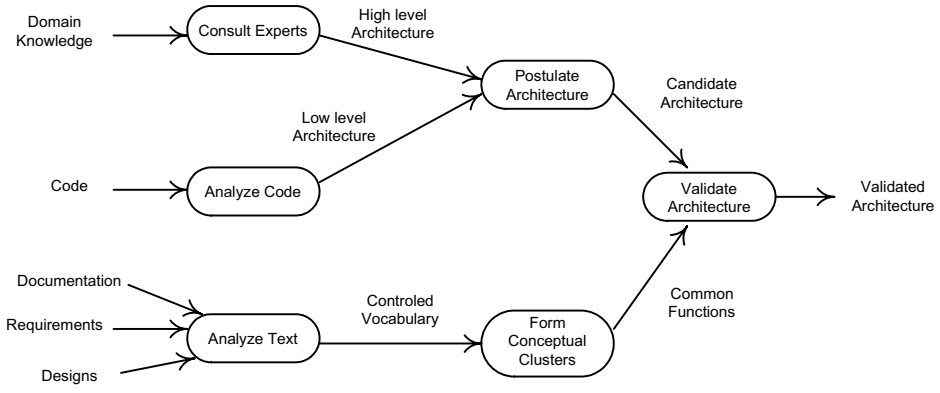

Figure 18: Overview of the DARE method (based on [Prieto-Díaz *et al.* 1995]).

When compared to other domain engineering methods, DARE has the particularity of aiming at automating the bottom-up approach to identify variability in the domain. The technique used is based on analyzing words and phrases that appear in documents that describe the domain. As mentioned earlier, DARE uses a faceted classification approach for the bottom-up process. Table 1 presents an example of a faceted classification for a domain of library systems.

Table 1: Partial faceted classification for the library systems domain (based on [Prieto-Díaz *et al.* 1995]).

Function	Items	Users	Request	Notice	Status	Subject-area
browse	aliases	borrower	change	address-change	delinquent	art
cancel	books	child	loan	available	lost	children
change	catalogs	delinquent	register	borrowed	on-shelf	engineering
check	dictionaries	librarian	reserve	loan-renewal	on-loan	fiction
lend	directories	privileged	search	new-acquisition	out-of-print	law
make	government-doc	registered	…	overdue	registered	science
process	indexes	regular		registration	suspended	…
register	journals	senior-cit		returned	…	
report	magazines	student		schedule-change		
reserve	maps	…		stolen-lost		
return	newspapers			task-change		
search	records			…		
send	textbooks					
update	…					
…						

DARE also uses the concept of *feature*. However, to our knowledge, this concept is limited when compared to FODA and ODM. In DARE, features are represented in feature tables. Rows in the table correspond to domain features while columns represent the values that the feature can take for different systems in the domain. This information seems to be heavily based on the domain expert's knowledge. However, *facets* are also used to help generate the system feature table that captures summary information about commonalities and variabilities of systems across the domain.

The DARE method does not cover all the activities related to the generation or development of reusable artifacts. DARE only supports asset identification and requirements definition within asset creation. As such, we will describe the domain architecture specification of DARE, since reusable assets should be conformant with that generic architecture.

The notation used for architectures in DARE is based on the concept that any architecture and architectural view can be represented as a network of frames similar to an attribute grammar. Each element of the architecture is defined by a standard *frame* with a fixed set of slot-value pairs. For each frame we must provide a *name* and a *type* identifier, where *name* is a unique name and *type* is the type of element in the architecture. Possible types are: system; subsystem; class; object; framework; function; data-structure, etc.

Slot values provide a way to specify relationships between elements of the architecture. Examples of relationship types are: requires; provides; consists-of; inherits; and implements. Although most of these values are domain independent, some of the values assigned to the slots can be domain specific, derived from a faceted domain vocabulary. If such is the case, then it becomes more clear the relationship between the domain architecture and the faceted classification resulting from the bottom-up process of domain analysis. This approach also facilitates *validations*. For instance, one could *validate* if the facets of a system are supported by the domain architecture.

A partial definition of the grammar of the architecture definition language in Backus–Naur form (BNF) is shown in Figure 19.

```
Domain-Architecture := context-diagram
                       high-level-decomposition-diagram
                       {domain-element}*
context-diagram := system
                       {external-entity, {relation}+}*
                       domain-boundary-statement
system := system-graphic-symbol
                       system-definition
external-entity := external-entity-graphic-symbol
                       external-entity-definition
high-level-decomposition-diagram := {domain-element, {relation}+}+
domain-element := element-type
                       element-name
                       {frame}*
                       {domain-element}*
element-type := subsystem | function | object | class | framework |
                       data-structure|...
frame := {slot-name, slot-value}+
slot-name := provides | requires | element-of | consists-of |
                       communicates-with | performs | computes |
                       has-children | child-of | represented-as |...
slot-value := domain-element | name
```

Figure 19: A partial definition of the grammar of the architecture definition language used in DARE (extracted from [Prieto-Díaz *et al.* 1995]).

2.6 Family-Oriented Abstraction, Specification, and Translation (FAST)

Family-Oriented Abstraction, Specification, and Translation (FAST) is a domain engineering method developed at Lucent Technologies [Weiss 1998]. FAST is aimed at providing a systematic

approach for analyzing potential families and supporting the rapid generation of family members by using a domain specific language suited for specifying the family members. A translator is used to automatically translate from the family specification into the individual applications.

FAST has two development phases: domain engineering and application engineering. The domain engineering phase includes defining the family and generating reusable assets. Defining the family is composed of discovering the family requirements, potential family members and identifying commonalities and variabilities between them. To identify the common and variable requirements, FAST uses a process called "commonality analysis". This is the core part of the FAST method.

Several reusable assets result from the commonality analysis: a domain specification language; product abstractions specified using the language; and the translator required to generate the final code. Since the development of a domain specific language may require significant resources, Nakatani *et al*. describe how this process can be facilitated in FAST by the use of *jargons* [Nakatani *et al.* 1999].

Commonality analysis is based on two primary sources of abstractions: the terminology used to describe the family and on assumptions that are true for all family members. To complete the scope of the family it is necessary to include variability. Variability makes it possible to determine the possible future members of the family. Variabilities define the scope of the family by predicting what decisions about family members are likely to change over the lifetime of the family. For each variability there must be a range of possible values. These ranges of values act as parameterizations of the variabilities, and are known as *parameters of variation*. In addition to specifying the range of values for each variability, the method also specifies the time at which the value is fixed, i.e., the binding time for the decision represented by the variability. This is similar to the concept used in FODA for the binding time of features.

The information gathered during commonality analysis facilitates the building of a generic architecture and reusable artifacts that supports actual family members as well as possible future family members. The results of the commonality analysis are compiled into a document that is similar to the domain book of the DARE method.

The process of building the commonality analysis document is iterative and human based. The process is based on meetings that include domain experts, a moderator and a recorder. The moderator must be an expert in the FAST method. The recorder is someone that registers the results of the meetings. The recorder and the moderator can be the same person. The moderator must be able to recognize well-formed, clear, and precise definitions, commonalities, variabilities, parameters of variation, and useful issues, and also know how to guide the discussion to produce them.

Domain-Specific Languages

One of the goals of FAST is to provide a way to rapidly support the generation of domain members. To achieve this goal, FAST uses domain specific languages to specify domain members. Such domain specific languages should be based on the variabilities identified and modeled during commonality analysis. In order to develop the required domain specific languages in a time that is *adequate* for the FAST projects, the jargons approach was proposed [Nakatani *et al.* 1999].

According to the authors, jargons are DSLs that are unusually easy to make. The key differences between jargons and DSLs are:

- All the expressions in jargons have the same abstract syntax

- All the jargons are processable with the same generic interpreter specialized at runtime with the semantics of the pertinent jargons.

Jargons can be made distinguished by:

- Having concrete syntax that is different from each other
- Having specific set of actions corresponding to the expressions that define the semantics of the jargon

There is also a set o tools that supports the development of jargons. These include a generic interpreter (the *InfoWiz*), a programming language for specifying actions (named *Fit*), and API functions for interfacing actions to the interpreter. The abstract syntax for the expressions has the following format:

```
;term(note-1| ...| note-n)[memo]
```

The ';' is used as a metacharacter, to distinguish jargons' expressions from plaintext. The *term* is the name of an expression. If the names for the *terms* of different jargons don't collide then the jargons can be used together. The *memo* is the information that is the focus of the expression. The *notes* are either attributes of the *memo*, or parameters to control the processing of the expression.

An *action*, which is a function written in the Fit programming language, specifies how the information associated with an expression (the *memo*) should be processed. To interpret a program written in a jargon, the jargon interpreter needs to read files that contain definitions of actions. These files are called *wizer*. The *InfoWiz* interpreter processes a program written in jargon by parsing the text and then traversing the parse tree in top-down, left-right, depth-first order and executing the action corresponding to the expression at each node of the parse tree. The result of the expression is appended to an output buffer. The *memo* of a parent expression is composed of the concatenation of the output buffers of the of the child expressions.

Lets assume the following jargon expression: `;greet[InfoWiz]`. A possible definition of an action for that expression is (using the *Fit* language):

```
A_greet
        WizOut "Hello, " GetWizMemo
```

If the presented jargon expression was interpreted the output would be: `Hello, InfoWiz`. In this example, `WizOut` and `GetWizMemo` are API functions that interface the Fit language with the jargon interpreter. `GetWizMemo` returns the memo of the expression and `WizOut` concatenates its arguments and appends the result to the output buffer.

The jargon approach to develop DSLs seams very simple and effective. Their authors state that the development of DSLs can be reduced from months or even years to days or weeks. They also defend other advantages when comparing the jargon approach to conventional DSL development. More details about these statements and the jargons approach can be found in [Nakatani *et al.* 1999].

The use of jargons in FAST is relatively straightforward since the commonality analysis document contains variability information that is organized in a structure similar to the abstract syntax of the jargons expressions. The *Variabilities* and *Parameters of Variation* sections of the commonality analysis document are used as a source to specify jargons. If needed, several jargons can be developed to model different variability *aspects* of the domain. Jargons are then used to model family members in terms of their variabilities. The jargon interpreters function as *translators*, by translating jargon models into *variability code*. Finally, the code developed to

implement commonalities can be integrated with the code resulted from the jargons in order to produce the family member.

2.7 Reuse-Driven Software Engineering Business (RSEB) and FeatuRSEB

Although not explicitly described as a domain analysis or domain engineering method, we include Reuse-Driven Software Engineering Business (RSEB) [Jacobson *et al.* 1997] in this analysis because of its detailed technical and organizational components. Also, as far as we know, the method was also the base for the first proposal to integrate use cases and features.

This method is based on Object-Oriented Software Engineering process (OOSE) [Jacobson *et al.* 1992]. It extends OOSE with a more reuse-oriented process. This method introduces an interesting approach: the adaptation of an existing object-oriented analysis/design method used for application engineering in order to use it in "engineering for reuse". RSEB extends OOSE with architectural constructs for families of related applications built from reusable components. It uses UML as the base notation [UML 2005].

The method has a strong focus on organizational issues. It provides technical support as well as a management perspective. According to the authors of RSEB, for reuse to be successful, the organization must be driven by the necessities of software reuse. As such, besides the technical perspective, RSEB includes process, organizational and business proposals for the organizations to achieve successful reuse. This focus on the organizational perspective makes the method similar to ODM. However, on the contrary to ODM, RSEB proposes an explicit notation as well as concrete technical components.

Figure 20 presents an overview of the RSEB method. The method is comprised of three major technical activities: Application Family Engineering, Application System Engineering and Component System Engineering.

In RSEB, an application system family is a set of application systems with common features. To support the application system family, the application family engineering activity is responsible for the development of a layered architecture. In this layered architecture, each layer is built on top of another more general layer. Upper layers are more application specific, lower layers are more general.

The method adopts the concept of *variant point* to represent variability in the models. A variant point identifies one or more locations at which the variation will occur. The graphical notation for variability points is a dot. This dot is presented inside the model element where the variation occurs. Model elements that represent variations are connected with a line to the correspondent variation point.

RSEB originally includes the concept of feature. In RSEB, a feature is described as being a use case, part of a use case or a responsibility of a use case. However, the previous statement is basically the only reference that RSEB makes to features. It is only with FeatuRSEB, an evolution of RSEB, that the feature concept is integrated into the method [Griss *et al.* 1998]. FeatuRSEB is described as an integration of FODA and RSEB. In FeatuRSEB, the feature model becomes the central model of the method. It is used to provide an abstract and concise syntax for expressing commonality and variability in the domain. By adopting features, the authors of FeatuRSEB claim that the role of use cases and features becomes clearer. The use case model provides the "what" of

the domain: a complete description of what systems in the domain do. The feature model provides the "which" of the domain: which functionality can be selected when engineering new systems in the domain.

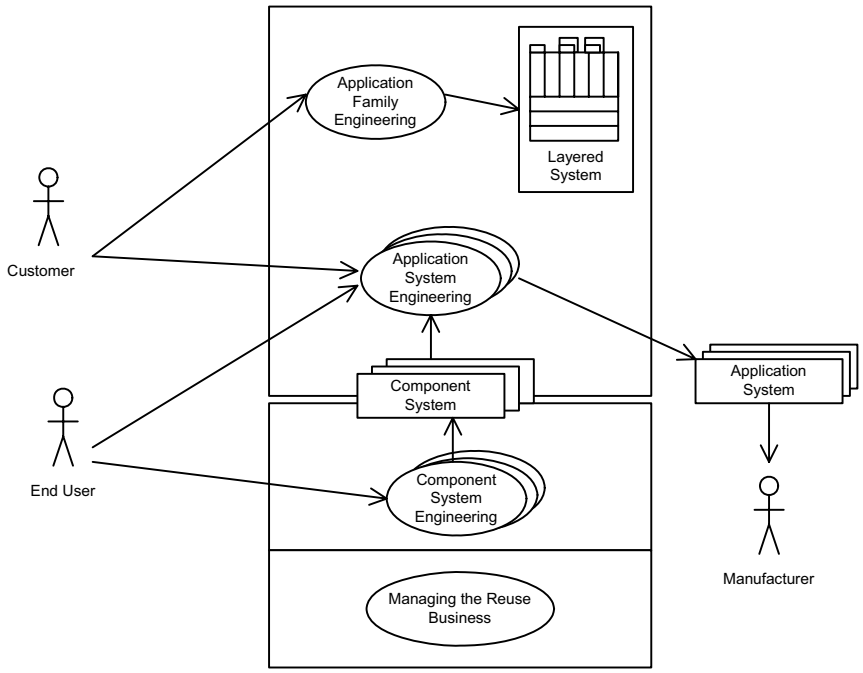

Figure 20: Overview of RSEB (based on [Jacobson *et al.* 1997]).

The relation between features and variation points is that variable features are exploited at variation points in components.

In the following sections that related to RSEB we will describe the original method as well as featuRSEB.

The major goal of RSEB is to produce a layered architecture that can be used to support the applications of a family of applications, i.e., a set of applications with common features. To achieve this goal, the method proposes the Application Family Engineering process. Figure 21 presents an overview of the process.

The typical layers of the architecture that results from the application family engineering process are:

- The application system layer. This layer contains one application system for each software system that offers a coherent set of use cases to some end users.
- The business-specific layer. This layer contains component systems specific to the type of business.
- The middleware layer. This layer contains components that provide utility and platform-independent services.
- The system software layer. This layer contains software for the computing and network infrastructure.

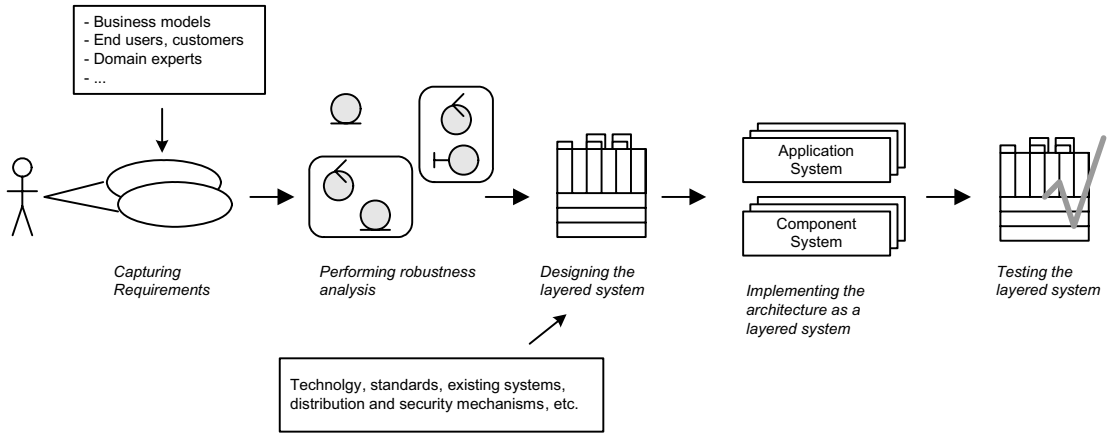

Figure 21: Overview of the Application Family Engineering process of RSEB (based on [Jacobson *et al.* 1997]).

RSEB proposes that the identification of analysis objects should be based on the "boundary-control-entity" pattern. In this pattern, control objects act as a central manager for interactions with several boundary and entity objects. This separates user interface concerns (*boundary* objects) from application functionality concerns (*control* objects) and structure (*entity* objects) and also use-case-specific behavior from entity objects. When applied to use cases, responsibilities that are specific to specific use cases are placed in distinct *control* objects. More general responsibilities common to several use cases can be placed in shared *entity* or *control* objects. Interactions of the system with actors become the responsibility of *boundary* objects. Control objects act also as coordinators of other objects and, as such, they have a responsibility similar to the *mediator* object in the Mediator pattern [Gamma *et al.* 1995]. This technique is used to identify the realization of use cases in all three processes of RSEB (family, application and component engineering). Next, we describe the major steps of this technique.

First, the "boundary-control-entity" pattern is used to map each use case into a collaboration of objects. Next, the method proposes the homogenization of the set of identified objects. This is done by deciding whether similarly named objects are in fact the same object, different instances of the same object type or distinct. According to the findings, the objects are renamed or merged as appropriate. The third step in the robustness process is the central one. In this step, objects are grouped into stable subsystems, possibly by re-grouping or merging objects. Principles of cohesion and coupling can be applied to increase the robustness of object clusters. Finally, use cases are examined to assign responsibilities and operations to corresponding objects. At the end of this step the there is one analysis object model for each use case.

As depicted in Figure 21, the other two main processes that compose RSEB are Application System Engineering and Component System Engineering. Application System Engineering is aimed at the development of specific applications of the application family. Since every application will be different there is no need for a major support for variability in this process, with the exception of supporting the reuse of *variable* components.

FeatuRSEB

We will now discuss the particularities of the FeatuRSEB evolution of the RSEB method. FeatuRSEB extends RSEB with the adoption of feature diagrams (as in FODA) and with a domain engineering perspective. In this perspective, the goal is not to provide and architecture that supports

a set of applications that share features, but to develop an architecture and reusable artifacts that can be used to support present and future applications in the domain.

In RSEB, the method is driven by use cases, and variability is identified at specific points in models. These points are identified as *variation points*, and their possible variants are also represented. In FeatuRSEB, a feature model needs also to be constructed to provide a global and integrated representation of the variability in the domain.

Since in FeatuRSEB the goal is to *engineer a domain* and not only an application family, the sources for requirements and the process are slightly different from RSEB. Similarly to other domain engineering methods, a significant part of requirements are based on the analysis of existing applications in the domain.

Similarly to RSEB, FeatuRSEB proposes that the *relations* between different models of the system should be explicitly maintained by using *traces* (i.e., links) between the correspondent elements.

Variability and Variation Points
The RSEB method is driven by use cases, and variability is identified and represented in all types of models by the concept of variation point and variant. In FeatuRSEB, features are used to be the central model that integrates and expresses variability. The feature model provides a "configuration roadmap" through the use case model, guiding through an understanding of what can be combined, selected and customized. Features can also be used to express constraints that although possible to represent in other models would pollute them with information that is out of their natural scope. They can also be used to model characteristics of a system that are difficult to express in other models, for instance, performance constraints.

Figure 22 presents an example of the notation used for variability. In this figure we can see how a variation point is represented in two distinct models. The concept of variation point is always represented in the same way, no matter the *type* of model. It is always depicted as a black dot with, optionally, an identification enclosed in curly brackets. The variants are connected to the corresponding variation point and a stereotype can be used to denote the *type* of variability support.

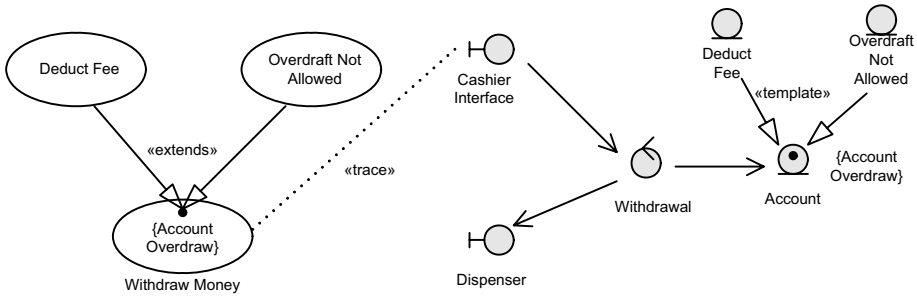

Figure 22: Example of variability notation used in RSEB (based on [Jacobson *et al.* 1997]).

Features
Feature notation in FeatuRSEB, although based in FODA is slightly different. Figure 23 presents an example of the notation adopted for feature diagrams. Basically in FeatuRSEB a distinction is made between two types of alternative features: *OR* alternatives and *XOR* alternatives. An alternative feature consists in a feature that acts as a variation point (called a *vp-feature*). The

subfeatures of this features become the variants. If the variant must be selected (i.e., bounded) at use time, then we have an *OR* alternative. If only one variant can be selected (i.e., bounded) at reuse time, then we have an *XOR* alternative. Or alternatives are usually bound at runtime. Therefore, all the variants must be included in the system distribution. As we will see throughout this thesis, particularly in Chapter 4, the tendency in the scientific community is to extend the modeling power of feature diagrams (by extending its metamodel) and also to add more details (i.e., attributes) to the concept of features.

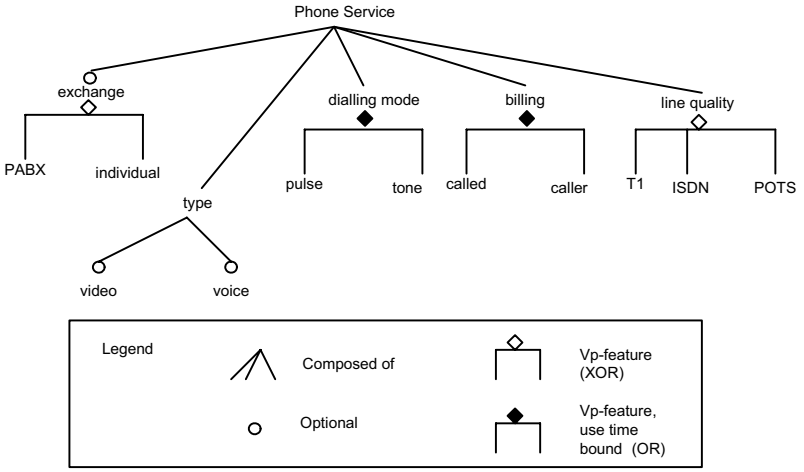

Figure 23: Feature notation used in FeatuRSEB (based on [Griss *et al.* 1998]).

In FeatuRSEB, the notation for features diagrams that is presented in Figure 23 is in reality a kind of *collapsed* view of a stereotyped UML class diagram. Each feature is then represented as a stereotyped class. The attributes presented in the stereotyped classes (i.e., features) correspond essentially to the information presented in the textual featured description used in FODA (see Figure 11). Figure 24 presents an example of this notation.

Variability Mechanisms

In RSEB and FeatuRSEB, variable features are exploited at variation points in a component. Several *variability mechanisms* are proposed to support (i.e., implement) variability. All of them are treated as a form of *generalization*. In particular, RSEB describes the adoption of the following variability mechanisms:

(1) *Inheritance*. Used to create subtypes or subclasses that specialize abstract types of classes at their variation points. A virtual operation can act like a variation point that is specialized with inheritance.

(2) *Extensions*. These are particular small type-like attachments that can be used to express variants (extensions) attached at variation points (extension points) in use case and object components.

(3) *Parameterization*. This is used for types and classes using templates (i.e., generics), frames, and macros.

Other mechanisms referenced in RSEB are, for instance, configuration, module-interconnection languages and generators.

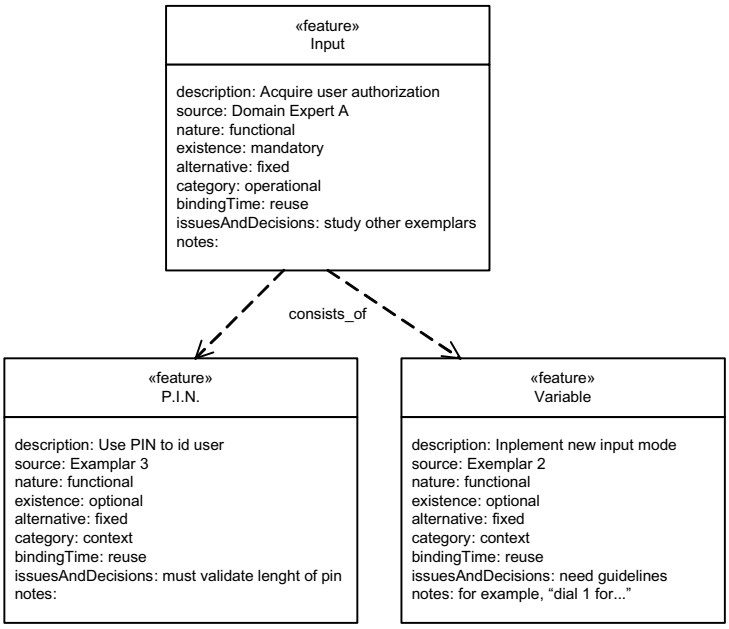

Figure 24: Example of the FeatuRSEB proposal for depicting feature diagrams as stereotyped UML class diagrams (based on [Griss *et al.* 1998]).

All of these mechanisms have specific characteristics that can make them more suitable to support particular variation points. For instance, extensions are well suited for variation points that need to support several variations at the same time. Besides this concern, variability mechanisms may also need to be adapted according to the programming language used. For instance, inheritance is a variability mechanism that is commonly supported by object oriented programming languages. Extensions, however, are not directly supported by the major object oriented programming languages. In this case design patterns can be used to support variability [Gamma *et al.* 1995]. Variability in design patterns is essentially based on inheritance, aggregation (i.e., composition) and *interfaces*. RSEB proposes the *Strategy* pattern to support variation points with only one possible variant. The *Decorator* pattern can be used if several variants can be active at the same time for the same variation point. Examples of applying design patterns to support variability can be found in [Fontoura 1999; Pree *et al.* 2002].

2.8 Product Line UML-Based Software Engineering (PLUS)

The Product Line UML-Based Software Engineering (PLUS) is a software product line development method that is based on UML [Gomaa 2005]. The method is described as being compatible with other object-oriented software development processes, such as the Unified Software Development Process (USDP) [Jacobson *et al.* 1999] and the spiral process model [Boehm 1988b].

This method is based on two major processes: software product line engineering and software application engineering. The software product line engineering process consists of developing a product line use case model, product line analysis model, software product line architecture, and reusable components. In the software application engineering, individual members of the product line are developed. The artifacts resulting from the software product line engineering process are used in the development of the product line members. The application requirements are used to adapt the product line use case model and obtain the application use case model. A similar

approach is used to obtain the application analysis model from the product line analysis model and the architecture of the application from the product line architecture. Reusable components are used to implement the application architecture. Figure 25 presents an overview of the method. The method is composed of two major activities: the software product line engineering and the software application engineering. The software product line engineering activity produces several reusable artifacts that model the product line domain. On of such artifacts is the feature model. In the application engineering activity, an application engineer develops a software application that is a member of the product line by using the feature model to derive the application from the product line architecture and components.

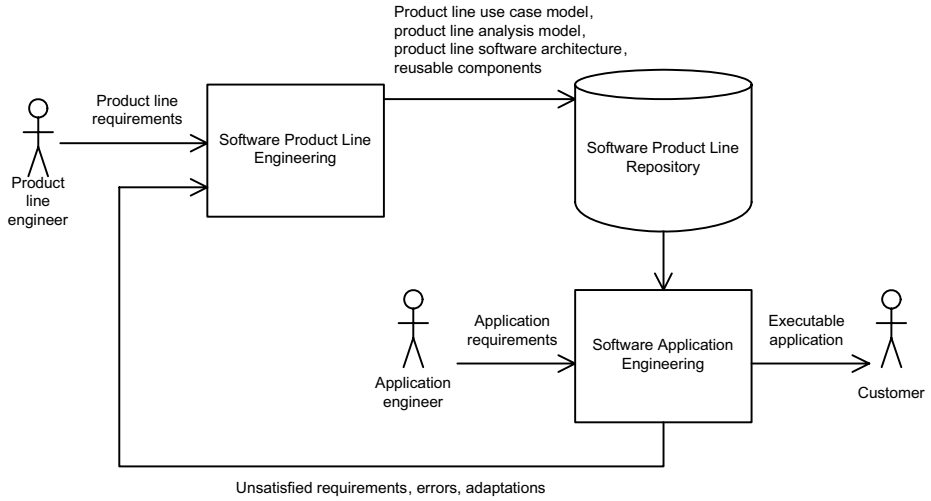

Figure 25: Overview of the PLUS method (based on [Gomaa 2005]).

Since PLUS is based on UML 2.0 it proposes the adoption of the majority of the UML diagrams including: use case diagrams; class diagrams; communication diagrams; sequence diagrams; state machine diagrams; composite structure diagrams; and deployment diagrams. Besides the UML diagrams PLUS also adopts features diagrams. These feature diagrams are based on FODA but use a notation that is based on UML class diagrams.

The method is compliant with the UML 2.0 notation. As such, it has improved support for modeling several aspects of a product line. One of such improvements is the use of *port*, *provided* and *required interface* notation in component models. The method proposes the use of stereotypes and special notation for modeling variability in the UML models. However, similarly to FeatuRSEB, it also proposes the adoption of feature diagrams for modeling variability. Regarding features the method proposes: a specific notation for feature diagrams that is based on UML class diagrams; modeling features using *feature tables*; stereotypes and specific notation for representing features in other UML diagrams; and a feature/class analysis method the determine the relationships between features and classes.

Figure 26 depicts the software product line engineering activity of PLUS. The major sub-activities and modeling tasks are:

- Software product line requirements modeling: use case and feature modeling.
- Software product line analysis modeling: static modeling; dynamic interaction modeling; dynamic state machine modeling; and feature/class dependency modeling.

- Software product line design modeling: software architectural patterns and component-based software design.

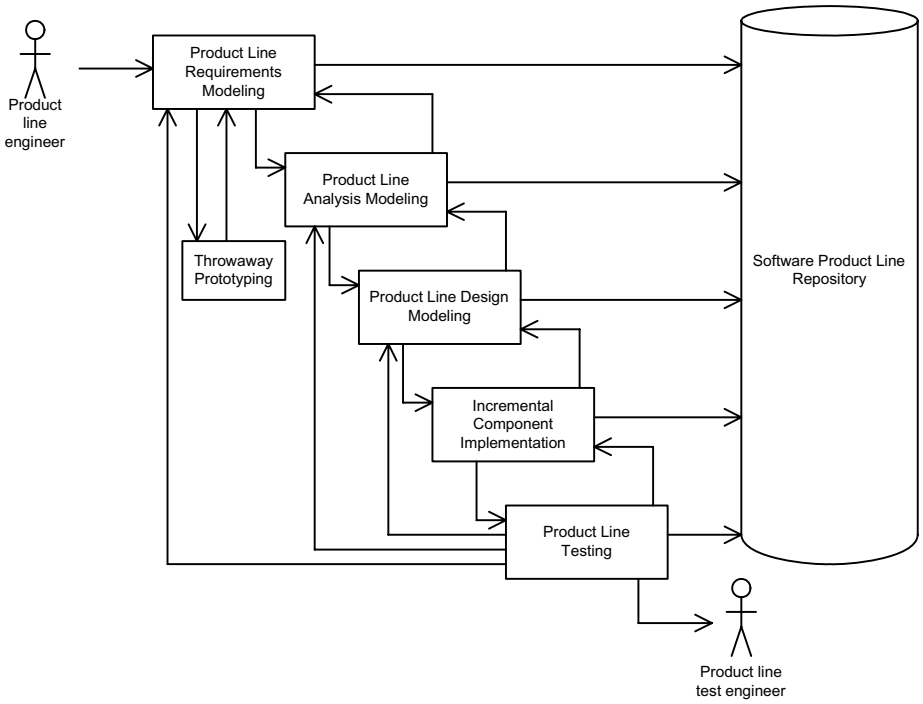

Figure 26: Software product line engineering in PLUS (based on [Gomaa 2005]).

In the analysis phase, several models are used to represent the possible realizations of the use cases. Basically, the method proposes the usage of sequence or communication diagrams to depict how objects collaborate to realize the use cases. State machine diagrams are used to model state-dependent classes and objects. At this phase of development, a product line context model and a product line information model are developed to depict, respectively, the product line boundary and the entity classes. PLUS also proposes a method to determine the dependencies between features and classes.

The software product line design phase is based on architectural patterns for software product lines. The method describes how some well-known patterns can be used in the context of product lines. In the design phase, component diagrams are used to model the architecture of the product line and also for the design of the reusable components.

PLUS proposes support for modeling variability in all of the above mentioned models. A metamodeling approach to PLUS is also discussed in [Gomaa *et al.* 2004].

Regardless of the engineering approach adopted, variation points are used to handle variability in use cases. *Small* variation points can be represented in the textual description of use cases. More complex variability points can be modeled either by the *include* or by the *extend* UML relationships between use cases. Figure 27 presents a use case diagram for an e-commerce product line.

In Figure 27 we have presented an example of how variability is usually modeled in use cases in the PLUS method. Basically, use cases are marked with the following stereotypes: «kernel», «optional» and «alternative». These stereotypes denote if a use case is mandatory, optional or

alternative. The method suggests the use of generalization/specialization relationships to model actor's variability.

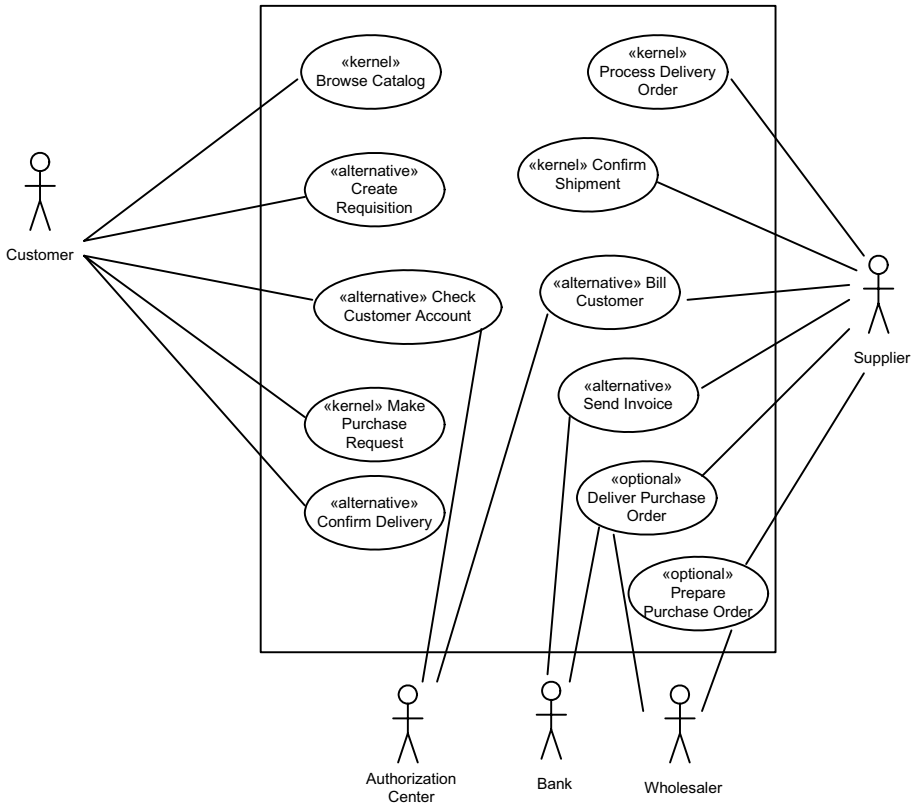

Figure 27: Example of use case modeling in PLUS (based on [Gomaa 2005]).

In PLUS, the description of the behavior of use cases is done textually, following guidelines that are similar to the ones presented in [Cockburn 2001]. In the context of this textual description of use cases, small variations (i.e., variations that occur *within* the behavior of the use case) are specified in a specific section of the textual specification of the use cases. These variations occur at specific points in the use case behavior called *variation points*. This concept of variation point is similar to the one of RSEB, however in PLUS it has a narrow scope since it is only used in use case models.

PLUS also proposes the use of relationships between use cases to model variability. The idea is to model in the relationship a condition of inclusion. If such a condition is true, then the relationship is included otherwise it is not included in the resulting model. These conditions are based on the features from the feature model. Basically, each feature acts like a boolean variable. Its value is true if it is included in a feature configuration or false otherwise. So, the conditions associated with the use case relationships are boolean expressions involving feature variables. A relationship is included in the use case model for a specific product of a product line if the associated condition evaluates to true.

Use Cases: Extend Relationship

Figure 28 presents an example of modeling variability with the *extend* relationship in PLUS. In this example we see how the use case *Check Out Customer* is extended by three extending use cases: *Pay by Cash*, *Pay by Credit Card* and *Pay by Debit Card.*

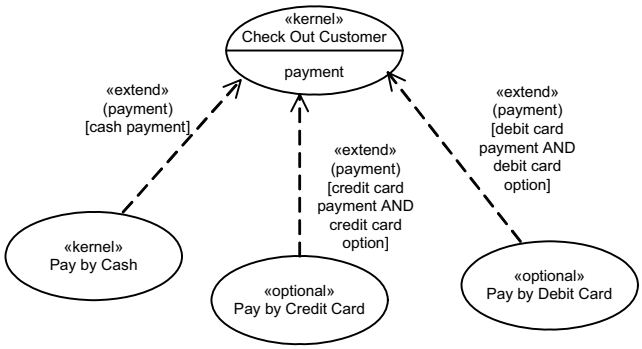

Figure 28: Example of modeling variability with the *extend* relationship in PLUS (based on [Gomaa 2005]).

According to the UML documentation, an *extend* (*Extend* element in UML 2.0 metamodel) is "A relationship from an extending use case to an extended use case that specifies how and when the behavior defined in the extending use case can be inserted into the behavior defined in the extended use case" [UML 2005]. Extended use cases are required to provide extension points, i.e., *points* in the behavior of the use case where that behavior can be extended by the behavior of extending use cases. Also, an *extend* relationship can contain an extension condition. If an *extend* relationship has a condition, then that condition must be true for the extension to take place. These are the base concepts regarding the *extend* relationship. As it is possible to observe from Figure 28, PLUS uses these concepts to model variability with the *extend* relationship. However, PLUS distinguishes between two type of extending conditions: *selection condition* and *product line condition*. The *selection condition* is used to identify which extension is selected during runtime execution of the use case. As such, it corresponds to the semantics of the original *extend* condition. As it name implies the *product line condition* identifies if the behavior specified by a particular extending use case is provided by a particular member of the product line. In PLUS, these to conditions are combined into one by the logical AND operator and specified as the condition of the *extend* relationship. Therefore, PLUS manages to represent variability with the *extend* relationship using the existing characteristics of this relationship as described in the UML documentation [UML 2005].

In the example of Figure 28, all three extensions refer *payment* as the extension location. The *extend* that connects *Pay by Cash* and *Check Out Customer* only provides the selection condition and, as such, *Pay by Cash* is a mandatory or *kernel* use case. In the case of the other two extending use cases a *product line condition* is present. The *Pay by Credit Card* will be present if the *credit card option* feature is selected and the *Pay by Debit Card* if the *debit card option* feature is selected. This is an example of using the *extend* relationship to model use cases that are optional. The *extend* relationship is also suited to model alternative behavior. It is just a matter of specifying the rules for the selection of the features. This can be done in the feature model.

Gomaa states that extension points can be used to model product line variability in the following ways [Gomaa 2005]:

- alternative variability: alternative extension use cases that can be mutually exclusive;
- optional variability: an optional use case is provided only if the product line condition is true;
- future product line evolution: the extension point is used as a placeholder for future extensions to the product line.

Use Cases: Include Relationship
In UML, an Include Relationship links two use cases and its semantics is that the behavior of the included use case is inserted into the behavior of the including use case. The UML documentation also states that "the included use case is not optional, and is always required for the including use case to execute correctly".

In PLUS the Include Relationship can also be used to model variability. The basic idea is similar to the approach used in the *extend* relationship. A *product line condition* is added to the relationship. When configuring a product in the product line, if this condition evaluates to true, the Include is part of the product otherwise it is not.

The described approach has, however, a problem because in the UML metamodel there isn't support for the condition in the Include Relationship. PLUS doesn't explicitly provide an approach to deal with this problem. This could be done, for instance, by using the UML profile mechanism or by extending the UML metamodel. In Chapter 3 and Chapter 4 we discuss this topic and a possible approach to tackle the described problem.

Features
After the determination of the product line use cases, a feature model is developed as a way to represent, in an integrated way, the variability of the product line. Basically, use cases give origin to features and the variation points of use cases, as well as the dependencies between them, are used as source for representing the dependencies between features. This process is also similar to the process used in FeatuRSEB to construct the feature model. As we will see next, the relationship between use cases and features can also be described by using tables.

The feature diagrams of PLUS are *inspired* in the FODA feature diagrams but based on UML class diagrams. Figure 29 presents an example of a feature diagram following the PLUS notation. This diagram regards the use case diagram for an e-commerce product line depicted in Figure 27.

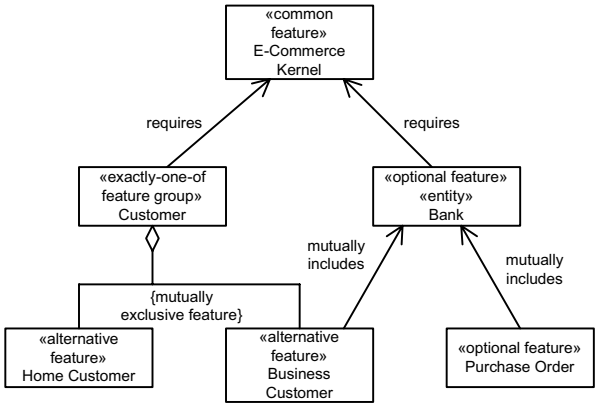

Figure 29: Example of feature diagram in PLUS (based on [Gomaa 2005]).

Basically, in PLUS, a feature corresponds to one or more use cases. For instance, the feature *Home Customer* corresponds to the use cases *Check Customer Account* and *Bill Customer*. On the other end, the feature *Business Customer* corresponds to the use cases *Create Requisition*, *Confirm Delivery* and *Send Invoice*. Since these two features are alternative and since only one of them can be selected in the configuration of a member of the product line they become sub-features of the *Customer* feature group. This feature group is represented by the stereotype *exactly-one-of feature group* that indicates that only one of the features of the group may be selected.

PLUS proposes the following UML stereotypes to denote feature *types*:

- «common feature»: a feature that is required for all members of the product line;
- «optional feature»: a feature that is optional;
- «alternative feature»: a feature that is alternative to some other(s) feature(s).

The following relations between features are proposed:

- *mutually inclusive features*: this relation is used when two features are always used together;
- *requires*: this relation is used to depict that the selection of a feature requires that some other feature needs also to be present in the configuration of the member of the product line.

Regarding constraints on grouping of features, PLUS proposes the following kinds of feature groups that state how the elements of the group can be selected:

- *mutually exclusive features* ("zero-or-one-of feature group");
- "exactly-one-of feature group";
- "at-least-one-of feature group";
- "zero-or-more-of feature group".

There is also the possibility that a feature corresponds to a functionality that is *smaller* than a use case. In this case it can correspond to a *variation point* in a use case or to a use case parameter. The notion of variation point is similar to the one found in RSEB (see Section 2.7). If a feature corresponds to a use case parameter, then the value of the parameter must be defined for the configuration of a member of the product line.

Since there is a close relationship between features and use cases in PLUS (at least for *functional* features), the method also proposes that features be modeled *within* use case diagrams by using *use case packages* to represent, for instance, two or more use cases that correspond to a feature. As discussed earlier, the *extend* and *include* relationships between use cases can also be used to model optional and alternative features.

PLUS follows an approach similar to that of RSEB in order to support the analysis process. The approach is based on the "boundary-control-entity" pattern but adds more specific stereotypes to classify objects. Figure 30 presents the specific stereotypes proposed in PLUS.

The analysis process is also similar to the one used in RSEB. Similarly to the use case development, the analysis process can follow a forward or reverse engineering approach. In both approaches, the analysis is based on the use cases identified previously. Objects that realize each use case are identified following one of the approaches.

Classes that are common to all members of the product line, i.e., participate in the realization of kernel use cases also become kernel classes. Classes that only participate in the realization of non-mandatory use cases become optional classes. Alternative classes can be captured by generalization relationships. The common attributes and operations are captured in the superclass and the differences are captured in the variant subclasses of the superclass.

Figure 31 presents an Entity class diagram for the product line described by the use case diagram of Figure 27.

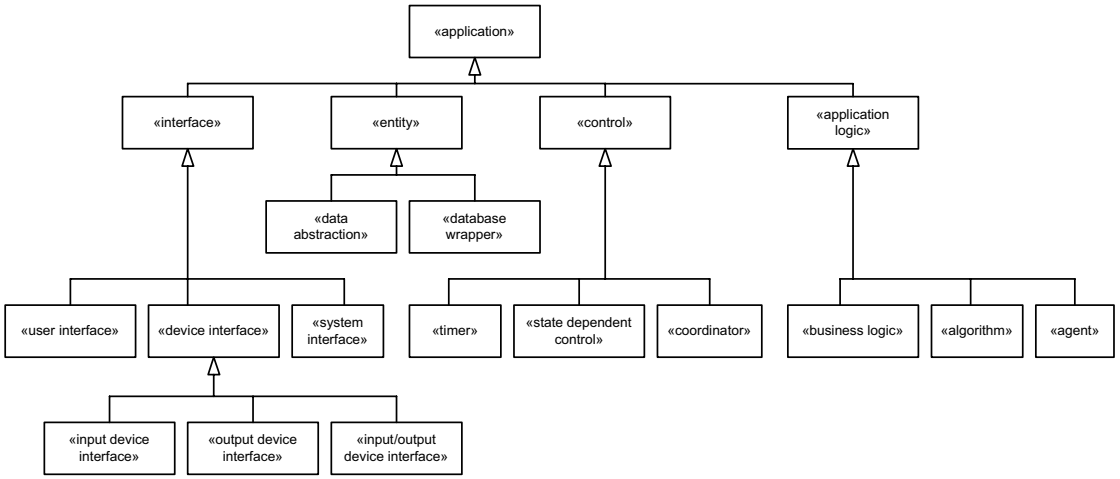

Figure 30: Classification of application objects/classes by stereotype in PLUS (based on [Gomaa 2005]).

To represent the collaborations of objects that realize use cases PLUS proposes the use of communication or sequence diagrams. In these diagrams, the object interactions are represented in time sequence. In communication diagrams the order of an event is usually represented by a number that labels the edge that represents the event. In PLUS a letter can follow the sequence number in order to represent alternative message sequences. Alternative sequences can also be depicted with a condition indicated after the message number. This condition can be based, for instance, on features of the product line.

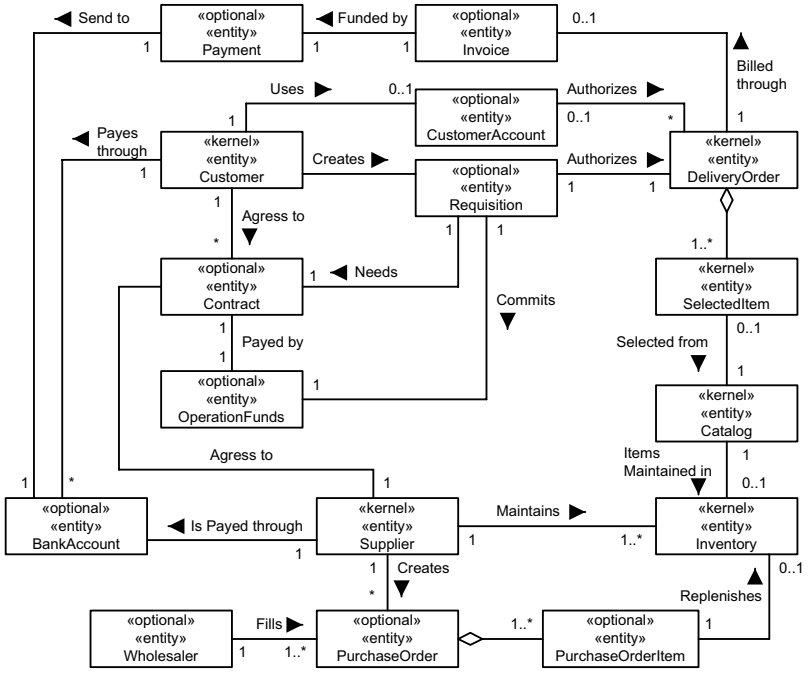

Figure 31: Example of entity class diagram in PLUS (based on [Gomaa 2005]).

The other two types of diagrams that are widely used in PLUS are the composite structure diagram and the finite state machine diagram (statecharts). The former is mainly used to represent

the architectural structure of the product lines while the latter is used to model state-dependent objects and their interactions.

Regarding the architecture of the product line, PLUS describes how several architectural design patterns can be applied. Regardless of the architectural pattern adopted, the main architectural activity is the design of the components, particularly its required and provided interfaces. PLUS describes some guidelines in how to perform this activity based on the other models. The main idea behind these guidelines is the design of "plug-compatible" components as a way to support architectural flexibility and variability.

In statecharts, variability can be supported by inheritance or by using parameterized statecharts with conditions, in a similar way to the communication diagrams. In the case of inheritance, the child state machine inherits the properties of the parent state machine and can modify them by: adding new states; adding new events and transitions; and adding or removing actions and activities. However, the child state machine must not delete states and events defined in its parent.

2.9 Product Line Software Engineering (PuLSE) and Kobra

Product Line Software Engineering (PuLSE) is a method developed at Fraunhofer IESE which purpose is to enable the conception and deployment of software product lines within a large variety of enterprise contexts [Bayer *et al.* 1999]. According to the authors of PuLSE, its main advantage comes from its focus on products, as opposed to domain engineering approaches which focus is on domains. According to its proponents, the focus on domain that traditional domain engineering approaches take imposes that the all domain must be supported. Domains are hard to scope and engineer for an enterprise because a domain captures many extraneous elements that are of no interest to an enterprise. According to the authors of PuLSE, the focus on products provides a more economic and practical approach for enterprises. In fact, products usually comprise multiple domains, but only cover parts of these domains. As such, scoping and manageability of product line approaches can be less demanding than in domain engineering approaches.

Figure 32 presents an overview of PuLSE. The method is divided in three major blocks: deployment phases; technical components; and support components.

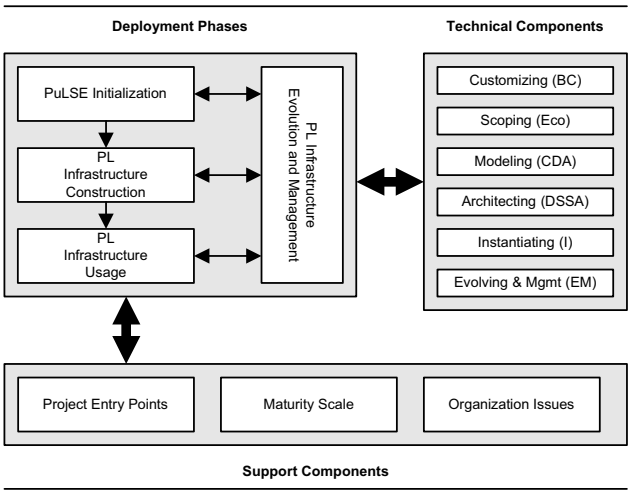

Figure 32: Overview of PuLSE (based on [Bayer *et al.* 1999])

The *deployment phases* are the logical stages that a product line goes through when it is implemented in an organization. The technical components provide the technical know-how needed to operationalize the product line development. They are used throughout the deployment phases. The *support components* are packages of information, or guidelines, which enable a better adaptation, evolution, and deployment of the product line. This separation of *roles* supports the flexibility and customization characteristics of PuLSE. It enables, for instance, partial and incremental implementation of PuLSE in organizations. It also provides, for instance, the adaptation of the technical components to specific tools or design methods already in use in the organization.

PuLSE can be seen as a software product line method template, since it can be customized for specific contexts. This method has also an object-oriented customization named KobrA [Atkinson *et al.* 2000]. KobrA is a customization of the PuLSE method aimed at object-oriented and component based implementations of PuLSE.

PuLSE is a very customizable method for developing software product lines. The customization of the method for the context of an organization is done in the *initialization* stage. After the method customization, the next stage is the *infrastructure construction*. This stage is divided in three parts: scoping (PuLSE-Eco), modeling (PuLSE-CDA) and architecting (PuLSE DSSA). These sub-stages of infrastructure have a direct correspondence with the technical components scoping (Eco), modeling (CDA) and architecting (DSSA). Variability identification is a concern that appears essentially in the scoping and modeling stages.

Here are some examples of information sources used for variability identification in PuLSE: books, standards, papers, users, domain experts and application engineers.

In order to provide a more useful description of the PuLSE method, we will base our analysis of PuLSE variability representation and implementation on KobrA. KobrA is a customization of the PuLSE method aimed at object-oriented and component based implementations of PuLSE. With this approach, not only we describe the generic phases and components of PuLSE, but we also present an example of a customization of PuLSE.

To represent variability, PuLSE uses decision models. A decision model contains a structured set o decisions. Each decision corresponds to a variability point in a workproduct together with the set of possible resolutions. The specification of a product can be obtained by resolving all the decisions of the decision model. For instance, all the variability existent in the generic storyboards should be present in the decision models. If there are other workproducts that model variability, there variability points and variants should also be present in the decision models.

Table 2 presents an example of decision models for a library product line. Table 2a presents an integrated decision model. Each variability has an identification and possible resolutions. The effect of resolutions is presented in the last column of the table. An effect can be a reference to other variabilities in other decision models or a modification in workproducts. For instance, the resolution *yes* for *CR-1* has an effect in the variability *CR1.1* of the decision model for the enterprise process diagram. In Table 2b we can see that a resolution of *no* on *CR1.1* has the effect of removing the process *reserveItem*.

Decision models are less intuitive when used to communicate variability, when compared to feature models. However, they provide a mechanism to *link* between the decisions about features and the other modeling artifacts used. They reflect the effect that feature decisions might have in the other models. Since features can have impact in several artifacts, in PuLSE there is a decision model for every model of the system.

Table 2: Example of PuLSE decision model for a library product line (based on [Bayer *et al.* 2001]).

ID	Variation	Resolution	Effect
CR-1	Reservation	Yes (default)	Yes: CR1.1, CR2.1, CR5.1
		No	No: CR1.1, CR2.1, CR5.1
CR-2	External Database	Yes	Yes: CR1.2, CR2.2, ...
		No (default)	No: LS13.2, LS14.1, ...
CR-3	Suggestion	Yes (default)	Yes: CR1.3, CR2.3, ...
		No	No: CR1.3, CR2.3, ...
CR-4	LibraryAccessCard	Yes	Yes: CR2.4, CR3.1, CR4.1, CR5.2, …
		No	No: CR2.4, CR3.1, CR4.1, CR5.2, …
...	...	...	...

a) Integrated Decision Model for the LibrarySystem Context Realization

ID	Variation	Resolution	Effect
CR1.1	Reservation	Yes (default)	-----
		No	Remove process *reserveItem*
CR1.2	External Database	Yes	-----
		No (default)	Remove process *data exchange* (and its subprocesses)
CR1.3	Suggestion	Yes (default)	-----
		No	Remove process *suggestItem*
...	...	...	...

b) Decision Model for the enterprise process diagram

In the case of KobrA, there are several artifacts that are used for domain analysis. Two of the artifacts that we already mention are the storyboards and the decision models. Other possible models are: Enterprise Model (with the Enterprise Process Diagram and the Enterprise Concept Diagram); Structural Model (with the Context Realization Class Diagram and the Context Realization Object Diagram); the Activity Model (with Activity Diagrams and the Use Case Model); and the Interaction Model (with Sequence Diagrams). The majority of these models, as their name implies, are based on the UML notation [UML 2005]. Next, we will briefly describe how these artifacts can be used.

The product map identifies the major tasks or business processes by domain. That information can be compiled and integrated into the enterprise process diagram. The enterprise process diagram is a tree of domain processes with a top node that represents the whole system. In this diagram, processes that are not mandatory are distinguished from regular processes. Figure 33 presents an example of an enterprise process diagram. Optional processes are depicted in gray.

Usually not all business processes are supported by the system. In KobrA, business processes, or parts of business processes, that are supported by the system can be modeled as use cases. Similarly to the other models, the «variant» stereotype is used to mark items that are not mandatory and, as such, represent variability. In Figure 34, we see that the following process nodes become use cases: *createNewAccount*, *loanItem*, *printAccountInformation*, *identifyAccount* and *identifyItem.* Nodes that are too fine-granular in their functionality are not represented as use cases. An example of this case is *registerLoan*. Figure 34 also shows three use cases that are not supported by the system: *printLibraryAccessCard*, *withdrawLibraryAccessCard* and *handOutInfoMaterial*.

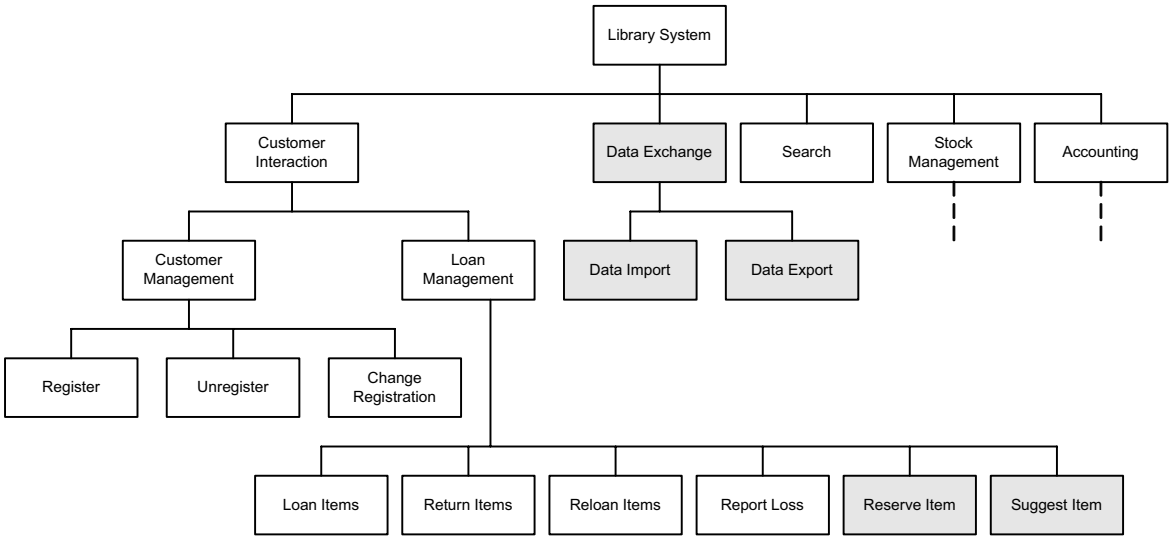

Figure 33: Example of KobrA enterprise process model (based on [Bayer *et al.* 2001]).

Processes, activities and use cases are used in KobrA to model behavior. For the structure of the system other models are used. Entities, roles and their associations are modeled in an enterprise concept diagram. This diagram is at an abstraction level similar to the enterprise process model. These two models provide an initial description of the system and enable the identification of the components that are created in the following phases. They describe the business of the organization and, as such, their construction requires expert domain information that can be obtained, for instance, by consulting domain experts and artifacts containing domain information, such as books and papers. Figure 35 presents an example of an enterprise context diagram.

The structure of the system is refined by using context realization class diagrams. Basically these diagrams are UML class diagrams. They contain components and classes that realize the concepts and processes of the enterprise diagrams that are to be support by the system. Other components and classes that support the realization of the system from a context perspective are also included. Some features (i.e., methods and fields) of the classes are also included in the model. Items of this model that are not mandatory are marked with the «variant» stereotype. The identification of the methods is based on the nodes of the activity diagrams for the business processes and also on the modeled use cases. The attribution of behavior (methods) to classes and components is based on the KobrA interaction model. KobrA interaction models provide a unified view of structure and behavior by using UML sequence diagrams.

All the variability in the different models proposed by KobrA is integrated into the decision models. Each model has its decision model. To obtain the requirements for a particular application of the domain it is necessary to execute all the effects that are consequences of resolving all the variations of the decision models (see Table 2).

An example of the adoption of feature diagrams for modeling variability in PuLSE can be found in an IESE report about a cellular phone product line [Muthig *et al.* 2004]. This report also demonstrates how variability can be represented in the textual description of use cases. In this case, variability is represented in the text of the use cases by using the <*OPT*> and <*ALT*> marks for, respectively, optional and alternative use case steps. The report also shows how a feature diagram can be built based on the variability annotations in the textual description of the use cases.

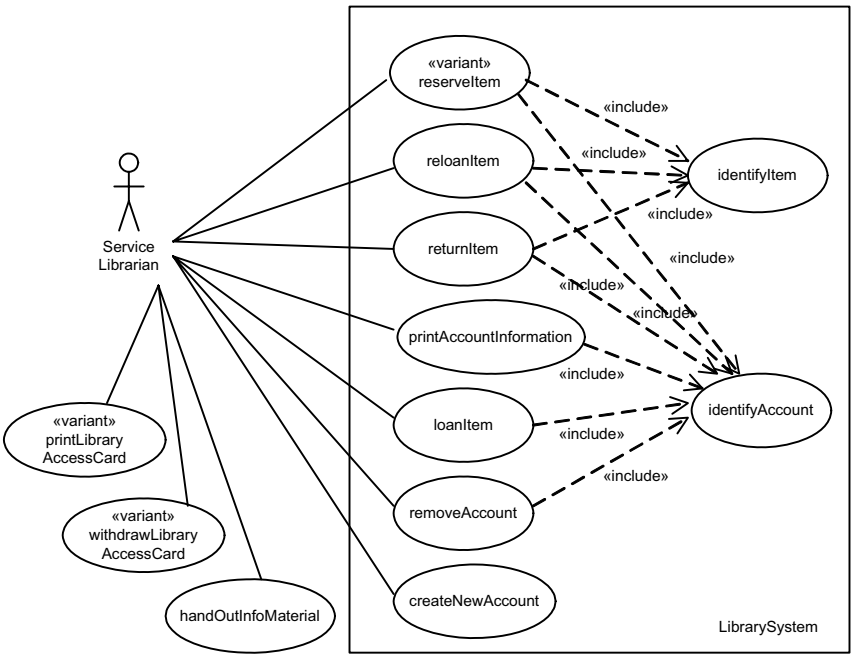

Figure 34: Example of KobrA use case diagram for the actor *ServiceLibrarian* (based on [Bayer *et al.* 2001]).

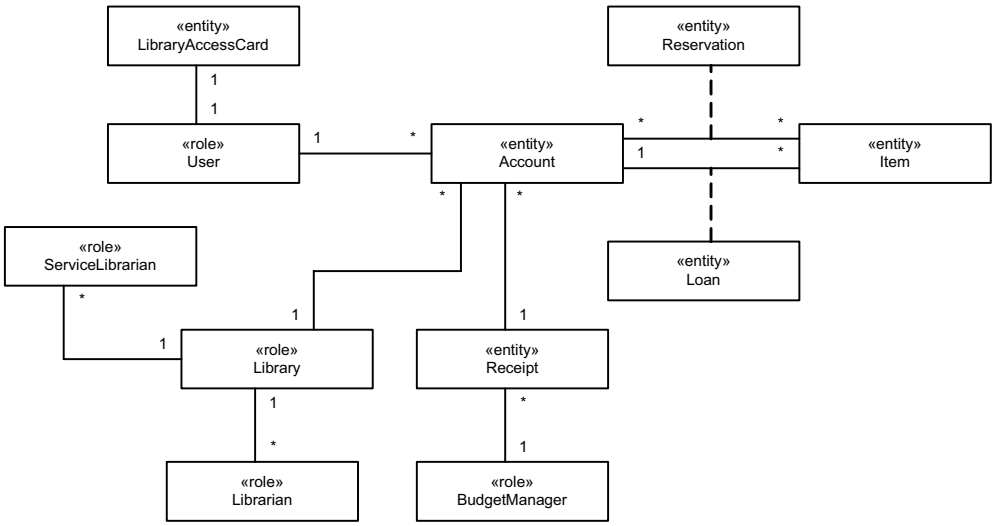

Figure 35: Example of KobrA enterprise concept diagram for a library (based on [Bayer *et al.* 2001]).

Architecting

PuLSE-DSSA is the sub-phase of PuLSE where the domain-specific software architecture is build. The process is incremental and based on scenarios. There are two types of scenarios: *generic* and *property-related*. The generic scenarios represent functional requirements and are derived from the generic storyboards and other artifacts of PuLSE-CDA. The property-related scenarios describe domain-independent quality aspects of the system. They are used to evaluate and rank the generic scenarios of possible candidate architectures. The result of the ranking is a reference architecture that is further developed. During the creation of the reference architecture, implementation-specific

decisions are collected that will have to be resolved during reference instantiation. These decisions and their possible resolutions are captured in a *configuration model* that extends the decision model.

Similarly to what we did for modeling, we will use KobrA as an example of customization of PuLSE to present how architecting can be achieved. We will use the same IESE report [Bayer *et al.* 2001].

A system developed with KobrA is represented as a static set of KobrA components (*Komponents*) organized in the form of a tree. Each component has an external and an internal representation. The external representation describes the visible properties and behavior of the component while the internal representation describes how the component fulfills the external behavior and properties by using lower level components.

The specification of a component (i.e., its external representation) is comprised of four main models: the structural model, the behavioral model, the functional model, and the decision model.

The structural model is based on class diagrams that represent the classes and relationships by which a component interacts with its environment, as well as any internal structure of the component, which is visible at its interface.

The behavioral model describes how a component reacts in response to external stimuli. It consists of an arbitrary number of UML statechart diagrams. The statecharts are used to describe user visible states of a component and state changes that are reactions on user visible events. Events represent requests for the execution of an operation. These operations are the same that appear in the specification class diagram of the component.

The functional model is based on operation schemas, similarly to the ones of Fusion [Coleman *et al.* 1994]. They describe the externally visible effects of the operations that are provided by a component. Each operation listed in the class diagram is described by an operation schema, which defines its effects in terms of input parameters, changed variables, output values, and pre- and post conditions.

Decision models have the same purpose as the ones described earlier.

We will now present a selection of models from the Library case study to illustrate the models used for the specification of the KobrA components.

Figure 36 presents the specification class diagram for the top level component of the library: the *LibrarySystem* component. The majority of the classes presented here come from the context realization class diagram discussed earlier. The methods presented are also based on information that resulted from PuLSE-CDA, particularly from the activity diagrams.

Items that realize variant features are also marked as variant. In the case depicted in Figure 36, the classes *Suggestion*, *Reservation* and *ExternalDatabase* are marked as variant since they support optional features (see Table 2a). The method *reserveItem* is marked as variant for the same reason.

Figure 37 presents another model of the specification of the *LibrarySystem* component: the supplied and required interfaces.

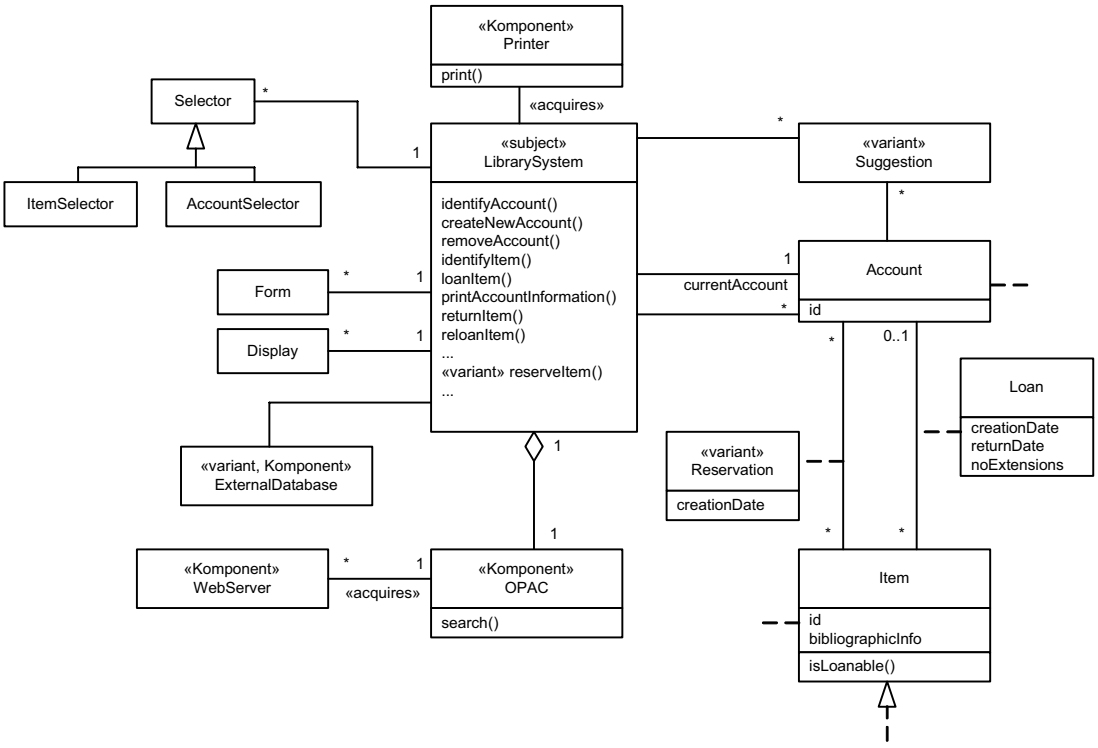

Figure 36: Specification class diagram for the *LibrarySystem* component (based on [Bayer *et al.* 2001]).

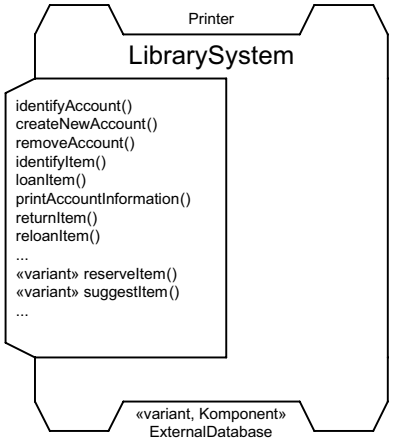

Figure 37: Supplied and required interfaces for the *LibrarySystem* component (based on [Bayer *et al.* 2001]).

In Figure 38 we see the operation schema for *loanItem*. We see the importance of the state diagram of the component, since there are some constraints that are based on the state of the component. For instance, the loan is only possible if the component has an account identified, i.e., is in the *accountIdentified* state. In the case of the operation schemas, variability is represented using tags that surround the variant text (<variant> and </variant>).

The realization of a KobrA component is comprised of four main models: the interaction model, the structural model, the activity model, and the decision model.

Name	*loanItem()*
Description	The loan of an Item to *currentAccount* is registered
Receives	*selector*: *ItemSelector*
Sends	<variant> Message “Reserved” </variant> Message “Already Loaned”
Rules	An *item* is loanable if it is not an *item* that must always stay in the library (e.g., antique books). An *item* is currently loanable if it is loanable and not loaned <variant> or reserved </variant> by another user.
Changes	new Loan
Assumes	Subject is in the state *accountIdentified* Selector selects exactly one Item
Result	*item* selected by *selector* has been obtained if *item* is currently loanable a new *Loan* object, loan, has been created that relates *item* and *currentAccount* and has the attribute values - *creationDate* = *today* - *returnDate* = *today* + <*loanPeriod*> and - *noExtensions* = 0 and, *loan* has been stored. if *item* is not currently loanable one of the messages has been displayed to the user <variant> - “Reserved” or </variant> - "Already Loaned"

Figure 38: Operation schema for *loanItem* (based on [Bayer *et al.* 2001]).

Interaction models define how groups of objects interact at run-time to realize component operations. UML interaction diagrams (either a UML collaboration diagram or a UML sequence diagram) are used to specify operations of the components. The construction of these diagrams is based on the operation schema from the component specification as well as on the activity diagrams that provide a process-oriented view of the realization of the component operations.

The realization structural models describe the classes and relationships from which components are realized. The realization class diagram is a refinement of the corresponding specification class diagram. Elements taken from the specification class diagram are described in more detail and new elements *discovered* in the process are also included.

Decision models are used in the same way as previously. They describe variability and the effects that resolving that variability has on the other models.

In Figure 39 we see the realization class diagram for the *LibrarySystem* component. We depicted in gray the elements that are new regarding the specification class diagram for the *LibrarySystem* component (depicted in Figure 36). These are the components (and eventually classes) that, in this case, were discovered to be necessary for the realization of the component. Each of these new components integrates the tree of components of KobrA. Each of these new components will need also to be specified and realized. The process is repeated until there is no more refinements to be done, i.e., all components are modeled.

In Figure 40 we see how activity diagrams are used for modeling the operations of the components. In this case, we present the activity diagram of *loanItem*, an operation of the *LoanManager* component. Each node in the activity diagram represents an operation. Each swim lane (partition) of the activity diagram represents a component or class that is responsible to perform operations. In KobrA, these activity diagrams are used as intermediate models to bridge

the step from the operation schemata of the specification models of a component to the interaction models used for the realization of components. These are models that are very close to the implementation level. In Figure 40 we see that variability is modeled in the activity diagrams in a similar way to the other models in KobrA, i.e., with the stereotype «variant». We see, for instance, that *ReservationManager* is a variant component. This is because reservation is an optional feature (see Table 2). This means that in *loanItem*, the operation *getItem* that is performed by *ItemManager* can be followed by the *isReserved* operation of the *ReservationManager* or by the *isLoaned* operation of the *LoanStore* class. Although this is a very detailed specification of the *loanItem* method of the *LoanManager* component it is, nevertheless, a platform independent model, which is one step away from implementation details for a specific platform or technology.

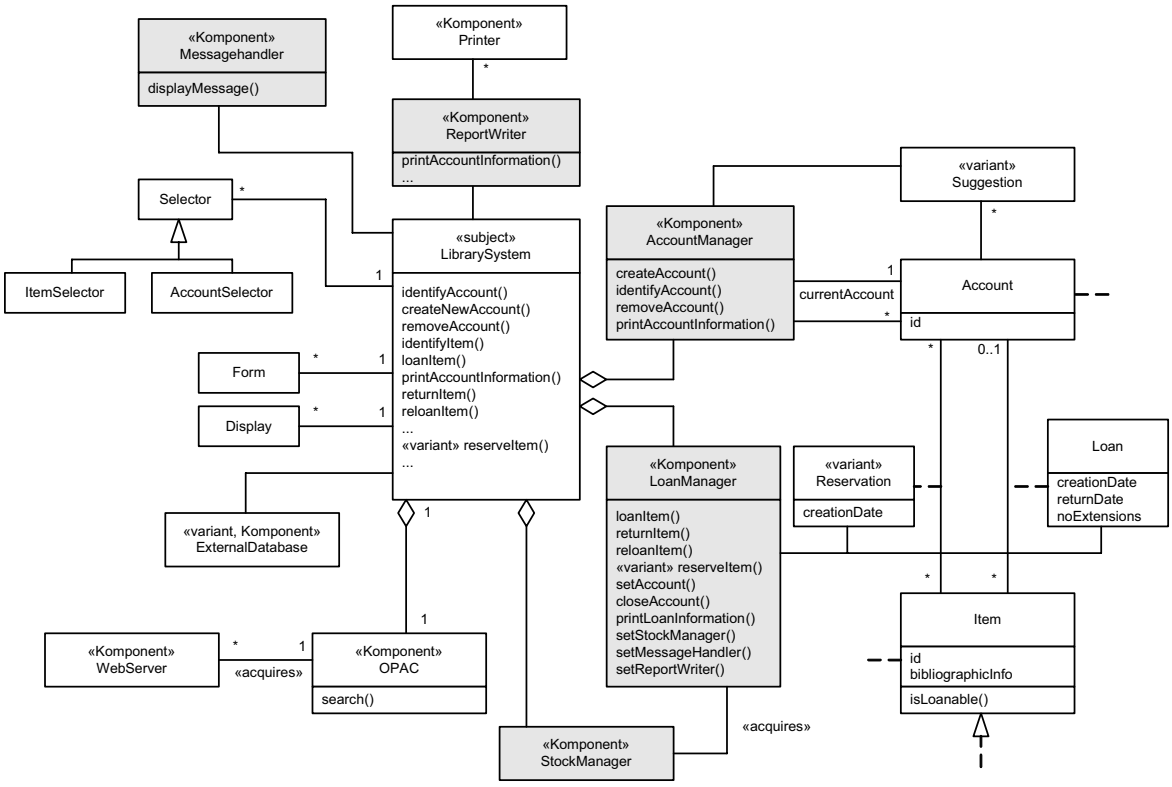

Figure 39: Realization class diagram for the *LibrarySystem* component (based on [Bayer *et al.* 2001]).

For the implementation of variability, PuLSE does not force specific technologies. Technologies must be selected according to the context of the project and the results of PuLSE-DSSA. For instance, design patterns can be used to implement variability. The mediator pattern is proposed as a way to achieve changeability and extensibility and, as such, variability [Muthig *et al.* 2004]. Another possibility is to use aspect-oriented programming [Kiczales *et al.* 1997] to support variant features that crosscut several classes or components. These are only possible approaches to implement variability.

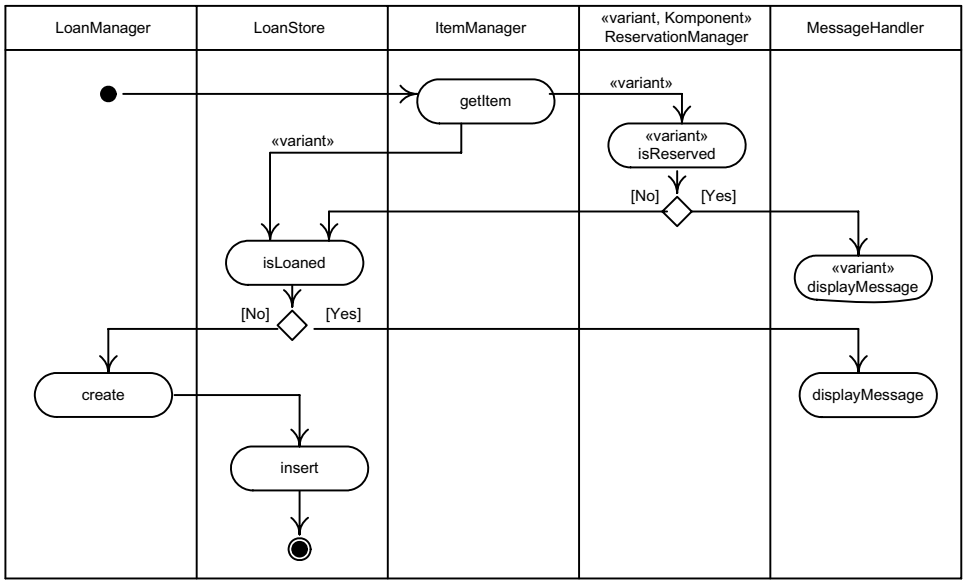

Figure 40: Activity diagram for the *loanItem* operation (based on [Bayer *et al.* 2001]).

2.10 Discussion

This section is concerned with a discussion of what we find to be the major topics on the research field. We start by analyzing the concepts of features, variation points and variants. These concepts relate to the notion of variability.

As we have seen, domain engineering focuses on supporting systematic and large-scale reuse by capturing both the commonalities and the variability of systems within a domain to improve the efficiency of development and maintenance of those systems. As such, the notion of variability is crucial in domain engineering since it supports the differentiation of systems that belong to the same domain. If we search for definitions for *variability* we most probably will find statements like "the state or characteristic of being variable" and synonyms like flexibility, adaptability, and alterability[2]. If we search for the term *variable* we find definitions such as "able to change". Therefore, in the context of domain engineering, variability is used to support the changes or differences that exist between systems that belong to the same domain.

Although commonality ("the sharing of characteristics or qualities"[2]) is what supports domain engineering, since without commonality there is no reuse, it is variability that makes the domain useful. If we are not capable of identify, represent and implement variability in a domain, then domain engineering is no different than single system engineering since all applications share the same characteristics. Therefore, we will focus our comparison of the methods presented in the previous sections on how they identify, represent and implement variability.

[2] Definition taken from MSN Encarta's online dictionary and thesaurus available at http://encarta.msn.com/.

2.10.1 Features

We have seen that a great majority of domain engineering methods uses features and feature diagrams. They are used to represent variability.

In FODA, features and feature diagrams are used to represent characteristics of concepts of the domain. To contextualize the discussion lets use the following definition of concept: “abstract idea or a guiding general principle, e.g. one that determines how a person or culture behaves or how nature, reality, or events are perceived”.

In ODM, a feature is *a difference* observed by modelers among multiple exemplars of a *concept of interest*, *that makes a difference,* i.e., is of interest to some domain stakeholders. From these two statements we see that there is a relationship between concepts and features. We can say that a feature represents a characteristic of a concept that enables the distinction between multiple exemplars of that concept. That is why feature diagrams are used to represent variability for concepts of a domain.

In fact, the human mind seams to use features as a way to distinguish between instances of the same concept. According to the *prototype theory*, concepts are organized around family resemblances, and consist of not defining, but characteristic features, which are weighted in the definition of the prototype [Aerts *et al.* 2005]. This has to do with the fact that people do not store all the information about the *objects* they encounter [Smith *et al.* 1981]. People manage to recognize new *objects* as instances of concepts they already know. *Objects* are recognized as instances of concepts by means of categorization. This categorization and the distinguishing between *objects* are achieved through features.

In Figure 41 we schematically represent the relationships between how the human mind *organizes* concepts and how these relate to elements of the mainstream software engineering paradigm.

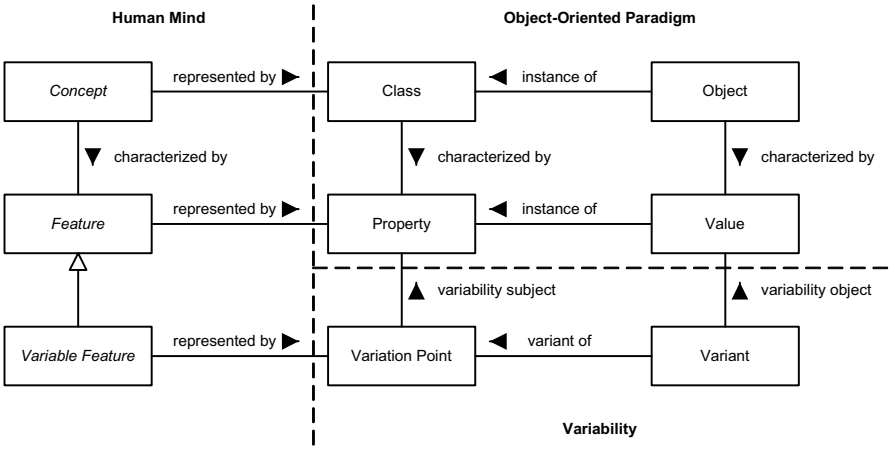

Figure 41: Relating variability, human mind and object-oriented terms.

We would like to present some notes regarding Figure 41. Concepts can be categorized according to its features in a similar manner to what we do with classes and its properties. So, why not use only classes and properties to model concepts and its characteristics? The answer is that

classes in the object-oriented paradigm describe sets of objects while, as we have seen in the beginning of this section, a concept is a much broader *concept*. A more precise mapping could be done using UML 2.0 elements. In this case, *Concept* could be represented by *Classifier*. In UML 2.0, *Classifier* is a more generic term than *Class* and it can be refined, for instance, to *UseCase* or *Actor* (see [UML 2005]). Even so, we are limited to *things* that can be modeled in UML 2.0. That is the reason why class diagrams and feature diagrams have different *natures*. However, if the reader is interested in the topic, deeper analysis concerning the relationship between features and classes can be found in [Czarnecki *et al.* 2006; Kim 2006].

2.10.2 Variation Points and Variants

The domain engineering methods that have been previously described have two other common terms that were used regarding variability: variation point and variant.

In RSEB, the concept of variation point is used to represent variability in the UML models. It identifies one or more locations at which variation will occur. These locations are represented using a dot inside the model element where the variation takes place. In RSEB, variable features are exploited at variation points.

In PLUS, variation points are used to handle variability within use cases. They are called *small* variation points and are represented in the textual description of use cases.

PuLSE adopts a more general approach to variation points. Here, variation points can occur in any workproduct of the method. They are used in decision models to model variability and its effect on the workproducts. A decision model contains a structured set o decisions. Each decision corresponds to a variability point in a workproduct together with the set of possible resolutions. The specification of a product can then be obtained by resolving all the decisions of the decision model.

A general approach to variation points and variants can be found in [Pohl *et al.* 2005]. In this work, variation points are based on the following two definitions: (1) variability subject: is a variable item of the real world or a variable property of such an item; (2) variability object: is a particular instance of a variability subject. Based on the previous definitions, variation point and variant are defined as follows: (1) variation point: is a representation of a variability subject within domain artifacts enriched by contextual information; (2) variant: is a representation of a variability object within domain artifacts. These definitions are used to support an orthogonal variability model that can complement any other model with the variability perspective.

In Figure 41 we see how variation point and variant can be related to the previously discussed concepts.

2.10.3 Method Comparison

We now present an analysis of the previously described domain engineering methods with respect to three different aspects: variability identification; variability representation and variability implementation. Table 3 presents a summary of our analysis.

Variability Identification

Domain experts, end-users, documents and applications are the main sources of information for variability identification. There are techniques used by some methods to help in the task of variability identification. For instance, FODA uses data flows for first variability identification in the scoping phase. DARE tries to automate the process of variability identification by adopting a

bottom-up approach that is based on a faceted conceptual clustering technique that uses documents and code as sources. RSEB and PLUS are UML-based methods and naturally adopt a use case driven approach to identify variability. In the case of PuLSE, since it is a *method framework* for product line development, it does not force particular techniques prior to the customization of the method. However it suggests the use of product maps and also storyboards (IESE has developed tool support for storyboards). In the KobrA customization of PuLSE, use cases are also used because KobrA is UML based.

Table 3: Domain engineering methods comparison.

	Variability Identification	Variability Representation	Variability Implementation
Draco	Documents; applications in the domain; and other domains in Draco.	Variability is defined by the grammar rules of the domain-specific language of the domain.	Components and domain-specific languages.
FODA	Domain experts; end-users; documents; and applications.	Variability mainly represented using feature diagrams.	Several implementation techniques suggested based on the binding time of features
ODM	Social and organizational sources of domain information.	Uses the concept of feature but does not prescribe any specific notation	Does not prescribe specific techniques for implementation.
DARE	Top-down based on domain experts and bottom-up Faceted Conceptual Clustering approach based documents and code.	Uses the concept of *domain book* and *feature tables* to represent the results of domain analysis.	Does not prescribe specific implementation techniques.
FAST	interviews with domain experts	A *commonality analysis document* contains sections that represent variability.	Based on domain specific languages.
RSEB and FeatuRSEB	Use case driven process of collecting domain information from users and customers.	Uses the concepts of variation point, variation and feature diagrams.	Based on the concepts of inheritance, extension and parameterization. Also proposes the adoption of design patterns.
PLUS	Use case driven process based essentially on user input.	Features; variation points; and UML stereotypes.	Architectural patterns; component-Based design; inheritance; and class parameterization.
PuLSE and KobrA	Information is elicited and collected from several sources in product maps and storyboards.	Uses storyboards and decision models. Other models can be used according to PuLSE customization.	KobrA proposes techniques such as AOP, component technology or design patterns.

From the analysis of the methods we can say that variability identification is mainly a manual task. Only DARE tries to automate the process with a faceted conceptual clustering technique. However, to our knowledge, the results of such approach are not conclusive [Prieto-Díaz *et al.* 1995]. The UML related methods support their variability identification tasks on use cases or similar techniques like storyboards.

Variability Representation

Variability representation is based on the concepts of feature, variation point and variant. Draco is the only method that does not explicitly use these concepts. This maybe a consequence of being

based on domain specific languages and, as such, variability is implicitly model in the grammar of the languages. FODA proposes features and feature diagrams not only to model variability, but also to parameterize other models like entity relationship and activity and state charts. DARE and FAST represent variability in documents that result from analysis: *domain book* in the case of DARE and *commonality analysis document* in the case of FAST. DARE uses feature tables, whereas FAST describes variabilities and parameters of variability in its analysis document. RSEB originally based its representation of variability in variant points and variants. In the FeatuRSEB extension to RSEB, features are included into the method to represent an integrated view on variability and as a guide to the reuse process. The PLUS method is based on UML. Therefore, it proposes to model variability with UML models. To represent variability, models are annotated with variability stereotypes. It also uses the concept of variation point, particularly to represent *small* variations in use cases and also variability in classes (class diagrams). To model features, it proposes the use of stereotyped class diagrams. In PuLSE, storyboards and decision models are used to represent variability. Other models can be used according to PuLSE customization. Decision models represent the effect that resolution of variability has on artifacts. It complements the variability that is modeled using, essentially, annotations and tags in other modeling languages such as UML.

Besides the variability representation approaches used in the discussed methods, other complementary approaches can be found in the literature. Here we reference significant proposals that are related to UML [Clauß 2001a; Clauß 2001b; Maßen *et al.* 2002; Pree *et al.* 2002; Philippow *et al.* 2003; Ziadi *et al.* 2003; Fantechi *et al.* 2004]

Variability Implementation
Regarding variability implementation, the majority of the described methods are not prescriptive. Exceptions are Draco and FAST, which propose the use of domain specific languages to implement variability. RSEB and PLUS, as a result of being based on UML, propose that variability can the implemented by inheritance and parameterization. They also propose the use of design patterns, particularly when the adopted programming languages have limited support for variability. PuLSE also does not prescribe specific implementation techniques. However, it proposes techniques such as aspect-oriented programming (AOP), component technology or design patterns to implement the modeled variability. Regarding the KobrA customization of PuLSE, we have seen that it uses UML models to specify system realizations. These models can be easily used to select the implementation techniques more suitable for each case.

As we have seen in Section 2.1 and also from the analysis of the domain engineering methods, variability realization techniques are essentially based on: inheritance; extensions and extension points; parameterization, templates and macros; configuration and module interconnected languages; and generation of derived components. However, other more specific techniques can be used like, for instance, aspect-oriented programming, frame technology, and feature-oriented programming. A more detailed analysis of some specific variability implementation techniques can be found in [Braganca 2003].

2.11 Conclusion

This chapter was dedicated to present and discuss related work in the research field of domain engineering. We start by contextualizing domain engineering, particularly with regards to software engineering, software reuse, and software product lines. We saw how domain engineering relates to single system development (i.e., application engineering). We presented commonality and variability as major concepts used in domain engineering. We have also briefly presented common variability implementation techniques as well as other innovative ways to support variability.

Since this thesis is essentially about methods to develop software in the context of product lines, we have presented and discussed a group of selected methods. We selected the methods based on their historical relevance and also on the degree of relationship with our own work, particularly in the case of FODA, RSEB, PLUS and PuLSE. Each of the methods were presented and discussed with a particular focus on how they identify, represent and implement variability. These characteristics of the methods were compiled into a comparison table that summarizes how each method supports variability identification, representation and implementation. We have also discussed and related major concepts used in domain engineering particularly, features, variation points and variants.

From the description and discussion of the methods presented in this chapter, and in the spirit that model driven approaches treat models as first-class artifacts of the software development process, we can conclude that we are still far from a model driven development paradigm for software product lines. We particularly identify the following major issues:

- the lack of *formalization* of relationships and transformations between models;
- the lack of a clear conceptualization of features that supports the definition of possible relationships between features and other modeling concepts;
- the lack of a global approach for modeling variability (and commonality) in the context of UML, either by the profile mechanism or by extending the metamodel;
- the lack of a clear approach to transformations between analysis and design models that could support the automation of the *bridging* between the *problem space* and the *solution space*;
- the lack of explicit support for multi-stage software development.

The presented issues are also research opportunities that we will attack in this thesis and discuss in the following chapters.

To provide a domain engineering state-of-the-art is an overwhelming task. Having said that, we hope this chapter provides enough knowledge background to facilitate the reading of the next chapters.

2.12 References

[Aerts *et al.* 2005] Aerts, D. and L. M. Gabora, "A Theory of Concepts and Their Combinations I: The Structure of the Sets of Contexts and Properties," *Kybernetes*, vol. 34, pp. 167-191, 2005.

[Anastasopoulos *et al.* 2000] Anastasopoulos, M., J. Bayer, O. Flege and C. Gacek, "A Process for Product Line Architecture Creation and Evaluation - PuLSE-DSSA - Version 2.0," IESE 038.00/E, 2000.

[Atkinson *et al.* 2000] Atkinson, C., J. Bayer and D. Muthig, "Component-Based Product Line Development: The KobrA Approach," First Software Product Line Conference, Denver, Colorado, 2000.

[Bassett 1997] Bassett, P. G., *Framing Software Reuse: Lessons From the Real World*: Prentice Hall, 1997.

[Batory 2004] Batory, D., "Feature-Oriented Programming and the AHEAD Tool Suite," International Conference on Software Engineering, Edinburgh, Scotland, UK, 2004.

[Batory *et al.* 1992] Batory, D. and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Transactions on Software Engineering and Methodology*, vol. 1, pp. 355-398, 1992.

[Bayer *et al.* 1999] Bayer, J., O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen and J.-M. DeBaud, "PuLSE: A Methodology to Develop Software Product Lines," Symposium on Software Reusability '99 (SSR'99), Los Angeles, 1999.

[Bayer *et al.* 2001] Bayer, J., D. Muthig and B. Gopfert, "The Library Systems Product Line - A KobrA Case Study," IESE 024.01/E, 2001.

[Beck 1999] Beck, K., *Extreme Programming Explained*: Addison-Wesley, 1999.

[Beck *et al.* 2001] "Manifesto for Agile Software Development," Available at http://agilemanifesto.org/, 2007.

[Boehm 1988a] Boehm, B., "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, vol. 21, pp. 61-72, 1988a.

[Boehm 1988b] Boehm, B. W., "A Spiral Model of Software Development and Enhancement," *Computer*, vol. 21, pp. 61-72, 1988b.

[Borgida *et al.* 1984] Borgida, A., J. Mylopoulos and H. K. T. Wong, "Generalization/Specialization as a Basis for Software Specifications," in *On Conceptual Modeling, Book resulting from the Interleave Workshop 1982*. New York: Springer-Verlag, 1984, pp. 87-117.

[Bosch 2000] Bosch, J., *Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach*: Addison-Wesley, 2000.

[Bosch 2002] Bosch, J., "Maturity and Evolution in Software Product Lines: Approaches, Artifacts and Organization," Second Software Product Line Conference (SPLC2), 2002.

[Braganca 2003] Braganca, A., "Run-Time Variability in Domain Engineering for Post-Deployment of User-Centric Software Functional Completion," U. Minho, Guimarães, PhD Report 2003.

[Chen 1976] Chen, P. P., "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, pp. 9-36, 1976.

[Clauß 2001a] Clauß, M., "Generic Modeling using UML extensions for variability," DSVL 2001, 2001a.

[Clauß 2001b] Clauß, M., "Modeling variability with UML," GCSE - YRW01, 2001b.

[Clements *et al.* 2002] Clements, P. and L. Northrop, *Software Product Lines - Practices and Patterns*: Addison Wesley, 2002.

[Cockburn 2001] Cockburn, A., *Writing Effective Use Cases*: Addison-Wesley, 2001.

[Coleman *et al.* 1994] Coleman, D., P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes and P. Jeremaes, *Object Oriented Development: The Fusion Method*: Prentice Hall International, 1994.

[Czarnecki 1998] Czarnecki, K., "Generative Programming Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models," in *Department of Computer Science and Automation*: Technical University of Ilmenau, 1998.

[Czarnecki *et al.* 2006] Czarnecki, K., C. H. P. Kim and K. T. Kalleberg, "Features are Views on Ontologies," SPLC 2006, Baltimore, Maryland, 2006.

[DARPA 1994] DARPA, "Software Technology For Adaptable, Reliable Systems (STARS). Army STARS Demonstration Project Experience Report.," STARS-VC-A011R/002/01, 1994.

[Dijkstra 1969] Dijkstra, E. W., "Notes on Structured Programming," Technological University of Eindhoven, Eindhoven, The Netherlands 1969.

[Fantechi *et al.* 2004] Fantechi, A., S. Gnesi, G. Lami and E. Nesti, "A Methodology for the Derivation and Verification of Use Cases for Product Lines," SPLC2004, Boston, 2004.

[Fayad *et al.* 1999] Fayad, M. E. and R. E. Johnson, *Domain-Specific Application Frameworks: Framework Experience by Industry*: John Wiley & Sons, 1999.

[Fontoura 1999] Fontoura, M., "A Systematic Approach to Framework Development," in *Computer Science Department*. Rio de Janeiro: Pontifical Catholic University, 1999.

[Foreman 1996] Foreman, J., "Product Line Based Software Development- Significant Results, Future Challenges," Software Technology Conference, Salt Lake City, 1996.

[Fowler 2002] Fowler, M., *Patterns of Enterprise Application Architecture*: Addison-Wesley Pub Co., 2002.

[Gamma *et al.* 1995] Gamma, E., R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[Garlan *et al.* 1994] Garlan, D. and M. Shaw, "An Introduction to Software Architecture," Carnegie Mellon University CMU-CS-94-166, 1994.

[Gomaa 1984] Gomaa, H., "A Software Design Method for Real-Time Systems," *Communications of the ACM*, vol. 27, pp. 938-949, 1984.

[Gomaa 2005] Gomaa, H., *Designing Software Product Lines with UML*: Addison Wesley, 2005.

[Gomaa *et al.* 2004] Gomaa, H. and M. E. Shin, "A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines," ICSR International Conference on Software Reuse, Madrid, 2004.

[Greenfield *et al.* 2004] Greenfield, J., K. Short, S. Cook and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*: Wiley, 2004.

[Griss *et al.* 1998] Griss, M. L., J. Favaro and M. d'Alessandro, "Integrating Feature Modeling with the RSEB," Fifth International Conference on Software Reuse, Victoria, Canada, 1998.

[Gurp 2003] Gurp, J. v., "On the Design & Preservation of Software Systems," in *Computer Science Department*. Groningen: University of Groningen, 2003.

[Hayes-Roth 1994] Hayes-Roth, F., "Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program," Teknowledge Federal Systems Informal Technical Report, 1994.

[Jacobson *et al.* 1999] Jacobson, I., G. Booch and J. Rumbaugh, *The Unified Software Development Process*: Addison-Wesley Professional, 1999.

[Jacobson *et al.* 1992] Jacobson, I., M. Christerson, P. Jonsson and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*: Addison-Wesley, 1992.

[Jacobson *et al.* 1997] Jacobson, I., M. Griss and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*: Addison Wesley Longman, 1997.

[Johnson 1995] Johnson, J., "CHAOS: The dollar drain of IT project failures," *Application Development Trends*, pp. 41-47, 1995.

[Kang *et al.* 1990] Kang, K. C., S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study Technical Report," Software Engineering Institute, Carnegie Mellon University CMU/SEI-90-TR-21, 1990.

[Kang *et al.* 1998] Kang, K. C., S. Kim, J. Lee, K. Kim, G. J. Kim and E. Shin, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures," *Annals of Software Engineering*, vol. 5, pp. 143-168, 1998.

[Kiczales *et al.* 1997] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin, "Aspect-Oriented Programming," European Conference on Object-Oriented Programming (ECOOP), Finland, 1997.

[Kim 2006] Kim, P., "On the Relationship between Feature Models and Ontologies," 2006.

[Maßen *et al.* 2002] Maßen, T. v. d. and H. Lichter, "Modeling Variability by UML Use Case Diagrams," REPL'02, Essen, Germany, 2002.

[McLeod 1978] McLeod, D., "A Semantic Data Base Model and its Associated Structured User Interface," Massachusetts Institute of Technology, 1978.

[Medvidovic 1997] Medvidovic, N., "A Classification and Comparison Framework for Software Architecture Description Languages," University of California, Irvine UCI-ICS-97-02, 1997.

[Muthig *et al.* 2004] Muthig, D., I. John, M. Anastasopoulos, T. Forster, J. Dorr and K. Schmid, "GoPhone - A Software Product Line in the Mobile Phone Domain," IESE 025.04/E, 2004.

[Myers 1988] Myers, B. A., "A Taxonomy of Window Manager User Interfaces," *IEEE Transactions on Computer Graphics & Applications*, vol. 8, pp. 65-84, 1988.

[Nakatani *et al.* 1999] Nakatani, L. H., M. A. Ardis, R. G. Olsen and P. M. Pontrelli, "Jargons for domain engineering," DSL-99, Austin, Texas, 1999.

[Naur *et al.* 1969] Naur, P. and B. Randell, "Software Engineering," Garmisch, Germany Report on a conference sponsored by theNATO Science Commite, 1969.

[Neighbors 1980] Neighbors, J., "Software Construction Using Components," in *Department Information and Computer Science*. Irvine: University of California, 1980.

[Ossher *et al.* 1994] Ossher, H., W. Harrison, F. Budinsky and I. Simmonds, "Subject-Oriented Programming: Supporting Decentralized Development of Objects," 7th IBM Conference on Object-Oriented Technology, 1994.

[Parnas 1976] Parnas, D., "On the Design and Development of Program Families," vol. 2, pp. 1-9, 1976.

[Peterson *et al.* 1994] Peterson, A. S. and J. Jay L. Stanley, "Mapping a Domain Model and Architecture to a Generic Design," Carnegie Mellon University/Software Engineering Institute CMU/SEI-94-TR-8, 1994.

[Philippow *et al.* 2003] Philippow, I., M. Riebisch and K. Boellert, "The Hyper/UML Approach for Feature Based Software Design," The 4th AOSD Modeling With UML Workshop, 2003.

[Pohl *et al.* 2005] Pohl, K., G. Böckle and F. v. d. Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*: Springer-Verlag, 2005.

[Pree *et al.* 2002] Pree, W., M. Fontoura and B. Rumpe, "Product Line Annotations with UML-F," Software Product Lines - Second International Conference, SPLC 2, San Diego, 2002.

[Pressman 2004] Pressman, R. S., *Software Engineering: A Practitioner's Approach*, 6 ed: McGraw Hill, 2004.

[Prieto-Diaz 1990] Prieto-Diaz, R., "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, vol. 15, pp. 47-54, 1990.

[Prieto-Diaz 1991] Prieto-Diaz, R., "Implementing Faceted Classification for Software Reuse," *Communications of the ACM*, vol. 34, pp. 89-97, 1991.

[Prieto-Díaz *et al.* 1995] Prieto-Díaz, R., B. Frakes and C. Fox, "DARE A Domain Analysis and Reuse Environment," Reuse, Inc., Fairfax, VA 1995.

[Reenskaug *et al.* 1996] Reenskaug, T., P. Wold and O. A. Lehne, *Working with Objects: The OOram Software Engineering Method*: Manning, 1996.

[Royce 1970] Royce, W., "Managing the Development of Large Software Systems," IEEE WESCON, 1970.

[RUP 2004] IBM, "Rational Unified Process web site at IBM," Available at http://www-3.ibm.com/software/awdtools/rup/, 2004.

[SEI 2007a] Software Engineering Institute, "Domain Engineering: A Model-Based Approach," Available at http://www.sei.cmu.edu/domain-engineering/domain_engineering.html, 2007.

[SEI 2007b] SEI, "Software Engineering Institute web site," Available at http://www.sei.cmu.edu/, 2007.

[Simonyi 1995] Simonyi, C., "The death of Computer languages, the birth of intentional programming," Microsoft Research, Technical Report MSR-TR-95-52, 1995.

[Simos *et al.* 1996] Simos, M., D. Creps, C. Klinger, L. Levine and D. Allemang, "Organization Domain Modeling (ODM) Guidebook, Version 2.0," Informal Technical Report for STARS STARS-VC-A025/001/00, 1996.

[Smith *et al.* 1981] Smith, E. E. and D. L. Medin, *Categories and concepts*. Cambridge, Massachusetts: Harvard University Press, 1981.

[Svahnberg *et al.* 2000] Svahnberg, M. and J. Bosch, "Issues Concerning Variability in Software Product Line," Third International Workshop on Software Architectures for Product Families, 2000.

[UML 2005] OMG, "Unified Modeling Language Version 2.0: Superstructure (formal/05-07-04)," Available at http://www.omg.org, 2005.

[Unisys 1993] Unisys, "STARS Conceptual Framework for Reuse Processes (CFRP), Vol. I: Definition, Version 3.0.," Unisys, Arlington, VA STARS-VC-A018/001/00, 1993.

[Weiss 1998] Weiss, D. M., "Commonality Analysis: A Systematic Process for Defining Families," Second International Workshop on Development and Evolution of Software Architectures for Product Families, 1998.

[wxWindows 2007] "wxWindows project web site," Available at http://www.wxwindows.org/, 2007.

[Yourdon *et al.* 1978] Yourdon, E. and L. Constantine, *Structured Design*: Yourdon Press, 1978.

[Ziadi *et al.* 2003] Ziadi, T., L. Hélouët and J.-M. Jezequel., "Towards a uml profile for software product lines," Fifth Internationl Workshop on Product Familly Engineering (PFE-5), 2003.

3. The MoDeLine Method

"Ironically, most of the processes in other industries are automated by software systems"
Jack Greenfield, in "Software Factories"

This chapter is concerned with methods for model driven development of software product lines. It covers the analysis and design phases in a transversal way, i.e., it is not specific to any method activity. The first half of the chapter describes an approach to extend an existing model driven method, called 4SRS (4-Step Rule Set), to explicitly handle variability. In the second half of this chapter, we present a proposal to extend 4SRS so that it can be used as the basis for a model driven method for the development of software product lines. This evolution of 4SRS method was called MoDeLine (Model Driven Development of Software Product Lines). We particularly discuss how MoDeLine supports the transformation of analysis models into architectural models. We also discuss some approaches to detail the first logical architecture of a system by integrating design patterns in the proposed approach.

3.1 Introduction

One of the most important artifacts of a product line is the product line architecture. In Section 3.2 we present an approach for adapting a model driven method called 4SRS (4-Step Rule Set) [Machado *et al.* 2005] so that it addresses commonality and variability explicitly. 4SRS started as a model driven transformational technique and evolved into a method. 4SRS has been developed and applied to obtain system architectures for single systems (as opposed to product lines) from requirements specified as UML use cases. The method is based on UML v1.4 [UML1.4 2001]. The resulting software architecture is modeled using object diagrams in which objects represent system level components. We experiment how such a technique can be applied to product lines and what adaptations are required to achieve that goal. For demonstration purposes we use the public available IESE report of the GoPhone product line [Muthig *et al.* 2004] that uses the UML modeling language.

The alignment of the software architecture and the functional requirements of a system is a demanding task because of the difficulty in tracing design elements to requirements. The 4SRS model driven method provides support to the software architect in this task. In the second half of this chapter we propose an approach to evolve 4SRS into the basis of a model driven method for the development of software product lines that we call MoDeLine (Model Driven Development of Software Product Lines). For that, we show how to evolve 4SRS to support UML 2.0. Particularly, we describe how to address the transformation of functional requirements (use cases) into component based requirements for the product line architecture based on UML v2.0 [UML 2005]. We present how we evolve 4SRS to incorporate the UML-F profile [Fontoura *et al.* 2000] notations and its extensions, as described in Chapter 4. We also show how to incorporate the UML 2.0 metamodel extensions that are proposed in Chapter 4. These extensions enhance the UML 2.0

metamodel to support the modeling of complementary variability situations in use case diagrams. We explore how the architectures that result from MoDeLine can be refined by using design patterns and design principles that have its origin in UML-F. We present our approach in a practical way and illustrate it with a case study.

3.2 Extending 4SRS for Variability Support

One of the most important artifacts of a product line is the product line architecture. A product line architecture is the basis for the derivation of the architectures of the members of the product line and also of the development of reusable product line components. As such, the product line architecture must encompass all the actual members of the product line as well as future members. This makes the architectural model a crucial artifact of the product line engineering process.

As for single systems development, the reference architecture for a product line is basically obtained from requirements. UML use cases are a widely adopted technique for functional requirements modeling. They are used with this perspective in single system development and also in product line approaches [Gomaa 2005]. In a product line approach requirements result from domain analysis. The domain analysis phase of product line engineering may involve several specific activities, besides functional requirements modeling, such as product line scoping and product portfolio definition.

The scoping activity aims at defining the products that the product line may include. In order to do so it is necessary to identify what is the domain of the product line and what are external and sub-domains. The result is usually a diagram representing the relations between domains.

The product portfolio aims at identifying the exact members of the product line, its characteristics and the timing for its development. To differentiate between products of the product line it is necessary to identify its features. Some features are common to several members of the product line while others are not. Feature diagrams are usually adopted for this purpose [Kang *et al.* 2002].

The two major techniques for dealing with requirements in a product line approach are use cases and feature models. They can be used together: the use case model is user oriented while the feature model is *reuser* oriented [Griss *et al.* 1998]. In this way, use cases focuses on requirements elicitation (what functionality should by provided by the product line), while features address better the functionality that can be *composed* for the members of the domain.

Regarding the reference architecture of a product line, use case models are the driving force that guides its development. Nevertheless, there are few documented processes in the product line area to help in the transition from use case requirements to high-level reference architectures. For instance, RSEB [Jacobson *et al.* 1997] (Reuse-driven Software Engineering Business) proposes that each use case gives origin to three kinds of objects, following the boundary-control-data pattern. But this is still just the starting point of the process. Other methods, like PuLSE [Anastasopoulos *et al.* 2000], simply provide a framework for guiding the design and evaluation of the product line architecture.

In this context, we find that the derivation of a high-level architecture from the requirements of a product line is still a topic of the product line engineering process that needs further research. In this section we address this problem. Our approach is based on a proven technique that has been used for the derivation of single system architectures from requirements modeled as UML use cases. The 4SRS technique applies transformational steps in order to derive a high-level

architecture (system-level object model) from the requirements of a system. In order to use this technique in the product line context, adaptations are needed. For instance, the technique has to address the variability concept that is essential to product lines. In order to best evaluate our approach we use, along this section, the publicly available IESE *GoPhone* product line technical report [Muthig *et al.* 2004]. This technical report presents a mobile phone product line engineered using PuLSE and KobrA. KobrA is an object-oriented customization of the PuLSE method [Bayer *et al.* 2001].

The development activities covered in this section are depicted in gray in Figure 42.

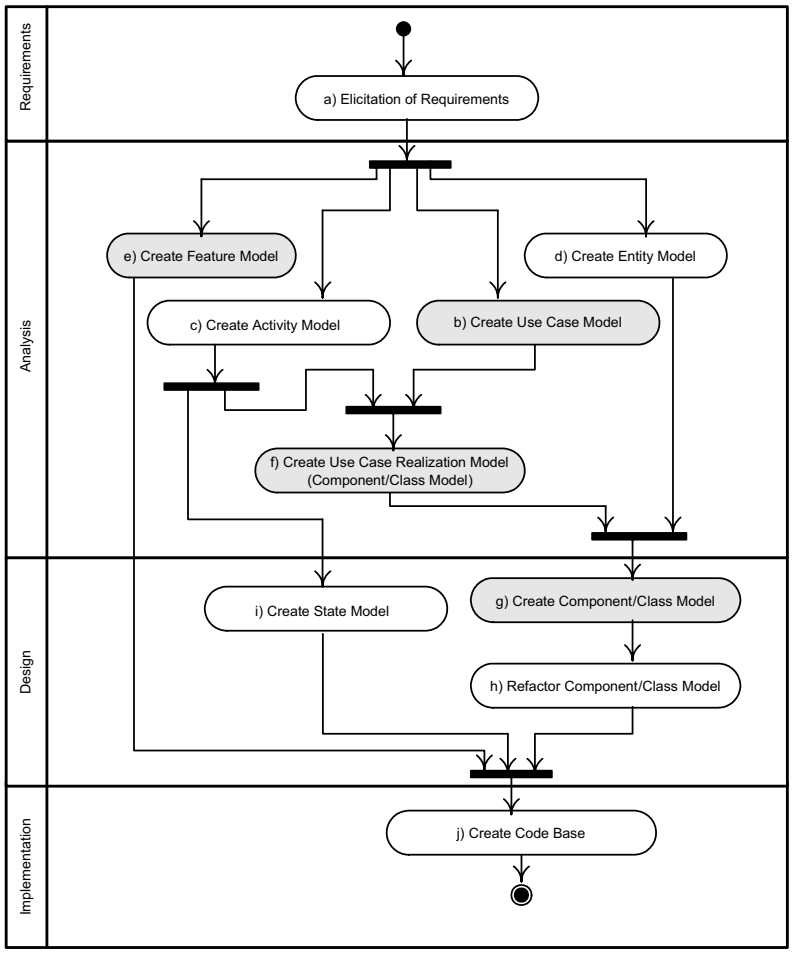

Figure 42: Development activities covered in section 3.2.

3.2.1 Requirements Modeling

Functional requirements of product lines can be modeled by use cases. Use case modeling in a product line must capture the requirements for all the possible members of the product line. As such, when adopting use cases to model the requirements of a product line, the major issue is the representation of variability. This means that each use case can vary, depending on the functional requirements of the members of the product line.

Variability is usually modeled using the concept of *variation points*. These variation points identify locations where variation will occur. In use cases, variation points can be expressed in different ways: includes relationship, extension points and use case parameters. To our knowledge, extension points are the more common way of expressing variability in use cases.

Variability can also be modeled in use case diagrams by using stereotypes to mark use cases. For instance, Gomaa proposes three stereotypes to classify use cases regarding variability: «mandatory», «optional» and «alternative» [Gomaa 2005].

In GoPhone, a variant use case has the stereotype «variant». A variant use case is a use case which functionality can vary between elements of the product line. Figure 43 shows the use case for the messaging domain of the GoPhone product line. From the model it is possible to observe that *send message* is a variant use case of the product line. Further details regarding the use case variability are specified textually, in the use case description. Figure 44 is an extract from the textual documentation of the *send message* use case in the GoPhone report [Muthig *et al.* 2004].

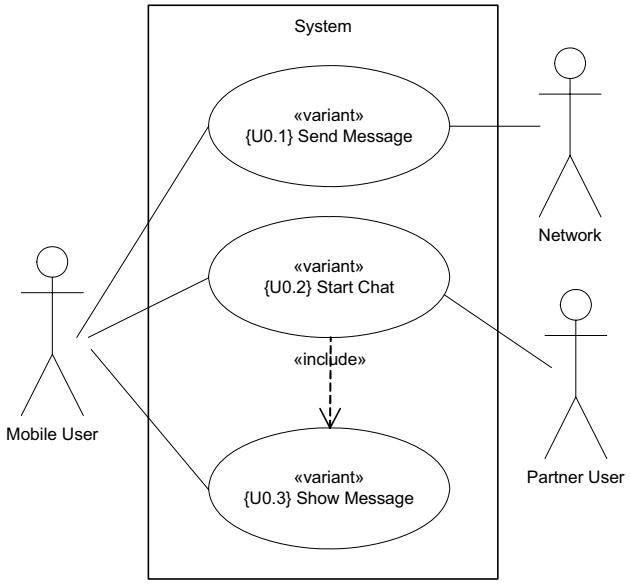

Figure 43: Use case diagram depicting the main functionality of the messaging domain (Based on the IESE's GoPhone Technical Report [Muthig *et al.* 2004]).

The *send message* description shows all the variation points of the use case. Variation points are identified by *OPT* or *ALT* tags. This approach explicitly points out all variation points of the use case but has disadvantages. For instance, if the use case is long, it may become very difficult to recognize a possible scenario for a member of the product line. Even further, this textual description is not adequate when the aim is the automation of tasks or the adoption of tools for dealing with variability.

In order to ease the automation of transforming the requirements of a product line into its high-level architecture (i.e., apply the 4SRS technique) we propose the explicit representation of the variation points in the use case model. In order to do so, a careful analysis of the initial use cases must be done.

The initial use cases, that are used to communicate the system functionalities with the stakeholders, must be transformed in order to express explicitly the functional variations of the

product line. This activity can be done without the intervention of the users of the system. The main idea is to extract the *include* and the *extend* relationships from the textual description of the use cases. The *include* relationships will result from functional decomposition and will allow the discovery of functional commonalities among use cases. The *extend* relationships will basically result from extracting alternative and optional functionality from the use cases.

Send Message

1. The user chooses the menu-item to send a message.
2. The user chooses the menu-item to start a new message.
3. Are there various message types?

<OPT> The system asks the user which kind of message he wants to send (Go Phone S, M, L, XL, Elegance, Com, Smart)

4. The system switches to a text editor.
5. The user enters the text message.
6. Is T9 supported?

<ALT 1> If T9 is activated, the system compares the entered word with the dictionary. (Go Phone XS, S, M, L, XL, Elegance)

7. Which kind of objects can be inserted into a message?

<ALT 1> The user can insert a picture into the message (Go Phone S, M, L, XL)
<ALT 2> The user can insert a picture or a drafted text-element into the message. (Go Phone Elegance, Com, Smart)
<ALT 3> (Go Phone XS)

8. Which kind of objects can be attached to a message?

<ALT 1> The user can attach files, business cards, calendar entries or sounds to the message. (Go Phone Smart)
<ALT 2> The user can attach business cards or calendar entries to the message.(Go Phone S, M, L, XL, Elegance, Com)
<ALT 3> (Go Phone XS)

9. The user chooses the menu-item to send the message.
10. The system asks the user for a recipient.
11. Which kind of message will be sent?

<ALT 1> The user types the phone number or chooses the recipient from the addressbook.(Go Phone XS, S, M, L, XL, Elegance)
<ALT 2> In case of a basic or extended SMS, the user types the phone number or chooses the recipient from the addressbook. In case of an email, the user types the email-address or chooses the recipient from the addressbook. (Go Phone Com, Smart)

12. The system connects to the network and sends the message, then the system waits for an acknowledgement.
13. The network sends an acknowledgement to the system.
14. The system shows an acknowledgement to the user that the message was successfully sent.
15. Is a sent message directly saved in the sent-message folder?

<ALT 1> The system asks the user if the message should be saved. If it should be saved, the system saves the message in the 'sent-message' folder (Go Phone XS, S, M, L, XL, Elegance)
<ALT 2> The system saves the message in the 'sent-message' folder.
(Go Phone Com, Smart)

16. The system switches to the main menu.

Figure 44: Description of the use case Send Message (Based on the IESE's GoPhone Technical Report [Muthig *et al.* 2004]).

We like to view these activities as the construction of a three dimensional space representing the functionality of the product line: commonality, detail and variability. For instance, for each use case, we can go deeper (y axis) and broader (x axis) by adding detail as we do functional decomposition and find commonality. In a third dimension (z axis) we can express variability. This approach simplifies use case diagrams when requirements are extensive and complex because, for a given use case, one can choose to view only one perspective from the three dimensional space. In our approach we focus only on product line variability, i.e., functionality that can vary according to product line members. Variability that is common to all members of the product line can also be represented in the use case diagrams. But this can clutter the diagrams. We also advocate that this kind of variability can be better expressed in other types of diagrams like, for instance, activity diagrams. In this section we will only address product line variability.

Next, we briefly present how to construct the three dimensional space of use cases.

Functional decomposition

The initial use cases of the product line should be developed following, for instance, the process described by Alistair Cockburn [Cockburn 2001]. This should result in use cases with a main

scenario description similar to the one presented in Figure 44. These use cases should be at a medium level of detail, also know as *user level*. Based on these initial use cases, an analysis should be made with the goal of factor out fragments that have high degrees of commonality between them. For instance, regarding the messaging domain of the GoPhone product line we have found three of such fragments that have become the use cases `{U0.1.1} Choose Recipient` (steps 10 and 11 of Figure 44), `{U0.1.2} Compose Message` (steps 3 to 8 of Figure 44) and `{U0.1.3} Send Message to Network` (steps 12 to 14 of Figure 44). These use cases are common to the initial use cases `{U0.1} Send Message` and `{U0.2} Start Chat`. According to the 4SRS technique, each use case name is prefixed, within curly brackets, with a 'U' followed by period separated numbers denoting the level of the use case.

We adopt Gomaa's notation [Gomaa 2005] for classifying use cases regarding their inclusion in the product line. As such, use cases can be marked with the stereotypes *mandatory*, *optional* or *alternative*. A mandatory use case is a use case that has to be included in all members of the product line. Optional and alternative use cases are only included in the members of the product line according to an inclusion condition. Alternative use cases must be in a group where usually one of the use cases is the default. This classification provides a very good foundation for viewing and analyzing the use case model according to the features of possible members of the product line.

When decomposing use cases, it is best to express the conditions regarding product line membership in the relationships, not the use cases. The reason is that these use cases can be included in several parent use cases, and the inclusion can vary depending on the parent. In Figure 45, `{U0.1} Send Message` has the stereotype *mandatory*, stating that this user level use case is to be included in all members of the product line. All the included relationships are mandatory, meaning that the use case `{U0.1} Send Message` requires all of the included use cases. Regarding decomposability, the *final* stereotype indicates that the use case is not decomposable any further. We also propose the stereotype *abstract*, to mark use cases which have all their functionality realized by others use cases, as a result of the decomposition. Since the default stereotype for the include relationship is *mandatory*, the diagram of Figure 45 does not show this keyword near the relationships. To be noted that non-mandatory functionality regarding `{U0.1} Send Message` should be left to the variability perspective.

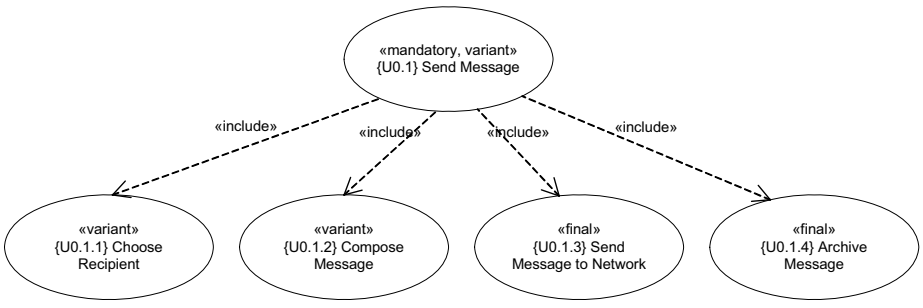

Figure 45: Decomposition of use case `{U0.1} Send Message`.

Variability externalization

The presented stereotypes do not provide hints regarding the variability of the use cases. So, in order to also express this information in the use case model we use the *variant* stereotype. When this stereotype appears on a use case it means that the use case has variability at the level of the product line. For instance, in Figure 46, use case `{U0.1.2} Compose Message` has the stereotype variant. This means that, at the product line level, this use case is variable. According to our three

dimensional approach, Figure 46 presents {U0.1.2} Compose Message in the variability perspective (z-axis). The extension points of the use case are visible and also are the conditions of inclusion of the extending use cases, according to the UML 2.0 notation. The information required to construct these perspectives can be easily extracted from use case textual descriptions. For instance, all the information required for Figure 46 can be extracted from Figure 44.

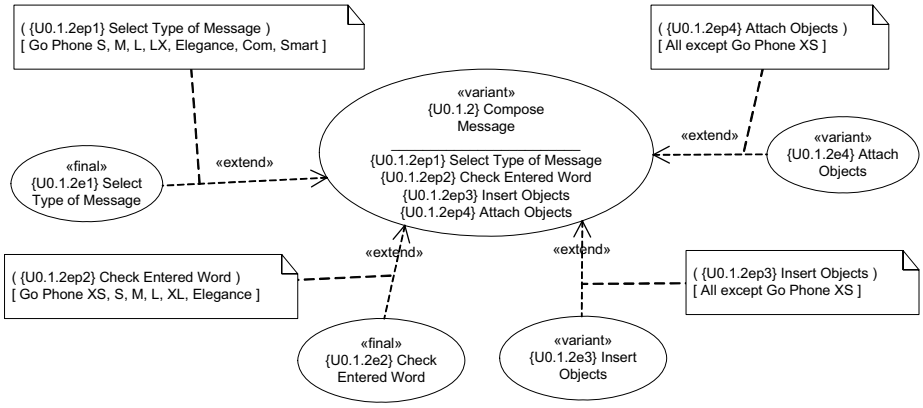

Figure 46: Variability perspective of use case {U0.1.2} Compose Message.

3.2.2 Architecture Derivation

4SRS is a method aimed at supporting the transition from system requirements to software architectures and design elements. The method is essentially based on transforming UML use case diagrams into UML object diagrams. It uses an approximation by which a use case is realized by a collaboration of objects of three kinds: interface, control and data; similarly as suggested in RSEB [Jacobson *et al.* 1997] (Reuse-driven Software Engineering Business). After this initial transformation, a series of steps with rules are proposed to transform the initial object model into a coherent object model that is compliant with the requirements. Basically, in each step, a set of refactoring rules are applied that modify the initial component model by grouping, splitting or discarding components. Some of these rules can be automated, others depend on human intervention. The method has been applied (and adapted) in several cases, from e-government [Machado *et al.* 2005] to protocol processing applications [Marcus Alanen *et al.* 2005].

This section presents the application of the 4SRS technique to the GoPhone product line use case models. We basically present a description of the transformational steps with some examples to better explain the involved transformations.

3.2.2.1 Step 1 – Object Creation

In this step, each use case originates three objects. This operation follows the same approach as RSEB, which proposes the creation of three objects for each use case: an *interface* object, a *control* object and a *data* object. For instance, in the example of Figure 45, the use case {U0.1} Send Message originates three objects: {O0.1.i}, {O0.1.c} and {O0.1.d}. This is an automatic step, and also a blind one, since each and every non-abstract use case originates three objects. Each object is named according to the corresponding use case with a suffix that identifies the type of object.

Regarding the original technique, the adaptation required for dealing with product lines is the need to detail the use case diagrams with all the extension points. For instance, in the GoPhone

case, this detail is never exposed in the use case model. The variability points are only described within the use case main scenario.

3.2.2.2 Step 2 – Object Elimination

This step of 4SRS is aimed at eliminating the unnecessary objects that resulted from the previous step. After this step, the object model should have only the objects that are functionally required, according to the requirements of the product line. The original 4SRS technique also states that "this step also supports the elimination of redundancy in the user requirements elicitation, as well as the discovering of missing requirements".

This is a major step of 4SRS and is comprised of several micro-steps.

Micro-step 2i: use case classification
In this micro-step, each use case is classified according to the *interface-control-data* heuristic that was used to automatically generate the objects in the previous step. The idea is that the classification of a use case can be a hint to eliminate unnecessary objects. Use cases are then classified according to one of the possibilities: "Ø", "i", "c", "d", "i-c", "i-d", "c-d", "i-c-d". Each letter is associated with one of the *interface-control-data* possibilities: "i"-*interface*, "c"-*control* and "d"-*data*. For instance, `{U0.1.4} Archive Message` is classified as "d", while `{U0.1.2e2} Check Entered Word` is classified as being "c-d".

Micro-step 2ii: local elimination
This micro-step regards the possible elimination of objects following the classification of the use cases in the previous step. To assist in this task, the description of the use cases should be used. For instance, the use case `{U0.1.2e2} Check Entered Word`, that was classified as being of type "c-d", is described in the GoPhone report as "If T9 is activated, the system compares the entered word with the dictionary". The value of this use case is based on the T9 functionality for validating and suggesting words. As such, the control and data facets are much more important than the interface. According to this, the object `{O0.1.2e2.i}` is removed from the object model.

Micro-step 2iii: object naming
This micro-step aim is to give proper names to objects that were not removed in the previous micro-step. Names can be derived from the base use case name, the description of the use case and also the classification of the object. For instance, object `{O0.1.2e2.d}` is named as `Word Repository`.

Micro-step 2iv: object description
All the existing objects should have a description. According to 4SRS, this description should be based on the use case description from which they resulted. Next we present an example of such a description.

{O0.1.2e2.c} Word Validator: *This object checks words as they are entered by the user. This functionality is typical of phones that have the "T9" feature. For checking and memorization of words, the object uses object {O0.1.2e2.d} Word Repository.*

Micro-step 2v: object representation
The aim of this micro-step is to globally validate the model. For instance, redundancy can be discovered and removed. Basically, this step performs a semantic validation of the object model and also of the use case model. For instance, objects `{O0.1.2e3e2.d} Picture Insertion`, `{O0.1.2e3e1.d} Draft Text Insertion`, `{O0.1.2e4e2.d} File Attach` and `{O0.3e3e1.d} File View and Save` all represent the functionality of a repository of files. As such, we maintain

only {O0.1.2e4e2.d} File Attach, since the semantic of this object includes the functionality of the other three objects.

Micro-step 2vi: global elimination
This is an "automatic" micro-step, since it is based on the results of the previous one. This step eliminates all the objects that were considered redundant in the previous step. For instance, resulting from the last micro-step, the objects {O0.1.2e3e2.d}, {O0.1.2e3e1.d} and {O0.3e3e1.d} are removed, since its functionality can be provided by the object {O0.1.2e4e2.d} File Attach. The result of this micro-step is a minimum number of objects that represent the product line functional requirements.

Micro-step 2vii: object renaming
The aim of this micro-step is to rename the objects that were not removed in the previous micro-step and that represent other objects. The documentation of such objects must also be updated. For instance, the {O0.1.2e4e2.d} File Attach object is renamed {O0.1.2e4e2.d} File Repository to proper represent its functionality, taking into account all the previous objects it represents.

3.2.2.3 Step 3 – Object Packaging & Aggregation

In this step, objects that make sense to be treated in a unified way can be placed in the same package. Aggregation can also be applied if there is a strong relationship between objects. This is usually the case of legacy objects in a sub-system. In the GoPhone product line this is not the case.

Since we are dealing only with the messaging domain of the product line, the packaging of objects follows this fact. As such, objects representing the user interface of the messaging domain are packaged in {P1} Messaging UI and objects representing messaging controlling and behavior are packaged in {P2} Messaging. Objects which major functionality is data persistence are included in {P4} Phone Database. We call this package *phone database* and not *messaging database* because it archives data regarding not only messages but other phone concepts like, contacts or files. {P3} Network is a package that includes objects with functionality regarding the mobile network, i.e., they represent the interface between the mobile phone and the network.

3.2.2.4 Step 4 – Object Association

This step introduces associations between objects that can be obtained from micro-step 2i. Also the relations between use cases can be used to generate associations between objects.

This is the last step in the 4SRS technique. Figure 47 presents the resulting object model for the messaging domain, including the packages. This object model, which resulted from the application of the 4SRS technique, is a system level object model. It provides high-level guidelines for the next phases of the development process. As such, it provides the basis for the requirements of a logical architecture that will support the following development phases. As it is possible to observe in Figure 47, the object model that result from the 4SRS technique includes all the functionality described in the source use cases. It is even possible to expose some hints regarding the product line variability, because, for instance, objects with an 'e' in their identification resulted from extending use cases. In the next Section we explore some issues regarding the logical architecture of a product line, namely variability representation and product member instantiation.

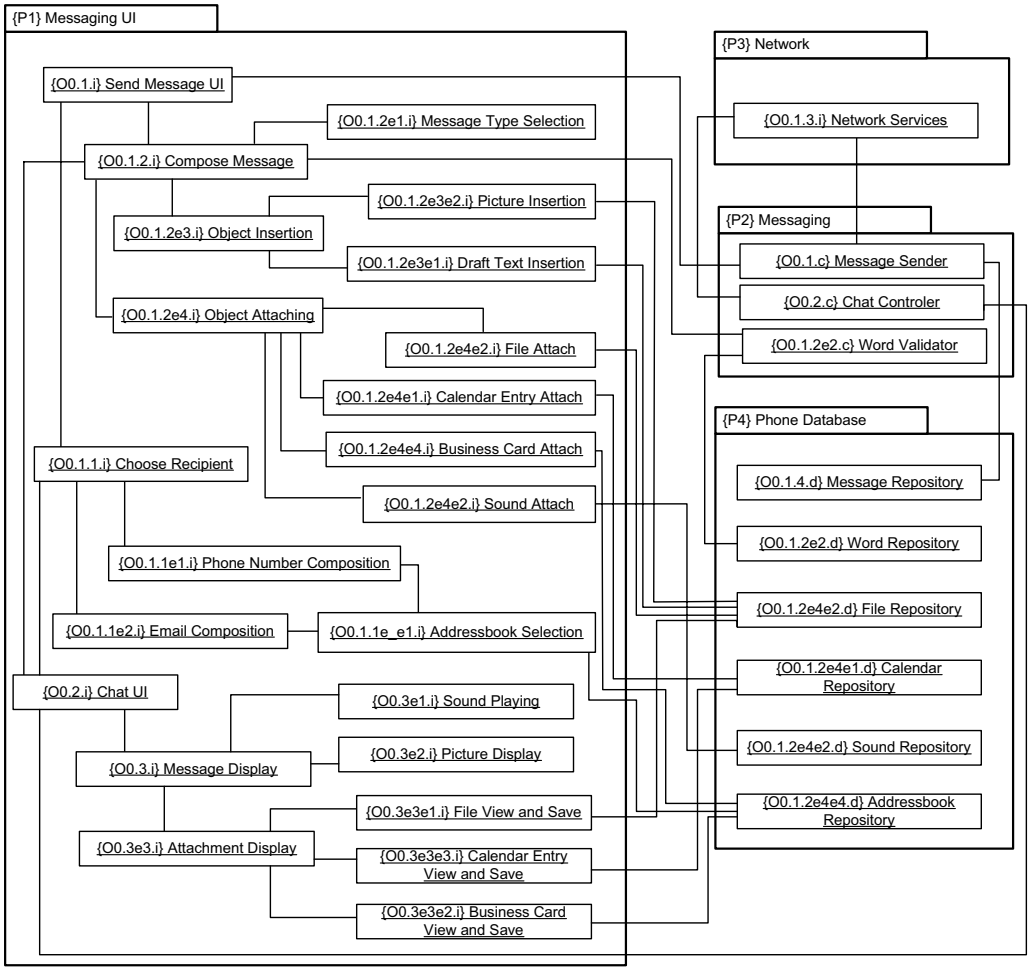

Figure 47: Object model of the messaging domain.

3.2.3 Logical Architecture

The major aim of a logical architecture is to serve as the basis for the design of a system. As such, it encompasses the description of the logical components of the system and also the interactions between them [Garlan *et al.* 1994]. As presented in the previous sections, the object model that results from 4SRS contains the components (objects) and interactions between them (object associations). As such, the object model that results from the 4SRS technique can be of great value for a system architect, because it clearly provides 'suggestions' for the logical components of a system and the interactions between them. This is very different from the usual gap that exists between requirements and the initial architecture for a system. This gap can be very 'dangerous' when the problem domain is new and there is not much knowledge in the solution space of the domain. In these cases it can be very difficult to design the system or even apply design patterns.

In the GoPhone technical report, the product line architectural design is based on the KobrA method and also on two design patterns: the mediator pattern and the state pattern. The objective of the mediator pattern is to achieve changeability and extensibility of the components and, as such, achieve flexibility in the product line. The justification for the state pattern is that it enables handling the small displays of mobile phones. These two patterns result from non-functional requirements: flexibility and state management. They impose some guidelines in the architecture

but they do not provide information regarding the functional components of the architecture. This is what we propose to achieve with the adoption of the 4SRS technique: a semi-automatic technique to obtain the product line's architecture functional requirements. The object model presented in Figure 47, which resulted from applying the 4SRS technique, depicts a partial view of such requirements for the GoPhone system. With such a model it is possible to design the system by applying well-known patterns, such as the mediator and the state pattern (such as in the GoPhone report). The difference from the GoPhone report is that, with our approach, we know which logical functional components are necessary to incorporate in the design. In this case, our logical architecture for the GoPhone product line is very similar with the one from the original report, the major difference being the fact that in our process we did not adopt KobrA.

The 4SRS technique was originally designed for obtaining the logical architecture of single systems. For this reason it does not deal explicitly with variability. As we saw, the main resulting artifact of the 4SRS technique is the object model. In our experimental approach to adapt 4SRS for product lines we have already proposed the need to externalize variability in the use case model. Regarding the logical architecture we also propose that other views of the system are needed to properly address product line development requirements. For instance, a class model may be more appropriate to express variability at the architectural level. Also, activity models are more appropriate to express fine grained variability. As such, we propose a multiple model approach for 4SRS. A similar approach can be also find in [Gomaa *et al.* 2004].

This multiple model approach is also more suited to deal with product line member instantiation. Product line member instantiation is based on the selection of features required for the member being instantiated. As previously mentioned, the usual approach is to build a feature diagram to guide this instantiation. The construction of a feature diagram can be done in parallel with the use case diagrams. In our approach, feature diagrams correspond to choices in the variability perspective (z-axis), when navigating through the use case model. A functional feature is basically realized by a use case. Extending use cases become optional or alternative features. Figure 48 presents a feature diagram for send message. The Figure also presents a possible example of the selection of features for a product line member, by showing them in gray.

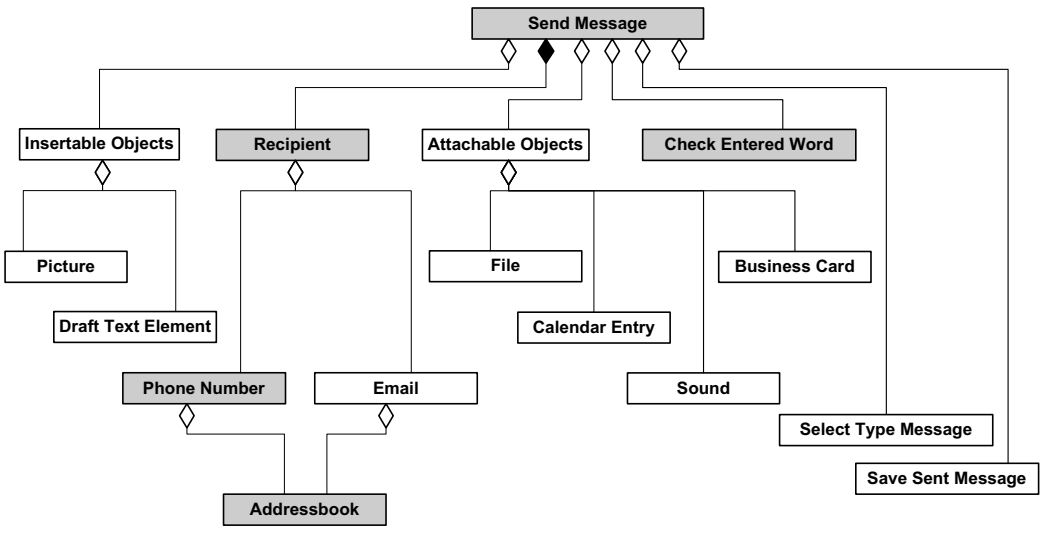

Figure 48: Feature diagram for *Send Message* (Based on the notation proposed in [Gomaa 2005]).

Figure 49 presents an excerpt of a possible class diagram depicting the *send message* feature according to the feature selections of Figure 48. This class diagram is based on UML 2.0 notation (the adoption of UML 2.0 in 4SRS is discussed in the next section). The class model should be constructed after the object model. The major goal of the class model, at this logical architectural level, is to be the first approximation to a structural model of the product line architecture. In the process of constructing the class model it is possible and even common that functionalities provided by several objects become realized by a single class or a hierarchy of classes.

The class diagram at this logical architectural level is used to represent the product line at a component level of abstraction. These class diagrams could be substituted by component diagrams. We have not done it in this case study because our aim was to follow as much as possible the models used by the 4SRS method.

The class model also provides a way to explicitly represent the product line variability. So, the construction of the class model is also based on the use case model and feature model. In this section we do not discuss the process for the construction of the class model.

As it is possible to observe in Figure 49, in this experimental approach we have adopted outgoing and incoming interfaces to model extension points. This seams to be an appropriate choice at this logical component level. This option does not compromise later design decisions of how to realize the extension points. In fact, other authors have proposed comprehensive feature variability realization techniques at the design level that are based on interfaces [Lee *et al.* 2004].

Since in our approach there is a very direct mapping between the use case model and the feature model and because it is easy to keep trace links from the class model to the object model and ultimately to the use case model, it is possible to derive the architectural requirements for a product line member based on its features. This topic will be addressed in the next section.

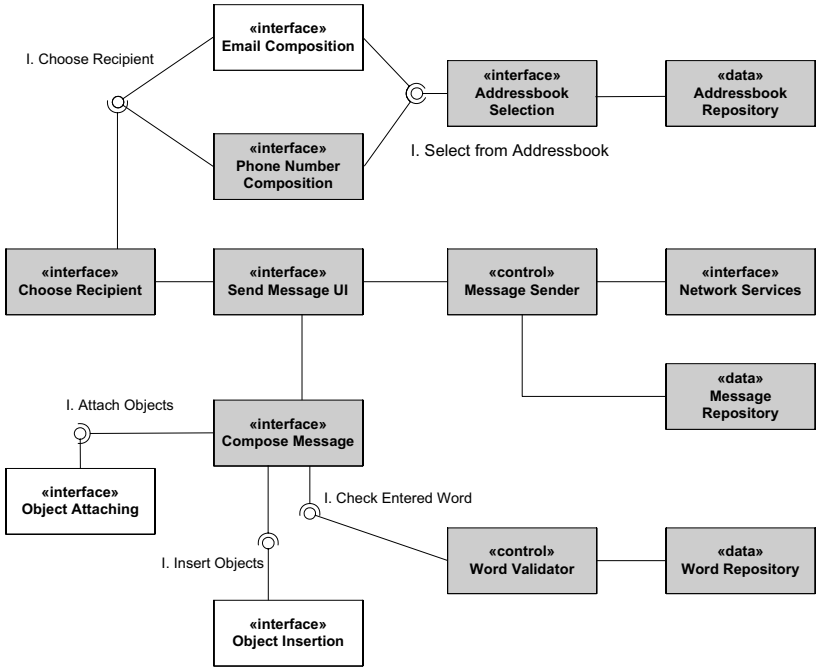

Figure 49: Excerpt of class diagram for *Send Message*.

3.3 Adopting CIM Models for Derivation of Architectural Requirements

The alignment of the software architecture and the functional requirements of a system is a demanding task. The conceptual gap that exists between the problem domain and the solution domain is very significant for the majority of the software projects. When this happens, it becomes difficult to co-relate requirements specifications and design decisions. The software architecture can rapidly become unsynchronized with the specified requirements of the system. To address this problem it is necessary to keep links between elements of the different levels of development. Achieving this without proper methodological and tool support is a daunting task. Model driven development is a promising approach since models are treated as first-class software artifacts, as traditional development does with code.

Model driven methods are still a research topic. Much of the actual effort is on supporting tools and languages for transformation of models. One example is QVT [QVT 2005], an OMG initiative to standardize model transformations in MDA [MDA 2007]. Also, reports of model driven approaches tend to focus on transformations and are usually applied to design and implementation models, and usually do not include requirement and analysis models. Nonetheless, requirements specification and analysis are crucial activities of all software development processes. They drive the design of the system's architecture. As such, they should be integrated into model driven methods.

Product lines are also another concept that is gaining popularity among the software industry. In this section, we will present an approach in which MoDeLine, an evolution of the 4SRS method aimed at supporting the model driven development of software product lines, is used to derive the architectural functional requirements of a product line from its requirements.

3.3.1 The Method

MoDeLine is a method that is composed of our methodological proposals for model driven development of software product lines. It is based on the 4SRS method with the following major additions: (1) adopts the UML 2.0 modeling language; (2) extends UML 2.0 metamodel for variability support; (3) adopts feature diagrams; and (4) adopts and extends the UML-F profile [Fontoura *et al.* 2000].

Figure 50 presents an overview of possible activities of model driven methods aimed at the development of product lines. These are also the activities that compose the MoDeLine method. In this section, we will only cover the activities that are marked in gray (*b*, *c*, *e*, *f, g* and *h*). The next sub-section describes how the MoDeLine method *formalizes* use case behaviors through activity diagrams. This enables MoDeLine to capture functional requirements in a precise way. These activity diagrams also capture a very fundamental concept of product lines: variation points. The next sections also briefly discuss feature diagrams and how the concepts of these three kinds of diagrams relate to each other. For clarity reasons, some details are left out of the diagram of Figure 50. For instance, activities *b*, *c*, *d* and *e* can be done in parallel but are not independent, they require coordination between them.

Experimental work on adapting the 4SRS method to handle variability in the context of product lines has been discussed in Section 3.2. From this previous work we have identified two fundamental issues regarding UML that require further clarification in order to fully integrate use case models into our evolution of the 4SRS method. These issues are:

- the semantics of the use case relationships: include; extend and generalization;
- formalization of use case behaviors.

These two issues are addressed by the extension to the UML 2.0 metamodel and the adoption of activity diagrams for the specification of the behaviors of use cases that are proposed and discussed in Chapter 4. The major extension proposed was that *Extend* relationships may require rejoin points that are different from the original extension point. In the next section, we briefly present the MoDeLine metamodel and discuss this situation.

3.3.1.1 Metamodel

Figure 51 presents an excerpt of the MoDeLine metamodel. The figure contains only the necessary elements to support the discussion of the topics covered in Section 3.3. From the figure we observe that this metamodel is an adaptation of the UML 2.0 metamodel. This essentially relates to the fact that the UML *Extend* relationship lacks the support for rejoin points, as indicated previously. As advocated in Chapter 4, alternative behavior needs the *rejoin* concept to be totally specified. So, in the MoDeLine metamodel, the new element *ExtensionFragment* adds this support to the original *Extend* relationship of UML.

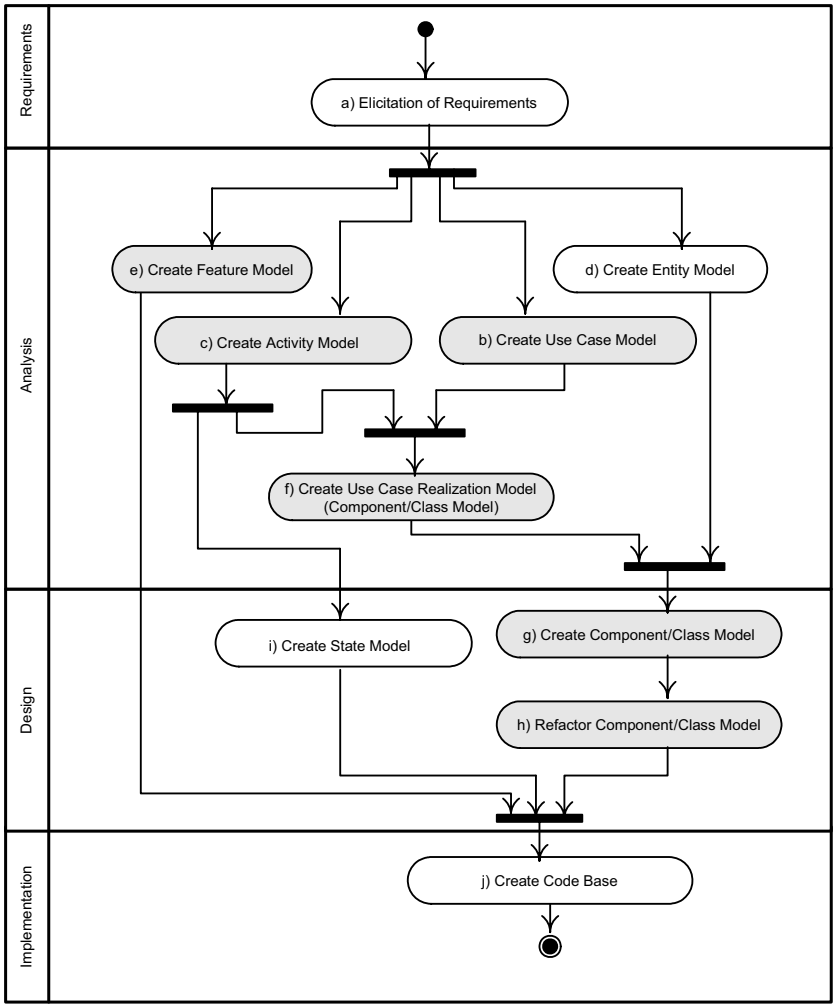

Figure 50: Development activities covered in Section 3.3.

The formalization of use case behavior by means of activities is also depicted in Figure 51. For each use case behavior there is an activity. Extend and rejoin points of use cases trace directly to nodes of activities.

The advantage of adopting a model driven approach, if it is supported by proper metamodeling tools, is that it is possible to add or adapt metamodels. One example of such adaptation is the *ExtensionFragment* element. Another example of altering the UML metamodel is the added support for feature diagrams that is also depicted in Figure 51. Although feature diagrams [Kang *et al.* 1990] are not part of the UML metamodel, they are a crucial artifact of product lines. They model the characteristics of a product line and how they relate to each other. A selection of the features represents a particular application of the product line. As such, features should relate to the requirements of the product line. Figure 51 also presents these relationships can be accomplished.

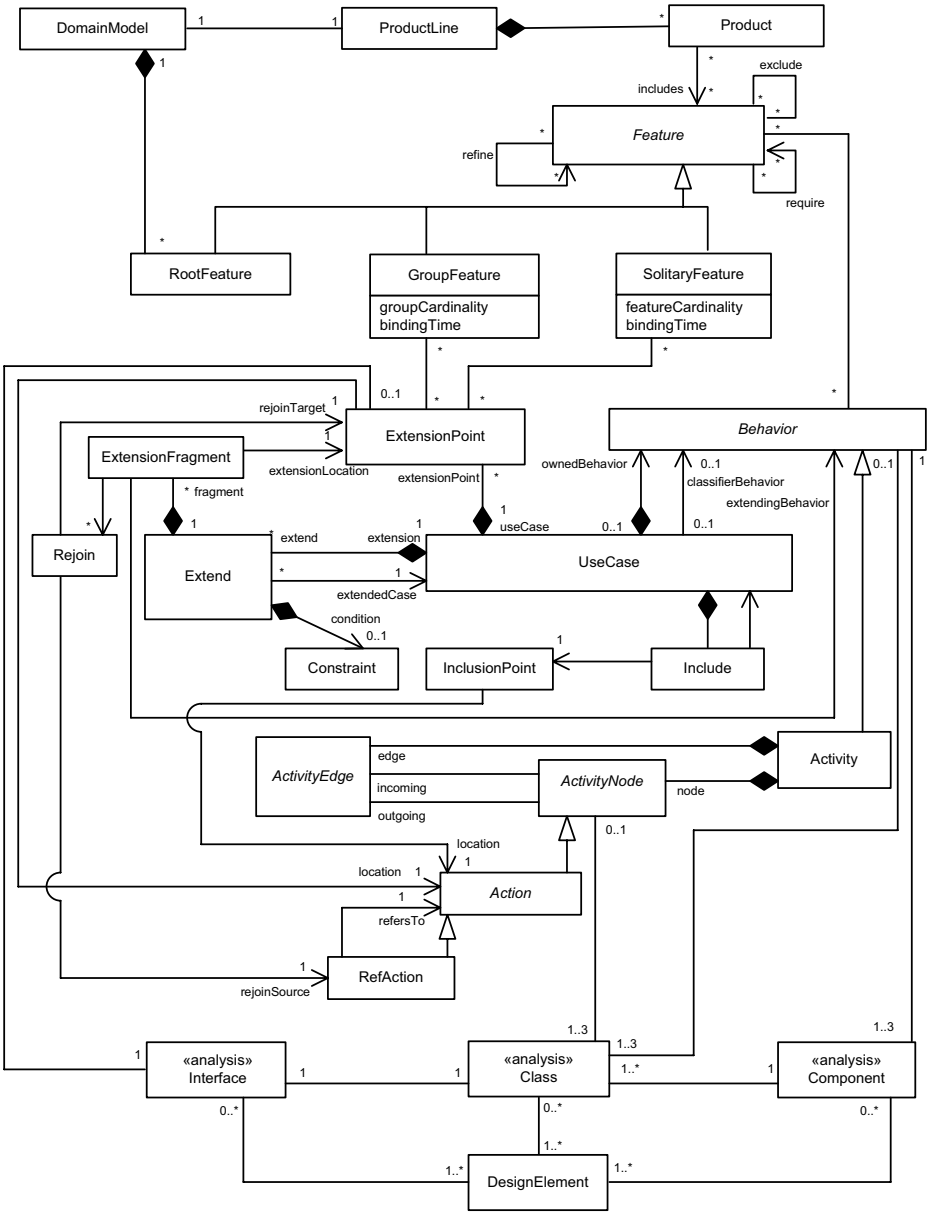

Figure 51: Excerpt of the MoDeLine metamodel.

3.3.1.2 UML-F

UML-F was proposed as a UML profile for frameworks [Fontoura *et al.* 2000] and later support was added to product lines [Pree *et al.* 2002]. With this profile it is possible to annotate UML elements with stereotypes that properly model variability. Unfortunately, the original UML-F profile only covers design elements. The profile lacks support for requirements and analysis models. As such, MoDeLine had to extend the UML-F profile to include support for requirements and analysis models. This topic is discussed in Chapter 4. Table 4 summarizes these stereotypes and informally defines their semantics.

Table 4: Summary of UML-F based stereotypes used in MoDeLine and their meanings.

Stereotype	Applies to element	Description
variant	UseCase	Indicates that the behavior of the use case can vary.
mandatory optional alternative	UseCase	Classifies use cases according to their *inclusion* in the product line.
inclusion_point	Action	Indicates that the *Action* is an inclusion point for the classifierBehavior of the included use case.
extension_point	Action	Indicates that the behavior of the use case can be extended at the *Action*.
vp	ExtensionPoint	Indicates that the *ExtensionPoint* is a variation point of the product line.
template	«analysis»Component «analysis»Class DesignElement	Indicates an element which behavior is affected by variants that relate to a hook (based on [Pree *et al.* 2002]).
hook	«analysis»Interface DesignElement	An element that represents (or contains) a location where variations occur, i.e., a variation point (based on [Pree *et al.* 2002]).
rejoin_point	RefAction	Indicates that this *RefAction* rejoins the flow at the referenced *Action* of the base behavior. Attributes: *Moment* (*before* or *after*).
application	DesignElement	Indicates that the *DesignElement* relates to a specific application of the product line (based on [Pree *et al.* 2002]).
framework	DesignElement	Indicates that the *DesignElement* is global to all applications of the product line (based on [Pree *et al.* 2002]).
variable	Method	Indicates that the behavior of the method varies (based on [Fontoura *et al.* 2000]). Attributes: *Instantiation* (*dynamic* or *static*).
extensible	Class	Indicates that new methods can be added to the class (based on [Fontoura *et al.* 2000]). Attributes: *Instantiation* (*dynamic* or *static*).
incomplete	Generalization	Indicates that the generalization set can be incomplete, i.e., it is possible to add new classifiers to the set (based on [Fontoura *et al.* 2000]). Attributes: *Instantiation* (*dynamic* or *static*).

3.3.1.3 Case Study: Library Product Line

To demonstrate the MoDeLine method, and particularly how it can be used to derive the architectural requirements of a product line, we will use a library product line. Suppose there is a software company that is planning to provide software management applications for libraries. Such

company could adopt a product line approach. It could develop products with different features to address the specific needs of its customers. It could also plan on release versions of its products with different features at different moments. Figure 52 presents a possible feature diagram for such a product line. We follow a notation for feature diagrams that is based on the one proposed in [Deursen *et al.* 2002]. In MoDeLine, the initial feature model can be automatically constructed from the use case model. This topic is further discussed in Chapter 5.

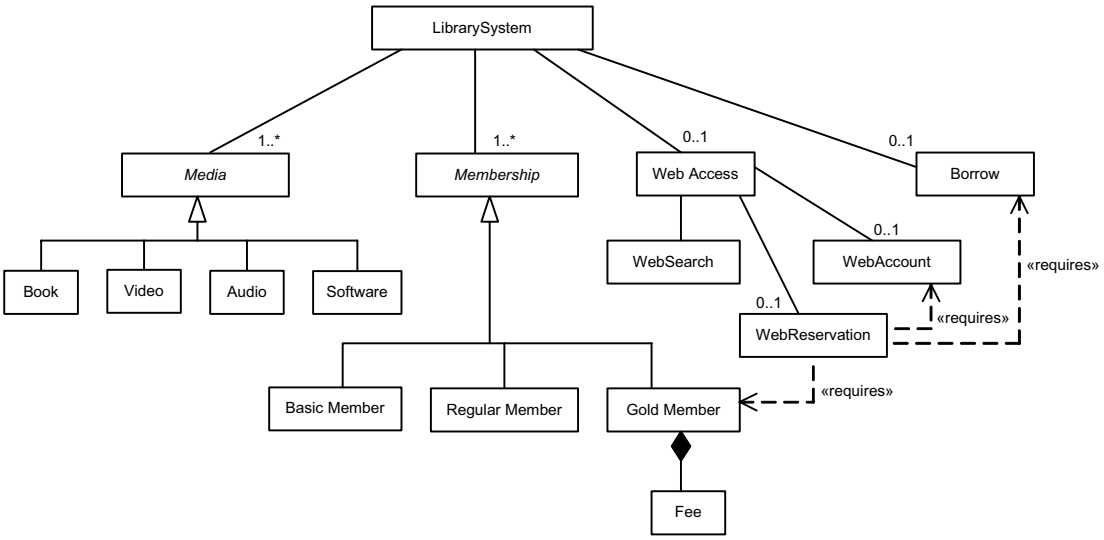

Figure 52: Feature diagram for a library product line, following notation proposed in [Deursen *et al.* 2002].

3.3.2 Modeling Requirements with Use Cases and Activity Diagrams

UML use cases are very useful in capturing requirements because of their simplicity but, as discussed in the previous section, they also have some informal characteristics that difficult their adoption in model driven methods. To address these issues, MoDeLine extends the UML use case metamodel and adopts *Activities* to formally specify use case behavior. In MoDeLine, for each use case behavior there is an activity diagram.

When use cases of the domain are identified, their behavior is modeled by activity diagrams. This is not so different than the traditional way of describing use case behavior by natural language, such as in [Cockburn 2001]. Basically, each step in a text description of a use case is modeled as an *Action* node in the activity diagram. Sequence diagrams can be a helpful tool in modeling diverse use case scenarios that together describe the global behavior of a use case. They also can help when building the activity diagrams.

Use case relationships are discovered during use case modeling. The *informal Include* (usually denoted as «include») and *Extend* (usually denoted as «extend») relationships become *formal* as they are modeled in MoDeLine, since they relate *Action* nodes and use case elements in a precise way. Common use cases for diverse applications become mandatory use cases of the product line. Optional and alternative use cases will participate in *Extend* or *Include* relationships (except if they are *root* use cases). As such, during this process of modeling use cases, we are also identifying features of the product line. However, feature diagrams should not become only direct mappings of use cases. For instance, all top-level mandatory use cases should relate to a single root feature. The feature diagram can also include non-functional features that are not related to use cases. In

MoDeLine, establishing relationships between features and use cases can be automated as discussed in Chapter 5. However, modeling feature diagrams is also a human task since it usually requires specifications that are hard to automate. For instance, there can be constraints between features that are based on marketing decisions and as such, must be manually edited in the feature model. Nonetheless, as we propose in Chapter 5, it is possible for a tool to automate part of this process and to suggest operations on the models based, for instance, on the stereotypes presented on Table 4.

Figure 53 presents the optional use case `{U0.1.1} Renew Loan` of the library product line (this use case is only present in an application of the product line if the feature *Borrow* is selected). This use case is extended by the use case `{U0.1.2} Handle Gold Member`. This extension was created since different kinds of membership impose different behavior in the system. Extension points usually give support to optional or alternative features. Since, in this case, an application of the library product line can support one or more member types, the feature *Membership* (see Figure 52) is modeled as a *GroupFeature*. As such, the extending use case `{U0.1.2} Handle Gold Member` becomes a realization of the feature *Gold Member* (one of the composing features of *Membership*). The *Membership* feature (a *GroupFeature*) relates to the extension points `Collect Fine`, `Get Item Status` and `Renew Loan` of the use case `{U0.1.1} Renew Loan`.

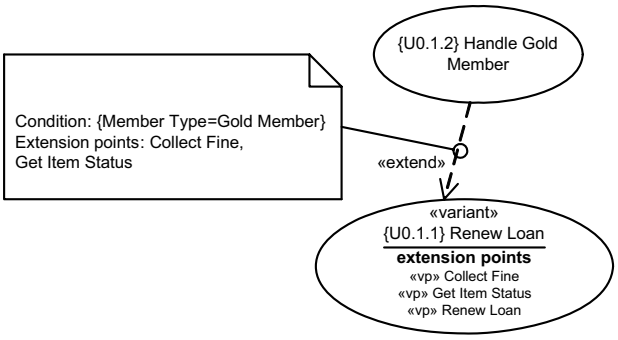

Figure 53: An «extend» relationship between use cases `{U0.1.2} Handle Gold Member` and `{U0.1.1} Renew Loan`.

Similarly to this simple case, it is possible to relate use cases and features as the requirements models are built. Other possible approach is to build the feature model after the use case model. The two approaches are possible in MoDeLine and both compatible with the goal of keeping relationships (*links*) between use case model elements and feature model elements.

In fact, since use case behavior is modeled by activity diagrams, a trace can be made from elements of the activity diagrams to features, through use case elements. Figure 54 presents the activity diagrams for the main behavior of use case `{U0.1.1} Renew Loan` and for two alternative behaviors of the extending use case `{U0.1.2} Handle Gold Member`. The activity nodes marked with the stereotype «extension_point» relate to the correspondent use case extension points with the same name. And, as we saw, these extension points relate to the *Membership* feature. Similarly, the alternative behaviors of use case `{U0.1.2} Handle Gold Member` that extend `{U0.1.1} Renew Loan` at the previous mentioned extension points should be related to one of the sub-features of *Membership*. In this case they relate to the sub-feature *Gold Member*.

Nodes of the MoDeLine activity diagrams are marked with additional information that is used to support the MoDeLine transformation rules. The stereotypes «interface», «control» and «data» are used to classify each node regarding its semantic role in the system. For instance, the node *Find Loan* represents a *data* operation in the system. We also use the concept of *partitions* to model 'who' has the responsibility for the operation of the node or is the major 'actor' of the node. For instance, the `System` is responsible for the node `Find Loan`.

When modeling activities, one very important aspect is the specification of object flows, i.e., input and output pins of the nodes. As they are specified, data types and associations are discovered. These are of great value since they provide input when creating the entity model of the domain (activity *d* of Figure 50).

3.3.3 Capturing Functional Architectural Requirements with Use Case Realizations

The adoption of activity diagrams for modeling use case behavior results in a precise specification of the functional requirements of the product line. Use case realizations are a very useful technique that helps in making the transition from requirements to analysis. A use case realization is usually modeled by a group of analysis classes that *collaborate* to perform the use case behavior. They represent the first step in the transition from the problem space to the solution space. As such, they should be the primary (eventually the only) input for the software engineer as he/she designs the product line.

The first task of the design is the specification of the architecture of the product line, i.e., the collection of computational components of the product line and the interactions between these components. These elements can be essentially derived based on the input of use case realizations. Other requirements may also influence the architecture. For instance, the architectural *style* of a product line can be influenced by the specific run-time platform or the topology of the hardware. In this chapter we will only address the functional requirements for the architecture. These are based on the use case realizations.

Traditionally, the specification of use case realizations is a very creative task. It requires a lot of experience from the software engineer, as he/she identifies the classes that realize the use case from the use case textual description. However, even a very experienced software engineer can misinterpret the requirements or forget some specification. What MoDeLine proposes is the automation of this task.

The automatic creation of use case realizations in MoDeLine is possible since use case behavior is totally specified by activity diagrams. The annotations made on the activity diagrams (as described previously) also support this automatic transformation. The basic idea is that each node of the activity diagrams gives origin to up to three analysis interfaces and analysis classes that implement the interfaces. The interfaces that originate from nodes have one method that is based on the input and output pins of the node.

Figure 55 and Figure 56 present the use case realization of, respectively, use case `{U0.1.1} Renew Loan` and use case `{U0.1.2} Handle Gold Member`. Since we are addressing design at an architectural level we find that component diagrams are more adequate than class diagrams to model use case realizations. As such, in MoDeLine use case realizations are component diagrams. Each use case gives origin to up to three analysis components. Each one is composed by the classes

and interfaces that resulted from the transformation of the activity nodes. For instance, for the use case realization of `{U0.1.1} Renew Loan`, the `Collect Fine` node gives origin to two analysis classes: `{c0.1.1b1.7.i} CollectFine` and `{c0.1.1b1.7.d} CollectFine` because the correspondent node was annotated with stereotypes «*interface*» and «*data*». Based on the *Extend* relationship that exists between the two use cases, it is also possible to automatically annotate the use case realization elements with the *hook* and *template* stereotypes of UML-F.

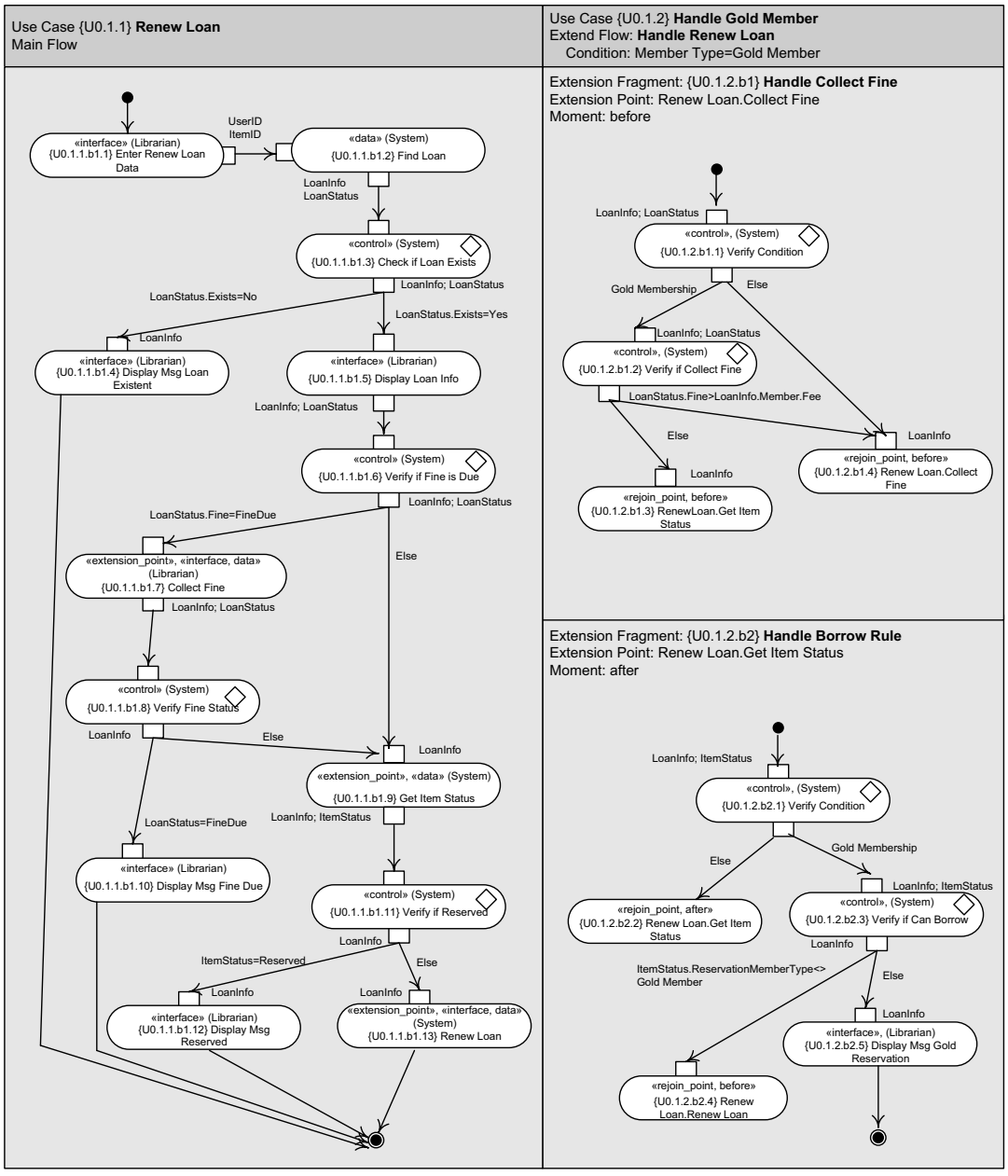

Figure 54: Activity diagrams for `{U0.1.1} Renew Loan` and `{U0.1.2} Handle Gold Member`.

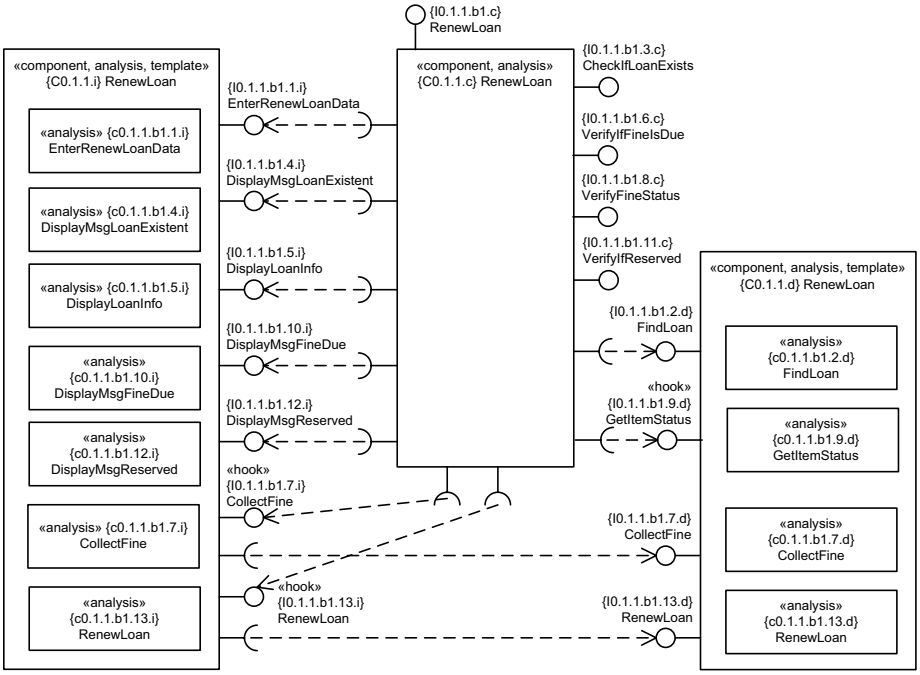

Figure 55: Use case realization diagram for `{U0.1.1} Renew Loan` (filtered view).

The creation of use case realizations in MoDeLine is a task that can be totally automated. As such, all the generated elements are *linked* to their origins. This makes it possible to trace, for instance, a feature to its realization elements. Another advantage of this approach is that use case realizations can be easily transformed into executable code for a specific language or platform following code generation approaches such as the ones presented in [Chauvel *et al.* 2005]. Thus, use case realizations can also provide simple prototypes of the product line that can be of great help for the user validation of use cases.

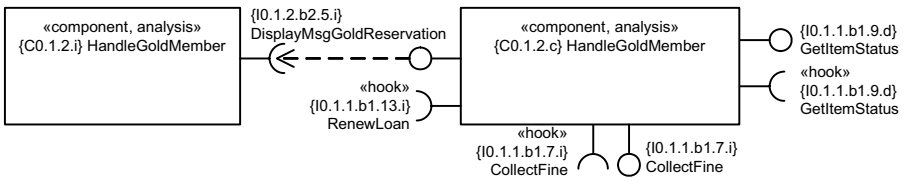

Figure 56: Use case realization diagram for `{U0.1.2} Handle Gold Member` (filtered view).

As we saw, the MoDeLine approach to use case realizations can be totally automated. One could argue that a complex use case could give origin to a complex use case realization. This is true, but eventually this would also probably happen if the creation of use case realizations were not automated.

Besides the previously discussed characteristics of MoDeLine use case realizations, the fundamental value of the method is that as a result of a simple approach, the product line engineer is able to reason with precise functional architecture requirements.

3.3.4 Logical Architecture

Each case realization provides a partial view of the product line. It is necessary to create a global model of the architecture of the product line. In MoDeLine this global architecture is based on the use case realizations and represents an integration of them. In contrast with the creation of the use case realizations, the creation of the architecture is a human based task. In this task, the product line engineer has to transform the analysis elements that resulted from the use case realizations into design elements. Each analysis element gives origin to a new design element or is incorporated into an existing one. For instance, all *«interface»* analysis components that are related to the librarian role can be incorporated into the `LibrarianUI` design component. Similarly, analysis classes give origin to new design classes or are incorporated into existing design classes. In Figure 57, we can see that the analysis class `{c0.1.1.b1.7.i} CollectFine` was incorporated into the design class `LoanUI`. The only method of the class `{c0.1.1.b1.7.i} CollectFine` becomes a part of the `LoanUI` class. To facilitate the trace to the original elements of the use case realizations, analysis interfaces are not transformed, i.e., analysis interfaces also exist in design. As the described transformations are performed the global architecture of the product line takes form. Associations between components become visible as matching required and provided interfaces are transformed from analysis to design. Variability points identified in the requirements phase are traceable to variability points in the architecture of the product line. For instance, the hook method `collectFine` of the `LoanUI` class originates from the `Collect Fine` extension point of use case `{U0.1.1} Renew Loan`.

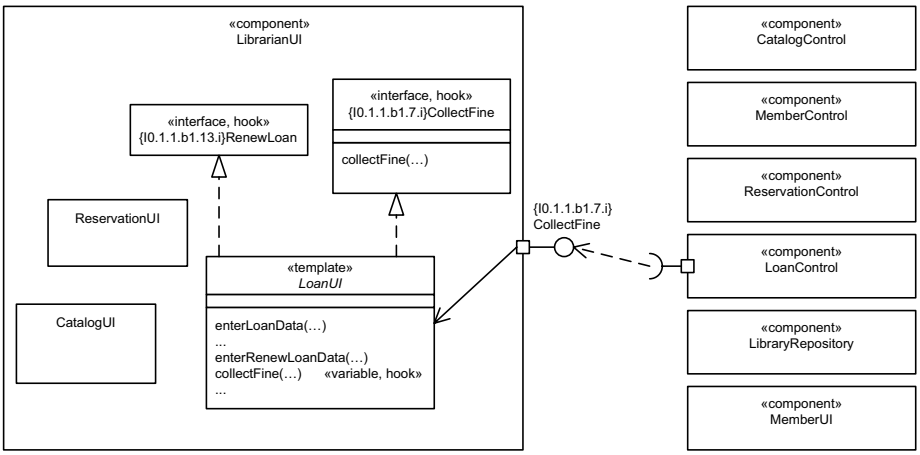

Figure 57: Architectural logical view showing `{I0.1.1.b1.7.i}CollectFine` connecting the `LibrarianUI` and `LoanControl` components (filtered view).

As Figure 50 shows, the design model is not based only on use case realizations. The entity model is also an input for the design model. Its classes also populate the design model.

The tasks described in this chapter enable the creation of an initial version of the architecture of a product line that is traceable to requirements and incorporates all functional requirements (as they were modeled). The design of a product line does not end with its architecture. More detailed design tasks are required to achieve the goal of activity *j* (see Figure 50): the creation of executable code. One example of such design tasks is the use of design patterns, as the ones proposed initially in [Gamma *et al.* 1995].

Figure 58 is an example of the result of applying the *abstract factory* design pattern to realize the variability point of the `collectFine` method of the `LoanUI` class. Originally (Figure 57), the *hook* (variability point) and the *template* were at the same class. The *abstract factory* design pattern separates the *template* from the *hook*. This realization of the variability point supports changing variants at runtime. Such a design decision should be made in accordance with the requirements. In this case, for instance, the *bindingTime* attribute of the feature *Membership* should have the value *runtime*. If only static binding time was required, the creation of subclasses of `LoanUI` would be sufficient to support the variability point.

This topic is out of the scope of the Chapter. Nonetheless, these are all examples of how the resulting artifacts of MoDeLine can be used to support detailed design activities.

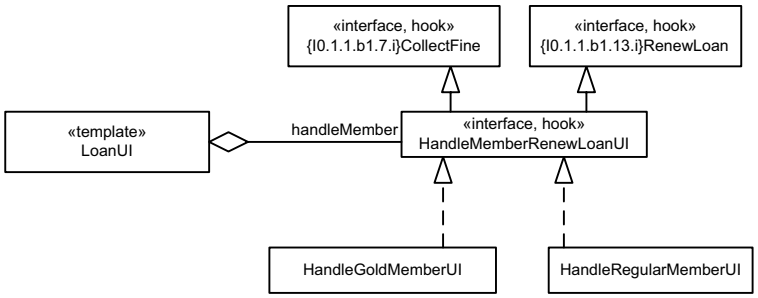

Figure 58: Applying the *abstract factory* design pattern to realize the variability point of the `collectFine` method of the `LoanUI` class.

3.4 Conclusion

In this chapter we have explored a possible methodological approach to support the model driven development of software product lines. The first part of this chapter was essentially concerned with adapting the 4SRS model driven method aimed at single system development to support variability and, therefore, be used to develop software product lines.

We have focused the discussion in the 4SRS transformational technique used for obtaining an object model from a use case model. We have discussed our approach by using a case study based on the GoPhone product line [Muthig *et al.* 2004].

The first part of this chapter contributes to the research field with a UML-based transformational technique which supports the derivation of the functional requirements of the logical architecture of a software product line in the form of an object model based on a use case model of the product line in which the variability 'dimension' is added to the use case model by using stereotypes and the Extend relationship.

The second half of this chapter presents a proposal to support the transformation of analysis models into architectural models based on an evolution of the 4SRS method to support the model driven development of software product lines that we called MoDeLine. It also delineates some approaches to detail the first logical architecture of a system by integrating design patterns in the proposed approach.

The method presented in the second half of this chapter also integrates some proposals that are only detailed in the next chapter, such as extending the UML 2.0 metamodel and the UML-F profile. We have also discussed the integration of feature models. However, the focus of the discussion was on *linking* analysis and design models. For the transition between analysis and design we have proposed *use case realizations*.

Use case realizations are a technique used to help the transition from the problem domain to the solution domain. We have presented how the creation of use case realizations could be totally automated in the MoDeLine method. This is possible because use case behaviors are *formally* modeled with activity diagrams and also because of the adaptations made to the UML 2.0 metamodel. These adaptations support the proper modeling of variability in all the activities of the method.

As a result of our approach, it is possible to maintain traces between elements at different conceptual levels. In the case of product lines, use case requirements are traced to architectural requirements: use case variability is traceable into architectural elements; features are related to architectural elements.

The MoDeLine model driven method can not be totally supported by common UML 2.0 tools, since it requires adaptations to the UML 2.0 metamodel. Tool support requires the use of metamodeling tools. This may have impact in the cost of applying MoDeLine or a similar approach. The possible increase in the cost of a project can be balanced by the increase in flexibility that such approach also brings. To be notice that product line approaches already require important setup effort. This initial cost is amortized in the future, as new applications of the product line are created. As such, we believe that MoDeLine, and similar approaches, are well suited for product lines.

Although we have not presented or discuss details regarding tool support of our approach, the results of our experimental development of tool support with metamodeling tools seems promising. We have supervised two undergraduate monograph projects ([Riqueza 2005] and [Pinto 2007]) that have made experimental developments of parts of our method with two metamodeling frameworks: GME [Ledeczi *et al.* 2001] and EMF [EMF 2007].

The second part of this chapter contributes to the research field with a proposal of a 'bridging' technique between the problem space and the solution space for the model driven development of software product lines that is based on the concept of use case realizations and its double view: activity models for the problem space view and component models for the solution space view.

The contents of this chapter reflect our methodological proposals for model driven development of software product lines. However, for a method to be adopted in a useful manner by practitioners this is not enough. For instance, a *clear* definition of the languages and notations used is required. In the next chapter, we address some particular issues regarding the languages and notations used in our approach. We particularly explore metamodeling and modeling issues that are related to the model driven development of software product lines.

3.5 References

[Anastasopoulos *et al.* 2000] Anastasopoulos, M., J. Bayer, O. Flege and C. Gacek, "A Process for Product Line Architecture Creation and Evaluation," IESE 038.00/E, 2000.

[Bayer *et al.* 2001] Bayer, J., D. Muthig and B. Gopfert, "The Library Systems Product Line - A KobrA Case Study," IESE 024.01/E, 2001.

[Chauvel *et al.* 2005] Chauvel, F. and J.-M. Jezequel, "Code generation from UML models with semantic variation points," MODELS/UML'2005, Montego Bay, Jamaica, 2005.

[Cockburn 2001] Cockburn, A., *Writing Effective Use Cases*: Addison-Wesley, 2001.

[Deursen *et al.* 2002] Deursen, A. v. and P. Klint, "Domain-Specific Language Design Requires Features Descriptions," *Journal of Computing and Information Technology*, vol. 10, pp. 1-17, 2002.

[EMF 2007] Eclipse Foundation, "Eclipse Modeling Framework," Available at http://www.eclipse.org/emf/, 2007.

[Fontoura *et al.* 2000] Fontoura, M., W. Pree and B. Rumpe, "UML-F: A Modeling Language for Object-Oriented Frameworks," ECOOP 2000-Object-Oriented Programming Conference, 2000.

[Gamma *et al.* 1995] Gamma, E., R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[Garlan *et al.* 1994] Garlan, D. and M. Shaw, "An Introduction to Software Architecture," Carnegie Mellon University CMU-CS-94-166, 1994.

[Gomaa 2005] Gomaa, H., *Designing Software Product Lines with UML*: Addison Wesley, 2005.

[Gomaa *et al.* 2004] Gomaa, H. and M. E. Shin, "A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines," ICSR International Conference on Software Reuse, Madrid, 2004.

[Griss *et al.* 1998] Griss, M. L., J. Favaro and M. d'Alessandro, "Integrating Feature Modeling with the RSEB," Fifth International Conference on Software Reuse, Victoria, Canada, 1998.

[Jacobson *et al.* 1997] Jacobson, I., M. Griss and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*: Addison Wesley Longman, 1997.

[Kang *et al.* 1990] Kang, K. C., S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study Technical Report," Software Engineering Institute, Carnegie Mellon University CMU/SEI-90-TR-21, 1990.

[Kang *et al.* 2002] Kang, K. C., J. Lee and P. Donohoe, "Feature-Oriented Product Line Engineering," *IEEE Software*, 2002.

[Ledeczi *et al.* 2001] Ledeczi, A., M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle and P. Volgyesi, "The Generic Modeling Environment," WISP'2001, Budapest, Hungary, 2001.

[Lee *et al.* 2004] Lee, K. and K. C. Kang, "Feature Dependency Analysis for Product Line Component Design," ICSR 2004 - International Conference on Software Reuse, Madrid, 2004.

[Machado *et al.* 2005] Machado, R. J., J. M. Fernandes, P. Monteiro and H. Rodrigues, "On the Transformation of UML Models for Service-Oriented Software," ECBS International Conference and Workshop on the Engineering of Computer Based Systems, Greenbelt, Maryland, 2005.

[Marcus Alanen *et al.* 2005] Marcus Alanen, J. Lilius, I. Porres and D. Truscan, "On Modeling Techniques for Supporting Model Driven Development of Protocol Processing Applications," in *Model Driven Software Development - Volume II of Research and Practice in Software Engineering*, vol. 2, S. Beydeda, M. Book and V. Gruhn, Eds.: Springer-Verlag, 2005, pp. 305-328.

[MDA 2007] OMG, "Model Driven Architecture Guide Version 1.0.1," Available at http://www.omg.org, 2007.

[Muthig *et al.* 2004] Muthig, D., I. John, M. Anastasopoulos, T. Forster, J. Dorr and K. Schmid, "GoPhone - A Software Product Line in the Mobile Phone Domain," IESE 025.04/E, 2004.

[Pinto 2007] Pinto, A., "Model-Driven Software Development: Sistema de Especificação de Produtos de Seguros," in *Departamento de Engenharia Informática*. Porto: Instituto Superior de Engenharia do Porto, 2007.

[Pree *et al.* 2002] Pree, W., M. Fontoura and B. Rumpe, "Product Line Annotations with UML-F," Software Product Lines - Second International Conference, SPLC 2, San Diego, 2002.

[QVT 2005] OMG, "MOF QVT Final Adopted Specification (ptc/05-11-01)," Available at http://www.omg.org, 2005.

[Riqueza 2005] Riqueza, J., "GME (Generic Modeling Environment): Estudo da ferramenta e sua envolvente," in *Departamento de Engenharia Informática*. Porto: Instituto Superior de Engenharia do Porto, 2005.

[UML1.4 2001] OMG, "Unified Modeling Language Version 1.4 (formal/01-09-67)," Available at www.omg.org, 2001.

[UML 2005] OMG, "Unified Modeling Language Version 2.0: Superstructure (formal/05-07-04)," Available at http://www.omg.org, 2005.

4. Modeling and Metamodeling

"A computer program will always do what you tell it to do, but rarely what you want to do."
Murphy's Laws of Computing

This chapter is concerned with modeling and metamodeling in the context of software product lines. The first half of the chapter describes a proposal to adapt the UML 2.0 metamodel in a way that effectively enables the adoption of use case diagrams in model driven approaches. The second half of the chapter describes a proposal to extend a UML profile for the design of frameworks and product lines called UML-F so that it includes requirements and analysis diagrams.

4.1 Introduction

Software product lines and related approaches, like software factories, are starting to capture the attention of the industry practitioners. Nevertheless, their adoption outside the research community and big companies is still very restricted. We believe that model driven approaches, like OMG's MDA [MDA 2007], with proper tool support, can bring the advantages of product lines to a broader audience. For this to become a reality, model driven methods should integrate requirements models into the software development process. In the first half of this chapter, we discuss the semantics of the use case relationships and their formalization using activity diagrams to support variability specification. Particularly, we propose an extension to the *Extend* relationship that supports the adoption of UML 2.0 [UML 2005] use case diagrams into model driven methods. We exemplify our approach with the MoDeLine method. In MoDeLine, use cases are the central model for requirements specification and model transformation.

Variability is a major concern when developing software product lines or object-oriented frameworks. In the context of the UML standard language, a profile has been proposed to represent variability at the design level. The UML-F profile provides a way to model variability at the design level in a unified way [Fontoura *et al.* 2000]. A method to design frameworks based on the same basic concepts of the UML-F profile has also been proposed [Fontoura 1999]. Nevertheless, variability identification and representation should be done as early as possible in the development process. In the second part of this chapter we present an approach to extend UML-F to support requirements and analysis diagrams. We present our proposals through an insurance framework case study that shows how extensions to use case and component diagrams of UML 2.0 can be used to capture variability as early as possible in a project and also how to map variability between diagrams at requirements, analysis and design levels.

4.2 Extending UML 2.0 Use Case's Metamodel

Software product lines enable high productivity levels in software development through proactive intra-organizational reuse. Nonetheless, such approaches imply relative demanding methods and, as such, are difficult to implement in small and medium sized companies. Model-driven approaches promise to facilitate the adoption of these demanding methods because they provide high levels of automation. One well known example of such fusion of approaches is the Microsoft software factories initiative [Greenfield *et al.* 2004].

Model driven methods are still a research topic. Much of the actual effort is on supporting techniques for transformation of models. One example is QVT [QVT 2005], an OMG initiative to standardize model transformations in MDA. Also, reports of model-driven approaches tend to focus on transformations and are usually applied to design and implementation models, and usually do not include requirement and analysis models.

Requirements and analysis are crucial activities of all software development processes. In the case of product lines, their importance is higher because they guide the design of the reference architecture of the product line and all the other common artifacts. As such, MDE approaches for product lines should integrate models of these phases.

To fully integrate requirements into model-driven approaches the requirement model has to be formalized. In the case of UML 2.0 this means the formalization of use cases. In product lines, a vital concern is the specification of variability. Figure 59 presents types of alternatives (variability in action flows) that are common in the textual description of use cases. UML 2.0 metamodel does not support all these types of alternatives. In this section we address this limitation and we propose an extension to the UML 2.0 metamodel to support model driven methods with such requirements for variability modeling. For the formalization of the behavior of use cases we propose the adoption of activity diagrams.

As an example of our approach, we present a process in which requirement models are fully integrated into the MoDeLine method. The initial goal of 4SRS, the MoDeLine precursor, was to provide a method to help analysts and designers develop large object-oriented systems through the use of models and rules for model transformation.

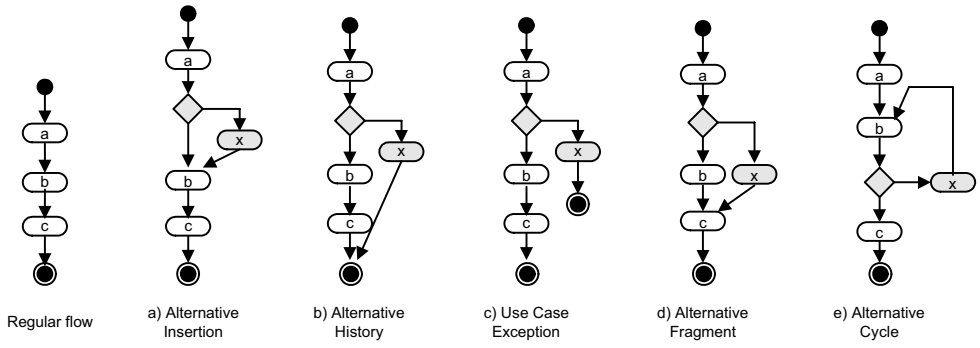

Figure 59: Types of alternative sequences of actions in use cases.

In Chapter 3 we have presented the first experimental results of adapting the 4SRS method for explicitly handle variability. In Chapter 3 we have also presented a global overview of the MoDeLine method. In this section we complement our previous work by addressing the

formalization of UML 2.0 use cases by extending its metamodel. In the context of the MoDeLine method, we also briefly address the transformation of these *new* use case models into components and classes, i.e., moving from the problem domain to the solution domain.

4.2.1 Use Case Relationships

According to the UML 2.0 metamodel, use cases can be associated by two major relationships: *Include* and *Extend*. The *Include* relationship is used when there are parts (behavior) of use cases that are common. When this happens, the common part can be extracted to a new use case. The new use case is related to the original use cases by an *Include* relationship. According to the UML 2.0 specification, an *Include* relationship acts like a procedure call, since the location of the inclusion of flow is coincident with the location of the rejoin of the flow.

The *Extend* relationship is used to model behavior that can extend the behavior of a base use case. There is always a condition associated with an *Extend* relationship. This condition is used to specify when the extension will be active.

Figure 60 presents an excerpt of UML 2.0 metamodel regarding use cases. From the presented semantics and the metamodel, it is clear that in UML 2.0 only the *Extend* relationship can be used to model variability. Nevertheless, other approaches have been proposed. For instance, Gomaa proposes that the *Include* relationship can be used to model optional use cases in product lines [Gomaa 2005]. In this section we will only address variability in use cases through the *Extend* relationship.

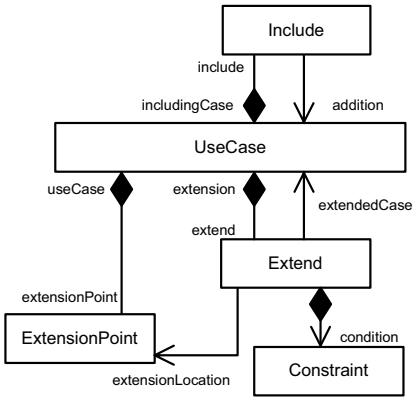

Figure 60: Excerpt of UML 2.0 use case metamodel.

When developing software product lines, features and feature diagrams are also commonly used to model variability. Features represent user-visible aspects or characteristics of a domain [Kang *et al.* 1990]. When they represent functional characteristics of a product line they can be related to use cases. Usually a feature can be modeled by one or more use cases [Griss *et al.* 1998; Gomaa 2005]. But, in the case of 'fine-grain' features, it is possible that a use case encompass several features. Features can be mandatory, optional or alternative. There can be also dependency relations between features. Optional and alternative features represent variability in a product line. These could be modeled by the *Extend* relationship.

Usually, use cases are described in natural language. In fact, there is a pattern for the textual description of use cases that is generally accepted by practitioners [Cockburn 2001]. In this pattern, use cases are composed by sequences of steps, or actions. There is usually one main sequence and many alternative sequences. There are five types of alternative sequences: conditional insertion; use case exception; alternative history, alternative part and alternative cycle [Metz *et al.* 2004]. Figure 59 presents a graphical representation of the possible sequence alternatives.

An alternative insertion (Figure 59a) is used to represent conditional behavior that is inserted into a precise point (extension point) of a flow. In this case the insertion point is coincident with the rejoin point, i.e., at the end of the alternative behavior the flow rejoins the main flow at the initial extension point. This is very similar to an *Include* relationship with a condition of insertion. Alternative insertions can be easily modeled by *Extend* relationships because the extension point and the rejoin point are coincident. In contrast, the other types of alternatives (alternative history, use case exception, alternative fragment and alternative cycle) are not directly supported by the UML 2.0 use case metamodel (see Figure 59). This is an important limitation since, in practice, it is not so unusual for extensions to have flows that are diverse from that of an alternative insertion.

To illustrate our approach we will consider the example of a library system and two use cases of that system, as presented in Figure 61 and Figure 62.

```
Use case Renew Loan:
- Main flow:
1. The Librarian enters the renew loan data (user ID and Item ID)
2. The system retrieves loan info
3. The loan info is displayed to the librarian
4. The system retrieves item info «extension point»
5. The system renews the loan «extension point»
Use case ends

- Alternative flows:
2a. Loan does no exist (after step 2)
  2a1. The system displays a message to librarian
  Use case ends
3a. A fine is due (after step 3)
  3a1. The librarian collects the fine «extension point»
  Use case rejoins (before step 4)
  Alternative flows:
    3a1a The fine is not totally paid (after step 3a1)
      3a1a1. The system displays a message to the librarian
      Use case ends
4a. The item is reserved (after step 4)
  4a1. The system displays a message to the librarian
  Use case ends
```

Figure 61: Excerpt of use case `Renew Loan`.

In Figure 61 there are 2 types of alternatives: exceptions (2a, 4a and 3a1a) and alternative flow (3a). These are internal alternatives of the use case. One of the reasons to use the *Extend* relationship is that it provides a way to extend already developed use cases with optional or alternative external behavior without interfering with them (it only requires the proper identification of the extension points in the extended use cases). This is a common process used to develop use cases: first identify and model usual and common behavior and only after reason about alternative and optional behavior. Alternative and optional behavior is usually related to entity instances and values of attributes that imply different behavior from the one of base use cases. The member type is an example of an entity type of the library domain that implies variability of behavior to base use cases.

Figure 62 presents an excerpt of the description of the extending use case `Handle Gold Member`. This excerpt contains only the extending behavior that regards use case `Renew Loan`. `Handle Gold Member` extends `Renew Loan` and uses the three extension points defined in `Renew Loan`.

We adopt a similar structure to specify regular and extending use cases, even if it is not common to have extending use cases with main flow. Also, we describe alternative flows and extension flows in the same section because their structure is basically the same, the major difference being the fact that in an extending use case the alternative flows usually relate to flows of other use cases.

```
Use case Handle Gold Member:
- Main flow: <empty>

- Alternative flows (Extension flows):
1. Handle Renew Loan
  Condition: MemberType=GoldMember
  1a. Handle Collect Fine
    (before Librarian collects the fine):
    1a1. If fine<member fee Rejoin base use case
      (before Retrieve item info).
    Rejoin base use case (before Librarian collects the fine).
  1b. Handle Borrow Rule
    (after Retrieve item info):
    1b1. If Item Reserved by non-gold member Rejoin
      base use case (before Renew loan)
    1b2. Display a message to the librarian
    Base use case ends

Referenced Extension Points:
-Librarian collects the fine:
    Renew Loan.The librarian collects the fine
-Retrieve item info:
    Renew Loan.The system retrives item info
-Renew loan: Renew Loan.The system renews the loan
```

Figure 62: Excerpt of use case `Handle Gold Member`.

Figure 62 presents common situations that reflect two types of alternatives that are not adequately handled by the *Extend* relationship of UML 2.0:

- The extending use case adds conditional behavior that can result in an alternative flow (`1a. Handle Collect Fine` and `1b. Handle Borrow Rule`), i.e., there are rejoin points that do not match the original extension point;
- The extending use case adds conditional behavior that can result in an alternative history (`1b. Handle Borrow Rule`), i.e., the new behavior can lead to an alternative ending in the base use case.

These situations could be handled by the incorporation of the alternatives into the base use case. Nonetheless, this option would lead to base use cases that would be difficult to read and understand. It would also become more difficult to handle variability in a product line, because important features, like *MemberType*, would be disperse into several use cases. Our approach (see next section) enables an effective way to model alternative flows that facilitates the process of discovering new dimensions of variable features and easily integrate them into the use case model.

For instance, Figure 63 presents how the alternative behaviors related to *MemberType* can be modeled in such a way. In this example, *MemberType* was identified as a dimension of variability because alternative behaviors that depend on the type of member of the library need to be incorporated into several base use cases. In order to address future new types of members we can model the extensions regarding member type in a single abstract use case. This type of abstract use case represents a dimension of variability. Specific instances of that variability dimension are modeled as sub-use cases. As such, in our example, `Handle Gold Member` could become a specialization of the abstract use case `Handle Member Type`.

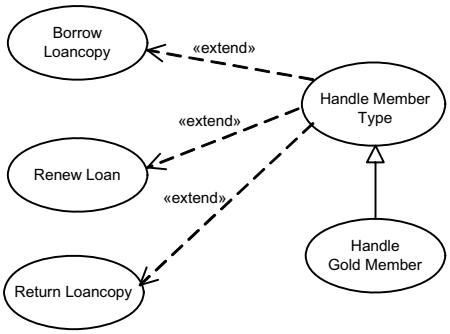

Figure 63: Modeling *MemberType* as a dimension of variability in use cases.

The use of the generalization/specialization relationship between the extending use cases (in our example, `Handle Member Type` and `Handle Gold Member`) permits the proper handling of dimensions of variability. The abstract extending use case functions as a template for the concrete extending use cases. For instance, the conditions of the *Extend* relationships are only specified in the concrete use cases.

As Figure 60 shows, the UML 2.0 metamodel only supports extensions that are basically conditional *Include* relationships. This represents a major limitation to the modeling of the diverse variability types that are commonly specified by textual use cases. In fact, it only supports alternative type *a* (Figure 59a). In the next section we present and discuss a proposal of an extension to the use case metamodel that addresses the modeling requirements identified in this section.

4.2.2 Extending the UML 2.0 Metamodel

In the past, several proposals have been made to formalize use cases [Hurlbut 1998; Overgaard *et al.* 1998; Porres 2001; Stevens 2001]. Some recent works also proposed approaches to manage variability in use cases in the context of product lines [Fantechi *et al.* 2004; Eriksson *et al.* 2005]. The main concern of their authors has been the lack of formalism of the usual use case text descriptions. Most well known proposals regard non-visual languages. In our specific case we aim at integrating requirements into a model driven method. In the context of UML 2.0, the modeling of behavior can be addressed by activity diagrams, so we have adopted activity diagrams for modeling use case behavior. Figure 64 presents an excerpt of the UML 2.0 metamodel adapted (extended) to support our proposal for formalization of use cases.

Figure 64 presents in gray those metamodel elements that correspond to extensions to the UML 2.0. Since, according to UML 2.0 specification, a use case is a specialization of a *BehavioredClassifier*, we use the *classifierBehavior* and *ownedBehavior* associations to model, respectively, the use case main flow and the alternative flows.

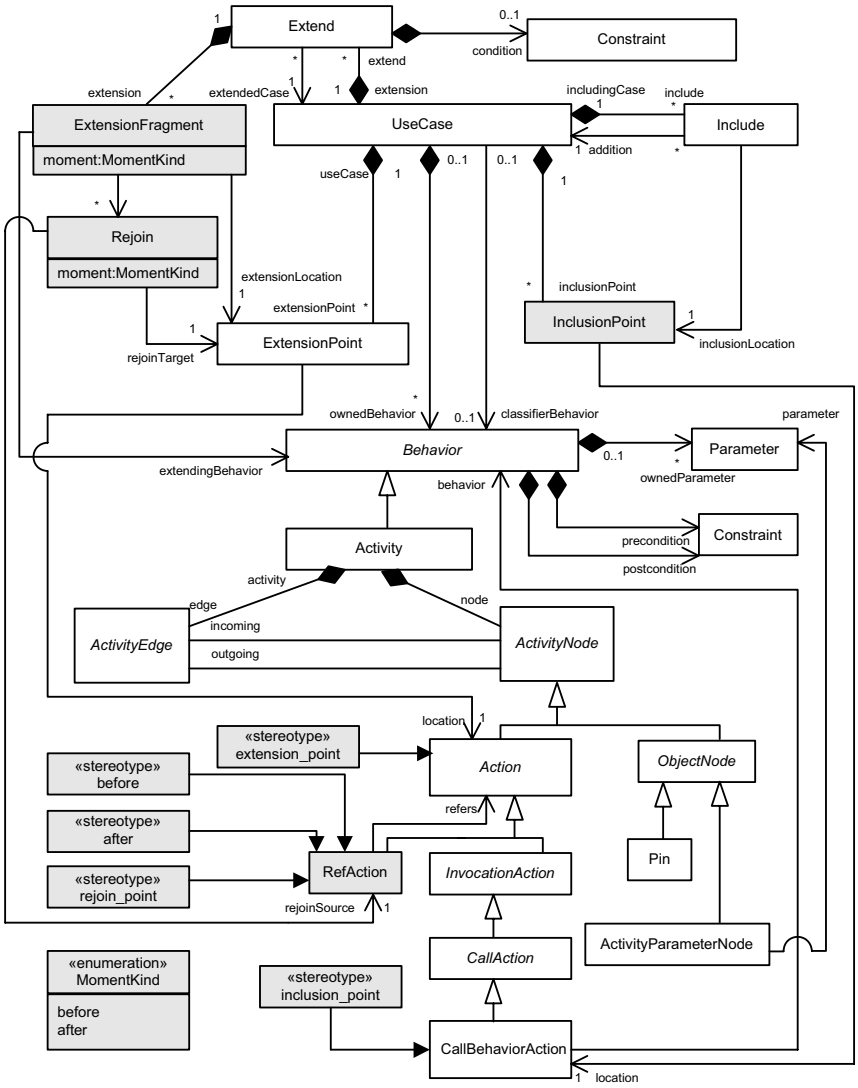

Figure 64: Excerpt of proposed metamodel.

We propose a new *ExtensionFragment* metaclass to support the issues identified in the previous section. In our proposed metamodel it is clear that an *Extend* relationship can have a condition and make several extensions (via *ExtensionFragment*) to a base use case. Each extension has one extension location but can have several rejoin locations. An extension also specifies which behavior of the extending use case will extend the base use case in the extension location. Since use case behaviors are formalized through activity diagrams, extension locations and rejoin locations refer to elements of type *Action* of the corresponding behavior. To clarify if the extension or the rejoin points are made before or after the corresponding *Action*, we propose the attribute *moment.*

Regarding the *Include* relationship, we propose the new *InclusionPoint* element so that we have a similar approach to the one used in the *Extend* relationship. An *InclusionPoint* refers to the

location where the behavior is to be included. This location has to refer to an element of type *CallBehaviorAction* of the same use case as the *Include* relationship. It is not necessary to specify what behavior is to be included because the semantic of the *Include* is to include the main behavior (*classifierBehavior*) of the included use case.

The stereotypes *extension_point*, *inclusion_point*, *rejoin_point*, *before* and *after* are used as a visual aid to more easily identify the semantics of the actions nodes of the activity diagrams.

In this section, we have briefly described the major characteristics of a proposed metamodel to support the formalization of use case models with the aim of supporting their integration into model driven methods. The next section describes how our approach is used to enhance variability support in the MoDeLine method. The development activities covered are: *Create Use Case Model*; *Create Activity Model* and *Create Use Case Realization Model* (see Figure 65).

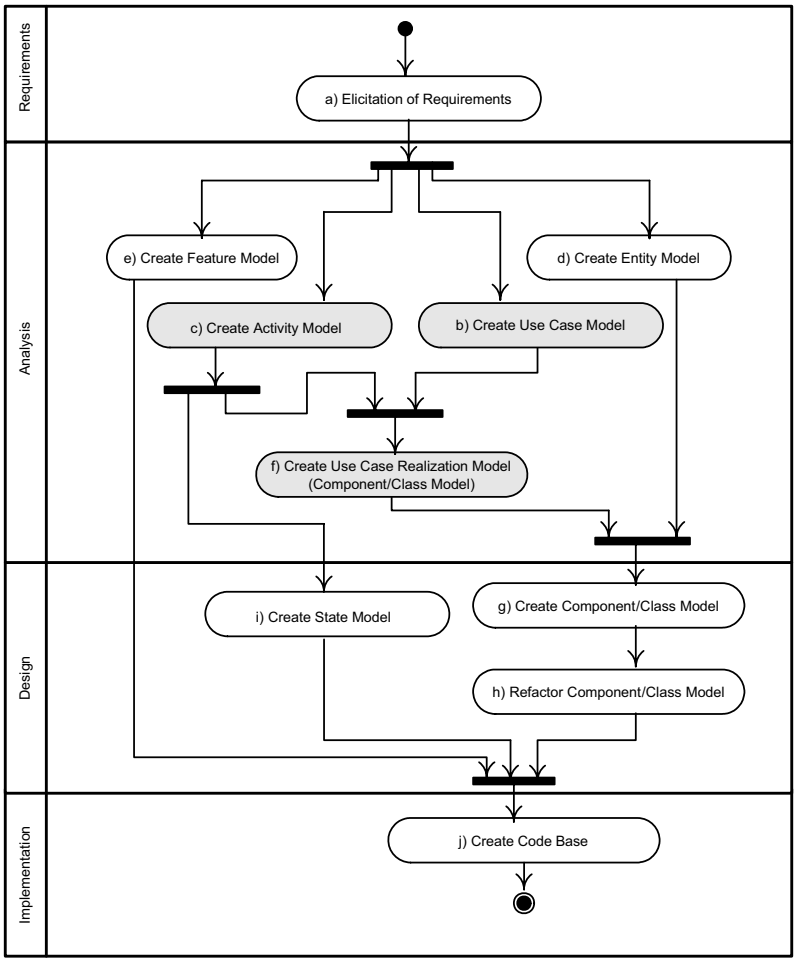

Figure 65: Development activities covered in Section 4.2.

4.2.3 From Problem to Solution Domain

This section describes the activities *b*, *c* and *f* of Figure 65, with the adoption of the UML 2.0 metamodel extensions proposed in 4.2.2 and the MoDeLine method applied for the use cases `Renew Loan` and `Handle Gold Member`, as presented in Figure 61 and Figure 62.

4.2.3.1 Creation of Use Case Model and Activity Model

The activity of the creation of the use case models is essential in the MoDeLine method, since use cases drive the creation of a very significant number of design elements, i.e., the design elements that are derived from functional requirements.

To illustrate this process we will describe how use cases textual descriptions with specifications similar to the ones described in [Cockburn 2001], can be modeled with our approach. Figure 61 and Figure 62 are cases of such textual specification.

Use case specifications usually contain main flow descriptions and alternative flows. For the construction of the use case models it is usual to address first regular use cases (i.e., non-extending use cases) and *Include* relationships. As the use case model is constructed, it is possible to start also developing the activity diagrams. For each regular use case, usually a single activity diagram is sufficient. According to the metamodel of Figure 64, there should be an activity diagram for each *Behavior* (*ownedBehavior* or *classifierBehavior*) of the use case. Since the *classifierBehavior* specifies the main flow of a use case, and main flow alternatives can also be specified in the *classifierBehavior*, a single activity diagram is sufficient for the majority of the use cases.

Activity Nodes. The construction of the activity diagrams is relatively straightforward. The base idea is that each step of the use case textual description becomes an *ActivityNode* in the activity diagram. Each *ActivityNode* refers or is performed by the system or an actor. As such, we adopt UML 2.0 *ActivityPartitions* associated with *ActivityNodes* to identify «who» is related to the *ActivityNode*. For instance, step 1 of Figure 61, "*The Librarian enters the renew loan data (user ID and Item ID)*" becomes the *Action* (*ActivityNode*) `Enter Renew Loan Data` associated with the *ActivityPartition* `Librarian`. Figure 67 presents the activity diagrams correspondent to the flows of use cases `Renew Loan` (Figure 61) and `Handle Gold Member` (Figure 62).

Decision Nodes. An alternative flow implies a *DecisionNode* in the activity diagram. The alternative flow "*2a. Loan does not exist (after step 2)*" of Figure 61 is transformed into the *DecisionNode* `Check if Loan Exists`. This kind of *DecisionNode* has usually two outgoing edges. One corresponds to the main flow and is traversed when the condition for the alternative flow is false. The other corresponds to the alternative flow. Usually, decision nodes in UML 2.0 are depicted with a diamond-shaped symbol. We represent all activity nodes in a uniform way. To identify control nodes we represent their symbols as small icons within the right side of the node visual symbol. This makes it possible to attach more information to control nodes (such as stereotypes and partition names), making their visual representation more meaningful.

Object Nodes. A very important aspect of using activity diagrams to model use case behavior is that it is possible to represent object nodes and their flow. The process of identifying the objects that are used as parameters of actions and behaviors can provide significant input to the entity model of the domain (see activity *Create Entity Model* of Figure 65). Another aspect of object nodes and parameters is that they provide an effective way to validate *Include* and *Extend*

relationships, since the parameters of the sources and targets of these relationships must be compatible. Also, when we reason about conditions for alternatives and *Extend* relationships, object nodes makes it possible to do it in a formal way, because they provide a way to constraint the modeler to only refer to objects that are accessible from the specific location of the condition in the activity diagram. These are all validations that are possible in our proposed metamodel.

***Include* relationship.** An *Include* relationship in which use case A includes use case B, means that there is a *CallBehaviorAction* node is use case A that *calls* the *classifierBehavior* of use case B. This means that the main flow of use case B is included by the *CallBehaviorAction* node of use case A. The parameters of the *CallBehaviorAction* of use case A must be compatible with the parameters of the *classifierBehavior* of use case B.

***Extend* relationship.** In order to support all possible alternative flows, our *Extend* relationship becomes significantly more complex than the original UML 2.0 *Extend* relationship. Regarding the extended use case, there are not significant changes. Basically, we only have to specify the extension points. In the case of the activity diagrams, this is done by marking the respective nodes with the stereotype *extension_point*. These become locations that can be used by extending use cases as extension or rejoin points. We maintain the term *extension point* in the base use case to expose nodes that can be used either as outgoing flows or as incoming flows of extending behavior. Maybe a more appropriate term would be *PublicPoint*. Figure 66 presents three examples of *extension_points* for the use case `Renew Loan`: `Collect Fine`, `Get Item Status` and `Renew Loan`.

Figure 66 presents the *Extend* relationship between `Renew Loan` and `Handle Gold Member` use cases and follows our proposal for the visual representation of the *Extend* relationship. This visual representation only differs from the actual notation of UML 2.0 in the contents of the note attached to the *Extend* relationship, since it reflects the new *ExtensionFragment* element of the metamodel (see Figure 64).

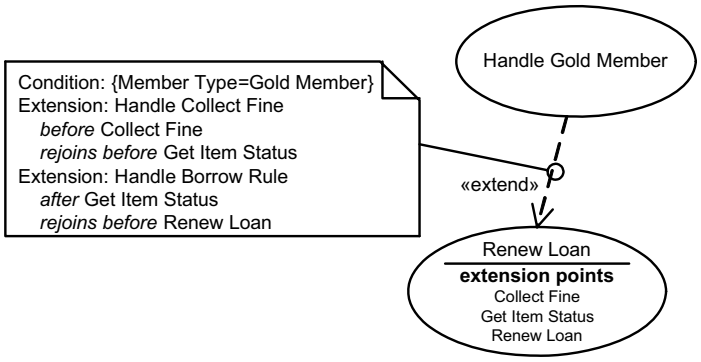

Figure 66: Proposed notation for the *Extend* relationship.

Regarding extending use cases, the *extendingBehavior* that will extend the extended use case in a specific *extensionLocation* has to be specified by an activity diagram, since it is an *ownedBehavior* of the extending use case. Figure 67 presents the activity diagrams that specify the behavior of the two extensions fragments of the *Extend* relationship from the use case `Handle Gold Member` to the use case `Renew Loan`. The figure also highlights the control flow that results from the *Extend* relationship (that is depicted in Figure 66). To clarify the extension and rejoin points of an extension we need to specify the *moment*: *before* or *after*. Since an *ExtensionFragment* has only

one *extensionLocation* this is coincident with the start flow node of the *extendingBehavior*. If the flow of the *extendingBehavior* rejoins the extended use case, this must be modeled in the activity diagram. We do this by using *RefAction* nodes. These are *references* to *ExtensionPoints* in the extended use case (`Renew Loan.Get Item Status`, `Renew Loan.Collect Fine` and `Renew Loan.Renew Loan` in Figure 67). They are marked with the *rejoin_point* stereotype and with a stereotype stating the moment of the rejoin (*after* or *before*).

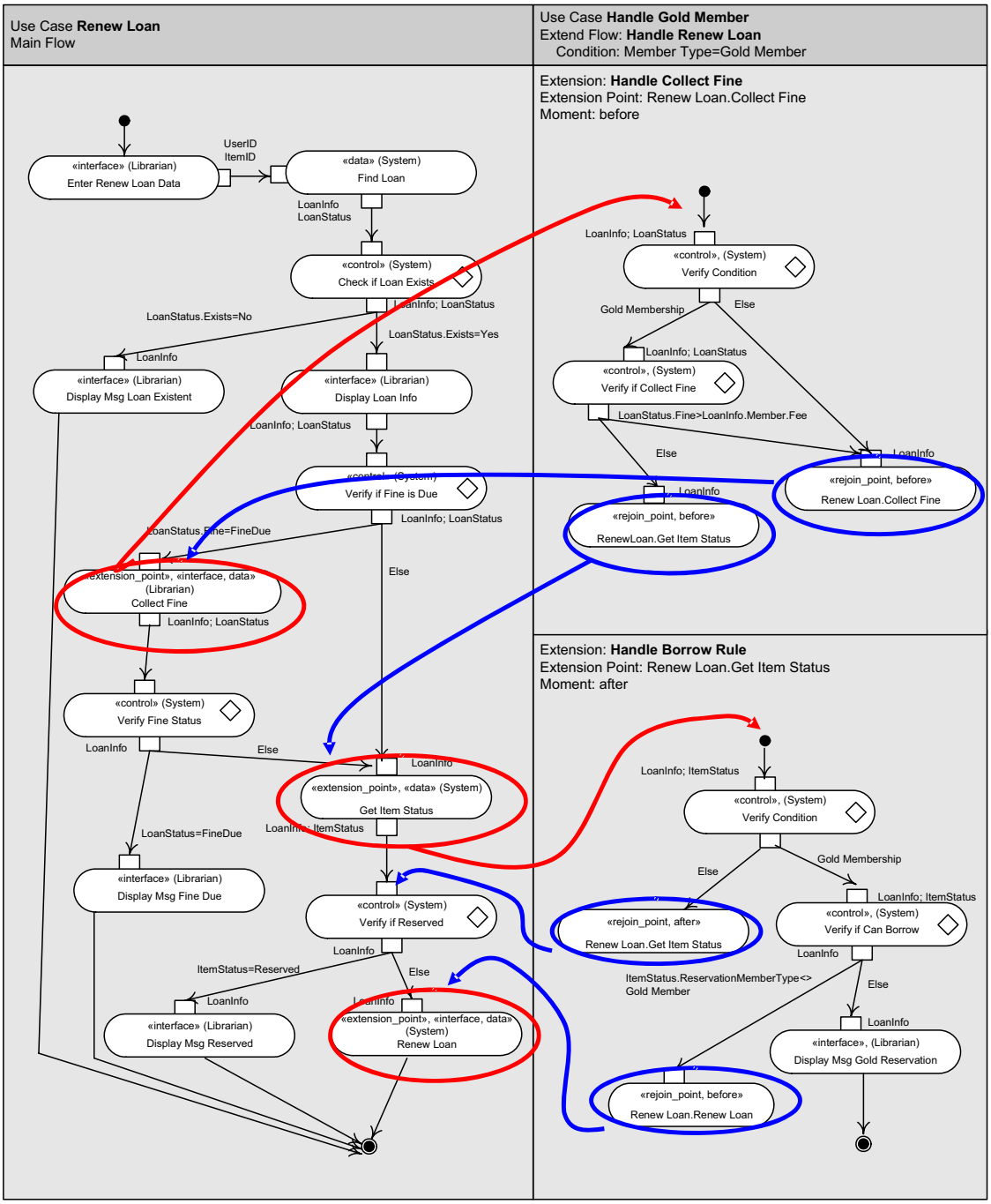

Figure 67: Extension points and rejoin points depicted in activity diagrams for base use case `Renew Loan` and extending use case `Handle Gold Member`.

4.2.3.2 Use Case Realization Model

A use case realization model acts like a 'link' between the problem domain and the solution domain. It has the responsibility of guarantee that all functional requirements of the problem domain are addressed in the solution domain, on a use case by use case basis.

Basically, the method follows a well-know applied practice introduced by [Jacobson *et al.* 1992] and creates three components for each use case: *interface*, *control* and *data* components. This corresponds to step 1 of 4SRS (see Chapter 3). Since, in our approach, each use case is complemented with activity diagrams, it is possible to use these activity diagrams to populate each of the use case components with classes and interfaces that are responsible to realize the behavior associated with the nodes of the activity diagrams. To guide this transformation each node in the activity diagrams must be marked with the following stereotypes: *interface*, *control* and *data*. This facilitates the allocation of the classes and interfaces to the three components that realize the use case.

For a model driven approach to be feasible there must be simple and direct mappings between the models. In this case, the mapping is done between *Actions* of activity models and *Methods* of component/class models. For each *Action* in the activity model we create an *Interface* with a *Method* in the use case realization model and also a *Class* that implements the interface. In the use case realization model these interfaces act like *roles* that are needed in the final system to address the behavior required by the specific use case. Extension, inclusion and rejoin points are realized through required/provided interfaces. Since the components are populated with classes and interfaces, micro-steps *2.i* and *2.ii* of 4SRS (*Component Elimination*) can be automated, i.e., components with no allocated classes and interfaces can be eliminated. Micro-steps *2.iii*, *2.iv* and *2.v* are also addressed at activity *Create Use Case Realization Model*, but they require the human intervention.

The elements that compose each use case realization will be incorporated into a global component/class model during activity *Create Component/Class Model* (see Figure 65). This activity addresses the remaining steps and micro-steps of 4SRS. In this activity, for instance, an interface and method resulting from the action `Get Item Status` could be reused to specify a global interface for a `Book` class. This interface of the `Book` class could also incorporate methods from other use case realizations.

4.3 Extending UML-F for Analysis Models

UML-F has been proposed as an extension to UML for modeling frameworks [Fontoura *et al.* 2000]. Basically, it supports the modeling of framework variation points. Although the product line approach encompasses more concerns than the framework approach, they share many similarities at the technical level. For instance, they both share the principle of reuse and their main concern is the management of variability. The applicability of UML-F to product lines has also been explained in [Pree *et al.* 2002].

Design activities are the UML-F profile focus. UML-F is based on extensions to class diagrams related elements such as classes, interfaces and relationships. We aim at exploring possible directions in order to extend the UML-F to fully support a product line engineering method. In this section, we address requirements and analysis. We present our approach in the context of our experience in using UML-F in the experimental modeling of an insurance product line in which we also adopted the MoDeLine model driven method. Figure 68 presents the development activities that are covered in this section (activities *b*, *c*, *e*, *f* and *g* of Figure 68).

By the time UML-F was proposed, UML 2.0 was not available, so UML-F is based on UML 1.3 and UML 1.4. As the specification of UML 2.0 is now available, this section also addresses the adaptation of UML-F to UML 2.0.

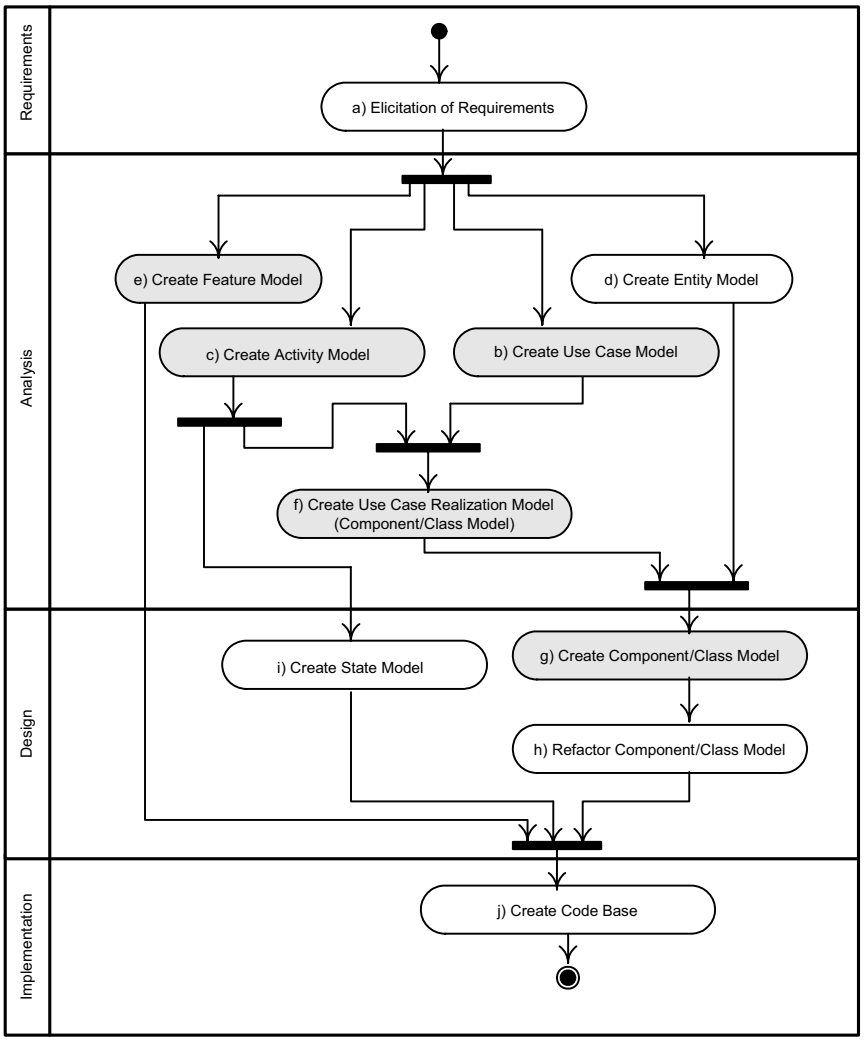

Figure 68: Development activities covered in Section 4.3.

4.3.1 UML-F: Variability at Design

UML 2.0 introduces some changes in the way profiles work. One of such changes is that tagged values can no longer exist by themselves; they exist in the context of a stereotype, in the form of attributes of the stereotype. UML-F is highly based on tags, as they existed in UML 1.3. One simple way to address the transition to UML2 (that is suggested in UML 2.0 official documentation) is to simply create a dummy stereotype to which the unattached UML 1.3 tags could be attached. Nonetheless, in our work, we propose some adaptations to conform UML-F to UML 2.0.

The original UML-F profile proposes three ways of classifying elements (classes, packages and interfaces) as framework or application assets: (1) the *application* stereotype marks an element as being application specific or belonging to a particular element of a product line; (2) the *framework* stereotype marks an element as belonging to the framework or as a common element in a product line; (3) the *utility* stereotype marks an element as belonging to a utility library or to the runtime system.

Regarding variability, UML-F proposes three tags that address three major types of variability points: (1) the tag *variable* is used to mark a method, stating that the behavior of the method varies, and that the method must be implemented during the framework instantiation; (2) the tag *extensible* is used to mark a class, meaning that new methods can be added to the class interface during framework instantiation; (3) the tag *incomplete* is used in a generalization relationship to indicate that it is possible to add new concrete subclasses during the framework instantiation.

Additionally, both methods and classes can be marked with the tags *static* or *dynamic*, which indicate if the variation's instantiation is to be made at compile (*static*) or runtime (*dynamic*). In this case, we propose that *variable*, *extensible* and *incomplete* tags become stereotypes with an enumerate attribute *instantiation* with two possible values: *static* or *dynamic*. Figure 69 presents an example of these stereotypes and respective attributes. This example is based on the motivation example for the template method design pattern presented in [Gamma *et al.* 1995]. In this example, the class `Application` can be extended and three methods must be implemented. The attribute *instantiation* states that the concrete subclasses (i.e., variations) of `Application` are specified until compile time.

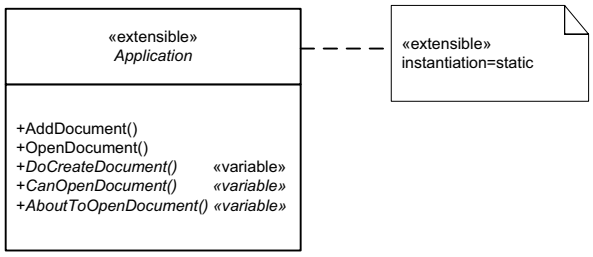

Figure 69: Example of *extensible* and *variable* stereotypes.

Concerning the tag *incomplete*, UML 2.0 already addresses this issue by labeling generalization sets with constraints, so the UML-F profile does not need to add any new elements. Figure 70 presents an example of UML 2.0 constraints labeling a generalization set. In this example, the abstract class `Document` has two framework realization classes (`TextDocument` and `XMLDocument`) and one application specific realization class (`MyDocument`) that is depicted with the *application* stereotype.

UML-F stereotypes provide a way to add more variability semantics to class diagrams. With UML-F it is possible to clearly identify at design the variability points of a system and the correspondent variations. For instance, in Figure 69, it is clear that `DoCreateDocument`, `CanOpenDocument` and `AboutToOpenDocument` are all variation points. Variations of these variability points can be added by implementing those methods in a specialization of the `Application` class. Using this approach of attaching stereotypes to model elements it is possible to add even more semantic to a model. For instance, we could identify which part of the model is affected by the variability points. It is common to classify elements which behavior is affected by

variants as *templates*, in the sense that they provide a template of behavior that is affected by variations at specific spots, also known as variation points or *hooks*.

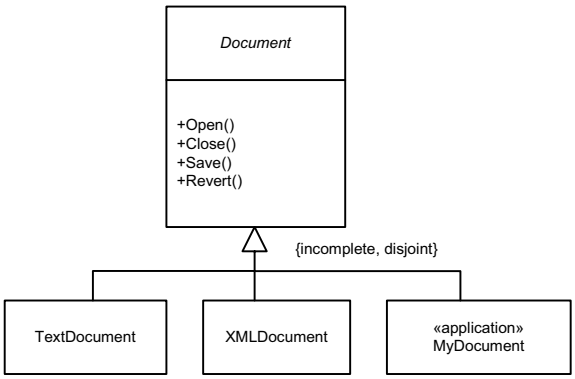

Figure 70: Example of *incomplete* interfaces.

In the example of Figure 69, and according to Figure 71, the method `OpenDocument` is a *template*, in the sense that its behavior is affected by variants attached to the *hooks* (or variation points). Basically, *template* and *hooks* can be in the same class, such as in the examples of Figure 69 and Figure 71 (which in UML-F is called the *unification construction principle*) or they can be in separate classes (which in UML-F is called the *separation construction principle*). When *template* and *hook* are in separate classes it is possible to change the behavior of the *template* at runtime by plugging a specific instance of the *hook*. This case can be illustrated by the example of Figure 70, where we have several variations for `Document`, which acts as a *hook*. By returning a different instance of `Document` in the implementation of the method `DoCreateDocument`, it is possible to change the behavior of the system at runtime. To support the annotation of model elements with the discussed semantics, UML-F proposes two stereotypes: *«template»* and *«hook»*.

```
class MyApplication extends Application {
…
  public void OpenDocument(String name) {
    if (!CanOpenDocument(name)) return;

    Document doc=DoCreateDocument();

    if (doc!=null) {
      docs.AddDocument(doc);
      AboutToOpenDocument(doc);
      doc.Open();
    }
  }
…
}
```

Figure 71: Implementation of the *template* method `OpenDocument`.

Generally speaking, one can say that all design variability is based on a *template/hook* meta-pattern. In fact, *GoF* design patterns [Gamma *et al.* 1995] rely on one or more *template/hook* related elements to provide its functionality. Further discussion of this topic can be found in [Pree *et al.* 2002].

Following this approach, any type of variability point can be implemented based on interfaces and combinations of *templates* and *hooks*. In fact, *variable* methods, *extensible* classes and *incomplete* interfaces semantically mean that it is possible to add or alter the behavior of a system by hooking new behavior in a manner that conforms to the structure and behavior of the remaining system, i.e., the new behavior must follow an interface that is used according to a template.

Since the UML-F approach to manage variability at design is based on the concepts of *templates* and *hooks*, we followed the same concepts when we reasoned about variability in requirements and analysis. The next Section presents the case we use to illustrate our approach to extend UML-F to support requirements and analysis models. We do it by presenting the problem domain and discuss some significant use cases. In sections 4.3.3 and 4.3.4 we explain our approach using examples from our case study. The case study is kept simple in order to facilitate the presentation of the central concepts.

4.3.2 Case Study

To illustrate our approach we use a case study based on an insurance product line. The aim of this product line is to support the business of several insurance companies. Insurance business varies significantly: insurance companies can have a specific line of business, i.e., life insurance or automobile insurance, or they can work in several lines of business; they can insure several types of objects or only insure persons; they can work with agents or not; they can be reinsured; etc. Although diverse, the insurance business has some basic simple concepts. In this Section we will use a simplified view of the insurance domain. Nevertheless, this view is sufficient to illustrate our approach.

The central concept behind an insurance is that a customer pays an insurance company some amount of money (the *premium*) so that the insurance company becomes responsible to pay for some possible lost, that may happen or not (according to the *risk*). The contract that is made between the customer and the insurance company is named policy. Figure 72 presents the use case `{U0.1} Operate an Insurance Policy`. The Figure presents also use cases that result from the functional decomposition of `{U0.1} Operate an Insurance Policy`. Since we will discuss our approach in the context of the MoDeLine model driven method we use its naming conventions. For instance, each use case name is prefixed with a sequence of numbers enclosed in brackets that reflect the *level* of functional decomposition of the use case.

The use case `{U0.1} Operate an Insurance Policy` is a use case at level 1 of the functional decomposition of our insurance product line. Use cases at this level are related to major activities of the insurance business (problem domain). In this case other possible activity could be reinsurance. Usually there are also top level use cases for administrative activities of the system, such as configuration and monitoring. For the purposes of this section we will only focus on `{U0.1} Operate an Insurance Policy`.

When reasoning about functional requirements it is usual to think about the behavior that the system, or parts of the system, has regarding interactions with users or other systems. When doing so it is also common, and easier, to address first the typical behavior of the system and leave the variations to a later moment. So, when applying the MoDeLine method, we start by addressing the common behavior of the insurance system. The results are very high-level use case diagrams that are then functionally decomposed and give origin to more specific use cases. For instance, the `{U0.1} Operate an Insurance Policy` use case can be functionally decomposed into other use cases, like `{U0.1.1} Buy a Policy` and `{U0.1.4} Claim against a Policy`. Within the same level of detail it may also make sense to create new use cases if, for instance, some behavior of a

use case can be reused by other use cases. If this is the case, then a new use case can be created with such behavior and that new use case can be included by the use cases that share that common behavior.

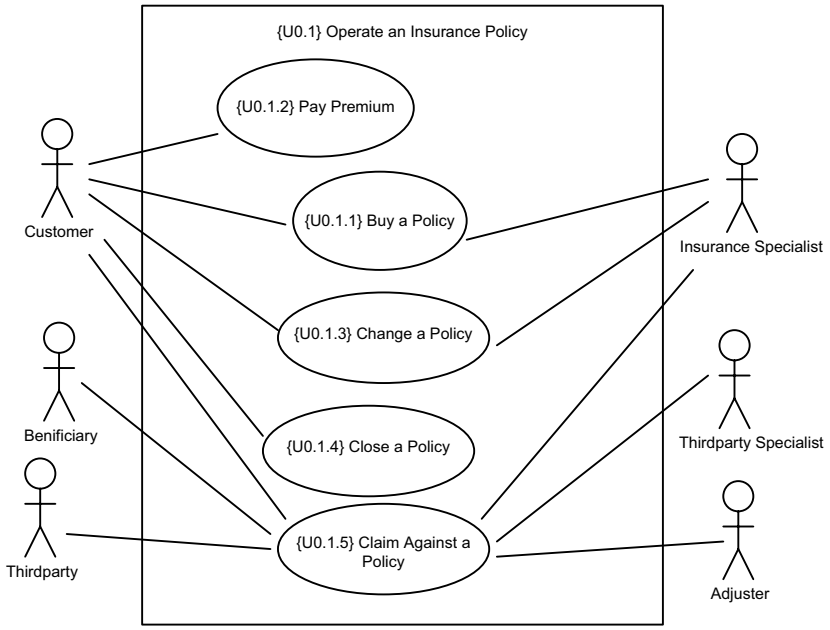

Figure 72: Functional decomposition of use case `{U0.1} Operate an Insurance Policy`.

Figure 73 presents how, for the use case `{U0.1.1} Buy a Policy`, tree new use cases are created: `{U0.1.1.1} Quote for Insurance`, `{U0.1.1.2} Risk Assessment` and `{U0.1.1.3} Pay for Policy`. These three use cases represent behavior that originally belonged to `{U0.1.1} Buy a Policy` but, because they can be reused by other use cases, they give origin to new use cases that are included by the use cases that reuse them.

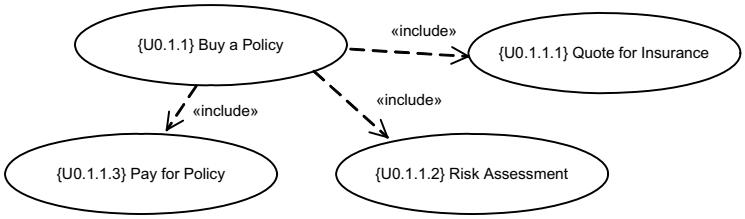

Figure 73: `{U0.1.1} Buy a Policy` functional decomposition.

During this functional decomposition of use cases, a proper characterization of the use cases is required as new use cases are created and existing use cases are modified. This can be done by following suggestions for textual description of use cases, such as the ones described in [Cockburn 2001].

The MoDeLine method only proposes that behavior variability be addressed after the initial functional decomposition of the use cases of the system. This can be done by reasoning on how

variants can affect the common behavior of the system. These variants can result from the addition of a new product in the case of a product line or from considering new variability points in the case of a framework. The next section presents our approach to model variability at requirements in such a way that makes it possible to transform requirements models into analysis models.

4.3.3 Variability at Requirements

One common source of problems in software development is the transition from the problem domain to the solution domain. The transformation is difficult and usually requirements get lost or are misinterpreted. In UML, requirements are modeled using use case diagrams. In order to address this transformation problem we propose that the behavior of use cases should be modeled with activity diagrams. Usually, practitioners describe the behavior of use cases in an informal way, by using textual descriptions. Formalization of use case behavior has been proposed in several previous works, such as [Overgaard *et al.* 1998; Porres 2001; Stevens 2001]. These proposals are all previous to UML 2.0 and do not adopt activity diagrams. In the context of UML 2.0 we propose the adoption of activity diagrams to model the behavior of use cases.

To explain our approach we discuss, in Section 4.3.3.1, some aspects of the use case metamodel that address variability. Section 4.3.3.2 presents how activity diagrams are used to model use case behavior and illustrate it with the insurance demonstration case. We show how variability concepts of requirements can be related to the variability concepts of design that were presented in Section 4.3.1.

4.3.3.1 Use Case Metamodel

Use case diagrams are very simple. This simplicity reflects the fact that they are used for modeling user requirements. According to the UML 2.0 specification, a use case is "the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system", i.e., it represents a declaration of an offered behavior of a system.

Typically, the behavior that use cases offer to actors is described by regular text [Cockburn 2001]. This is mainly because text is a good mean of communication with non-technical users of the system. Nonetheless, in order for transformation between requirements and analysis or design to become feasible, use case behavior need to be specified in a more *formal* way. UML has been introducing some formalization into use cases as the result of the revisions of the standard. But, what forms of variability are possible to formalize with use cases? In order to adopt use cases in a model-driven approach, as in our demonstration case, it is necessary to clearly specify the open/unclear semantics of use cases in UML 2.0. Next we will address this specific question.

***Extend* Relationship**
One of the results of the evolution of UML is the formalization of extending use cases. In UML 2.0, a use case can extend other use case through an *Extend* relationship. According to the UML 2.0 specification, "the extending use case defines a set of modular behavior increments that augment an execution of the extended use case under specific conditions". This definition clearly states that *Extend* relationships can be used to specify variability, since there are conditions associated with them. However, as we address variability at requirements, we find that to fully understand what the *Extend* relationship really means it is necessary to investigate these three topics, extracted from the previous definition:
a) *a set of modular behavior increments*
b) *that augment an execution of the extended use case*

c) *under specific conditions*

According to the UML 2.0 specification, an extending use case is not a regular use case, i.e., the extending use case typically defines behavior that may not necessarily be meaningful by itself, since it defines modular behavior increments that extend the base use case. These extensions occur at *extension points* of the base use case. An *extension point* identifies a point in the behavior of the base use case where that behavior can be extended by the behavior increments of the extending use case. The UML 2.0 specification does not force any specific definition for the location of the *extension point*. However, since a *UseCase* is a generalization of *BehavioredClassifer*, it inherits its members *classifierBehavior* and *ownedBehavior*. They are both of type *Behavior*, which is an abstract class. The *classifierBehavior* represents the behavior of the classifier itself. Since *UseCase* inherits from *BehavioredClassifier*, it is acceptable to say that *classifierBehavior* represents the behavior of the *UseCase* and that *ownedBehavior* represents other behaviors of the *UseCase* (which can be invoked by the *classifierBehavior*). Also, according to UML 2.0, an *Activity* is a specialization of a *Behavior*. In the light of these UML 2.0 specifications, it is acceptable to say that the usual text description of the behavior of a use case can be substituted by activity diagrams. In the next Section we will exemplify in detail how this can be done.

Since we are now assuming that the behaviors of use cases can be modeled by activity diagrams it is possible to clarify the previous doubts. Regarding topic *a)* and *b)*, a behavior increment refers to an *ownedBehavior* of the extending use case (that is realized by an activity). This extending behavior occurs at the extension points of the extended use case. These extension points correspond to nodes of activities that realize the behavior of the extended use case.

Regarding topic *c)*, and according to UML 2.0 metamodel, an *Extend* relationship has a condition that must be true for the base use case to be extended at the *extension points*. Using well-known concepts of the product line field, *extension points* can act as *variation points* and extending use cases act as *variants*. It is possible to bind a *variation point* to a *variant* at several stages in the development cycle [Gurp 2003]. Hence, the condition of the *Extend* relationship can have different realizations according to the stage when the condition is evaluated. If the condition is to be evaluated at runtime, then its operands can be based on the available values, i.e., the values of the parameters of the behaviors. If we are resolving the *variation point* at a pre-runtime phase (which is common in product lines and frameworks) then it is possible to use other operands like, for instance, configuration values and feature options.

Figure 74 represents an excerpt of the MoDeLine metamodel where it is possible to observe the previously discussed concepts and also some proposed UML-F stereotypes that facilitate the modeling of those concepts. The figure also presents how modeling elements from distinct development levels relate to each other. This figure is different from the one presented in Chapter 3 because here we want to describe how UML-F stereotypes can be applied in the standard UML 2.0 metamodel. Therefore, contrary to Chapter 3, here we do not show the MoDeLine proposed extensions to the UML 2.0 metamodel.

***Include* Relationship**

According to the UML 2.0 specification, an *Include* relationship between two use cases *A* and *B* means that the behavior defined in the *B* use case is included in the behavior of the base use case (*A*). It also states that the included use case is not optional, and is always required for the included use case to execute correctly.

Include relationships are used to extract common parts of use cases and therefore represent behavior that is reused across use cases. It can be also used to address functional decomposition in

large systems. According to the same formalization we used for the *Extend* relationship, an *Include* relationship acts like a procedure call. So, in the base use case, some part of its *classifierBehavior* is realized by the behavior of the included use case. In an analogous manner to the *Extend* relationship, in a point in the base use case (the *inclusion point*; a *Node* of an activity that realizes the behavior of the base use case) the behavior (*classifierBehavior*) of another use case is included.

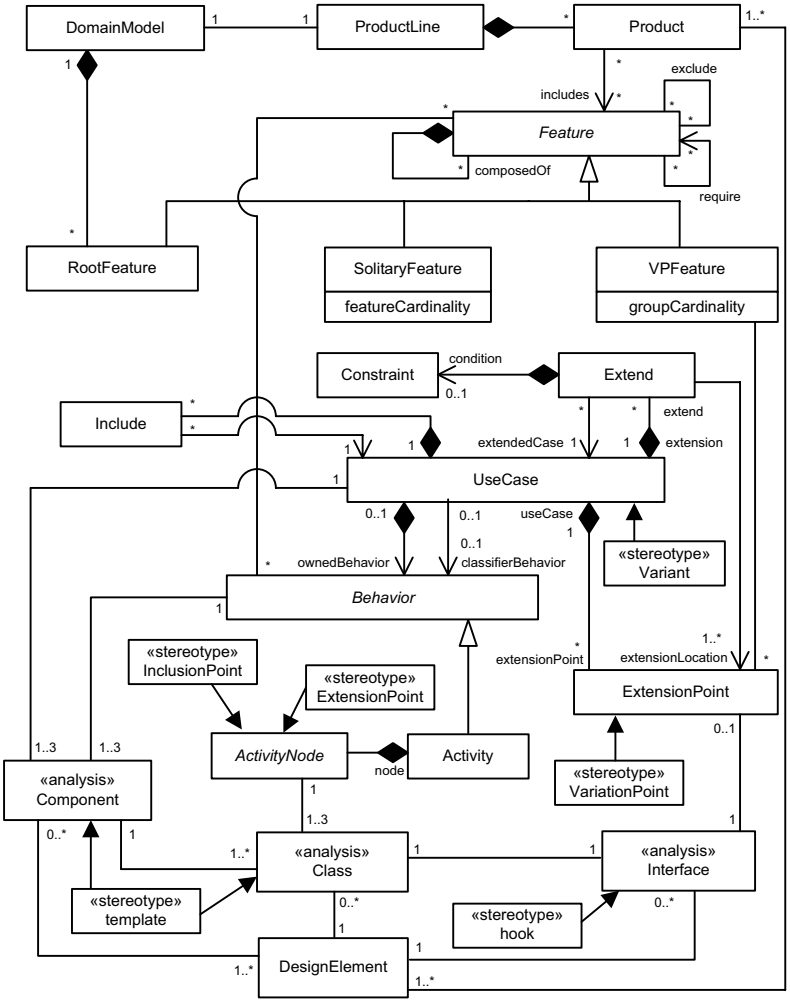

Figure 74: Excerpt of MoDeLine metamodel with proposed stereotypes to support variability.

4.3.3.2 Modeling the Behavior of Use Cases with Activity Diagrams

It is now possible to address behavior variability by reasoning on how variants can affect the common behavior of the system. Variants can result from the addition of a new product in the case of a product line or from considering new variability points in the case of a framework.

Activity diagrams can be constructed from the textual descriptions of the use cases. Basically, from the text description of the use case we identify excerpts that can be modeled as sub-behaviors of the use case. These sub-behaviors become activity nodes in the activity model. The global behavior of the use case (i.e., *classifierBehavior*) can then be modeled by connecting the activity nodes in a way that reflects the flow of behaviors of the use case. One possible way to construct

these activity diagrams is to create first several sequence diagrams of the use case in order to better reason about the global behavior of a use case. Figure 75 presents a simplified version (for clarity reasons) of the activity diagram for the use case `{U0.1.1} Buy a Policy`.

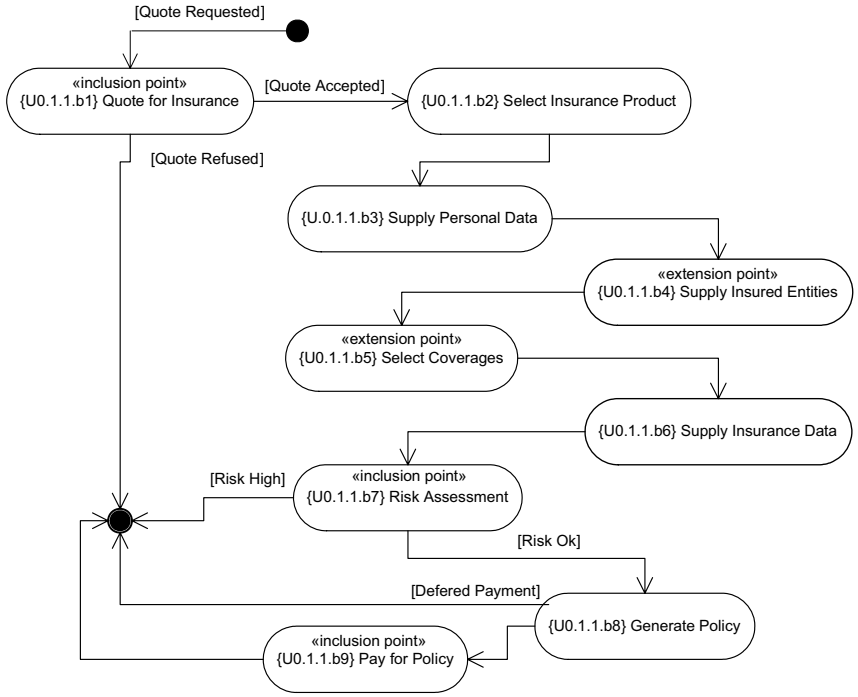

Figure 75: Activity diagram for the *classifierBehavior* of use case `{U0.1.1} Buy a Policy`.

All the *Include* relationships are modeled as activity nodes with the stereotype *InclusionPoint*. This indicates that the *Node* is realized by the behavior of another use case. For instance, the activity node `{U0.1.1b0.7} risk assessment` is an *InclusionPoint* for the *classifierBehavior* of use case `{U0.1.1.2} Risk Assessment`.

Figure 75 also presents how the *Extend* relationship is modeled in activity diagrams. Regarding the use case `{U0.1.1} Buy a Policy`, its behavior could be extended by adding support for different lines of insurance business, for instance, life insurance and automobile insurance. If we do so, the behavior of buying a policy needs to be extended to support the specificities of the new insurance lines. For instance, insured entities in life insurance are persons but in automobile insurance are vehicles. The possible insurance coverages are also different for different insurance lines. Accordingly, the nodes `{U0.1.1b0.4} Supply Insured Entities` and `{U0.1.1b0.5} Select Coverages` are marked with the stereotype *ExtensionPoint*. With this approach we relate the extension points of the use case diagram with the correspondent nodes in the activity diagram. Figure 76 presents this situation by adding the use cases `{U0.1.1e1} Handle Automobile Line` and `{U0.1.1e2} Handle Life line` to the diagram of use case `{U0.1.1} Buy a Policy`.

What we are advocating is the adoption of activity diagrams for modeling the behavior of use cases, either the main scenario (*classifierBehavior* in the use case metamodel) or the alternative scenarios (*ownedBehavior* in the use case metamodel). As we saw, with this approach it becomes possible to clearly model *Extend* and *Include* relationships. Since extension points can act like variation points, they become like hooks at the requirements level. Accordingly, use cases which behavior is affected by these variation points (hooks), i.e. the use case which own the

correspondent *ExtensionPoints,* act like templates. We are identifying the concepts of templates and hooks (and also of possible variants) as soon as possible in the development cycle. These model elements will give origin (after some possible transformations) to the design hooks and templates that were discussed in Section 4.3.1. Usually, they are first traced into analysis elements. The next section briefly presents how these elements are transformed into analysis concepts.

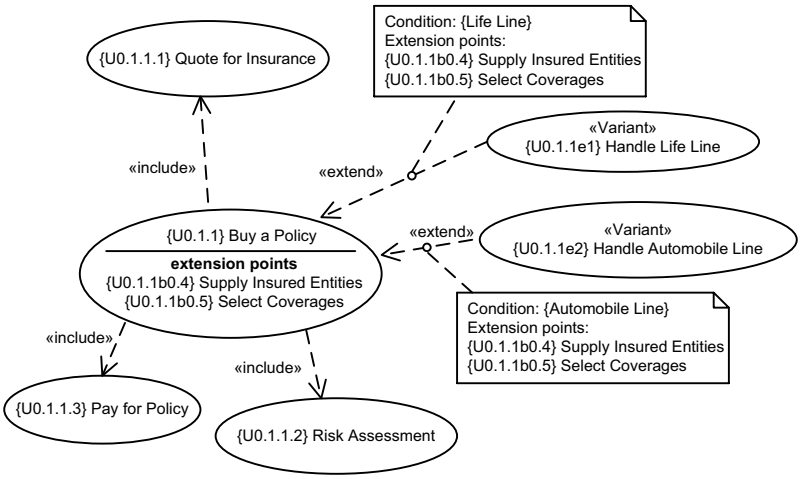

Figure 76: *Extend* relationships of use case `{U0.1.1} Buy a Policy`.

4.3.4 Variability at Analysis

Bridging from problem space to solution space is a very difficult task that requires a lot of experience. The MoDeLine method proposes a series of transformation steps to support this task. Further details regarding this topic were discussed in Chapter 3. To facilitate this transformation we adopt use case realization models. As presented in Chapter 3, use case realization models are analysis component models that realize use cases. In the case of product lines two other diagrams are common: feature diagrams and entity diagrams (in this section we will not address entity diagrams). In order to construct feature diagrams we follow the same approach as [Griss *et al.* 1998]. Feature diagrams are constructed usually after identifying variability in use cases. According to the metamodel of Figure 74, a *Feature* is realized by one or more behaviors of use cases. A *VPFeature* (Variant Point Feature) is a feature that corresponds to optional or alternative behavior. So it relates to one or more *ExtensionPoints* with stereotype *VariationPoint*. With this simply mappings it becomes possible to construct a feature model and trace between features and elements of use case diagrams.

Figure 77 presents a feature model of our case study that follows the notation proposed by [Deursen *et al.* 2002]. In this case, the feature *InsuranceLine* relates at least with the *ExtensionPoints* `{U0.1.1b0.4} Supply Insured Entities` and `{U0.1.1b0.5} Select Coverages`. The feature *LifeLine* is realized by at least the behavior of the use case `{U.0.1.1e1} Handle Life Line`. Other extension points and behaviors could be related to these features if we considered all the use cases of the product line.

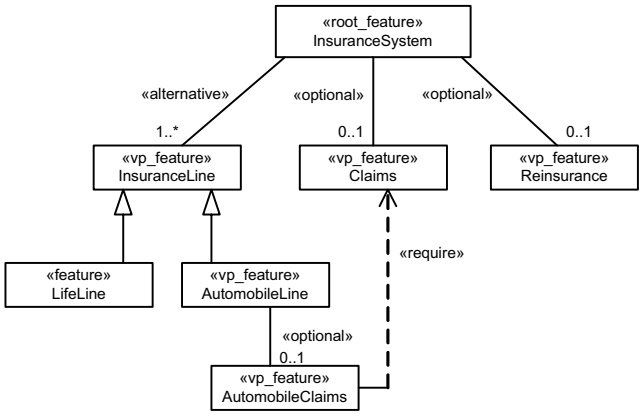

Figure 77: Simplified feature diagram for insurance product line.

One common attribute of a feature is the binding time, i.e., the moment (compile-time, runtime, setup-time, etc.) when a variant point feature is bound to a specific variant. This attribute can be used as a hint for the instantiation attribute of the design variability stereotypes discussed in Section 4.3.1.

In MoDeLine, behaviors of use cases are realized by at most three collaborating analysis components. This follows the well-known *interface-control-data* analysis heuristic. Each *ActivityNode* is also realized by at most three analysis classes and each analysis class implements one analysis interface. Activity nodes with the stereotype *VariationPoint* become analysis interfaces with the *hook* stereotype. Accordingly, the correspondent analysis components can be annotated with the *template* stereotype. With this approach we annotate the analysis elements with the same stereotypes used to annotate UML-F design elements. It becomes possible to transform analysis elements into design elements and keep the trace into requirements and features. Figure 78 presents how elements of the *interface* analysis component `{C0.1.1.i} BuyAPolicyUI` (that collaborates in the realization of use case `{U0.1.1} Buy a Policy`) relate to elements of the abstract design class `PolicyUI`. In Figure 79, the abstract design class `PolicyUI` is transformed according to the *separation construction principle* (discussed in Section 4.3.1). This could be done in order to support a possible binding time of type *runtime* attached to the *InsuranceLine* feature that is traceable to the *hook* methods of `PolicyUI`.

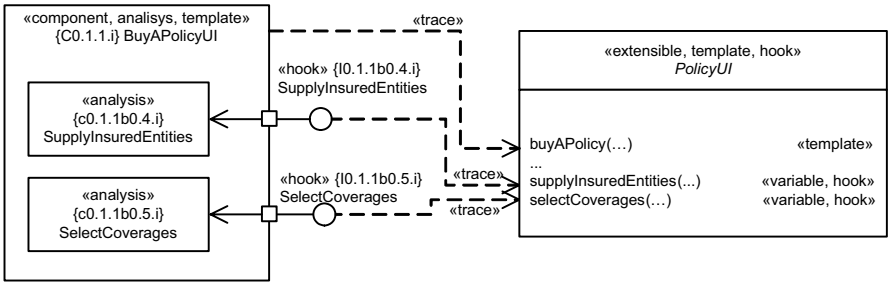

Figure 78: Example of traces between analysis and design elements.

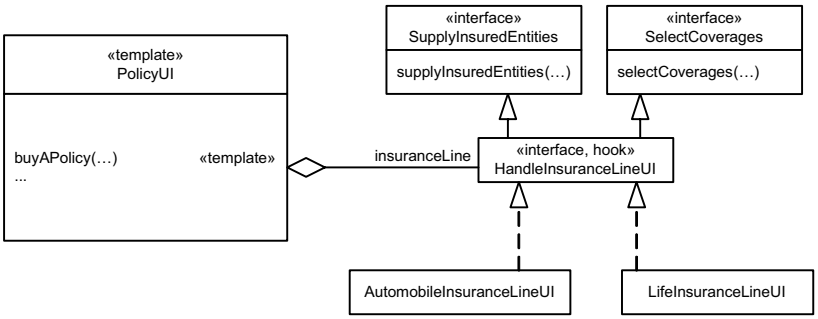

Figure 79: Example of applying the *separation construction principle*.

4.4 Conclusion

A generalized adoption of product line approaches can only become a reality if supported by model driven methods. In order to accomplish this goal, model driven methods should incorporate support for all phases of the software development process, including the analysis phase and requirements models.

In the first half of this chapter we have proposed extensions to the UML 2.0 metamodel so that use case models can be effectively adopted in model driven methods to develop variability focused systems. Our proposal is based on UML 2.0 use case models and on the previous work related to the MoDeLine method that was discussed in Chapter 3. We have identified and discussed the UML 2.0 metamodel restrictions regarding the *Extend* relationship, particularly the issue that it only supports *alternative insertions*. In that context, we have proposed a complementary extension to the UML 2.0 metamodel since it adds support for new types of alternative flows.

We have also discussed how to transform the proposed use case models into their respective realization models. Within the example, we have also discussed how use case realization models could play an important role in supporting an incremental approach to system design which provides an effective way to maintain the trace to requirements and enables model driven development methods in which requirements models are first class citizens.

The first part of this chapter contributes to the research field with an extension to the UML 2.0 metamodel so that use case models can be adopted to model variability intensive systems and support the following kinds of alternatives: alternative insertion; *alternative history; use case exception; alternative fragment; and alternative cycle.*

UML-F is a UML profile for modeling frameworks and product lines, which major concern is variability. Originally its focus was design level diagrams. In this chapter we have presented a proposal to extend UML-F to also support requirements and analysis diagrams. We have discussed our proposal in the context of an insurance product line case study and the MoDeLine model driven development method. We have also described how UML-F could be upgraded to comply with UML 2.0. Even if we manage to model variability at different levels of abstraction, a major problem remains: how to address traceability and transformations between levels. We have also presented an approach in the context of the MoDeLine method.

The second part of this chapter contributes to the research field with an extension to the UML-F profile so that it supports the modeling of variability in requirements and analysis models

and maintains an integrated trace of the 'hook' and 'template' concepts throughout the analysis and design phases of the software development process.

However, for a method to be adopted in a useful manner by practitioners it usually requires its support by tools. This is even truer in the context of the complexity intrinsic to software product lines. In the next chapter, we address this issue by exploring modeling, metamodeling and model transformations in a more *formal* way. We use the term *formal* meaning clear specifications that can be supported by existing tools as opposed to *formal methods*[3].

4.5 References

[Cockburn 2001] Cockburn, A., *Writing Effective Use Cases*: Addison-Wesley, 2001.

[Deursen *et al.* 2002] Deursen, A. v. and P. Klint, "Domain-Specific Language Design Requires Features Descriptions," *Journal of Computing and Information Technology*, vol. 10, pp. 1-17, 2002.

[Eriksson *et al.* 2005] Eriksson, M., J. Borstler and K. Borg, "The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations," SPLC2005, Rennes, France, 2005.

[Fantechi *et al.* 2004] Fantechi, A., S. Gnesi, G. Lami and E. Nesti, "A Methodology for the Derivation and Verification of Use Cases for Product Lines," SPLC2004, Boston, 2004.

[Fontoura 1999] Fontoura, M., "A Systematic Approach to Framework Development," in *Computer Science Department*. Rio de Janeiro: Pontifical Catholic University, 1999.

[Fontoura *et al.* 2000] Fontoura, M., W. Pree and B. Rumpe, "UML-F: A Modeling Language for Object-Oriented Frameworks," ECOOP 2000-Object-Oriented Programming Conference, 2000.

[Gamma *et al.* 1995] Gamma, E., R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[Gomaa 2005] Gomaa, H., *Designing Software Product Lines with UML*: Addison Wesley, 2005.

[Greenfield *et al.* 2004] Greenfield, J., K. Short, S. Cook and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*: Wiley, 2004.

[Griss *et al.* 1998] Griss, M. L., J. Favaro and M. d'Alessandro, "Integrating Feature Modeling with the RSEB," Fifth International Conference on Software Reuse, Victoria, Canada, 1998.

[Gurp 2003] Gurp, J. v., "On the Design & Preservation of Software Systems," in *Computer Science Department*. Groningen: University of Groningen, 2003.

[3] In computer science and software engineering the term *formal method* is usually applied to denote mathematically-based techniques for the specification, development and verification of software and hardware.

[Hurlbut 1998] Hurlbut, R., "Managing Domain Architecture Evolution Through Adaptive Use Case and Business Rule Models," in *Graduate College*. Chicago: Illinois Institute of Technology, 1998.

[Jacobson *et al.* 1992] Jacobson, I., M. Christerson, P. Jonsson and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*: Addison-Wesley, 1992.

[Kang *et al.* 1990] Kang, K. C., S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study Technical Report," Software Engineering Institute, Carnegie Mellon University CMU/SEI-90-TR-21, 1990.

[MDA 2007] OMG, "Model Driven Architecture Guide Version 1.0.1," Available at http://www.omg.org, 2007.

[Metz *et al.* 2004] Metz, P., J. O'Brian and W. Weber, "Specifying Use Case Interaction: Clarifying Extension Points and Points of Rejoin," *Journal of Object Technology*, 2004.

[Overgaard *et al.* 1998] Overgaard, G. and K. Palmkvist, "A Formal Approach to Use Cases and Their Relationships," «UML»98: Beyond the Notation, Ecole Superieure des Sciences Appliques pour l'Ingenieur - Mulhouse, Universite de Haut-Alsace, France, 1998.

[Porres 2001] Porres, I., "Modeling and Analysing Software Behavior in UML," in *Department of Computer Science*. Turku, Finland: Abo Akademi University, 2001.

[Pree *et al.* 2002] Pree, W., M. Fontoura and B. Rumpe, "Product Line Annotations with UML-F," Software Product Lines - Second International Conference, SPLC 2, San Diego, 2002.

[QVT 2005] OMG, "MOF QVT Final Adopted Specification (ptc/05-11-01)," Available at http://www.omg.org, 2005.

[Stevens 2001] Stevens, P., "On Use Cases and Their Relationships in the Unified Modelling Language," FASE'01, 2001.

[UML 2005] OMG, "Unified Modeling Language Version 2.0: Superstructure (formal/05-07-04)," Available at http://www.omg.org, 2005.

5. Formal Model Transformations

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."
Sir Charles Antony Richard Hoare

Chapter 5 is dedicated to formal model transformations. The first half of the chapter presents a proposal of mappings between use cases and feature diagrams. It also presents how these mappings can be supported by the QVT operational language. The second half of the chapter presents a proposal to implement multi-staged software development in the context of model driven and software product lines. This is also one of the scenarios of usage for software factories.

5.1 Introduction

Features have been widely used by the product line community to model variability. They represent the common and variable characteristics of the members of a product line. They are very well suited for the configuration of product line members. Outside the product line community, use cases are also widely used to model the functionality of systems at a similar level of abstraction but from a user perspective. Significant work has been done by several authors regarding the possible relationship between these two perspectives of a system. Nonetheless, this has been done in an informal way. In the first half of this chapter we explore the relationships between these two perspectives and describe a possible approach to automate the transformation from UML [UML 2005] use case to feature models.

Model driven approaches are shifting software development from a code based activity to a model based activity. Models can be refined and transformed from requirements into code specific to a platform. Although several model transformations can occur, they usually take place at a single software development stage. In the case of software product lines, and particularly of software factories, the modeling of a system can occur at several stages, for instance, at the software-house, at the systems integrator and at the final customer site. Basically, this requires that the model used at a particular stage can be refined at the next stage. In the second half of this chapter, we explore the issues related to such an approach and we propose model transformation patterns that can be generically applied to models so that they can be used in multi-staged modeling approaches. We show how to realize the approach with EMF (Eclipse Modeling Framework) [EMF 2007] and present an insurance case study.

To contextualize the proposal discussed in this chapter we briefly discuss model transformations in the next section.

5.2 On the Transformation of Models

Model driven engineering is a promising approach to software development that could become the mainstream paradigm for software development in the near future [Bezivin 2005]. In this new paradigm, models play the central role, as the code does for traditional approaches. Models are used to construct abstractions of the system at several levels and from different perspectives. Models at higher abstraction levels can be transformed into models at lower abstraction levels and, eventually, models are transformed into code that can be executed by a specific platform. Usually, this is done at a single stage. For instance, a software-house can apply this approach to build its software packages. In the case of software product lines, and particularly of software factories, the modeling of a system can occur at several stages, for instance, at the software-house, at the system integrator and at the final customer site. Generically, one can say that in this case, the software system can be specialized at all the tiers (or stages) of the supply chain. Such scenario requires that the models used at a particular stage can be refined at the next stage.

The Model Driven Architecture (MDA) is the Object Management Group (OMG) approach to model driven development [MDA 2007]. At the core of this architecture is the Meta Object Facility (MOF) standard [MOF 2006]. MOF provides a metadata management framework and a set of metadata services to enable the development and interoperability of model and metadata driven systems. Figure 80 presents an example of the MOF metadata architecture for UML [UML 2005]. The figure represents the relationships between models at different levels of the MOF architecture. This figure also represents very well the metadata architecture for single-staged software development approaches.

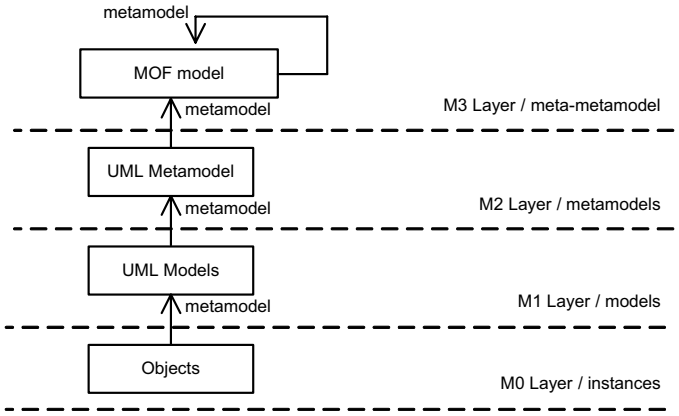

Figure 80: Example of MOF metadata architecture.

Model transformations can be classified as [Mens *et al.* 2005]: number of source and target models; technological space; endogenous versus exogenous transformations; and horizontal versus vertical transformations. Figure 81 presents a schematic representation of the elements involved in model transformations that can be used to illustrate the discussion of the classifications of model transformations.

Number of source and target models

We will adopt the following definition of model transformation [Kleppe *et al.* 2003]: "*A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together*

describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language."

We can generalize this definition and say that in a model transformation one or more input models are transformed into one or more output models. In the case the source and target models are the same we are in the presence of a refactoring of the source model[4].

Model refactoring can take place on two levels [Biermann *et al.* 2006a] (See Figure 80):
1) Refactoring rules are typed over the metamodel (M2) and are applied to models (M1);
2) Refactoring rules are typed over some model (M1) and refactor instance models (M0).

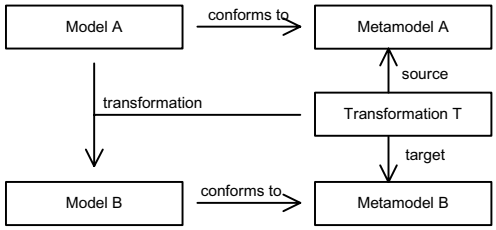

Figure 81: Model transformations.

Technological space
The source and target models of a model transformation may belong to the same or to different technological spaces [Bezivin *et al.* 2003]. Two models share the same technological space if they have the same meta-metamodel (M3-level). For instance, in Figure 81, if *Metamodel A* and *Metamodel B* have the same metamodel, then the technological space is the same. Models form different technological spaces are hard to combine or work in an integrated manner because their technological support is different. Examples of different technological spaces are EMF and the DSL Tools for Visual Studio [DSL 2007].

Endogenous versus exogenous transformations
Endogenous and exogenous are terms used to classify transformations based on the language in which the source and target models are expressed. Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations are transformations between models expressed using different languages. In Figure 81, if *Metamodel A* and *Metamodel B* are the same metamodel, then the source and target models share the same language and, therefore, the transformation is endogenous.

Horizontal versus vertical transformations
A horizontal transformation is a transformation where the source and target models reside at the same abstraction level. Possible examples are refactoring (an endogenous transformation) and migration (an exogenous transformation). In vertical transformations, the source and target models are at different abstraction levels. A typical example is refinement, where a specification is

[4] If we take to the extreme the model driven approach, this is similar to source code refactoring, since a program source code can be viewed as a model that is conformant to its metamodel – the grammar of the programming language.

gradually refined into lower abstraction levels, by means of successive refinement steps that add more concrete details.

Table 5 illustrates that the dimensions horizontal versus vertical and endogenous versus exogenous are orthogonal.

Table 5: Orthogonal dimensions of model transformations.

	horizontal	vertical
endogenous	*Refactoring*	*Formal refinement*
exogenous	*Language migration*	*Code generation*

Czarnecki *et al.* applied a domain analysis approach to elaborate an in deep classification of model transformation by using feature models [Czarnecki *et al.* 2003]. In their work, the top level features that characterize a transformation approach are: Transformation rules; Rule application scoping; Source-target relationship; Rule application strategy; Rule scheduling; Rule organization; Tracing; and Directionality.

According to those features, they identify the following major categories for existing model transformation tools:

- Model-to-code approaches (Ex: JET [JET 2007], AndroMDA [AndroMDA 2007]);
- Model-to-model approaches
- Direct-manipulation approaches (Ex: Jamda [Jamda 2007])
- Relational approaches (Ex: QVT [QVT 2005])
- Graph-transformation-based approaches (Ex: VIATRA [VIATRA 2007], GreAT [GReAT 2007])
- Structure-driven approaches (Ex: OptimalJ [OptimalJ 2007])
- Hybrid approaches (Ex: ATL [ATL 2007])

EMF Refactoring
An EMF transformation is a rule-based modification of an EMF source model resulting in an EMF target model [Biermann *et al.* 2006a]. Both the EMF source and target models are typed over an EMF core model which itself is again typed over Ecore. Refactoring can take place on two levels:
1) Refactoring rules are typed over the Ecore model (M2) and are applied to EMF models (M1);
2) Refactoring rules are typed over some EMF core model (M1) and refactor EMF instance models (M0).

5.3 Automating Mappings between Use Cases and Features

In this section, we present an approach to *formalize* the mappings between use cases and features. We do so in the context of the MoDeLine model driven approach and present a possible implementation roadmap based on open source modeling and transformation tools. The activity diagram presented in Figure 82 depicts our approach.

The approach presented in this section is usually performed at a single stage. However, modeling feature configurations can also be performed at several stages [Czarnecki *et al.* 2005b]. We will address multi-stage modeling only in Section 5.4.

Figure 83 presents the development activities that are addressed in this section. In Figure 83, the activities *create use case model* and *create feature model* seem to be independent, while in Figure 82 they are correlated. Both figures are correct since the dependence between use cases and features only exists if we use the automation approach presented in this section and, even so, it is also possible to add features to the feature model that do not have use cases as source.

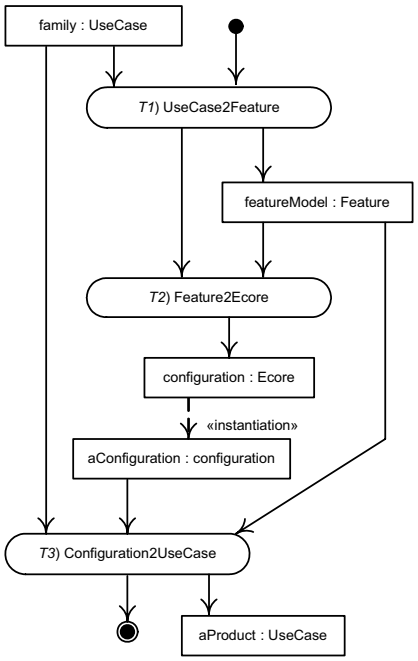

Figure 82**:** Process for obtaining a product use case model from a family use case model.

5.3.1 Feature Models

Feature modeling is widely used to model commonality and variability in software systems, particularly software product lines. Although this is true, implementations vary significantly and there is no common globally accepted metamodel for features. In this section we present our approach to feature modeling that is partially compliant on [Asikainen *et al.* 2006] and [Czarnecki *et al.* 2004].

Figure 84 presents our metamodel for feature diagrams. We adopt the notion that a feature represents a characteristic or property of a system that is relevant to some user or stakeholder. From this perspective, a feature diagram represents the properties or characteristics that a system may have. Since features can be further characterized by subfeatures, a feature diagram is usually represented as a tree of features with adornments that visually represent relationships between the features. The feature at the root of the tree is called *root feature* and it is usually a conceptual feature that represents the whole system. Because features represent characteristics that may (or not) be present in a system, feature diagrams are well suited to represent common (for features that

are always present) and variable (for features that may not be present) characteristics of a system. The process of removing the variability out of the feature diagram (by selecting -or not- optional features) results in the configuration of a system (*feature configuration*). Basically, a feature is said to be *mandatory* if it is included in all configurations. A feature that may not be present in all configurations is called *optional*. *Alternative* features are features that form a group from which they are selected according to some rule (usually the rule states that only one feature of the group can be selected).

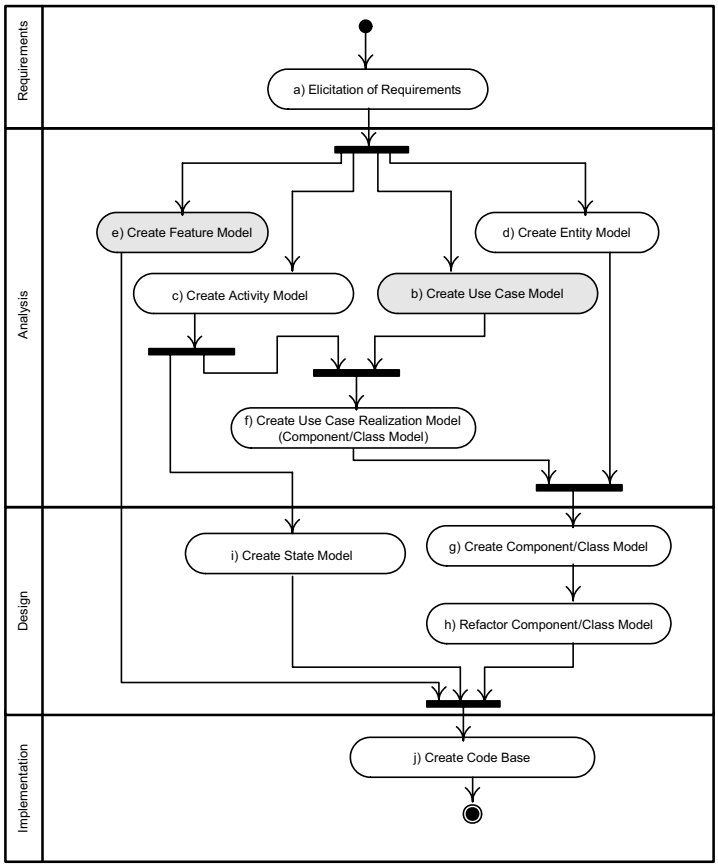

Figure 83: Development activities covered in section 5.3.

Our metamodel for feature diagrams supports all the presented concepts. We use *Subfeature* to represent containment relationships between features. A mandatory feature is a child (*Subfeature*) of some other feature for which the *minCardinality* and *maxCardinality* are 1. An optional feature is a child of some other feature for which the *minCardinality* is 0 and the *maxCardinality* is 1. A feature group can be modeled by a *SubFeature* with the alternative features as *childs* and the specific cardinality of the group stated by *minCardinality* and *maxCardinality*. The *similarity* enumeration is used to state if the selected alternatives must by of the same kind[5]. Since a feature can only be contained be another feature, we use the concept of *Reference* to enable a feature to be referred by several *Subfeature* relations.

[5] The *similarity* concept was proposed by [Asikainen *et al.* 2006] Asikainen, T., T. Mannisto and T. Soininen, "A Unified Conceptual Foundation for Feature Modeling," SPLC2006, Baltimore, 2006..

Figure 85 presents an excerpt of a feature model for a family of library applications. *Features* are represented within rectangles, in a way that is similar to UML classifiers [UML 2005]. *Subfeatures* are represented as links between two or more *Features*. These links can be adorned with the values for the attributes of *Subfeatures*. For instance, `collectFine` is a Subfeature that relates the *parent* Feature `BorrowLoanCopy` with the *childs* `CollectPartialFine` and `CollectTotalFine`. In this case, *minCardinality* and *maxCardinality* are both 1, which means that when configuring a member of the family we must select only one of the *Subfeatures* of `BorrowLoanCopy`. Figure 85 also presents examples of the use of *References*. For instance, a *Reference* is used to state that the feature `RenewLoan` can also have `CollectPartialFine` as a *Subfeature*. *Attributes* can be used to further characterize features. In this example, the feature `BorrowLoanCopy` can be further characterized by the attribute `days` that represent the maximum number of days that a library member can borrow a book.

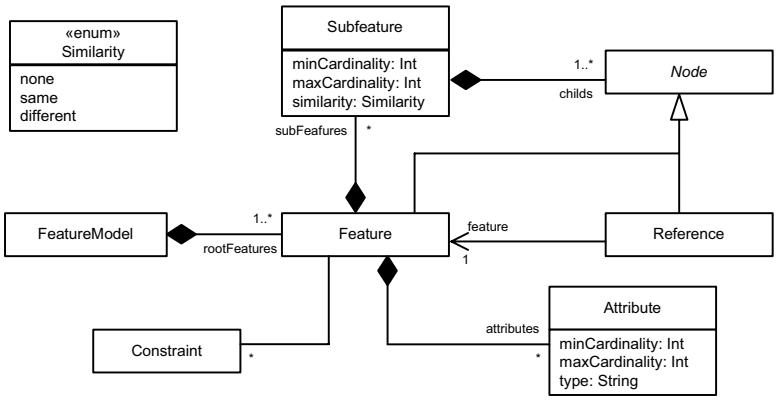

Figure 84: Feature metamodel.

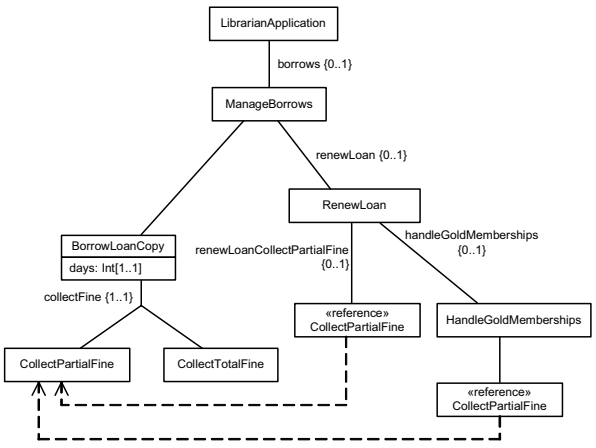

Figure 85: Excerpt of a library feature model.

Finally, it is possible to model dependencies between features using *constraints*. A constraint language, such as OCL, can be used to express these dependencies [OCL 2006]. For instance, Figure 86 presents an example of an OCL constraint that disables the subfeature `renewLoanCollectPartialFine` if the feature `CollectTotalFine` is selected.

```
context ManageBorrows inv:
self.borrowLoanCopy.collectFine
  ->oclIsTypeOf(CollectTotalFine) implies
  self.renewLoan.renewLoanCollectPartialFine
    ->isEmpty();
```

Figure 86: Example of OCL constraint implementing a feature dependency.

A feature model, such as the one presented in Figure 85, represents all the possible features for applications of a family of applications. We *configure* a specific application of the family, by removing all the variability from the feature model. If we follow the analogy that a feature is similar to a classifier, then a *configuration* is achieved by a *valid* instantiation of the features (classifiers). We will elaborate such approach in section 5.3.4, when we describe a possible implementation roadmap.

In the next section, we present and discuss the use case metamodel and in section 5.3.3 we discuss how use cases can be used as a source for feature modeling.

5.3.2 Use Cases

Use cases have been widely adopted since its introduction [Jacobson *et al.* 1992]. They have become an integral part of the UML standard modeling language. Use cases are used essentially for functional requirements modeling, as a source for the initial design of a system and for documentation. However, with the recent model driven approaches, such as MDA [MDA 2007], and the appearance of supporting tools, using computational independent models - such as use cases - as first-class development artifacts can become a reality. However, to achieve this goal with use cases, it is necessary to remove all ambiguities existent in the UML use case metamodel, specially for modeling variability [Maßen *et al.* 2002; Eriksson *et al.* 2005]. In this section we present our approach to achieve that goal. Figure 87 presents an excerpt of the UML 2.0 metamodel that is related to use cases. Our main extensions to the original metamodel are depicted in gray. Next, we explain these extensions.

According to the UML 2.0 specification, a use case is the "specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system". As such, these set of actions represent behavior of a system. As it is possible to observe from Figure 87, a *UseCase* has one *mainBehavior* and can have several *alternativeBehavior*'s. The UML 2.0 specification does not state how the behavior of use cases should be specified but, since our approach needs a formal specification, we will use *activities*. So, each behavior of a use case is specified by an *Activity*.

Use cases can have relationships between them. Basically, a use case can *include* (or be included by) other use cases and can *extend* (or be extended by) other use cases.

The *Include* relationship acts like a procedure call, i.e., at some specific point of a use case the behavior of another use case is executed. We have introduced the *InclusionPoint* element that represents the point in the use case where the inclusion occurs (see Figure 87). The *location* attribute is a reference to a node of an activity that models one of the behaviors of the use case.

On the other end, the *Extend* relationship acts like a deviation of the normal flow of a use case. This deviation is usually conditional, so the base behavior is unaware of the extension. We formalize the UML original notion of extension fragment (*ExtensionFragment* element) and add

the notion of rejoin (*Rejoin* element). As such, if the extend condition is true, the use case behavior is extended at one or more extension points, by the corresponding fragments (which are alternative behaviors of the extending use case). The attribute *location* of the *ExtensionPoint* element is a reference to a node of an activity that models one of the behaviors of the extended use case.

The *rejoinSource* of the *Rejoin* element is a reference to a node of an activity that models the alternative behavior of the fragment of the extending use case. A more deep discussion of the *Extend* relationship can be found in Section 5.3.

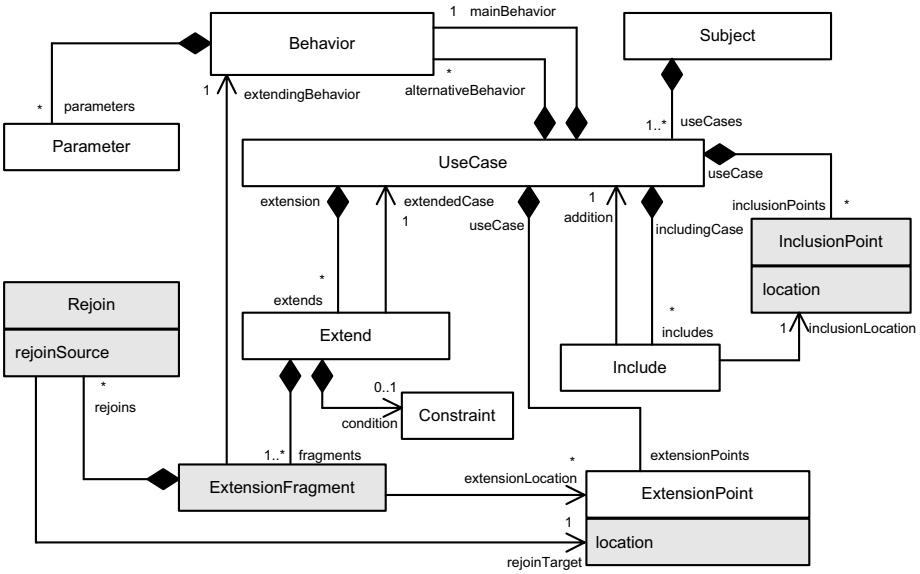

Figure 87: Excerpt of UML 2.0 metamodel relating to use cases.

With these extensions to the original UML 2.0 use case metamodel we remove the existing limitations that restricted its application into model driven approaches. Of course, further specifications can be added, notably constraints to validate the model, such as the ones presented in Figure 88.

Constraints such as the ones presented in Figure 88 can assure that the use case model is well formed. A well formed use case model is also required to help achieving valid results for the transformations that will be discussed next.

```
context Include
inv: self.inclusionLocation.useCase =
       self.includingCase;
inv: self.addition <> self.includingCase;

context Extend
inv: self.extension <> self.extendedCase;
inv: self.fragments->forAll( f |
       f.extensionLocation.useCase =
       self.extendedCase );
inv: self.fragments->forAll( f |
       f.rejoins->forAll( r |
       r.rejoinTarget.useCase =
       self.extendedCase ) );
```

Figure 88: Example of OCL constraints for *validating* the use case metamodel.

In this section we have presented an approach to remove ambiguities from the UML 2.0 use case metamodel. It is now possible to analyze the semantic and syntactic relations between use cases and features. In the next section we present our view on this topic.

5.3.3 Relating Use Cases and Features

The issue of relating use cases and features is not new. Notably, there is the much referenced work of Griss, Favaro and d'Alessandro [Griss *et al.* 1998]. In their work they propose an approach by which functional features are extracted from the domain use case model. They also propose that the structure of the feature model can be created according to the structure of the use case model (by using the «include» and «extend» relationships). As the authors suggest, further types of features can be discovered and added along the development process, such as features resulting from architectural or design modeling tasks. More recent works in this field are also aligned with this approach [Eriksson *et al.* 2005; Gomaa 2005; Jacobson *et al.* 2005]. We also follow this approach, since feature modeling requires an extensive knowledge of the domain, which is only possible after the effective modeling of such a domain. This is true, particularly for the functional features of the domain. So, the initial feature model is build from the domain use case model. In the remainder of this section we will discuss this mapping based on the use case model example of Figure 89. Figure 89 already contains visual annotations that are used to model variability. For the moment we will disregard these annotations. As the figure shows, a library system has functionality that regards to the librarian. The librarian can use it to manage borrows. He can borrow loan copies to library users and also renew loans. Such borrows can be subject to fines if they surpass a certain duration. In the case the user is a *gold member* of the library, special treatment applies.

Use Cases
According to our approach, each use case is mapped to a feature. Top use cases become root features (see feature metamodel in Figure 84). The complete structure of the feature model can only be created by examining the relations between use cases. As such, we cannot say a use case is mandatory or optional without a context. This context results from the relationships the use case has with other use cases. For instance, if the functionality of a use case is always *referenced* by other use cases, then we can say that such a use case is mandatory. This is the case for the top level use case `Interact with Librarian Application`. This use case is an example of an abstract use case, used as an *umbrella* use case in our model driven method. It helps in our functional decomposition of use cases. In that perspective our method shares similarities with the KobrA method discussed in Chapter 2. In other approaches such *kind* of use cases may not be necessary.

Next, we examine each of the use case types of relationships and elaborate on how they can be used to model variability and how they can be mapped to the feature model.

***Include* Relationship**
In the UML 2.0 standard documentation there is nothing to support that the *Include* relationship can be used for modeling variability. The documentation states that "The including use case may only depend on the result (value) of the included use case. This value is obtained as a result of the execution of the included use case". However, in the context of a variability focused system, like a product line, it is common to have alternative or optional includes [Gomaa 2005]. In the example of Figure 89, we have two alternative includes from the use case `Borrow Loan Copy`: one includes the use case `Collect Total Fine` and the other includes the use case `Collect Partial Fine`. In this case, if the two included use cases match the requirements for the inclusion point, no harm is done, since one of them will supply the expected behavior to the including use case.

In the case we need to model only one include as optional, some extra care is needed. If such include does not take place, the result is the suppression of the inclusion point on the including use case. Regarding the behavior of the use case, this results in the removal of the activity node corresponding to the inclusion point. If the node had only one incoming and one outgoing control flow, we simply connect the outgoing flow of the previous node with the incoming flow of the next node. We must also make sure that the output object nodes of the previous node are connect to the input object nodes of the next node and that they are compatible. This situation is presented in Figure 90, where an inclusion point node (a) is removed (b). If such requirements are meet, then it is also safe to have optional includes. More complex scenarios may also be possible but require human intervention or more contextual knowledge from the modeling tool.

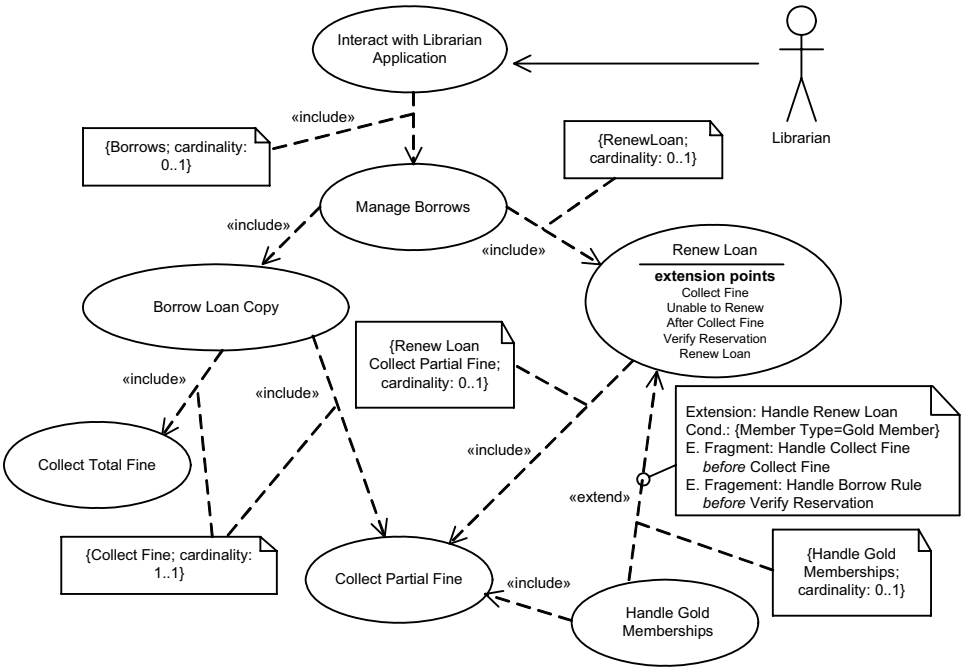

Figure 89: Example of a use case diagram for a Library product line.

Since we are using a metamodeling approach and extending existing metamodels (i.e., the UML 2.0 metamodel) we will also use a metamodeling approach to model variability. In Chapter 4, we used stereotypes to model variability in UML models. Here, we are going to use a complementary approach based on creating a metamodel to support the concept of *variability annotation*. This approach is somewhere between using stereotypes as we did in Chapter 4 and the orthogonal variability model proposed in [Pohl *et al.* 2005].

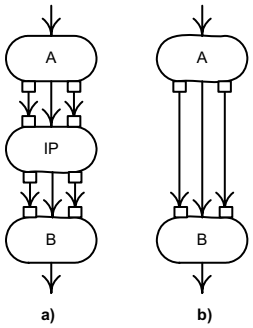

Figure 90: Removing a node from an activity diagram.

In Figure 89, *variability annotations* are represented as notes linked to the *Include* and *Extend* relationships. They represent variability points with a name, a minimum and a maximum cardinality and the respective options. For instance, the variability annotation `Collect Fine`, has a cardinality `1..1` that says that one and only one of the options must be selected. The two options are the includes that related the use case `Borrow Loan Copy` to the included use cases `Collect Total Fine` and `Collect Partial Fine`. Since it is the modeler of the use case domain model that is editing these use cases and relationships, he/she is also capable of making these annotations.

With this variability information annotated in the use case model it becomes possible to map the use case relationships to the feature model. In the case of the *Include* relationship, each include annotation is mapped into a *Subfeature*. The including use case is mapped to the parent feature of the *Subfeature* and the included use cases are mapped to the *childs* of the *Subfeature* (see Figure 84). Since a use case can be referenced by more than one *Include*/*Extend* it can also become a *child* in several *Subfeatures*. Because a feature definition exists only once, a use case is mapped only once to a *Feature* and the subsequent references are mapped to a *Reference* in the feature model.

***Extend* Relationship**
Contrasting with the *Include* relationship, the *Extend* relationship is used to model variability. As we can observe from Figure 87, an *Extend* has an associated condition. If this condition evaluates to true, the use case is extended by the extension fragment's behaviors. On the other end, if the condition is not true, no extension is performed, and the behavior of the base use case remains unchanged and unaware of the extending use case.

In the example of Figure 89, the use case `Renew Loan` can be extended by the use case `Handle Gold Memberships`. As the extend note states, the extension only takes place if the member that is renewing the loan is a gold member. As the example shows, these conditions typically relate to alternative or extending behavior at an application level, not at a product line level. As such, and also not to alter the semantics of the *Extend* relationship, we also use variability annotations to mark the *Extend* relationship. In Figure 89, the annotation `Handle Gold Memberships` states that the corresponding *Extend* relationship is optional. The possibility of also annotating the *Extend* relationships with variability annotations permits, for instance, the modeling of groups of alternative extends.

5.3.4 Implementation Roadmap

In this section we present a possible implementation roadmap to the approach described in the previous section. For that we use Eclipse Modeling Framework (EMF) version 2.2.0 [EMF 2007] and SmartQVT version 0.1.3 [SmartQVT 2007]. The EMF provides a modeling and code generation framework for Eclipse applications based on *Ecore* models. These Ecore models support Essential MOF (EMOF) as part of the OMG MOF 2.0 specification [MOF 2006]. We note that the code presented is in compliance with such versions and may not be valid in other versions of the tools. For the validation of the Ecore models, a possible approach is to use an implementation similar to the one described in [Damus 2006].

The process of mapping use cases to features is the one presented in Figure 82. The main goal of the process is to obtain a use case model for a specific application of a domain based on a feature configuration model. For that, we use the approach to map use cases to features as discussed in the previous sections. Basically, it consists of three transformations: transform a family use case model into a feature model (*T1*); transform a feature model into a configuration metamodel (Ecore model) (*T2*); and finally, transform a configuration model and a family use case model into an application use case model (*T3*).

T1: Family Use Case Model to Feature Model
The family use case metamodel is similar to the one presented in Figure 87 with the addition of two new elements used to annotate variability: *ExtendVariability* and *IncludeVariability* (see Figure 91). These enable the annotation of variability into *Extend* and *Include* relationships, as described in the previous section. Figure 89 presents an example of these annotations in a family use case model. The resulting feature model must be in conformance with the feature metamodel presented in Figure 84. Figure 92 presents an extract of the QVT operational transformation that map a use case family model into a feature model. Basically, the program starts by mapping each use case to a feature (line 6). Features resulting from use cases have the name of the corresponding use case. The program then maps each *Include* and *Extend*, that are not referenced by variability annotations, into *Subfeatures* (lines 7 and 8). For that, it verifies if the *Feature* that maps to the included use case is already member of a *Subfeature* (lines 34 and 35). If so, it uses a *Reference* to reference that *Feature*. If not, the *Feature* becomes a direct child of the *Subfeature*. Obviously, these *Subfeatures* are mandatory (line 38).

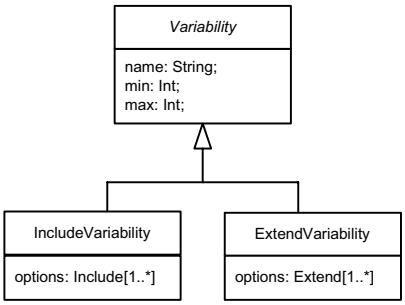

Figure 91: Variability *annotations* for use case models.

After mapping non-annotated *Include* and *Extend* relationships, the transformation program maps *IncludeVariability* and *ExtendVariability* elements to *Subfeatures* (lines 9 and 10). The transformation involves mapping each of the *options* of the *IncludeVariability* (or the *ExtendVariability*) into a *Subfeature* child (lines 55 to 61). These *options* will become *Subfeature* children of type *Reference* or *Feature*, according to the previously described logic. The information regarding cardinality of the variability option groups is mapped directly to the cardinality of the respective *Subfeature* (line 64 and 65).

With this transformation program, if we take as input the family use case model presented in Figure 89, we obtain a feature model similar to the one presented in Figure 85.

T2: Feature Model to Configuration Metamodel
With the previous transformation we obtain a feature model that is in conformance with the feature metamodel of Figure 84. What we would like to do now is to build configuration feature models that are in conformance with the feature model that resulted from the previous transformation, i.e., the feature model should become the metamodel for the configuration models. To achieve this, we use an approach in which a model is *promoted* to a metamodel. In this case, the feature model that resulted from the previous transformation is transformed into an Ecore metamodel. If we map *Features* to *Classes*, then *Subfeatures* become naturally associations between *Classes*. With this approach, feature configurations are simply instances of the corresponding *Classes*. This is similar to the approach proposed in [Asikainen *et al.* 2006]. Although it is a recent discussion topic, at least among practitioners [Merks *et al.* 2006], the generic process of promoting models to metamodels is out of scope of this section. This subject will be addressed in Section 5.4.

Next, we briefly describe the transformation between feature models and Ecore feature configuration metamodels. An extract of the QVT operational transformation is presented in Figure 93.

```
transformation Usecases2Features(in ucModel:USECASE, out fModel:FEATURE);

main() {
  ucModel.objects()[Subject]->map subject_to_feature_model();

  ucModel.objects()[UseCase]->map usecase to feature();
  ucModel.objects()[Include]->map include_to_subfeature();
  ucModel.objects()[Extend]->map extend_to_subfeature();
  ucModel.objects()[IncludeVariability]->map includeVariability_to_subfeature();
  ucModel.objects()[ExtendVariability]->map extendVariability_to_subfeature();
}

mapping Subject::subject_to_feature_model () : FeatureModel {
  rootFeature := ucModel.objects()[UseCase]->map usecase to root feature();
  name := self.name;
}

mapping UseCase::usecase_to_root_feature () ...

mapping UseCase::usecase_to_feature () : Feature ...

helper includeInVariability(i: Include) : Boolean {
  var x :=ucModel.objects()[IncludeVariability]->select(iv | iv.options->exists(i1|i1=i) );
  var y := x->first();
  return if y = null then false else true endif;
}

mapping Include::include_to_subfeature () : Subfeature
  when { not includeInVariability( self ); } {
  var f: Feature; var r: Reference;

  parent := self.includingCase.resolveone(Feature);
  f := self.addition.resolveone(Feature);
  r := if repeatedFeature(f) then object Reference{ name:=f.name; feature:=f; }
       else null endif;

  name := self.name;
  minCardinality:=1; maxCardinality:=1;
  childs := if repeatedFeature(f) then Sequence { r.asType(Node) }
            else Sequence { f.asType(Node) } endif;
}

helper repeatedFeature(f: Feature) : Boolean {
  var x := fModel.objects()[Subfeature]->select(sf | sf.childs->exists( f1| f1=f) );
  var y := x->first();
  return if y = null then false else true endif;
}

mapping IncludeVariability::includeVariability to subfeature () : Subfeature {
  var f: Feature; var r: Reference;

  parent := self.options->first().includingCase.resolveone(Feature);

  childs := Sequence { };
  self.options->forEach(i) {
    f := i.addition.resolveone(Feature);
    r := if repeatedFeature(f) then object Reference{ name:=f.name; feature:=f; }
         else null endif;
    childs += if repeatedFeature(f) then Sequence { r.asType(Node) }
              else Sequence { f.asType(Node) } endif;
  };

  name := self.name;
  minCardinality := self.min;
  maxCardinality := self.max;
}
```

Figure 92: Extract of QVT Operational transformation from use case to feature model.

```
1  transformation Features2Ecore(in fModel:FEATURE, out eModel:Ecore);
2
3  main() {
4    fModel.objects()[FeatureModel]->map feature_model_to_epackage();
5  }
6
7  mapping FeatureModel::feature model to epackage () : EPackage {
8    var fm: EClass;
9    name := self.name;
10
11   -- first pass
12   fm := self->map feature_model_to_eclass();
13   eClassifiers := Sequence { fm };
14   -- Each Subfeature becomes an abstract class
15   eClassifiers += fModel.objects()[Subfeature]->map subfeature to eclass();
16   eClassifiers += self.rootFeature[Feature]->map feature_to_eclass();
17   fm.eStructuralFeatures := self.rootFeature[Feature]->map rootfeature_to_ereference(fm);
18   -- Transform all other features...
19   eClassifiers += fModel.objects()[Feature]->select(x | x.resolveone(EClass)=null)
20                    ->map feature_to_eclass();
21   -- second pass
22   fModel.objects()[Feature]->map subfeatures();
23 }
24
25 mapping FeatureModel::feature_model_to_eclass () : EClass ...
26
27 mapping Feature::rootfeature_to_ereference ( fm: EClass) : EReference ...
28
29 mapping Feature::feature_to_eclass () : EClass ...
30
31 mapping inout Feature::subfeatures () {
32   var c:EClass;
33   c := self.resolveone(EClass);
34   c.eStructuralFeatures := self.subFeatures[Subfeature]->map subfeature_to_ereference();
35 }
36
37 mapping Subfeature::subfeature_to_eclass () : EClass ...
38
39 mapping Subfeature::subfeature_to_ereference () : EReference {
40   var c: EClass;
41
42   name := self.name;
43   containment := true;
44   lowerBound := self.minCardinality;
45   upperBound := self.maxCardinality;
46
47   c := self.resolveone(EClass);
48   eType := c;
49
50   self.childs[Feature]->select(c|c.oclIsKindOf(Feature)).resolveone(EClass)
51                        ->map childs to subtype(c);
52   self.childs[Reference]->select(c|c.oclIsKindOf(Reference)).feature.resolveone(EClass)
53                          ->map childs_to_subtype(c);
54 }
55
56 mapping inout EClass::childs_to_subtype (superType: EClass) ...
```

Figure 93: Extract of QVT Operational transformation from feature to Ecore model.

In the transformation, each *Subfeature* is mapped to an abstract *EClass* with the same name (line 15). Later, this abstract *EClass* will become the *eSuperType* of the types that will map to the *childs* of the *Subfeature* (lines 50 to 53). After transforming all *Subfeatures* to abstract *EClasses*, the program maps each *rootFeature* to a non-abstract *EClass* (line 16) and to an *EReference* from the *EClass* resulting from the *FeatureModel* element to each of the new *EClasses* (line 17). This will map all top level *Features*. Next, all not yet mapped *Features* are also mapped to non-abstract *EClasses* (lines 19 and 20). In a second pass, all *Features* are again processed (line 22). This time, for each *Feature*, its *Subfeatures* are mapped to *EReferences*. The *eType* for each of these *EReferences* is the abstract *EClass* that resulted from the initial transformation of the *SubFeatures* (line 47 and 48). This abstract *EClass* becomes the *eSuperType* of the *EClasses* that were mapped

from the *childs* (*References* or *Features*) of the *Subfeature* (lines 50 to 53). With this transformation we obtain an Ecore metamodel that is equivalent to the feature model. This transformation regards the activitity *T2* depicted in Figure 82. We can now use the EMF generation capabilities to generate an editor from which we can create feature configurations that are in conformance with the metamodel.

T3: Family to Application Use Case Model
The last step in the transformation process involves the generation of application use case models from feature configurations (activity *T3* in Figure 82). This requires a transformation that must have at least the family use case model and the feature configuration model as input and a product (or application) use case model as output. Basically, the transformation involves including in the output model only the use cases, includes and extends relationships that are referenced by the feature configuration model. This transformation can be similar to the other two described in the paper.

A more generic transformation program could be required if we wanted our transformation to support changes in the family use case model (which can by very probable). Such changes result also in changes to the feature model that acts as metamodel for the feature configurations. To have only one *T3* transformation process regardless of number of configuration metamodels, the *T3* activity transformation could also have as input the feature model. Since this model is the source for obtaining the configuration metamodels, it could serve as guide to process the feature configuration models regardless of their metamodels (they would have to be processed as generic Ecore models), thus allowing a generic transformation process that can be applied to all possible feature models. In this case, the QVT transformation would follow the declaration presented in Figure 94

```
transformation configuration2usecases(
  in fu:USECASE,     -- family use case
  in fc:Ecore,       -- untyped feature configuration
  in fmm:FEATURE,    -- feature meta-metamodel
  out u:USECASE);    -- product use case
```

Figure 94: Declaration of *untype* transformation *configuration2usecases*.

In Figure 94 we see that instead treating the feature configuration model `fc` as a typed model it is handled as an Ecore model. The feature `fmm` metamodel that resulted from the *T1* transformation is used to dynamically *guide* the transformation. Model metadata can also be accessed dynamically to support the transformations. In Figure 95, we present a helper function that dynamically verifies if a use case is included in the feature configuration.

```
helper usecase_in_configuration(u: UseCase): Boolean
{
   var o:=fc.objects()[EObject]->select(x | x.metaClassName=u.name )->first();
   return if o = null then false else true endif;
}
```

Figure 95: QVT helper function that *dynamically* verifies if a use case in included in a feature configuration.

5.4 Transformation Patterns for Multi-Staged Development

In this section, we will address in a practical way multi-staged model driven software approaches and how they differ in their nature from single-staged approaches. As we will see, multi-staged approaches result in a series of method recipes for applying model driven technologies in a way similar to design patterns [Gamma *et al.* 1995]. As such, we will present the multi-staged model driven software approach as a series of model driven transformation patterns. We will do so using the Eclipse Modeling Framework (EMF) [EMF 2007], an Eclipse based metamodeling framework that is based on the MOF standard, and an insurance software supply chain as a case study.

Figure 96 presents the development activities covered in this section.

5.4.1 Multi-Staged Modeling Approach

Multi-stage development is already a common approach in the software industry. In fact, it is almost impossible to discover a software project that does not use software artifacts provided by other players in the industry, be that open source code, software components, documentation, integrated development environments (IDE) or even software platforms. Although this is a reality, it is usually done in an informal way, i.e., the supplier and the customer do not usually share much more than the exchange of the artifact. Examples from other industries demonstrate that supply chains with strong relations between the players can be very beneficial (the well know example is the automobile industry). Software product lines are an approach to software engineering that is based on this principle [Clements *et al.* 2002]. Software factories extend the product line approach beyond the frontier of the organization, into the supply chain [Greenfield *et al.* 2004]. In this case, we say that a software supply chain becomes a multi-staged software development approach if it achieves a level of integration of the software development phases that is similar to that of a software development approach that is based on a single site (single-staged or traditional approach). In this section we describe our view on supporting multi-staged software development in the context of model driven approaches.

A model driven approach in the context of multi-stage software development implies that models are used across the stages. They must support the specialization of the system and also its *instantiation* (when models are transformed from platform independent formats to platform specific formats). We will illustrate this approach with an insurance software supply chain case study.

Figure 97 presents a possible scenario for a multi-stage software insurance supply chain. The figure presents how insurance agreements can be specialize in several stages. Insurance agreements are contracts established between insurance companies and its customers which are usually materialized as insurance policies. The structure and rules that apply to these agreements can vary significantly according, for instance, with the insurance branch (life, property, etc.), the insurance coverages, or the type of insured object.

The stages presented in Figure 97 represent players of an insurance company. In the figure we can see the insurance company, an insurance company division and an insurance company branch. Each of the stages runs a domain-specific platform [Czarnecki *et al.* 2006], in this case, an insurance information system platform. The domain-specific platform can be *configured* for a particular purpose through domain-specific modeling. Domain-specific modeling is done by a domain-expert. In this case, insurance agreements are modeled and used to *configure* the

domain-specific platform. As Figure 97 shows, agreement models can be also specialized in subsequent modeling stages.

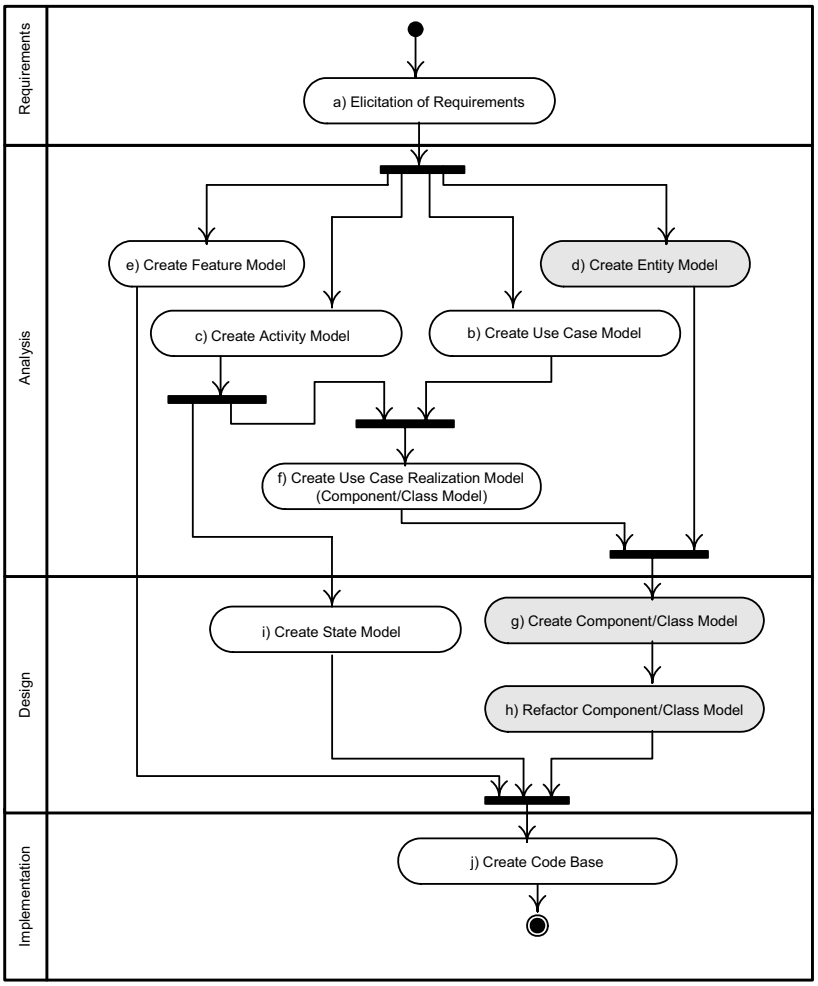

Figure 96: Development activities covered in Section 5.4.

Figure 98a represents the typical architecture for EMF based applications, with three architecture levels. The Ecore model is the equivalent of the MOF model. Figure 99 presents the kernel of the Ecore model. Typically, using EMF, we construct a core model (which metamodel is the Ecore model) based on which the EMF framework can generate java source code to support the creation of model instances; support the edition of model instances using a tree based editor and also testing classes. The EMF also supports the serialization of the model instances to XML. So, we can say that EMF can be used to generate code to support the creation of domain-specific modeling environments (the graphic part is usually complemented with the use of GMF [GMF 2007]). In addition, since it generates java code to support the model, we can also use EMF as a source code generation tool.

Figure 98b represents a possible usage of EMF to model insurance agreements. In Figure 98c, we see how an insurance agreement model can be specialized into a car insurance model using EMF. This can be done with EMF since the specialization relationship is supported by the Ecore model and the EMF resource infrastructure allows references between models. So, the car

insurance agreement model can reference the insurance agreement model and *EClass* elements from the car insurance model can have *EClass* elements of the insurance model as their supertypes.

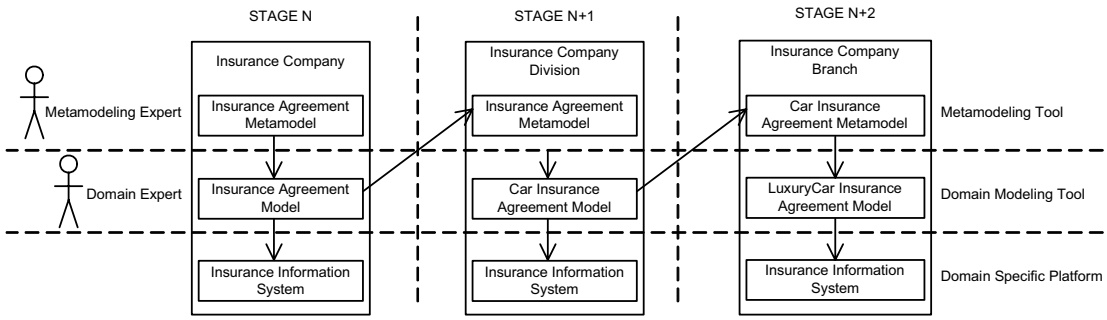

Figure 97: Multi-staged model driven insurance supply chain.

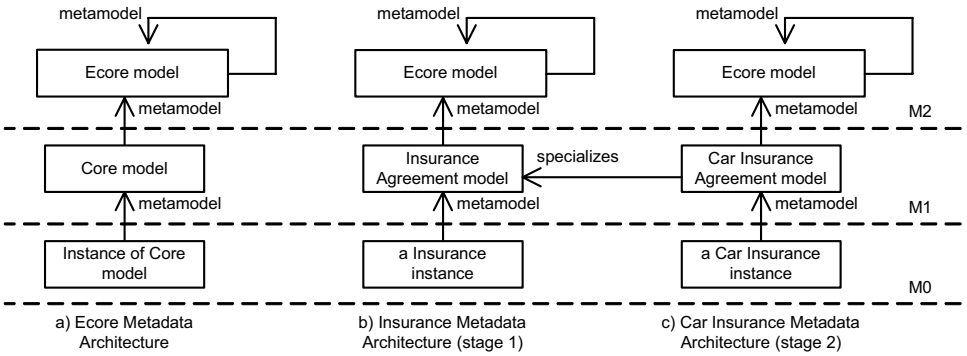

Figure 98: EMF multi-staged model driven metadata architecture for insurance supply chain with one modeling level (M1).

Figure 98 can also serve as an example of a possible multi-staged modeling approach. In fact, since EMF supports model references and the specialization relationship between *EClass* elements, Figure 98b and Figure 98c could represent two stages of a software insurance supply chain: Figure 98b could represent modeling at insurance company headquarters while Figure 98c could represent modeling at an insurance company division. Although this is true, the situation depicted in Figure 98 is not the most common because specialization between the modeling stages is done at the core modeling level. As we saw in Figure 97, we usually want to specialize the domain-specific models (at the domain expert level), not the core models (at the metamodeling expert level).

In fact, the usual approach for insurance supply chains requires an EMF architecture with four levels, similar to the UML example presented in Figure 80. Figure 100 presents the four EMF modeling levels (or layers) for insurance agreements.

Usually, the first metamodel (the M2 layer of Figure 100) is used to specify a specialized modeling environment, i.e., a domain-specific modeling environment. With this environment, the modeler at the M1 layer no longer needs to use Ecore abstractions, it can use specific abstractions of the domain. In this case, at the M1 layer, the modeler specifies insurance agreement models. The models created at the M1 layer can be used to create instances of agreements at the M0 layer.

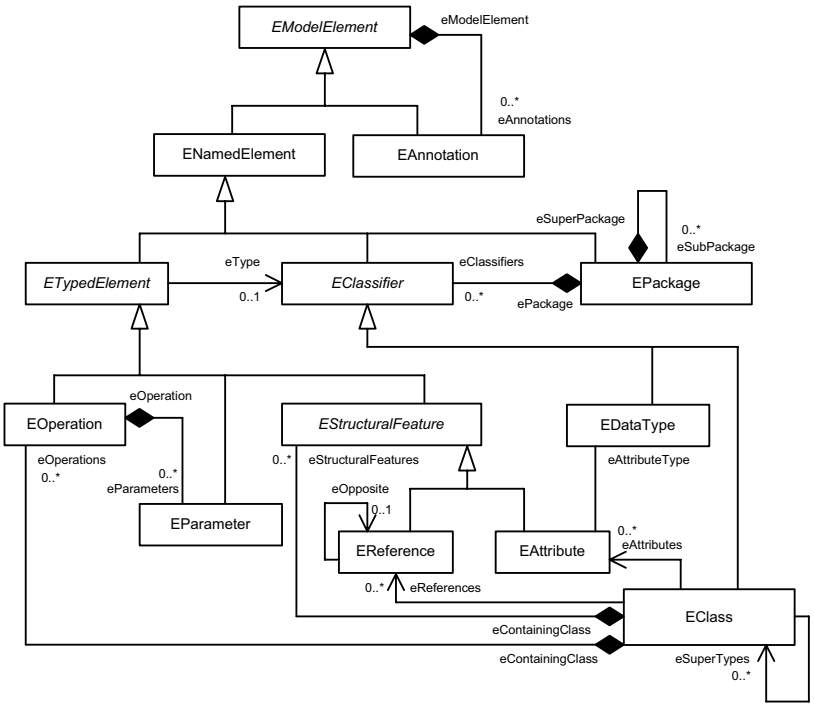

Figure 99: Kernel of the Ecore model.

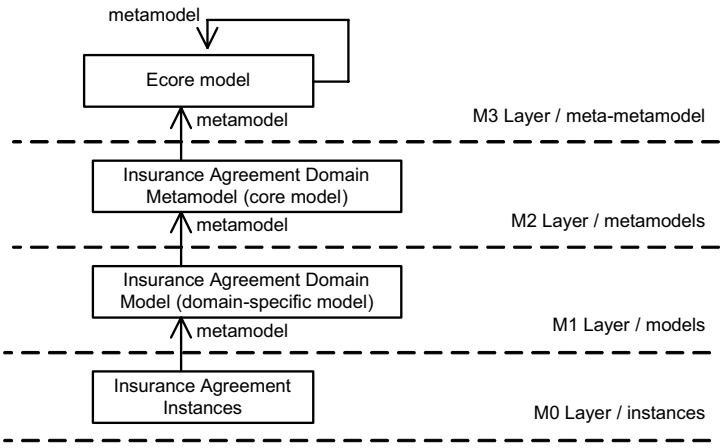

Figure 100: EMF modeling layers for insurance agreements.

The scenario presented in Figure 100 has, however, a restriction: since models at the M1 layer are not core models it is not possible to use the EMF infrastructure to generate support for the M0 layer (M0 usually is supported by platform specific code, for instance, java code). Of course, a possible solution is to use a specific generative approach to transform the M1 models. In the case of EMF, this can be done using JET [JET 2007]. This approach is probably suitable to single layer scenarios, such as the one of Figure 100. However, a multi-level or a multi-staged model driven scenario may not cope very well with this approach if support for the layers is based on specific code, since it requires a significant development effort and it is hard to maintain. What we advocate is an approach which reuses the metamodeling framework to support multiple modeling layers. Basically, we propose that domain-specific models be *promoted* to metamodeling models (in the

case of EMF, to core models). As such, all the generative infrastructure of the metamodeling framework can be reused at the several modeling layers. Figure 101 depicts the approach. In the next section we will explain the multi-staged modeling approach and the involved activities (mainly model transformations) and roles.

5.4.2 Multi-Staged Model Transformations

To explain the multi-staged model driven scenario it is important to understand the involved roles. There are basically three main roles: *metamodeler*; *domain modeler* and *executer*. The *metamodeler* uses the metamodeling framework directly. In the case of EMF, the *metamodeler* edits directly core models. A *domain modeler* is someone that edits domain-specific models, usually according to a metamodel that is specified by a *metamodeler*. The *executer* is the system that runs instances of models which metamodel is specified by the *domain modeler*. The distinction between the *executer* and the other roles is that models at this level are not used as metamodels of other levels, as such, this modeling level is terminal. As presented in Figure 97, usually the *executer* is a domain-specific platform.

In our insurance software supply chain, the domain modeler must be an insurance expert, since he/she must specify insurance agreements. The metamodeler is someone who will specify the metamodel that will support the modeling environment of the domain modeler. As such, he/she must be an expert in the metamodeling framework (for instance, in EMF) and must acquire the necessary domain knowledge using some domain analysis method (e.g., FODA [Kang *et al.* 1990]). In Figure 101, it is possible to observe the responsibility of these three roles for a multi-staged modeling approach: the metamodeler has the responsibilities at the M2 layer; the domain modeler has responsibilities at the M1 layer; and the executer (the domain-specific platform), at the M0 layer. In fact, the M1 layer is divided into M1 and M1'. The M1' layer is where the domain modeling takes place. The domain model must support two perspectives: one that supports the creation of the instantiation model (M1) and other that support the creation of the specialization model (M2). To reuse the metamodeling generative infrastructure, we propose that these two perspectives be supported by transforming the domain model into the metamodeling native format (in this case core models). Next, we will detail the multi-staged modeling approach and the activities involved.

Figure 101 displays an overview of our approach to multi-stage model driven (the figure only displays automatic activities). Since the approach is based on domain-specific models and those require a domain-specific modeling environment, the process *bootstrapping* is done by the metamodeler. The first metamodel (core model) is used to introduce the domain-specific modeling concepts that will be used by domain experts in all the stages to create or specialize domain-specific models (in this case, insurance agreements). Such metamodel will provide domain modelers their modeling concepts in a way similar to the concepts that ecore provides to the metamodeler. As a *bootstrapping* metamodel, it must also integrate with the concepts of the domain-specific platform. Since the domain-specific platform will integrate domain concepts through the instance models, the discussed integration is essentially achieved by the support for the instantiation perspective of the initial metamodel. Figure 102 depicts the models at the *bootstrapping* of the multi-staged model driven approach for the software insurance supply chain case study. The *bootstapping* process corresponds to the first stage (*stage N*) depicted in Figure 101.

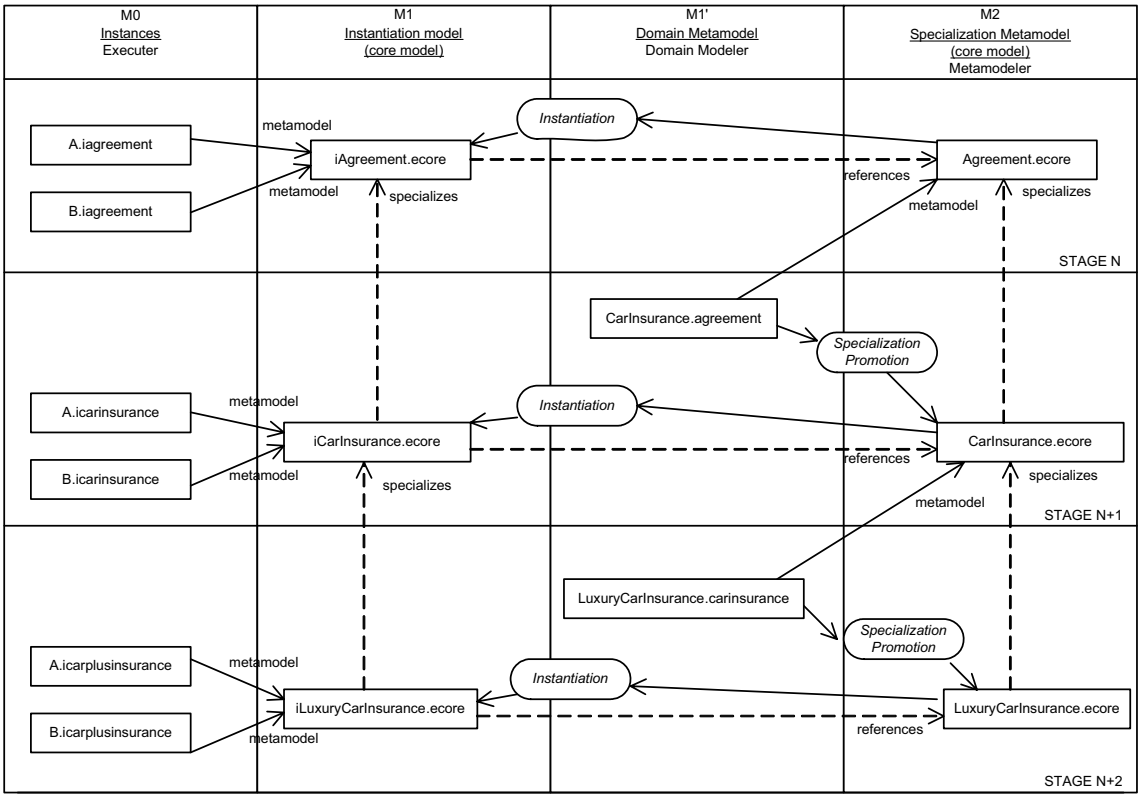

Figure 101: EMF multi-staged modeling of insurance supply chain with two modeling levels (M2 and M1').

Figure 102 is divided in three parts: the specialization metamodel (*Agreement.ecore* in *stage N* of Figure 101); the instantiation model (*IAgreement.ecore* in *stage N* of Figure 101); and the domain model (part of the domain-specific platform, modeled using core models, not depicted in Figure 101). The instantiation model is automatically generated from the specialization metamodel by the *instantiation* automatic activity (see Figure 101). The *instantiation* activity basically creates the instantiation model by filtering out of the metamodel elements that regard only its specialization perspective. The result is the *IAgreement.ecore* instantiation model, which contents are depicted in the lower part of Figure 102. The instantiation model can be used to generate code (using the metamodeling tool generative capabilities) that integrates its concepts with the domain-specific platform at this stage. The elements of the instantiation model can maintain references to their origins in the metamodel if access to metadata is required by the domain-specific platform. To distinguish between the two perspectives (*instantiation* and *specialization*) of the metamodel we use *annotations*. This approach requires that the meta-metamodel support annotations. As Figure 99 shows, EMF ecore supports annotations. In Figure 102 we can see annotated elements. The annotations are depicted in a similar way to that of stereotypes in UML diagrams. We will detail the major role of annotations in the next section.

The specialization metamodel is also used to support the generation of the domain-specific modeling environment used by the domain experts. Figure 101 shows how the *Agreement.ecore* specialization metamodel (*stage N*) is used to support the domain modeling activity of *stage N+1*. Figure 103 presents a possible model of a car insurance agreement that conforms to the *Agreement.ecore* metamodel.

Since domain metamodels (such as the one depicted in Figure 103) are not native metamodels of the metamodeling framework they can not be directly used by the framework to generate a specialization for the next stage. As such, they need to be transformed into native metamodels (core models, in the case of EMF). Once again, we propose to do it by using the annotations of their metamodels as guides to the transformation process (*Specialization Promotion* in Figure 101). In the case presented in Figure 101, the *CarInsurance.agreement* domain metamodel can be transformed into a core model by using the annotations of its *Agreement.ecore* metamodel.

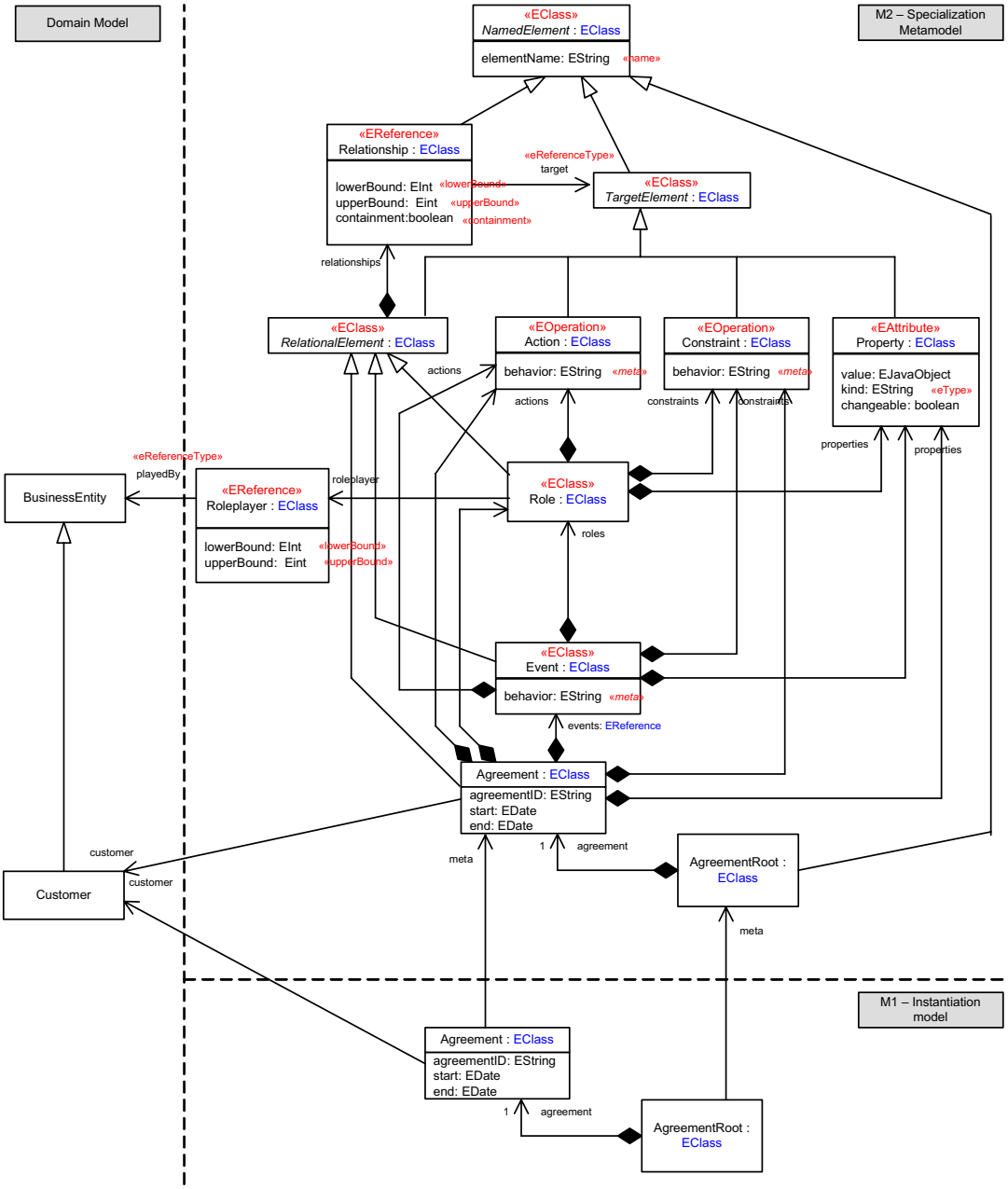

Figure 102: Specialization metamodel *vs* instantiation model *vs* domain model.

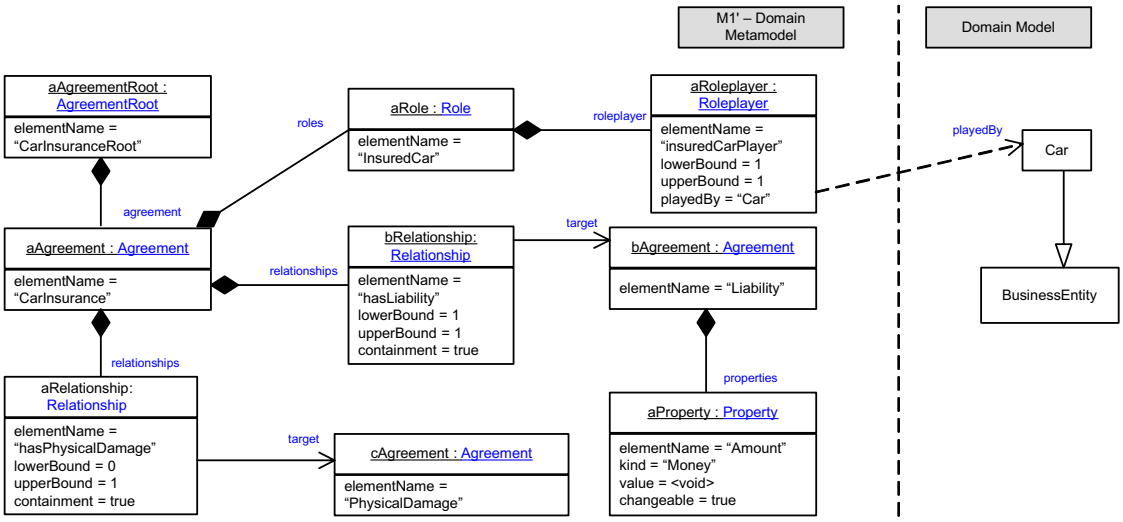

Figure 103: Domain metamodel for a car insurance agreement (*CarInsurance.agreement*).

In this section we have discussed the principles by which a multi-stage model driven approach can be supported by a metamodeling framework. The goal is to provide an approach that frees the developer (as much as possible) of doing specific model transformations, i.e., provide a *generic* approach that supports multi-staged model driven development. Similarly to the way design patterns are presented as generic solutions to design problems, in the next section we will present our approach as layers of metamodeling patterns that can be adopted to support multi-stage model driven scenarios.

5.4.3 Transformation Patterns

In this section we present our approach to multi-stage model driven software development. Since a multi-stage model driven approach can be applied in several scenarios (being the insurance software supply chain only one of them) we explain our approach as a set of model driven development patterns. We follow the spirit of the original description of design patterns and describe the problem, solution and consequences of each model driven pattern. In fact, each presented pattern is a part of a more large scale pattern that we call *Multi-Stage Domain Specific Modeling*. If we continue to make the analogy with traditional development patterns we could say that this is an architectural style pattern [Shaw *et al.* 1996].

As we will see next, the description of the *solution* of a model driven pattern consists essentially in describing metamodeling approaches and transformation between models (or/and metamodels).

5.4.3.1 Model Driven Architectural Pattern

Name
Multi-Stage Domain Specific Modeling

Problem
How to support domain-specific modeling and domain-specific model specialization for several domain-specific modeling stages in the context of a domain-specific platform.

Solution
The proposed solution adopts *off-the-shelf* metamodeling tools. By this we mean that the solution is essentially based on existing generative and transformational support of publicly available metamodeling tools. Eclipse EMF is one example of such a metamodeling tools.

To support the multi-stage model driven approach we propose that the models of the native metamodel format be annotated in a manner that marks their elements as being *instance* elements (*instantiation* perspective) or *meta* elements (*specialization* perspective). Such annotations can then be used to guide two generic transformation activities: the *instantiation* transformation and the *specialization promotion* transformation. *Instance* elements are the base for the creation of the instantiation model (*instantiation* transformation) and *meta* elements are the base for the creation of the specialization metamodel (*specialization promotion* transformation).
Figure 101 presents an illustration of this pattern for an insurance supply chain.

Consequences
The solution proposed for this model driven pattern requires that we solve two sub-problems (or sub-patterns): the instantiation transformation pattern and the specialization promotion transformation pattern.

The adoption of this model driven pattern has several consequences. These consequences were largely discussed in the previous section, so we will not further detail them.

5.4.3.2 Model Transformation (Sub-)Patterns

Instantiation (metamodel to metamodel) Transformation Pattern

Problem
How to support instantiation from models (metamodels) where some elements have metadata semantics, and as such should not appear in the *instances* of the models.

Solution
Here we propose a solution that adds a further constraint to the problem. Since this pattern occurs in the context of multi-stage modeling, the instantiation models should support specialization from stage to stage.

The general solution has already been discussed in the previous section. Basically, the original metamodel should be annotated. These annotations should mark the elements that have meta semantics. As such, the transformation consists of creating a metamodel that leaves out elements with meta semantics. For the insurance supply chain, we can see an example of the result of the transformation in the metamodel that appears at the bottom of Figure 102.

The multi-stage modeling approach implies that the instantiation models are specialized at each stage. Figure 104 and Figure 105 can be used to illustrate the problem (and also the solution). Figure 104 presents the specialization metamodel at stage N+1 of the insurance case study. This specialization model was obtained from the domain model depicted in Figure 103. Figure 105 presents the output of the instantiation transformation, when the source metamodel is the one

depicted in Figure 104. As it is possible to observe from both figures, the resulting instantiation metamodel not only contains elements that are *not* annotated as meta elements in the source metamodel but also contains the annotated elements that resulted from the specialization process. These elements are those that subset or refine meta annotated elements of the previous stage. For instance, in Figure 104 we can see that the *hasLiability* relationship between *CarInsurance* and the *HasLiability EClass* subsets[6] the *relationships* relationship of the previous stage. Therefore, *HasLiability* is included in the resulting instantiation metamodel. Since this element has an annotation stating that it has the meta semantics of an *EReference*, it becomes an *EReference* element in the resulting metamodel.

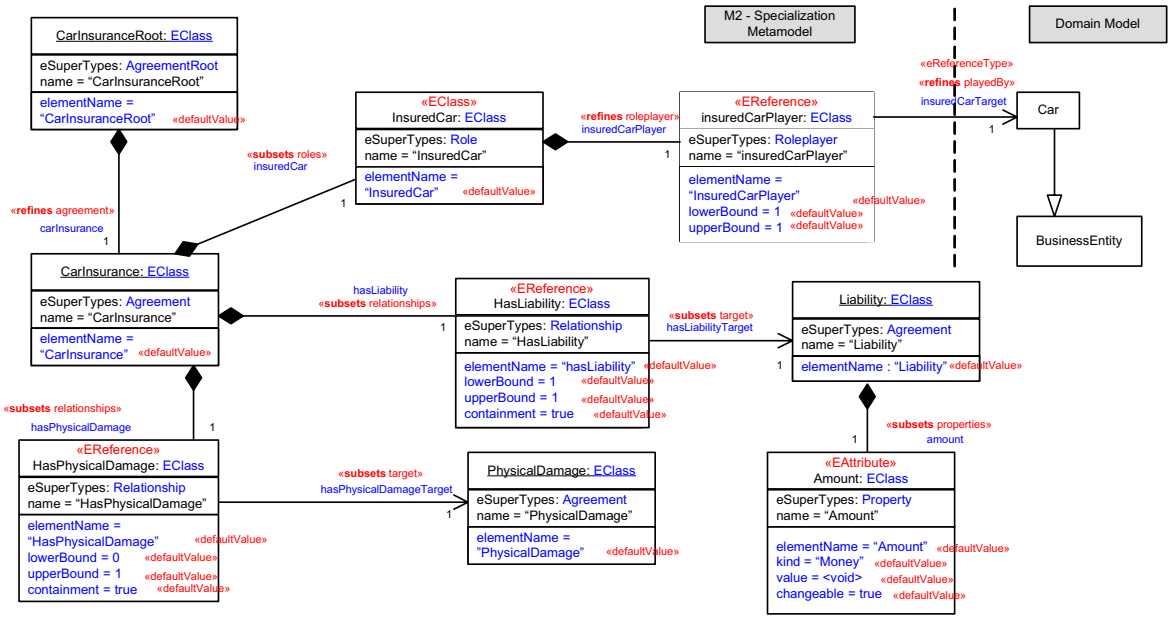

Figure 104: Native metamodel for a car insurance agreement (*CarInsurance.ecore*).

Consequences

The solution proposed for this pattern is straightforward if we consider it only in the context of single stage development. When we consider it in a multi-stage approach we have to take into account the refinements (specializations) made in the previous stage. The annotations in the source elements regarding such refinements as well as their meta semantics can guide the creation of the instantiation model. The annotations regarding the meta semantics are used to preserve the intention of the original metamodeler. As it is possible to observe in the previous examples, such annotations are done using the names of the elements of the meta-metamodel of the modeling tool (or the *native* metamodel). In the case of EMF, Figure 99 presents such elements. From Figure 102, we see that the original metamodeler intention was that the non-abstract elements *Event* and *Role* should have a meta semantic of an *EClass*; the non-abstract elements *Action* and *Constraint* should have a meta semantic of an *EOperation*; the non-abstract elements *Roleplayer* and *Relationship* should have a meta semantic of an *EReference*; and the non-abstract element *Property* should have a meta semantic of an *EAttribute*. These examples represent the four most typical element transformations in the context of the instantiation transformation pattern: *Class* to *Attribute*; *Class* to *Operation*; *Class* to *Reference* and *Class* to *Class*.

[6] We use the term *subset* with similar semantics to that found in the *subset* annotations used in UML associations.

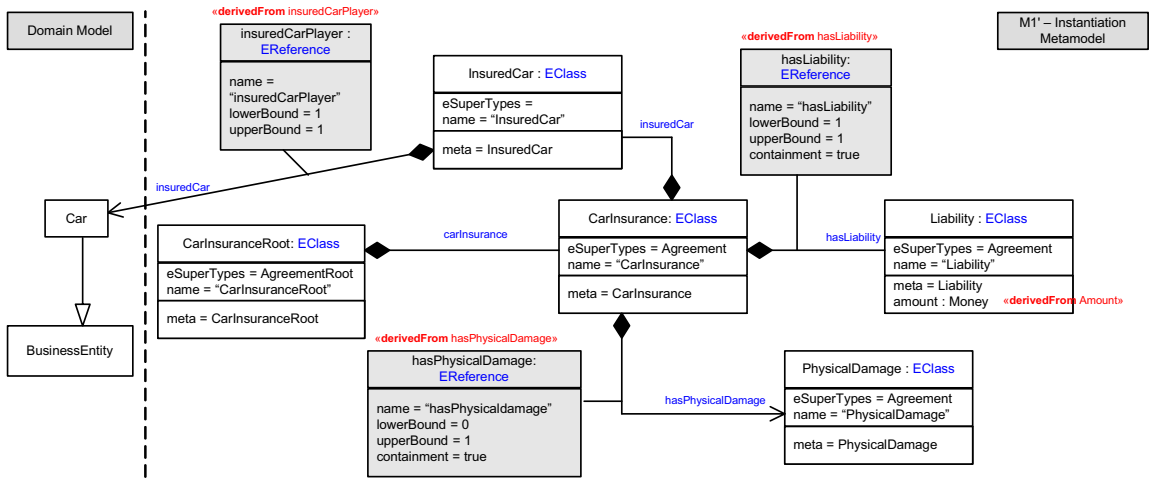

Figure 105: Instantiation metamodel for a car insurance agreement (*ICarInsurance.ecore*).

Transformation Details

Essentially, the solution for this pattern consists on how elements of a metamodel can be transformed into a resulting instantiation metamodel in the context of the general solution suggested by the pattern. In a native metamodel similar to Ecore, we identify four major transformations: *Class* to *Attribute*; *Class* to *Operation*; *Class* to *Reference* and *Class* to *Class*. They transform source elements into target elements as their names imply. In Figure 105, we can see the result of applying these transformations to the source metamodel of Figure 104. We will now consider the example of the source element *InsuredCar*. This source element is annotated as having the meta semantics of an *EClass*. As such, if transformed, it must become also an *EClass* element in the target metamodel. Other possible annotations in the source element may be used to further specify the value of target element attributes.

The resulting elements must also indicate if they are specializing elements of the previous stage. This is not the case of the *InsuredCar*. On the other end, *CarInsurance* is an example of an element that is specializing the *Agreement* element from the previous stage.

As we have discussed previously, only source elements that pertain to the instantiation perspective of the metamodel are transformed. This includes elements that do not have an annotation describing their meta semantics and elements that are annotated but have been refined (specialized) by the domain modeler (these are elements resulting from the *specialization promotion transformation pattern*). Such elements are linked to other elements by references with the *refine* or *subset* annotations.

Specialization Promotion (model to metamodel) Transformation Pattern

Problem

How to support the specialization of domain models that are not native metamodels of the metamodeling tool and, therefore, do not have the native support for specialization.

Solution
We propose a solution that is based on the promotion of the domain model to a native metamodel of the metamoldeling tool in a way that preserves the semantics of the domain model. We call this transformation a specialization promotion because we are transforming a model into a metamodel, i.e., we are *promoting* a model into a metamodel. Our solution is proposed in the context of the multi-stage model driven pattern and therefore, in conformance with the other patterns, we use annotations to guide the transformation process.

Figure 103 presents an example of a domain model (in fact it is *acting* as a metamodel) which metamodel is the one presented in Figure 102 (*M2 Specialization metamodel*). The result of applying the specialization promotion transformation to the model of Figure 103 results in the native metamodel of Figure 104. Basically, each object instance of the domain model becomes a *Class* (*EClass*) in the native metamodel. Each reference instance (or object link) becomes a *Reference* (*EReference*) in the native metamodel. Similarly to the instantiation transformation pattern, these represent sub-patterns of the specialization promotion transformation pattern: *Object instance* to *Class* and *Reference Instance* to *Reference*. These sub-patterns will be explained next.

Transformation Details
Here we discuss the possible transformations involved in the solution of the *specialization promotion transformation pattern*. The goal is to transform domain models into their equivalent native metamodels. A domain model is an instance of a metamodel, and as such is composed of *objects* and *links* or reference instances between objects. The *objects* are instances of *Class* elements (*EClass*) of the metamodel. The *links* are instances of *Reference* elements (*EReference*) of the metamodel.

When the domain modeler creates an instance of a *Class* he/she is making a specialization of the *Class*. As such, in the *Object* to *Class* transformation, an object is transformed into a *Class* that must specialize (become a subtype of) the meta-class of the source object. For instance, the *aRole* object of Figure 103 becomes the *InsuredCar EClass* in Figure 104. The *InsuredCar EClass* is a specialization of the *Role EClass*, which is the meta-class of the *aRole* object. With this approach domain models can be specialized in each stage of the multi-stage approach.

Links also follow an approach similar to that of the objects. As we have mentioned, they become references in the resulting metamodel. But, because they are instances of references, they are annotated as *subsets* or *refines* of the original reference. For instance, the *roles* link of Figure 103 that links *aAgreement* and *aRole* becomes the *insuredCar* reference between *CarInsurance* and *InsuredCar* target elements (see Figure 104). This reference is annotated as being a *subset* of the *roles* reference of the metamodel of the previous stage.

5.5 Conclusion

This chapter is essentially dedicated to formal transformations in model driven approaches. We particularly address specific issues that are related to the development of variability focused systems, such as software product lines. Our proposals are also discussed in the context of available tools in a way that facilitates its adoption and implementation.

In the first part of this chapter, we have presented a model driven approach to map use case to features. This approach is inspired by the original work of *Griss et al.* [Griss *et al.* 1998].

Gomaa also proposes a similar relationship between use cases and features [Gomaa 2005]. We differ from their works because we base the variability annotations in the *Extend* and *Include* relationships and not in the use cases. We have explained why this is more appropriate to model variability.

Czarnecki *et al.* presented an approach to map features to design models [Czarnecki *et al.* 2005a]. Basically, in their approach, a template design model is annotated with presence conditions that are logic expressions based on features. Non-annotated elements have an implicit true presence condition. If these expressions evaluate to true, the design element is included in the result model. In their approach, design elements must be annotated after the feature model. In our approach, the use case variability annotations have a similar effect, but we differ, since the design elements included in a configuration will result from the ones that are necessary to realize the product use case model that results from the transformation process.

Eriksson *et al.* also describe a model based approach that relates use cases and features [Eriksson *et al.* 2005]. However, their work is focused on used cases being described by scenarios and sequences of steps. As such, it does not explicitly deal with mappings at a UML use case diagram level (as we do) but at an inner use case level. They document the transformation process but do not precisely specify it, so it is difficult to tell if such approach is possible to implement with transformation languages such as QVT. Their approach to modeling use case variability has similarities with the one of Fantechi *et al.* [Fantechi *et al.* 2004]. Our work differs from these authors because our approach to model variability within use cases is based on models, not textual representations in natural language. We use activities to model the use case's behavior. Similar variability annotations as the ones present for the *Extend* and *Include* relationships can be used to annotate activity model elements and a similar transformation approach can also be used to map such elements to feature model elements.

We have discussed and proposed mappings between use case and feature models in a *formal* way that supports its implementation. Regarding applicability of our approach, we have showed that an implementation is feasible even with a not yet mature QVT implementation. Clearly, transformations could be improved by a complete QVT implementation enabling, for instance, the use of the QVT relational language. This would facilitate, for instance, the specification of two way transformation mappings between use cases and features. As discussed, the proposed method seams to open a feasible approach to implement mixed use case and feature model driven based product line engineering methods.

The first part of this chapter contributes to the research field with a 'formalization' of mappings between use cases and feature models that supports its automation by standard model transformation languages and metamodeling tools like QVT and EMF.

As models become first class artifacts of software development approaches they are treated in a similar manner as code is in *traditional* software development. Therefore, it is also natural that, as model driven approaches become widely adopted, issues and concerns that are related to code become concerns at the modeling level. For instance, we have seen this regarding refactoring of models [Biermann *et al.* 2006b]. As model driven approaches are adopted by the industry we will see also more experience reports. The results from these experiences will undoubtedly support the identification of *mode driven patterns*.

In the second half of this chapter, we have presented and discussed a problem that is common in the software product lines and software factories: multi-stage development. We have described this problem following the spirit of architectural and design patterns. The solutions proposed for the

patterns were based on using as much as possible the functionalities of actual publicly available metamodeling tools and their generative capabilities. We have presented the problem and the patterns based on EMF and an insurance software supply chain example. We believe this is just one in many model driven patterns that are yet to be identified. We have exemplified the described approach with an insurance supply chain case study. However, the approach can be used in other contexts like, for instance, to support multi-stage configuration of feature models such as we have discussed in first part of this chapter. In fact, the promotion of a feature model to an Ecore model, such as we have presented in Section 5.3, could be achieved by the approach we have presented in Section 5.4. Multi-stage configuration of features is also discussed in [Czarnecki *et al.* 2004].

The solution part of the presented patterns was based on annotating models and on doing transformations based on those annotations. We have not proposed a specific tool for the transformations, however, in the spirit of following as much as possible standards and similarly to the work presented in the first part of this chapter, an implementation of the QVT [QVT 2005] language, such as SmartQVT [SmartQVT 2007] could be used.

The second part of this chapter contributes to the research field with a proposal to support the multi-stage domain modeling of software systems based on model to metamodel transformations that reuse the generative capabilities of metamodeling tools such as EMF and are guided by model annotations.

5.6 References

[AndroMDA 2007] AndroMDA, "AndoMDA generator framework," Available at http://www.andromda.org/, 2007.

[Asikainen *et al.* 2006] Asikainen, T., T. Mannisto and T. Soininen, "A Unified Conceptual Foundation for Feature Modeling," SPLC2006, Baltimore, 2006.

[ATL 2007] INRIA, "Atlas Transformation Language," Available at http://www.eclipse.org/m2m/atl/, 2007.

[Bezivin 2005] Bezivin, J., "Model Driven Engineering: Principles, Scope, Deployment and Applicability," Generative and Transformational Technics in Software Engineering, Braga, Portugal, 2005.

[Bezivin *et al.* 2003] Bezivin, J., G. G. Dupe, F. Jouault, G. Pitette and J. E. Rougui, "First experiments with the ATL model transformation language: Transforming XSLT into XQuery.," 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture, 2003.

[Biermann *et al.* 2006a] Biermann, E., K. Ehrig, C. Kohler, G. Kuhns, G. Taentzer and E. Weiss, "EMF Model Refactoring based on Graph Transformation Concepts," *Electronic Communication of the EASST*, vol. 3, 2006a.

[Biermann *et al.* 2006b] Biermann, E., K. Ehrig, C. Kohler, G. Kuhns, G. Taentzer and E. Weiss, "Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework," Models 2006, Genova, Italy, 2006b.

[Clements *et al.* 2002] Clements, P. and L. Northrop, *Software Product Lines - Practices and Patterns*: Addison Wesley, 2002.

[Czarnecki *et al.* 2005a] Czarnecki, K. and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," GPCE'05, Tallinn, Estonia, 2005a.

[Czarnecki *et al.* 2006] Czarnecki, K., M. Antkiewicz and C. H. P. Kim, "Multi-level Customization in Application Engineering," *Communications of the ACM*, vol. 49, 2006.

[Czarnecki *et al.* 2003] Czarnecki, K. and S. Helsen, "Classification of Model Transformation Approaches," OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, Anaheim, 2003.

[Czarnecki *et al.* 2004] Czarnecki, K., S. Helsen and U. Eisenecker, "Staged Configuration Using Feature Models," SPLC2004, Boston, 2004.

[Czarnecki *et al.* 2005b] Czarnecki, K., S. Helsen and U. Eisenecker, "Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models," *Software Process Improvement and Practice, special issue on "Software Variability: Process and Management"*, vol. 10, pp. 143-169, 2005b.

[Damus 2006] "Implementing Model Integrity in EMF with EMFT OCL," Available at http://www.eclipse.org/articles/Article-EMF-Codegen-with-OCL/article.html,

[DSL 2007] Microsoft, "DSL Tools for Visual Studio," Available at http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx, 2007.

[EMF 2007] Eclipse Foundation, "Eclipse Modeling Framework," Available at http://www.eclipse.org/emf/, 2007.

[Eriksson *et al.* 2005] Eriksson, M., J. Borstler and K. Borg, "The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations," SPLC2005, Rennes, France, 2005.

[Fantechi *et al.* 2004] Fantechi, A., S. Gnesi, G. Lami and E. Nesti, "A Methodology for the Derivation and Verification of Use Cases for Product Lines," SPLC2004, Boston, 2004.

[Gamma *et al.* 1995] Gamma, E., R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[GMF 2007] Eclipse Foundation, "Graphical Modeling Framework," Available at http://www.eclipse.org/gmf/, 2007.

[Gomaa 2005] Gomaa, H., *Designing Software Product Lines with UML*: Addison Wesley, 2005.

[GReAT 2007] ISIS Institute for Software Integrated Systems, Vanderbilt University, "Graph Rewrite And Transformation (GReAT)," Available at http://www.escherinstitute.org/Plone/tools/suites/mic/great, 2007.

[Greenfield *et al.* 2004] Greenfield, J., K. Short, S. Cook and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*: Wiley, 2004.

[Griss *et al.* 1998] Griss, M. L., J. Favaro and M. d'Alessandro, "Integrating Feature Modeling with the RSEB," Fifth International Conference on Software Reuse, Victoria, Canada, 1998.

[Jacobson *et al.* 1992] Jacobson, I., M. Christerson, P. Jonsson and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*: Addison-Wesley, 1992.

[Jacobson *et al.* 2005] Jacobson, I. and P.-W. Ng, *Aspect-Oriented Software Development with Use Cases*: Addison Wesley, 2005.

[Jamda 2007] Jamda, "Jamda Framework for Building Application Generators," Available at http://jamda.sourceforge.net/, 2007.

[JET 2007] Eclipse Foundation, "Eclipse JET - Java Emitter Templates," Available at http://www.eclipse.org/emft/projects/jet/, 2007.

[Kang *et al.* 1990] Kang, K. C., S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study Technical Report," Software Engineering Institute, Carnegie Mellon University CMU/SEI-90-TR-21, 1990.

[Kleppe *et al.* 2003] Kleppe, A., J. Warmer and W. Bast., *MDA Explained, The Model-Driven Architecture: Practice and Promise*: Addison Wesley, 2003.

[Maßen *et al.* 2002] Maßen, T. v. d. and H. Lichter, "Modeling Variability by UML Use Case Diagrams," REPL'02, Essen, Germany, 2002.

[MDA 2007] OMG, "Model Driven Architecture Guide Version 1.0.1," Available at http://www.omg.org, 2007.

[Mens *et al.* 2005] Mens, T. and P. V. Gorp, "A Taxonomy of Model Transformation and its Application to Graph Transformation," International Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia, 2005.

[Merks *et al.* 2006] eclipse.tools.emf newsgroup thread, "Making EMF models valid Ecore models for a two-level code generation," Available at http://dev.eclipse.org/newslists/news.eclipse.tools.emf/msg20713.html,

[MOF 2006] OMG, "Meta Object Facility (MOF) 2.0 Core Specification (formal/06-01-01)," Available at http://www.omg.org, 2006.

[OCL 2006] OMG, "Object Constraint Language Specification v2.0 Final Adopted Specification (formal/06-05-01)," Available at http://www.omg.org, 2006.

[OptimalJ 2007] Compuware, "OptimalJ," Available at http://www.compuware.com/products/optimalj/, 2007.

[Pohl *et al.* 2005] Pohl, K., G. Böckle and F. v. d. Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*: Springer-Verlag, 2005.

[QVT 2005] OMG, "MOF QVT Final Adopted Specification (ptc/05-11-01)," Available at http://www.omg.org, 2005.

[Shaw *et al.* 1996] Shaw, M. and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*: Prentice Hall Publishing, 1996.

[SmartQVT 2007] France Telecom, "SmartQVT - Open Source Transformation Tool Implementing the MOF 2.0 QVT-Operational Language," Available at http://smartqvt.elibel.tm.fr/, 2007.

[UML 2005] OMG, "Unified Modeling Language Version 2.0: Superstructure (formal/05-07-04)," Available at http://www.omg.org, 2005.

[VIATRA 2007] Budapest University of Technology and Economics (BME), "VIsual Automated model TRAnsformations - VIATRA," Available at http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html, 2007.

6. Conclusion

"The important thing is not to stop questioning"
Albert Einstein

This chapter is dedicated to the analysis of the work and proposals presented in this thesis. We also discuss possible future research topics and the applicability of our proposals.

6.1 Discussion

This thesis proposes the adoption of a model driven approach to the development of software product lines. The combination of model driven and software product line approaches promises new possibilities to the field of software engineering. A symptom of this is the number of presentations, tutorials and demonstrations that explored this approach in the last Software Product Line Conference[7]. However, similarly to model driven approaches applied to single system development, this has been done essentially in the context of platform independent and platform specific models. The focus of this thesis and its contributions are essentially in the context of computational independent models and its usage in model driven development of software product lines.

6.1.1 Research Contributions

We will now discuss the research contributions of this thesis and how they address the research goals.

Chapter 3 was concerned with a methodological approach for model driven development of variability focused systems, particularly software product lines. We have explored how a variability *dimension* could be added to computation independent models and how they could be used to specify the functional requirements of product lines. We also described how these specifications could be used to support the derivation of the architectural requirements of a product line. We exemplified our approach by adapting 4SRS, a single system UML-based model driven development technique, to support the development of software product lines. We demonstrated the approach by applying it to a cellular phone product line described in [Muthig *et al.* 2004].

Research Contribution 1: *A UML-based transformational technique which supports the derivation of the functional requirements of the logical architecture of a software product line in the form of an object model based on a use case model of the product line in which the variability 'dimension' is added to the use case model by using stereotypes and the Extend relationship.*
This contribution addresses research goals 1 and 2 of this thesis that were presented in Chapter 1.

The second half of this chapter presents a proposal to support the transformation of analysis models into architectural models. Our proposal was based on an evolution of the 4SRS method to support the model driven development of software product lines that we called MoDeLine. We also

[7] Program available at http://sec.ipa.go.jp/SPLC2007/.

delineate some approaches to detail the first logical architecture of a system by integrating design patterns in the proposed approach. We have proposed the concept of use case realization as a way to bridge between the problem domain and the solution domain. We discuss how use case realizations can be supported by activity diagrams in the problem space and by component diagrams in the solution space. We have also discussed how use case realizations could integrate design models and design concepts such as the ones of UML-F. The approach was exemplified with a library product line case study.

Research Contribution 2: *A proposal of a 'bridging' technique between the problem space and the solution space for the model driven development of software product lines that is based on the concept of use case realizations and its double view: activity models for the problem space view and component models for the solution space view.*
This contribution addresses research goals 2 and 3 of this thesis that were presented in Chapter 1.

Chapter 4 was concerned with modeling and metamodeling to support variability focused systems. The first half of Chapter 4 describes a proposal to adapt the UML 2.0 metamodel in a way that effectively enables the adoption of use case diagrams in model driven approaches to software product line development. Particularly, we identified that the *Include* and *Extend* use case relationships only supported *alternative insertions*. We discussed the fact that other types of alternatives (alternative history, use case exception, alternative fragment and alternative cycle) are not directly supported by the UML 2.0 use case metamodel. Therefore, we have proposed an extension to the UML 2.0 metamodel to overcome this limitation. We have also proposed the adoption of activity diagrams to model use case behavior. We exemplified our approach with a library product line case study.

Research Contribution 3: *An extension to the UML 2.0 metamodel so that use case models can be adopted to model variability intensive systems and support the following kinds of alternatives: alternative insertion*; *alternative history; use case exception; alternative fragment; and alternative cycle.*
This contribution addresses research goal 1 of this thesis that was presented in Chapter 1.

The second half of the Chapter 4 describes a proposal to extend a UML profile for the design of frameworks and product lines called UML-F so that it includes requirements and analysis diagrams. Originally the UML-F profile only addressed UML design models [Fontoura *et al.* 2000]. We have discussed how the UML-F's *hook* and *template* base concepts used to manage variability at design could also be used and integrated *earlier* in the software development process, at the requirements and analysis phases. We have discussed our proposal in the context of an insurance product line case study and the MoDeLine model driven development method. We have also described how UML-F could be upgraded to comply with UML 2.0.

Research Contribution 4: *An extension to the UML-F profile so that it supports the modeling of variability in requirements and analysis models and maintains an integrated trace of the 'hook' and 'template' concepts throughout the analysis and design phases of the software development process.*
This contribution addresses research goals 1 and 3 of this thesis that were presented in Chapter 1.

Chapter 5 was dedicated to model transformations. In the first part of Chapter 5 we have proposed an approach of a mapping between use cases and features *formalized* through the QVT model transformation language. We also provided an implementation roadmap based on EMF and SmartQVT. The approach was exemplified with a library product line case study.

Research Contribution 5: *A 'formalization' of mappings between use cases and feature models that supports its automation by standard model transformation languages and metamodeling tools like QVT and EMF.*
This contribution addresses research goals 1 and 4 of this thesis that were presented in Chapter 1.

The second half of Chapter 5 presented a proposal to implement multi-staged domain modeling of software systems. We have seen that this is one of the common scenarios for software factories. We have explored how such an approach could be realized by the metamodeling and generative capabilities of tools such as EMF. The basic idea behind our approach is that a model in one stage can be the metamodel of the following stage. Therefore, we propose an approach to transform models into metamodels that is based on annotating models and reuse of the generative capabilities of the metamodeling tools. We argue that such an approach can be seen as a pattern of metamodeling. As such, we have detailed our proposals in a way similar to the description used for design patterns [Gamma *et al.* 1995]. We illustrated our approach with an insurance case study that resulted from our experimental work at I2S; a software house specialized in solutions for the insurance market[8].

Research Contribution 6: *A proposal to support the multi-stage domain modeling of software systems based on model to metamodel transformations that reuse the generative capabilities of metamodeling tools such as EMF and are guided by model annotations.*
This contribution addresses research goals 1 and 4 of this thesis that were presented in Chapter 1.

6.1.2 Publications

A PhD work is much more than the sum of its publications. However, they usually represent a significant effort that is required of the student. Having said that, we now present a list of publications related to this PhD in which we are author or co-author with our PhD supervisor. All the publications were peer reviewed.

1. "*Runtime Variability in Domain Engineering for Post-Deployment of User-Centric Software Functional Completion*", **Alexandre Bragança**, 1st Year PhD Technical Report, University of Minho, Guimarães, Portugal, December, 2003.

2. "*Runtime Variability in Domain Engineering"*, **Alexandre Bragança**, Software Engineering Doctoral Consortium, SEDES 2004, part of the 1st Portuguese Conference of Software Engineering, Coimbra, Portugal, April, 2004.

3. *"Run-time Variability Issues in Software Product Lines"*, **Alexandre Bragança** and Ricardo J. Machado. In "Implementation of Software Product Lines and Reusable Components", IESE-Report No. 122.04/E, Proceedings of the Implementation of Software Product Lines and Reusable Components Workshop at the 8th International Conference on Software Reuse (ICSR 8), Madrid, Spain, July, 2004.

4. *"Run-time Feature Realization based on Domain-Specific Platforms"*, **Alexandre Bragança** and Ricardo J. Machado, Poster session, 8th International Conference on Software Reuse (ICSR 8), Madrid, Spain, July, 2004.

5. *"Engenharia de Domínio no Suporte ao Aumento de Flexibilidade nos Sistemas de Software"*, **Alexandre Bragança** and Ricardo J. Machado, Proceedings of 5th International Conference on the Quality of Information and Communications Technology, Quatic2004, pp 15-21, Porto, Portugal, October, 2004 [ISBN 972-763-069-3].

[8] I2S company web site accessible at http://www.i2s.pt

6. *"A Methodological Approach to Domain Engineering for Software Variability Enhancement"*, **Alexandre Bragança** and Ricardo J. Machado, Proceedings of the Second Workshop on Method Engineering for Object-Oriented and Component-Based Development at OOPSLA 2004, pp 39-49, Vancouver, Canada, October, 2004 [ISBN 0-9581915-3-0].

7. *"Deriving Software Product Line's Architectural Requirements from Use Cases: an Experimental Approach"*, **Alexandre Bragança** and Ricardo J. Machado, Proceedings of the Second International Workshop on Model-Based Methodologies for Pervasive and Embedded Software at ACSD 2005, pp 77-91, St. Malo, France, June, 2005 [ISBN 952-12-1556-9].

8. *"Extending UML 2.0 Metamodel for Complementary Usages of the «extend» Relationship within Use Case Variability Specification"*, **Alexandre Bragança** and Ricardo J. Machado, Proceedings of 10th International Software Product Line Conference, SPLC 2006, pp 123-127, Baltimore, Maryland, August, 2006, IEEE CS Press [ISBN 0-7695-2599-7].

9. *"Adopting Computational Independent Models for Derivation of Architectural Requirements of Software Product Lines"*, **Alexandre Bragança** and Ricardo J. Machado, Proceedings of the 4th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software at ETAPS 2007, pp 91-101, Braga, Portugal, March, 2007, IEEE CS Press [ISBN 0-7695-2769-8].

10. "*Model Driven Development of Software Product Lines"*, **Alexandre Bragança**, Software Engineering Doctoral Consortium, SEDES 2007, part of the 6th International Conference on the Quality of Information and Communications Technology, Quatic 2007, pp 199-203, Lisbon, Portugal, September, 2007, IEEE CS Press [ISBN 0-7695-2948-8].

11. *"Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines"*, **Alexandre Bragança** and Ricardo J. Machado, Proceedings of 11th International Software Product Line Conference, SPLC 2007, pp 3-12, Kyoto, Japan, September, 2007, IEEE CS Press [ISBN 0-7695-2888-0].

6.1.3 Research Validation

In Chapter 1 we have discussed our research approach and also research validation. Figure 106 presents in gray the development activities that were, to some extent, covered by this thesis. Although we are very much aware of the limitations of our work, we are also aware of the extension of the topics covered and of time and other resources constraints. We now remember what Mary Shaw presents as being the validation approaches used in software engineering research works [Shaw 2003]: Analysis, Evaluation, Experience, Example, and Persuasion. Obviously, being this thesis about software engineering, we would like to have adopted an empirical research method based on the application of our proposals to case studies in order to evaluate the results. Unfortunately, that was not possible. Therefore, the validation of our work was done essentially based on our own experience and on feedback from users and domain experts during projects and meetings relating to the topics covered by this thesis and that were essentially experimented at I2S. Other source of validation were the experimental software developments we have done during this thesis that demonstrated that at least part of the methods and techniques presented in this thesis are technologically feasible and provide the results we have discussed. Usually these results were transmitted in the form of case studies.

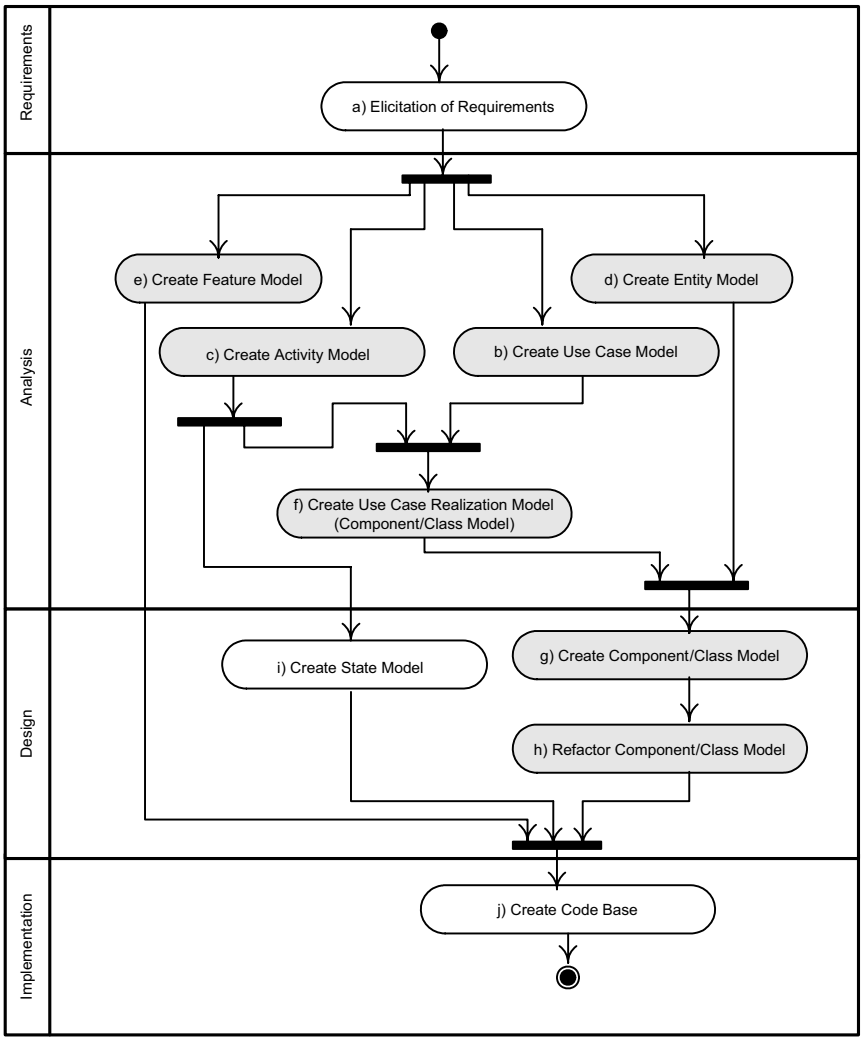

Figure 106: Development activities covered by this thesis.

By the time we are writing this thesis, an experimental project regarding the multi-staged model driven approach presented in Chapter 5 is running in a software house (I2S). We are certain that in the near future we will be able to report on the results of this project. We are also working on the application of our proposals to an open source ERP. We plan on report on how a model driven approach can impact the usually very complex tasks associated with the configuration management and implementation of ERP systems.

6.2 Perspectives and Future Work

The work presented in this thesis shows, at least in part, that model driven approaches to software product line development are feasible and have a potential to become the next major evolution in the software development practices. Signs of that evolution can be found in the software factories proposal from Microsoft [DSL 2007] or the integration of model driven and product line tools such as Gears from Biglever [Gears 2007] and Raphsody from Telelogic [Rhapsody 2007]. Our experimental developments with available metamodeling and model transformation tools also show that significant parts of the approach can be automated by available tools that can be integrated into

the major integrated development environments. However, some open issues must be addressed before mixed model driven and software product line approaches become a reality for the mainstream software development practitioners.

Fundamental issues
The major fundamental issue we can think of is: will models substitute code and, therefore, modeling substitute programming as we know it?

Of course we can not predict future, but we are convinced that in the near future models will substitute traditional coding in a way similar to what exists today between mainstream object-oriented programming languages and assembly programming. What happened in the past with object-oriented programming languages, class frameworks and component technology could happen in the near future with the conjunction of the promising emerging technologies and approaches: model driven development; product line development; and web services. So, what is missing for that promise to become a reality? We are still in the beginning, exploring essentially each technology in its one. An integrated foundation framework is missing, similarly to what happened with the Java and .Net programming platforms relatively to the object-oriented paradigm, software reuse and software components. In this thesis, we have only grasped such a foundation, by promoting modeling and, particularly, metamodeling and model transformations as first class tasks in the software development process. We have also focused our contributions on the notion of variability and, particularly, we have showed how the focus of the software development process can be promoted to concepts as high as domain-specific features and variation points. Naturally, nowadays, outside contexts with significant domain knowledge, domain artifacts and resources available, the generalized adoption of this approach by software practitioners will be restricted. Since the theoretical foundations of each *field* are becoming gradually stable and are shared by the community, the missing parts for the realization of a foundation framework are tool and methodological support.

Tool support
All the metamodeling and transformation approaches presented in this thesis where implemented and tested with several tools. These tools included EMF [EMF 2007], GMF [GMF 2007], GME [Ledeczi *et al.* 2001] and Microsoft DSL Tools for Visual Studio [DSL 2007] for metamodeling and SmartQVT [SmartQVT 2007], ATL [ATL 2007b] and openArchitectWare [openArchitectureWare 2007] for model transformation. Although we have not explored transformations between model and code there are also several available tools for this particular task. Therefore, we could say that there is already a significant offer of tools and, at least, they cover the technical requirements of the approaches discussed in this thesis.

The open issues we see as requiring further research are the ones relating to building and sharing reusable models and metamodels. For instance, if someone wanted to reuse the UML 2.0 use case modeling extensions proposed in Chapter 4 it could simple reuse the EMF metamodel. However, an EMF metamodel only contains the abstract syntax of the composition of the modeling elements. It does not contain concrete syntax or semantics associated with the metamodel. These parts of a domain-specific modeling environment are usually specified in code (in the Eclipse modeling project, GMF provides support for modeling a significant part of the graphical user interface) with the exception, probably, of model constraints that can be also specified, for instance, in an OCL model. As such, the resulting domain-specific modeling environment is still a code base project and is usually deployed as an Eclipse plugin. An Eclipse plugin is reusable according to code reuse rules, not *model reuse rules*. An approach could be to provide the metamodeling tools with support for modeling semantics, for instance, by supporting action modeling. An example of how EMF can be extended in such a way can be found in the Kermeta project [Kermeta 2007].

Another related technical issue that must be tackled before a mainstream adoption of model driven approaches is the automatic support for maintaining and enforcing relations between model elements. Actually the invocation of queries to verify relations and transformations to enforce them is a manual activity. These are the fundamental technical issues that, in our perspective, are required to be addressed by the research community in order to model driven development approaches become mainstream technology. We see that a possible approach could be addressed by the concept of *metamodel and model modules*. These reusable modules should include: metamodels with abstract and concrete syntax (visual and/or textual); validation rules; mapping rules (relations and transformations); metamodel semantics; and models (i.e., *instances* of metamodels). These modules could then be shared, for instance, using web services. Actually, for instance, ATL transformations sources can be accessed at the *ATL transformation zoo* web site and some transformations are also accessible as web services [ATL 2007a]. Of course, all of these elements should be specified also as models. Metamodeling tools would have to support these concepts.

An issue that naturally would rise is that of interoperability between different metamodeling tools. This kind of issue is actually being addressed by the concept of *technical spaces* [Bezivin 2005]. These are generic issues of model driven approaches, not specific to software product line approaches. These are also deep technical issues which require significant research resources and, as such, will probably not be our next research topics.

In the near future we see our research effort being focused on more pragmatic issues that are directly related to the work presented in this thesis that we see feasible to accomplish with the actual existing tools. Similarly to the approaches discussed in this thesis, we propose to tackle these issues with techniques and methods which can be supported by current tools or by adapting existing tools.

Technical and Methodological support

This thesis is fundamentally about software development methods. Particularly, we have approached very specific issues in tasks that generically can be a part of a process for the model driven development of software product lines. These tasks are presented in Figure 106. There are much more components of a software engineering method. For instance, we have left out several very important tasks, such as: scoping; management; deployment; configuration; quality assessment; and evolution. We also did not explore the global process flow, e.g., data-flow between tasks, iterations, and roles. So, there are a lot of potential research fields to explore. Next, we will, however, only discuss the topics that we see as natural follow up research issues of this thesis.

If we observe Figure 106, we see three development tasks that have not been addressed by this thesis: elicitation of requirements; create state model and create code base.

Regarding requirements, there are essentially three *kinds* of sources of domain knowledge: human; non-structured textual documentation; structured technical documentation (e.g., source code and models). We see much space for automating as much as possible the task related to requirements identification and elicitation. Some approaches were discussed in Chapter 2. This kind of approaches could be of great help for companies trying to convert traditional approaches of software development to the product line approach. For such, tools for manipulating source code artifacts, such as, ASF+SDF could be adopted [Brand *et al.* 2001]. However, some issues still have to be tackled. For instance, how to identify variation points, features, and use cases from existing source code?

Other important aspect in introducing a software product line approach in a company, which can also be automated to some degree, is the reuse of existing components for the implementation of features. Here, we see some potential to with our approach of mappings between features, use cases and use case realizations (i.e., component collaborations). These mappings could help the matching between features and its characteristic (e.g., binding time) and the characteristics of potential reusable components.

For the *create code base* task (see Figure 106) we see two possible approaches. One is to consider that code is also a model. This is a more radical approach that we do not see as a realistic research topic for us, at least before the previously discussed model driven fundamental issues are tackled. Another approach is to adopt a mixed model and code based approach. In this case, code automatically generated and maintained based on models and hand written code must coexist. One approach to keep of booth code bases coherent is to join them by using interface based design patterns, such as the ones presented in [Gamma *et al.* 1995] (e.g., factory, strategy and bridge). For the generation of code from models several approaches can be used, such as MOF Model to Text [Mof2Text 2007] and JET [JET 2007]. Related to this issue is the task *create state model* (see Figure 106). Using state machines to model the state of components and classes is a common development task. Modeling variability in state machines has also been discussed in [Gomaa 2005]. Possible approaches for the generation of code from state machine models with variability have been presented and discussed in [Chauvel *et al.* 2005].

Multi-staged domain specific modeling is a topic of particular interest to us. This interest has its origins in a problem we have identified during a recent project with a software development company. The approach we have presented in Chapter 5 was to some extent inspired by a similar approach described in [Czarnecki *et al.* 2005]. Although our approach is aimed at tackling the specific problems that were described in Chapter 5, we see the approach as having a lot of potential for covering related issues, for instance, to support the multi-staged configuration of feature models. Therefore, the investigation of the possibility of generalization of the approach is a natural follow up of the work presented in this thesis.

We are certain that we have only tackled specific issues in this field of research and that we are still far from a complete model driven approach to software product line development. However, we truly hope this work can help practitioners in the current overwhelming task of adopting model driven and product line approaches. For the research community, this thesis contributes to the foundations of the novel model driven approach to software product line development, particularly at the computation independent model level. We hope this thesis contributes to a deeper interest in research topics related to the model driven development of software product lines, which we see as a very promising approach in software engineering.

6.3 References

[ATL 2007a] INRIA, "ATL Transformations Zoo," Available at http://www.eclipse.org/m2m/atl/atlTransformations/, 2007.

[ATL 2007b] INRIA, "Atlas Transformation Language," Available at http://www.eclipse.org/m2m/atl/, 2007.

[Bezivin 2005] Bezivin, J., "Model Driven Engineering: Principles, Scope, Deployment and Applicability," Generative and Transformational Technics in Software Engineering, Braga, Portugal, 2005.

[Brand *et al.* 2001] Brand, M. G. J. v. d., A. v. Deursen, J. Heering, H. A. d. Jong, M. d. Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, Scheerder, J. J. Vinju, E. Visser and J. Visser, "The Asf+Sdf Meta-Environment: a Component-Based Language Development Environment," IEEE Conference in Computational Complexity, 2001.

[Chauvel *et al.* 2005] Chauvel, F. and J.-M. Jezequel, "Code generation from UML models with semantic variation points," MODELS/UML'2005, Montego Bay, Jamaica, 2005.

[Czarnecki *et al.* 2005] Czarnecki, K., S. Helsen and U. Eisenecker, "Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models," *Software Process Improvement and Practice, special issue on "Software Variability: Process and Management"*, vol. 10, pp. 143-169, 2005.

[DSL 2007] Microsoft, "DSL Tools for Visual Studio," Available at http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx, 2007.

[EMF 2007] Eclipse Foundation, "Eclipse Modeling Framework," Available at http://www.eclipse.org/emf/, 2007.

[Fontoura *et al.* 2000] Fontoura, M., W. Pree and B. Rumpe, "The UML Profile for Framework Architectures," 2000.

[Gamma *et al.* 1995] Gamma, E., R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[Gears 2007] BigLever, "Gears," Available at http://www.biglever.com/solution/product.html, 2007.

[GMF 2007] Eclipse Foundation, "Graphical Modeling Framework," Available at http://www.eclipse.org/gmf/, 2007.

[Gomaa 2005] Gomaa, H., *Designing Software Product Lines with UML*: Addison Wesley, 2005.

[JET 2007] Eclipse Foundation, "Eclipse JET - Java Emitter Templates," Available at http://www.eclipse.org/emft/projects/jet/, 2007.

[Kermeta 2007] Triskell Project, "Kermeta," Available at http://www.kermeta.org/, 2007.

[Ledeczi *et al.* 2001] Ledeczi, A., M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle and P. Volgyesi, "The Generic Modeling Environment," WISP'2001, Budapest, Hungary, 2001.

[Mof2Text 2007] OMG, "MOF Models to Text Transformation Language, Beta 2, (ptc/07-08-16)," Available at http://www.omg.org, 2007.

[Muthig *et al.* 2004] Muthig, D., I. John, M. Anastasopoulos, T. Forster, J. Dorr and K. Schmid, "GoPhone - A Software Product Line in the Mobile Phone Domain," IESE 025.04/E, 2004.

[openArchitectureWare 2007] openArchitectureWare.org, "openArchitectureWare platform for model driven software development," Available at www.openarchitectureware.org, 2007.

[Rhapsody 2007] Telelogic, "Rhapsody," Available at http://modeling.telelogic.com/, 2007.

[Shaw 2003] Shaw, M., "Writing Good Software Engineering Research Papers," 25th International Conference on Software Engineering, 2003.

[SmartQVT 2007] France Telecom, "SmartQVT - Open Source Transformation Tool Implementing the MOF 2.0 QVT-Operational Language," Available at http://smartqvt.elibel.tm.fr/, 2007.

Appendix A: Experimental Implementation of Use Case Modeling Environment

This appendix presents details about the experimental implementation of a use case modeling environment for supporting the MoDeLine method. This experimental implementation was done using the Generic Modeling Environment (GME).

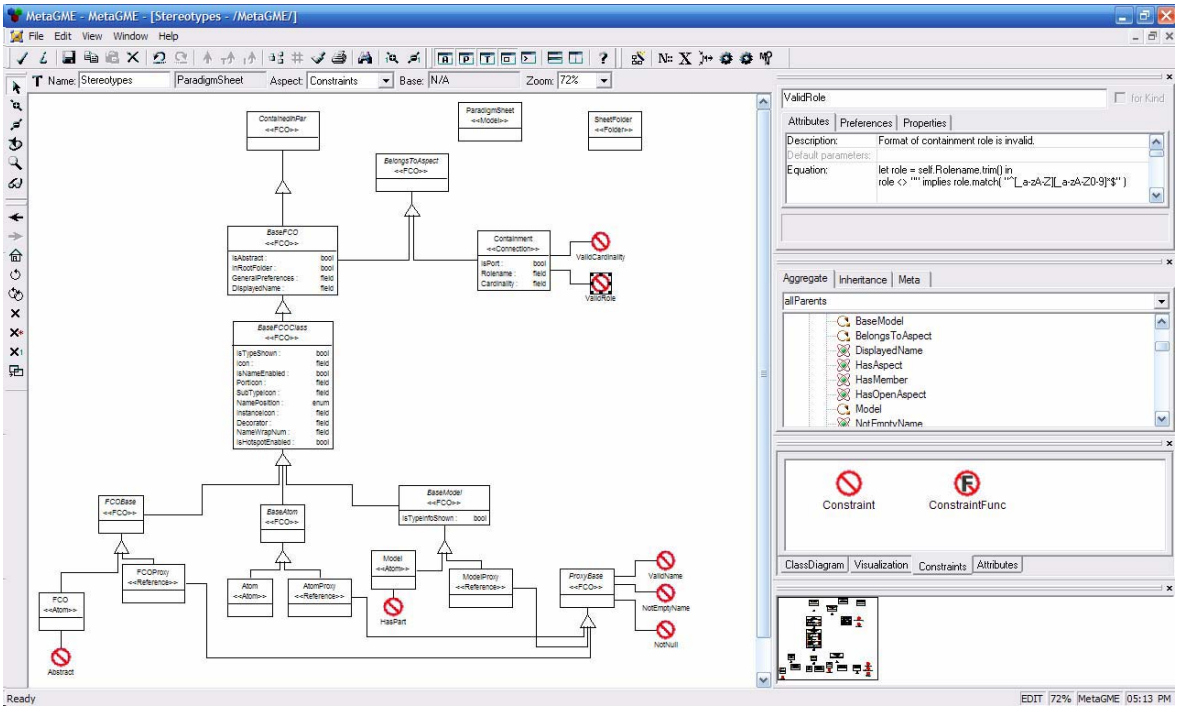

Figure 107: Editing constraints in the MetaGME paradigm.

GME is part of the MultiGraph Architecture (MGA), a toolset for creating model-integrated program synthesis (MIPS) environments. These tools are publicly available at http://escher.isis.vanderbilt.edu/.

MIPS environments provide a means for evolving domain-specific applications through the modification of models and re-synthesis of applications. In GME, a modeling paradigm defines the family of models that can be created using the resultant MIPS environment. A metamodel is a formalized description of a particular modeling language, and is used to configure GME itself. In Figure 107 we see how the GME environment is used to model the *MetaGME* paradigm. The *MetaGME* is the paradigm that is used in the synthesis of a modelling environment that *transforms* GME in a MIPS environment. Not all MIPS environment is implemented using models; some components are implemented using Microsoft's COM technology as extensions to GME (e.g., model decorators and model interpreters). It is also possible to observe in Figure 107 some of the modeling concepts used to create paradigms.

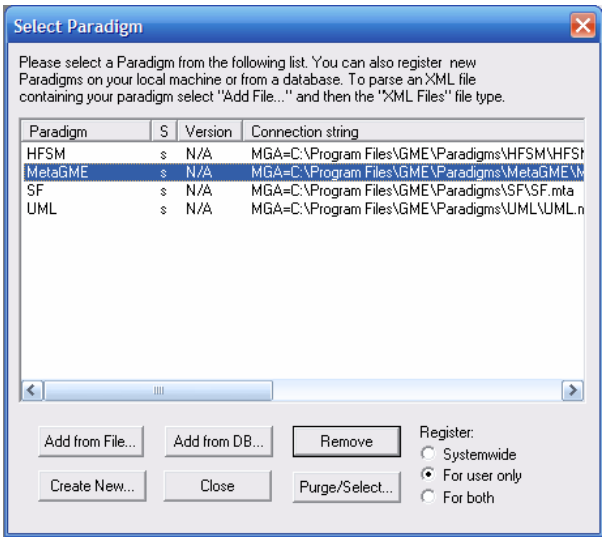

Figure 108: Creating a new modeling project based on MetaGME.

In GME, all models must be created from registered metamodels (i.e., paradigms). For instance, for creating the use case modeling environment presented in this appendix the *MetaGME* paradigm must be selected, as illustrated in Figure 108.

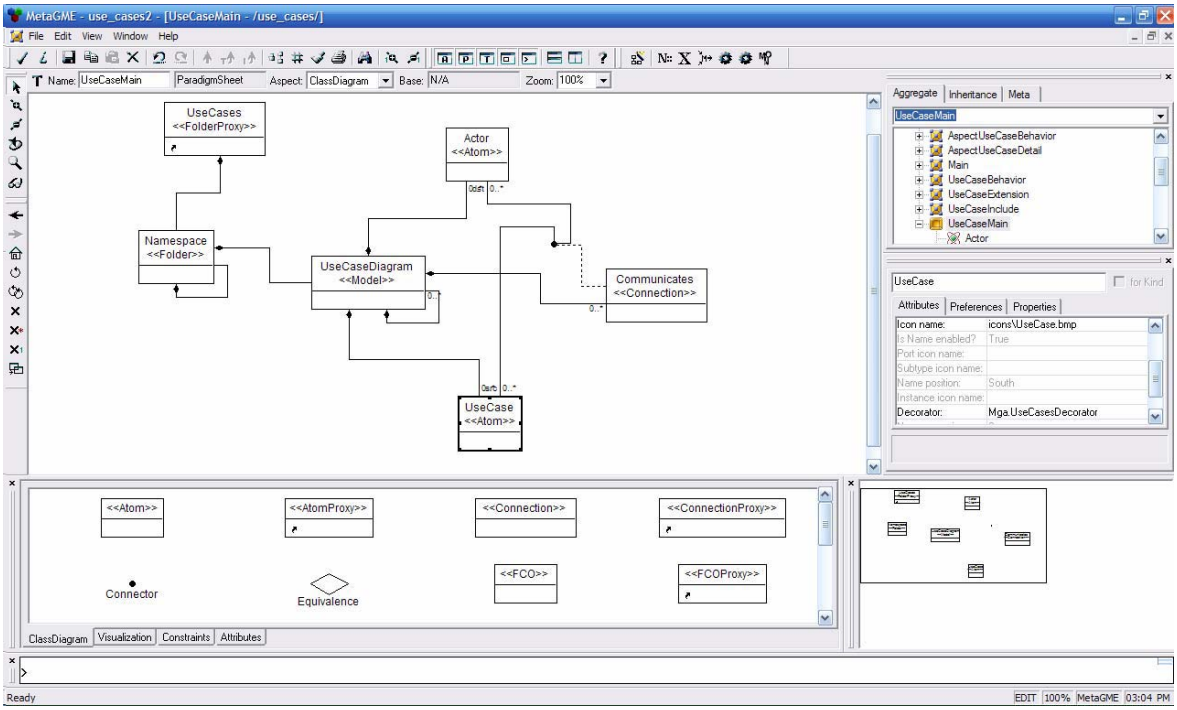

Figure 109: GME metamodel paradigm sheet for *Use Case*.

Figure 109 presents GME *adapted* according to the *MetaGME* paradigm. It is also possible to observe part of the use case metamodel specified using the *MetaGME* modeling concepts that appears on the bottom of Figure 109.

Figure 110 illustrates another part (*paradigm sheet*) of the use case metamodel. The part of the metamodel that is visible contains details of the specification of the *Extend* relationship.

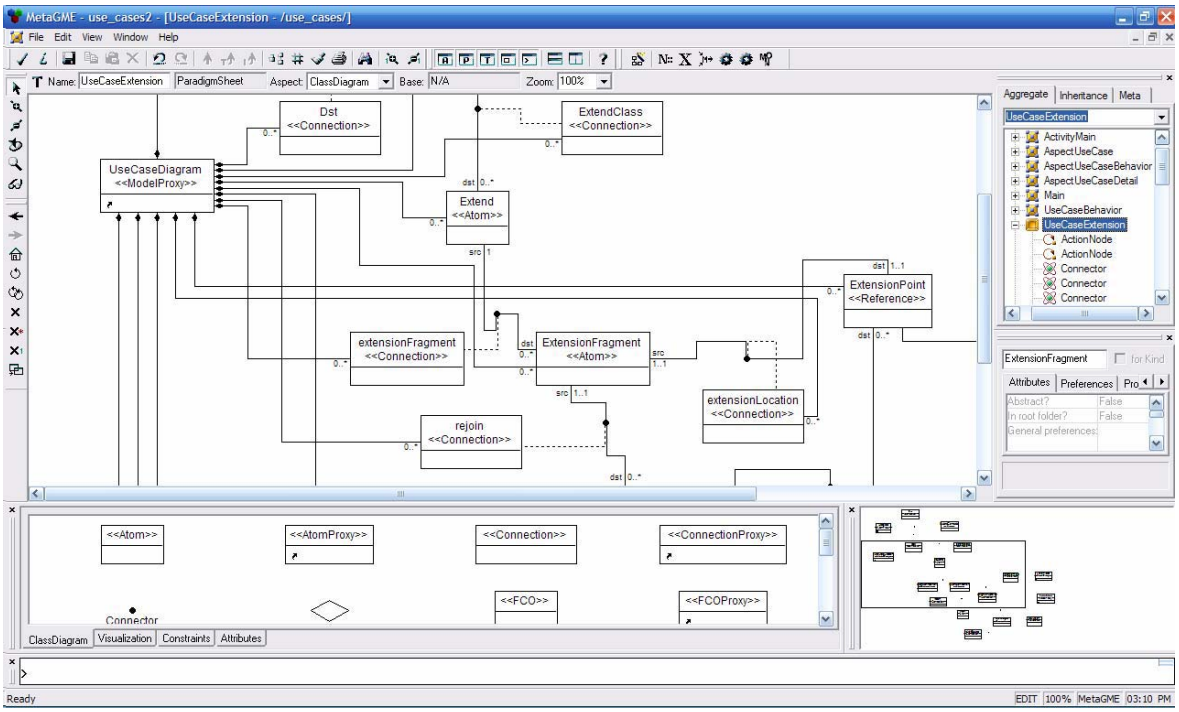

Figure 110: GME paradigm sheet for the *Extend* relationship.

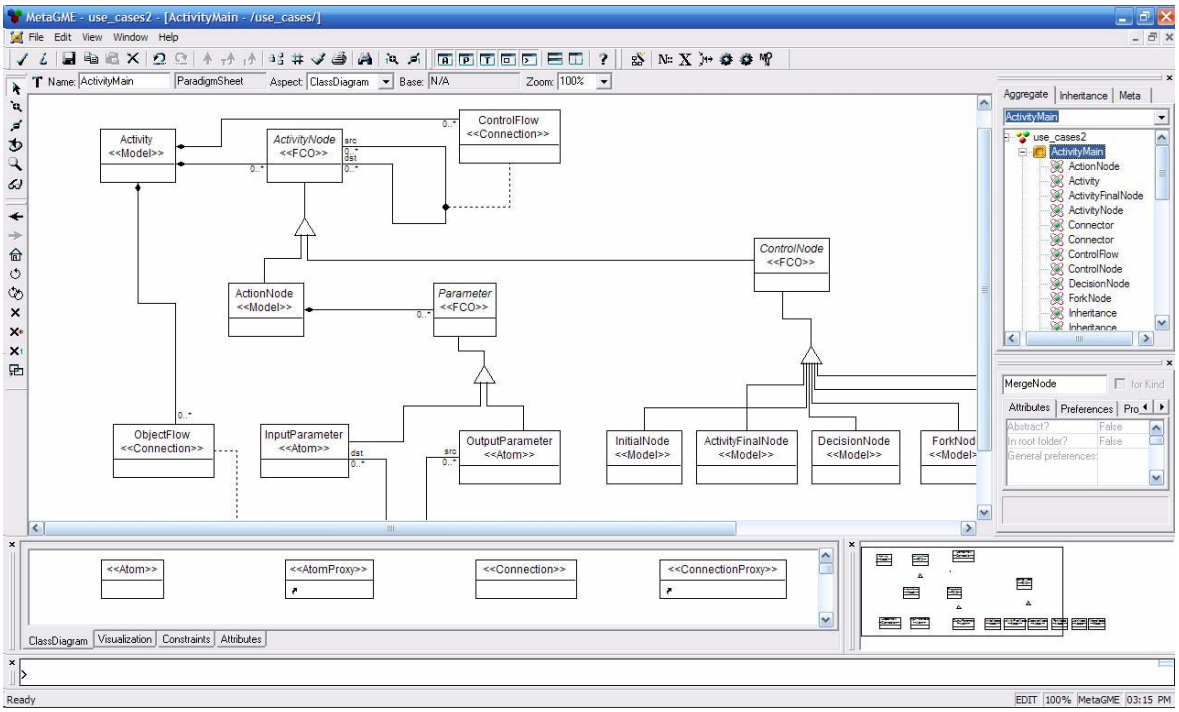

Figure 111: GME paradigm sheet for modeling the behavior of use cases with *Activities*.

Figure 111 presents the part of the metamodel that specifies the behavior of use cases. As the reader can observe from Figure 111, the behavior of use cases is based on activity diagrams.

Not all components of the MIPS environment are specified by models. For instance, Figure 112 illustrates the *decorator* that was developed in Microsoft's Visual Studio to adapt some parts of the visual representation of model elements (for which GME automatically provides default visual representations and visual editors) according to the visualization requirements for our use case modeling environment.

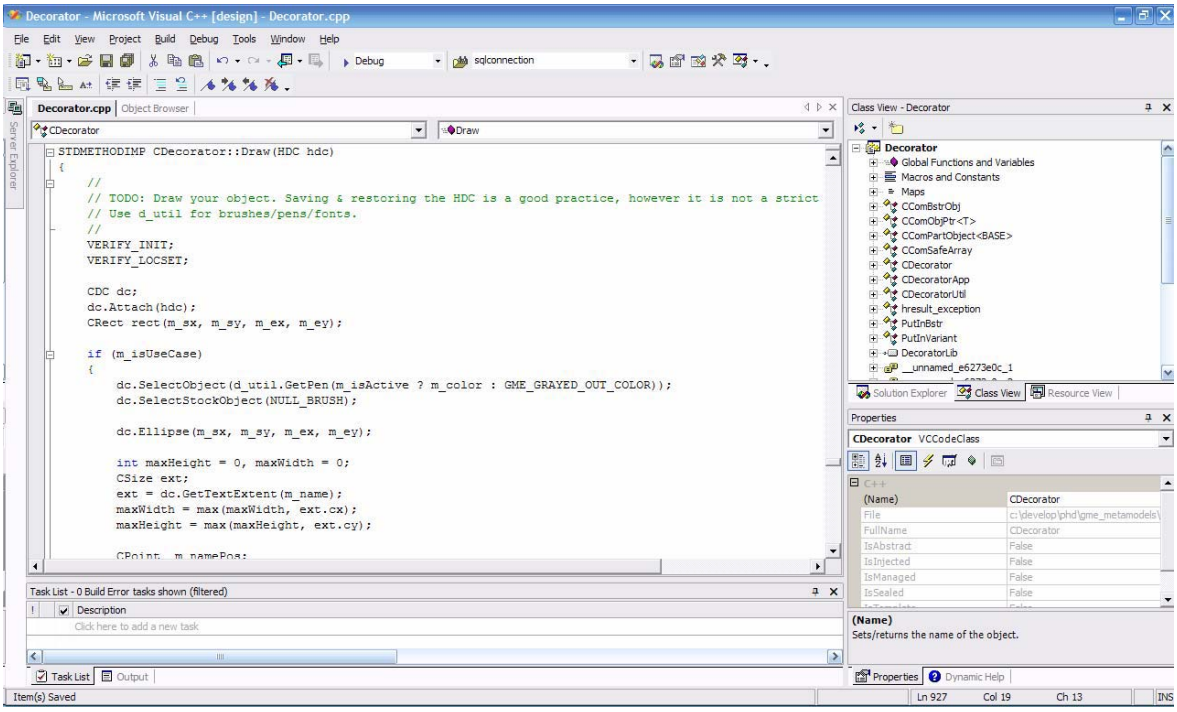

Figure 112: Implementing the COM use case decorator in Microsoft Visual Studio.

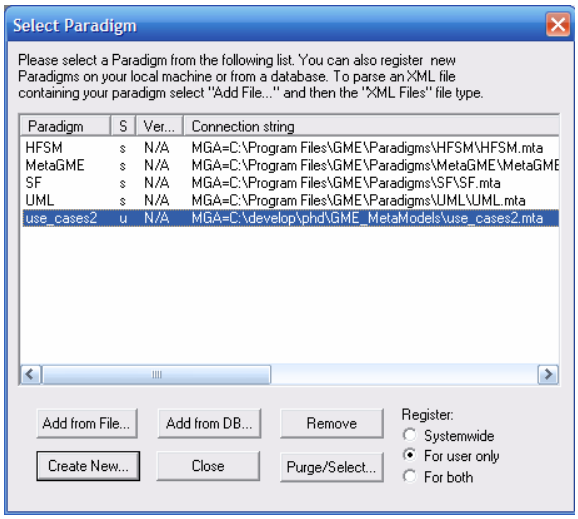

Figure 113: Creating a new GME project based on the new use case paradigm (metamodel).

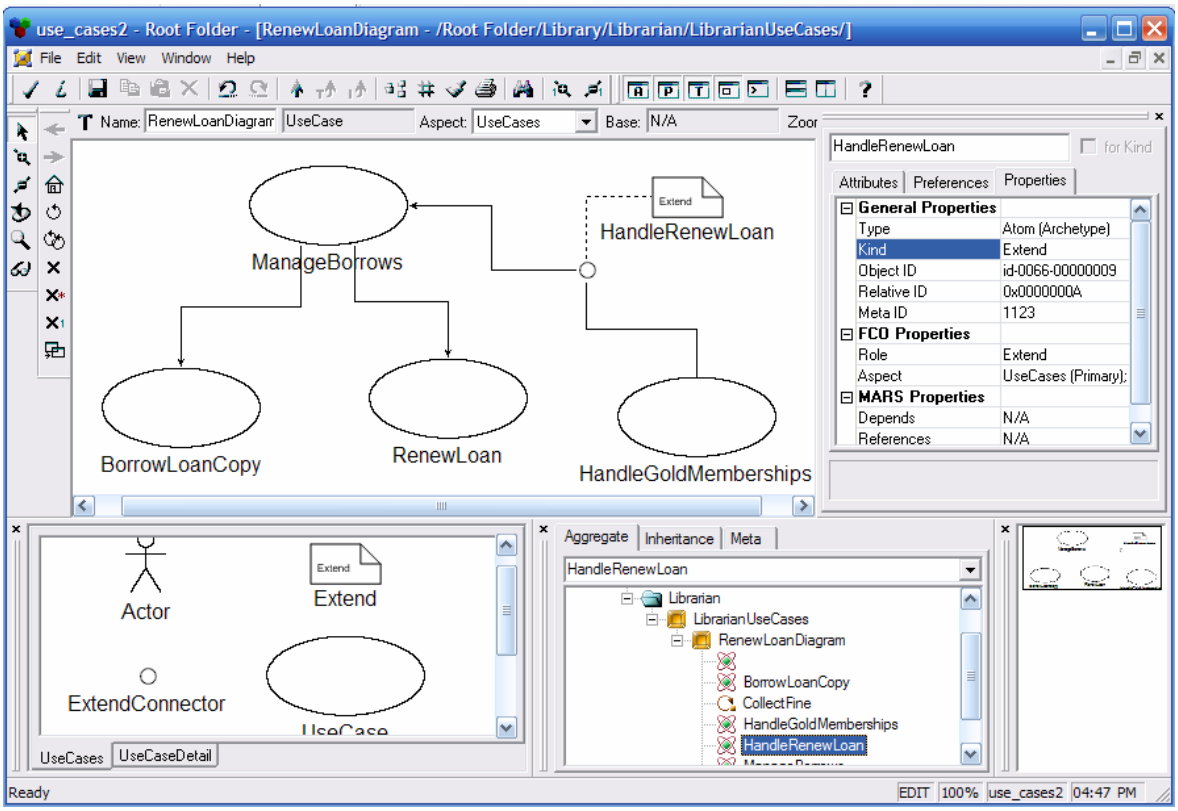

Figure 114: Editing use case models with the GME environment *adapted* to the new use case paradigm.

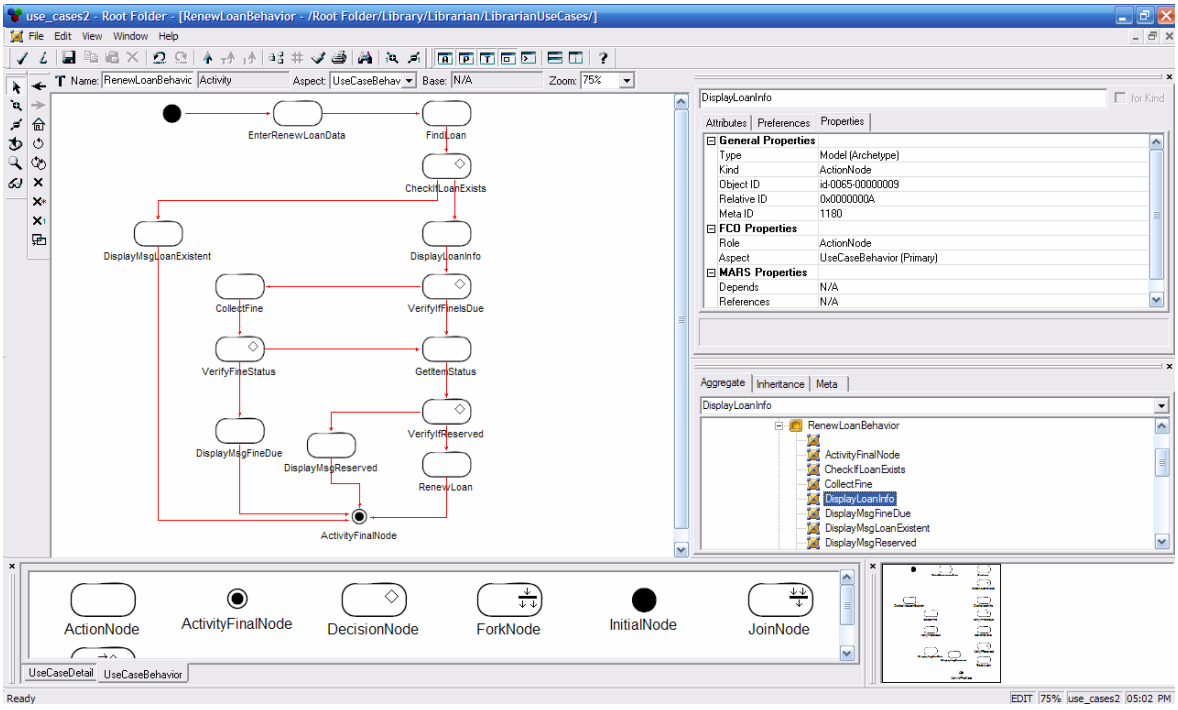

Figure 115: Modeling the behavior of use case *Renew Loan* with activity diagrams.

Once the metamodel is completed and validated (i.e., passes all the validation constraints on the *MetaGME* paradigm) it can be registered and used to create models. Figure 113 presents the use case paradigm registered and selected for *adapting* GME to be used as a use case modeling environment.

Figure 114 presents GME being used to model use cases for a library software product line. In Figure 115 we see another perspective of the use case modeling environment. In this case it is being used to model the behavior of a use case with an activity diagram.

Appendix B: Experimental Implementation of Model Transformations

This appendix is about experimental implementation of model transformations between use cases and feature models with EMF and SmartQVT. These experimental implementations are related to the topics discussed in the first part of Chapter 5.

EMF is a project of Eclipse that consists of a modeling framework and code generation facility for building tools and other applications based on a structured data model. This tool is available at http://www.eclipse.org/modeling/emf/.

SmartQVT is a model transformation tool implementing the MOF 2.0 QVT-Operational language. This tool is available at http://smartqvt.elibel.tm.fr/. The SmartQVT accepts as input for the model transformations models in the EMF format.

EMF and SmartQVT are plugins of the Eclipse open development platform available at http://www.eclipse.org/.

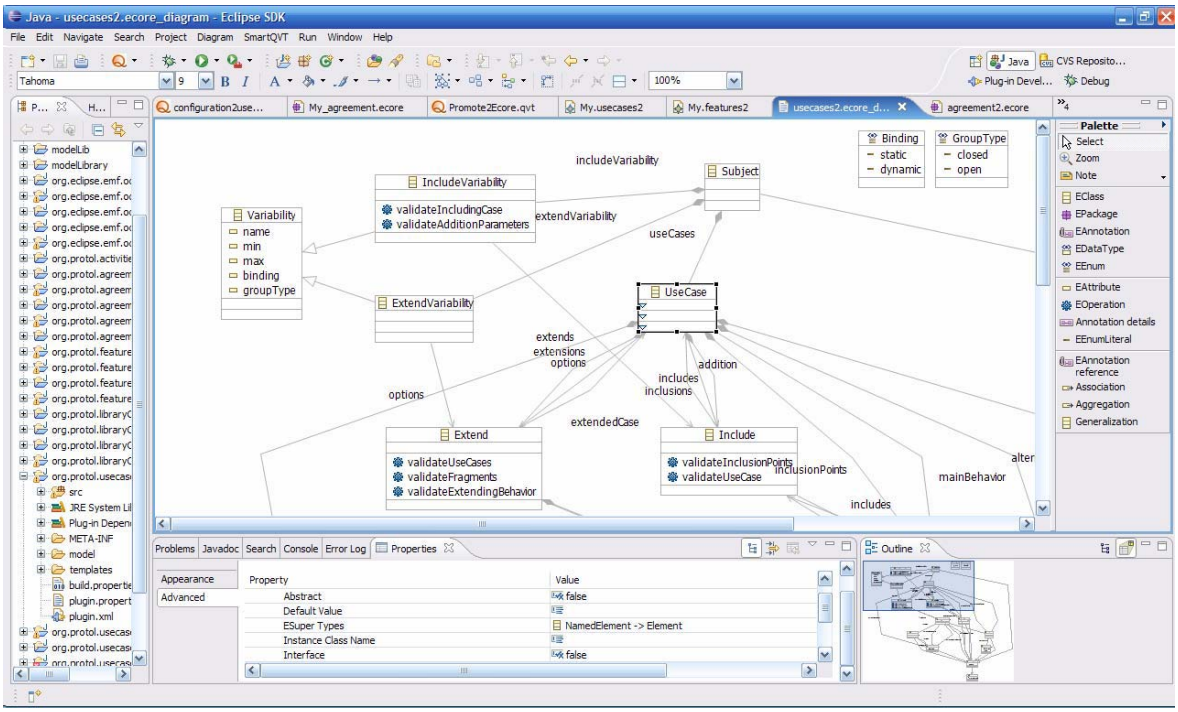

Figure 116: Editing, with EMF, a domain use case metamodel, i.e., a use case metamodel with support for variability annotations.

Figure 116 presents how EMF can be used to construct a metamodel for supporting the creation of an environment for modeling domain use cases. The modeling concepts that can be used to create metamodels are visible in the tool palette. Since the initial design of EMF was inspired by

the MOF standard, the concepts that appear in the tool palette are based on the MOF concepts. In fact, the metamodel used in EMF (Ecore) corresponds to a subset of MOF called EMOF (Essential MOF).

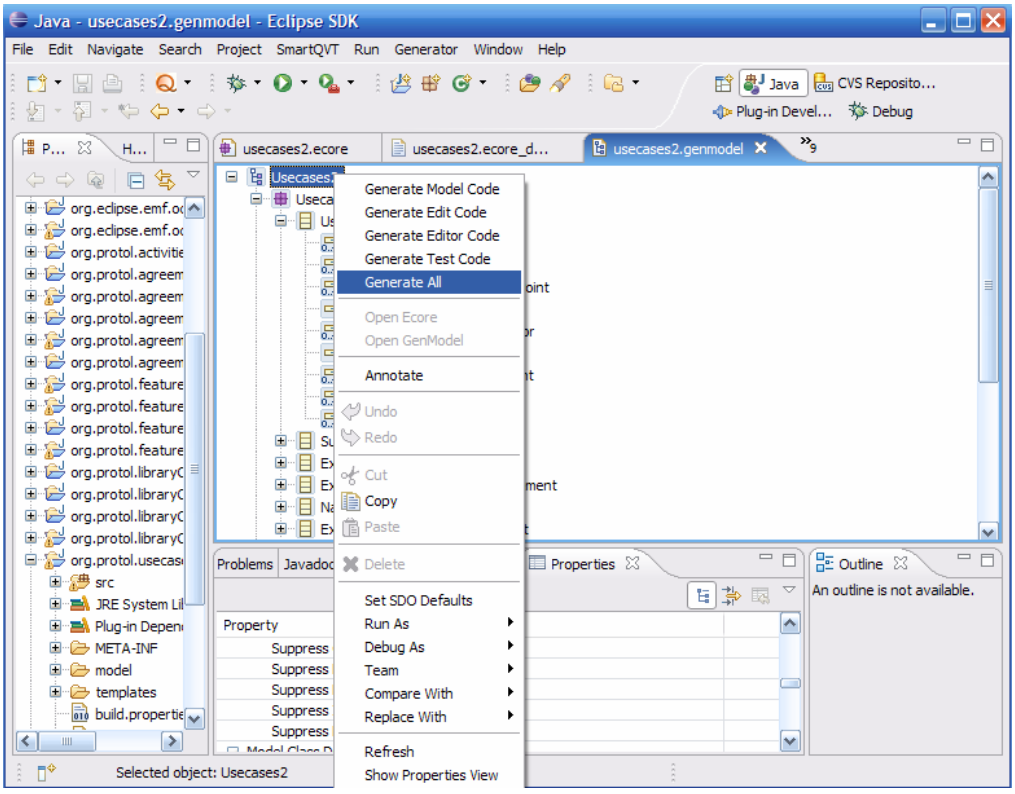

Figure 117: Generating model, edit, editor and test code from the use cases *genmodel*.

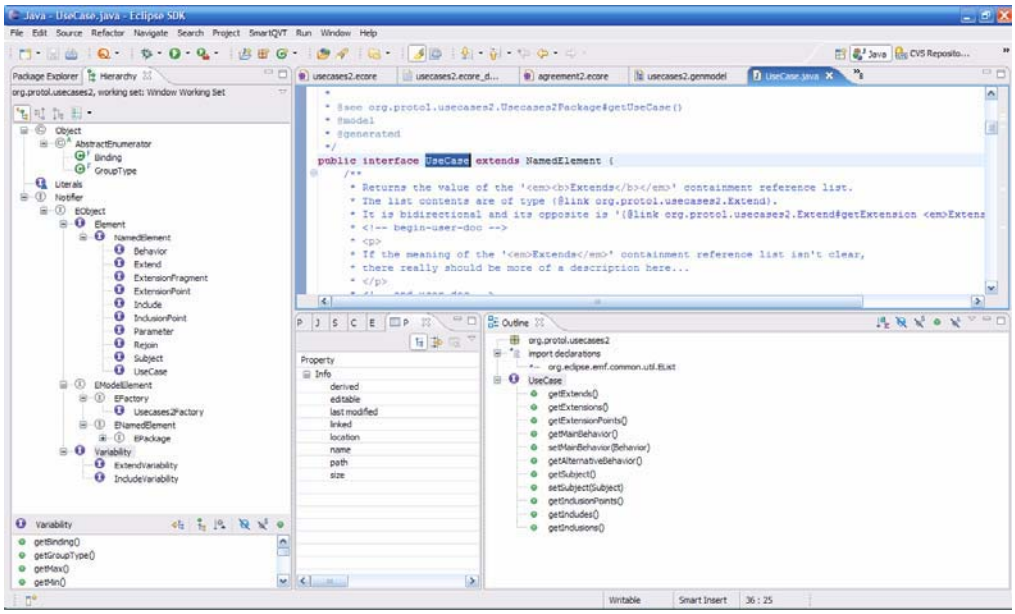

Figure 118: Inspecting the generated code for the *UseCase* element.

After creating a metamodel with EMF it is possible to generate code to support an environment for constructing models that are in conformance with the metamodel. For that, a *genmodel* must be created. A *genmodel* is basically a model that is based on the created metamodel, but has annotations that configure the generation process. It can be automatically created by EMF based on a metamodel. In Figure 117 we see how the generation process can be invoked.

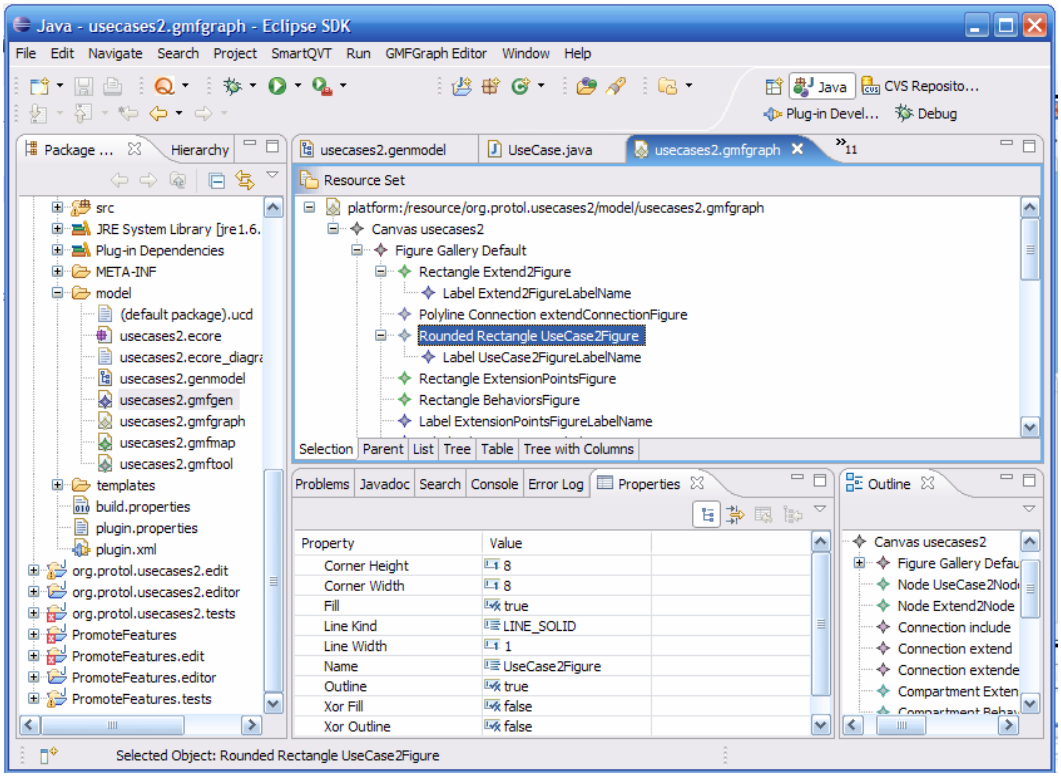

Figure 119: Editing GMF graph metamodel for the domain use case metamodel.

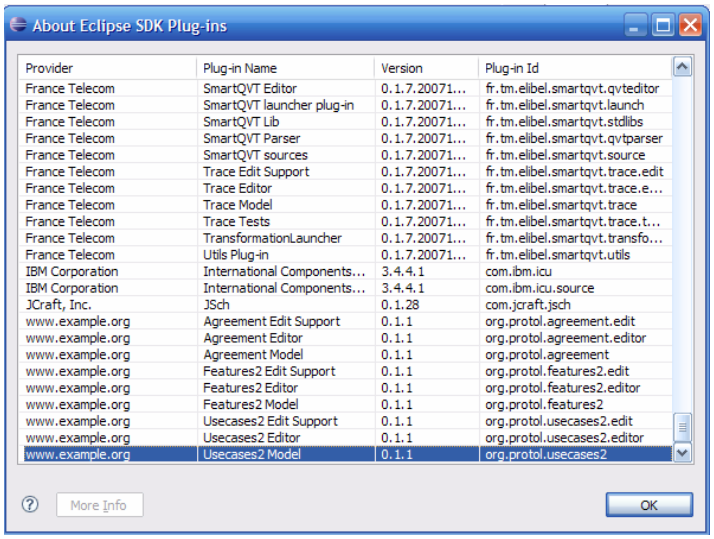

Figure 120: Domain use case model plugins installed in Eclipse.

Figure 118 presents several views of the code that was generated by EMF based on the metamodel of the domain use cases presented in Figure 116. The code generated is included in Eclipse plugin projects. Therefore, if properly deployed to an Eclipse installation they extend that Eclipse installation to support the generated modeling environments. Figure 120 presents an Eclipse installation with some modeling plugins configured.

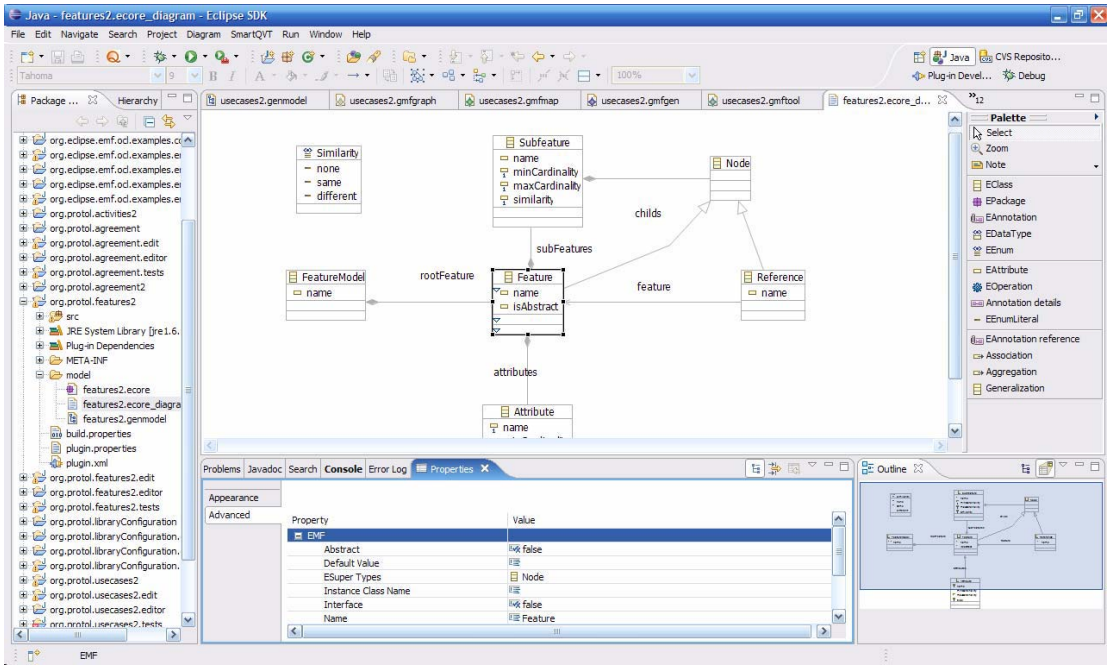

Figure 121: Editing feature metamodel in EMF.

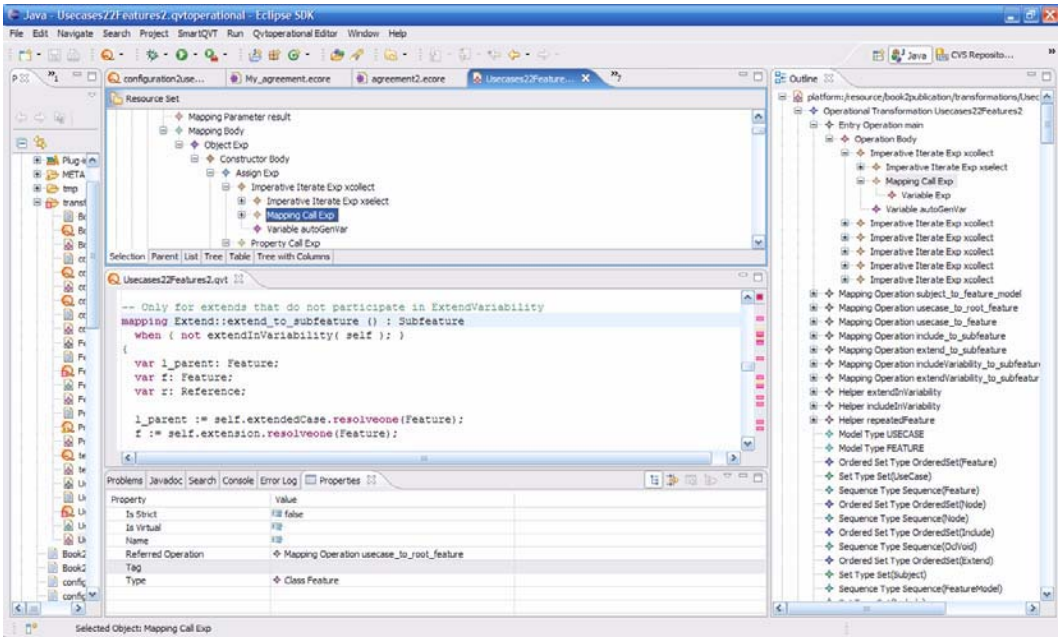

Figure 122: Editing QVT code and inspecting generated QVT operational model for use case to feature transformation.

Figure 121 presents the metamodel for features. This metamodel is used for the generation of the modeling environment for supporting modeling feature diagrams. This process is similar to the one described for domain use case models.

The graphical support generated by EMF for the edition of models is based on a *tree* interface. If the generated code is not adequate for specific requirements it can be adapted manually, since EMF has support for integrating generated code and manually edited code. Other possibility consists in using GMF (Graphical Modeling Framework) to generate more complex graphical visualization and editing support for EMF metamodels. GMF is also an Eclipse project and is available at http://www.eclipse.org/gmf/. For instance, Figure 119 presents how GMF was used to generate a graphical representation for the domain use case models that was more similar to the presentation notation proposed by the UML standard.

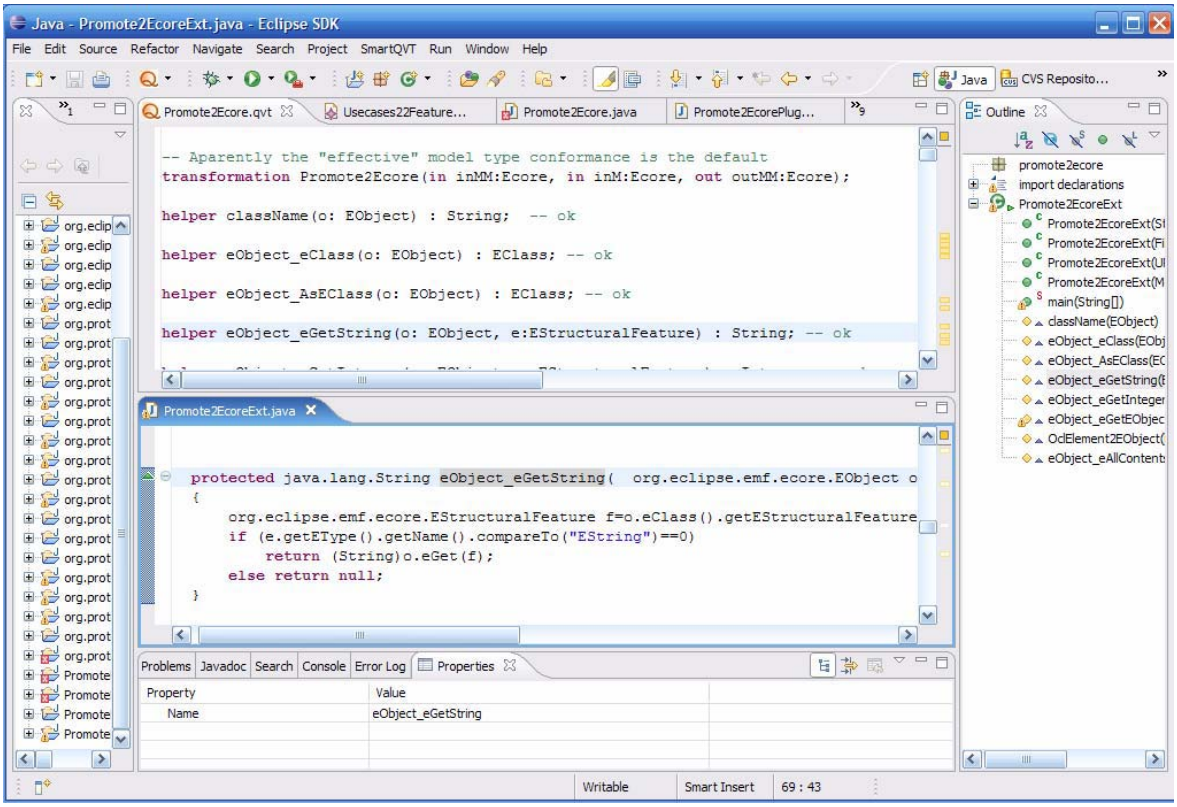

Figure 123: Coding, in Java, a helper function used in the feature to *Ecore* transformation.

Figure 122 presents the creation of the domain use case model to feature model transformation specification using SmartQVT. In the figure it is possible to observe not only the source code of the transformation in the QVT operational language but also the model that represents the transformation code. This model is generated by the SmartQVT parser. The model that results from the parser is used as input for the SmartQVT compiler. The result of the compiler is Java code that implements the transformation.

The QVT language supports its extension by means of *external functions* that are called *black box*. These can be used to declare operations (*queries*, *helpers* or *mappings*) that are implemented in another language than QVT. In the case of SmartQVT these operations can be implemented in

Java. Figure 123 presents how this characteristic was used to implement several *extensions* to the QVT language that were required to implement the feature to *Ecore* model transformation. Essentially, our extensions added support in QVT for accessing model metadata and to be able to manipulate models in an *untyped* way.

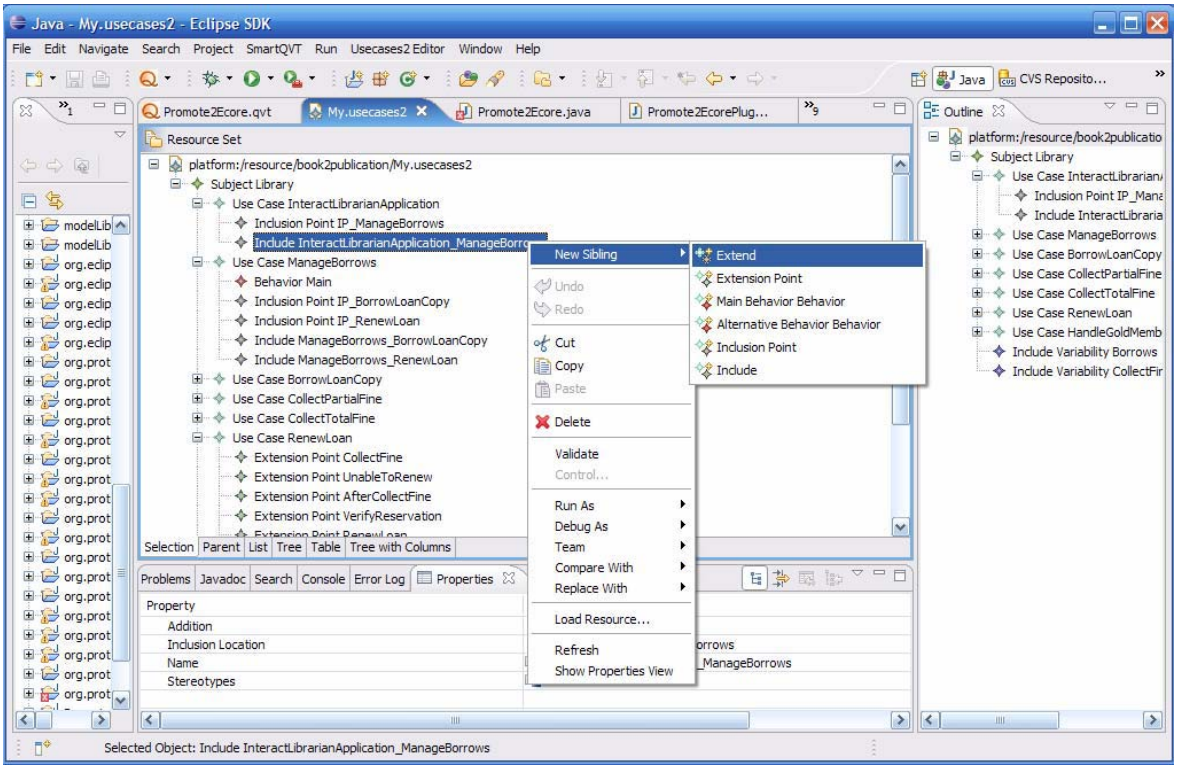

Figure 124: Using the use case editor generated with EMF to create a domain use case model for a library product line.

Figure 124 presents Eclipse being used to construct a domain use case model for a library product line. The model is being created using the modeling environment that was generated based on the metamodel presented in Figure 116. After the domain use case model is created the transformation *domain use case model to feature model* can be invoked to obtain a feature model. In the middle of Figure 125 we see part of the generated feature model. This feature model can be transformed into a configuration metamodel by invoking the transformation partially depicted in Figure 123. This results in an Ecore model that can be used to automatically generate a modeling environment to construct feature configurations that conform to the feature model that resulted from the domain use case model.

The final step in this process is to obtain a use case model that reflects a feature configuration. For that, the transformation *domain use case model to product use case model* must be invoked. As described in Chapter 5, this transformation has three input models and one output model. The input models are: the domain use case model; the feature model; and the configuration model. The output model is the product use case model that results from selecting from the domain use case model only the elements that are related to the features selected in the feature configuration model. In the top of Figure 125 we can observe the resulting product use case model.

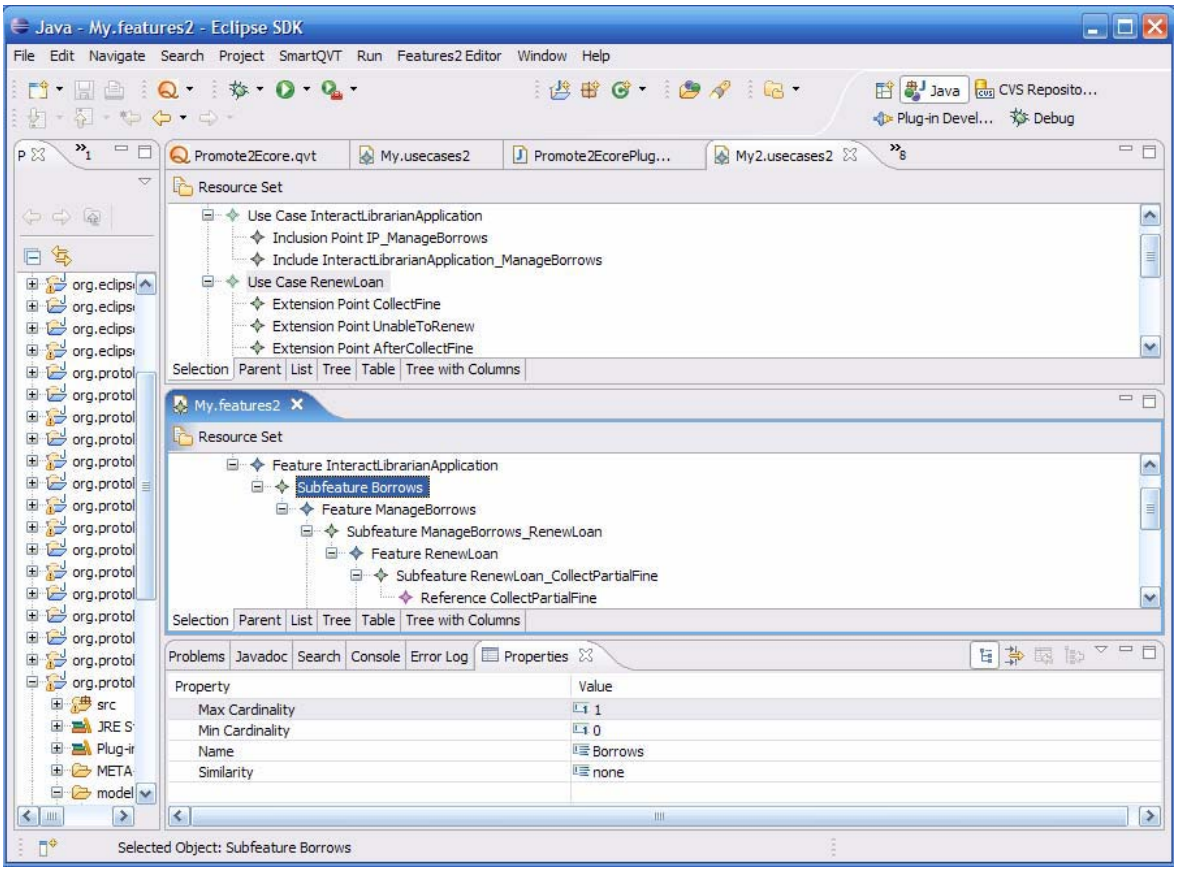

Figure 125: Inspecting the feature model generated from a domain use case model of the library product line and a use case model that resulted from a feature configuration.

Appendix C: Experimental Implementation of Multi-Staged Modeling Approach

This appendix presents an experimental implementation of the multi-staged modeling approach that was discussed in Chapter 5 with EMF and SmartQVT.

EMF is a project of Eclipse that consists of a modeling framework and code generation facility for building tools and other applications based on a structured data model. This tool is available at http://www.eclipse.org/modeling/emf/.

SmartQVT is a model transformation tool implementing the MOF 2.0 QVT-Operational language. This tool is available at http://smartqvt.elibel.tm.fr/. The SmartQVT accepts as input for the model transformations models in the EMF format.

EMF and SmartQVT are plugins of the Eclipse open development platform available at http://www.eclipse.org/.

Figure 126 presents Eclipse and EMF being used to edit a metamodel for insurance agreements (stage N of the multi-stage approach). This metamodel is called the *specialization* metamodel. According to our approach, this metamodel must be annotated in a way that can be used to guide the model transformations according to the patterns that were presented in Chapter 5. Figure 127 presents the agreement metamodel being annotated.

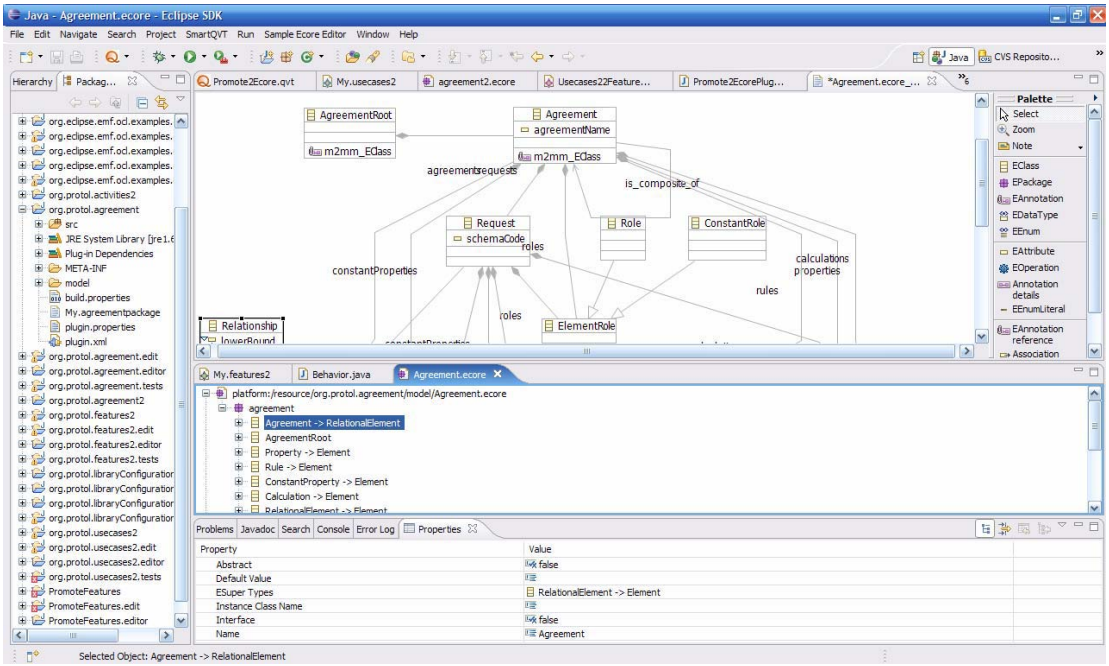

Figure 126: Editing a metamodel for insurance agreements (stage N).

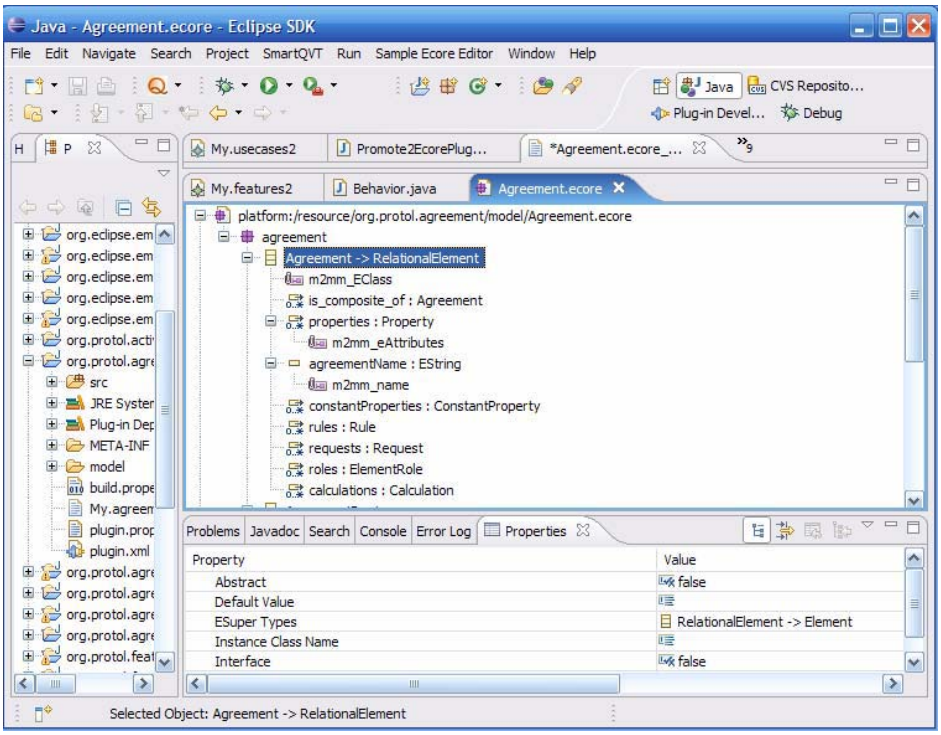

Figure 127: Annotating the insurance agreement metamodel with annotations to guide the transformations (stage N).

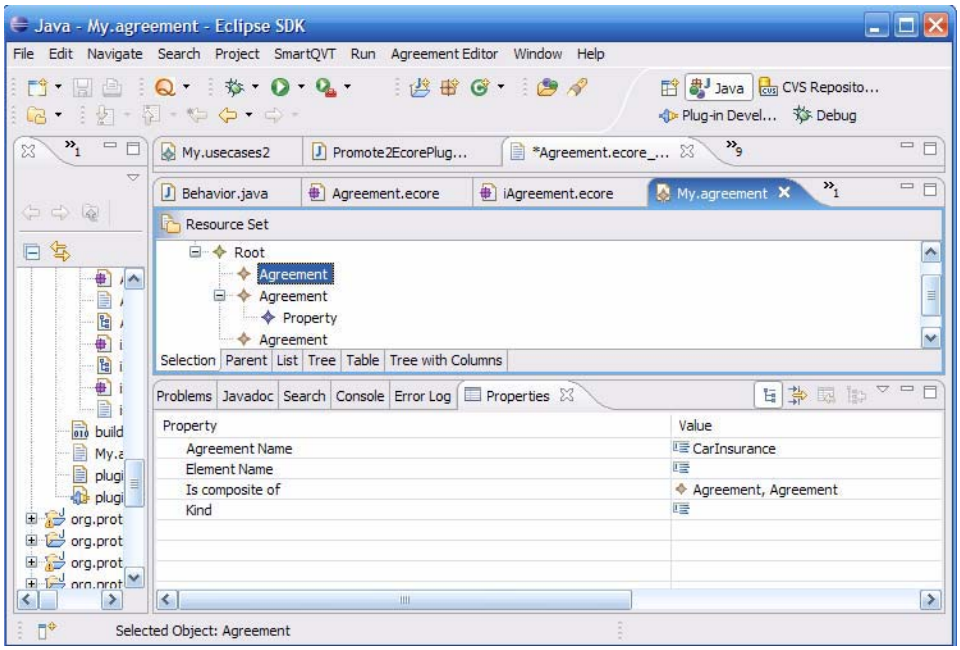

Figure 128: Entering a model of an insurance car agreement with the agreement modeling environment (stage N+1).

After creating the agreement metamodel it is possible to generate code to support an environment for constructing models that are in conformance with that metamodel. This generated

environment is used, according to our multi-stage modeling approach, as a domain-specific modeling environment. Figure 128 presents Eclipse being used as a domain-specific modeling environment. In this case it is being used to model a insurance car agreement.

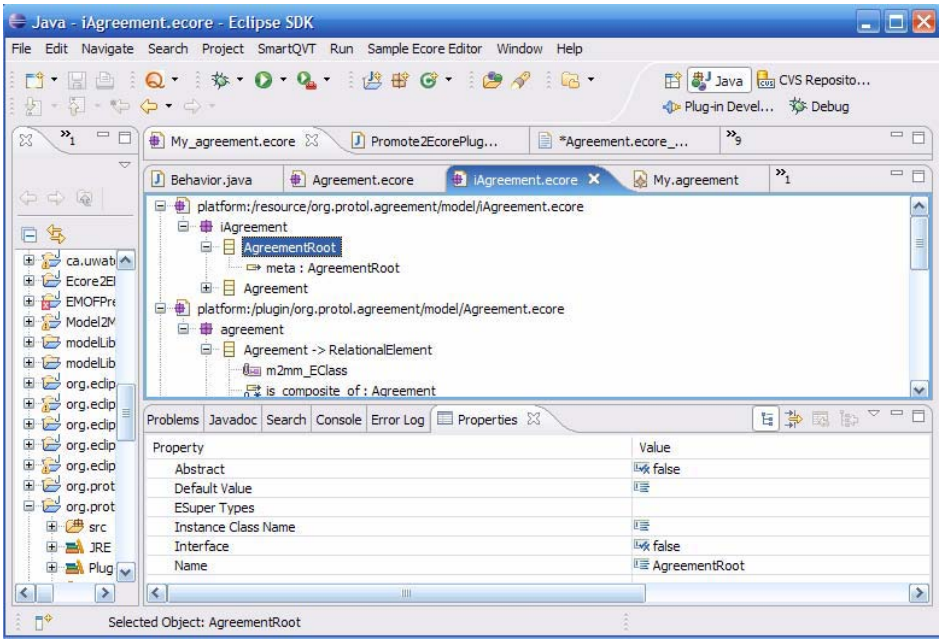

Figure 129: Inspecting the instantiation model (ecore model) of stage N obtained from the annotated agreement metamodel.

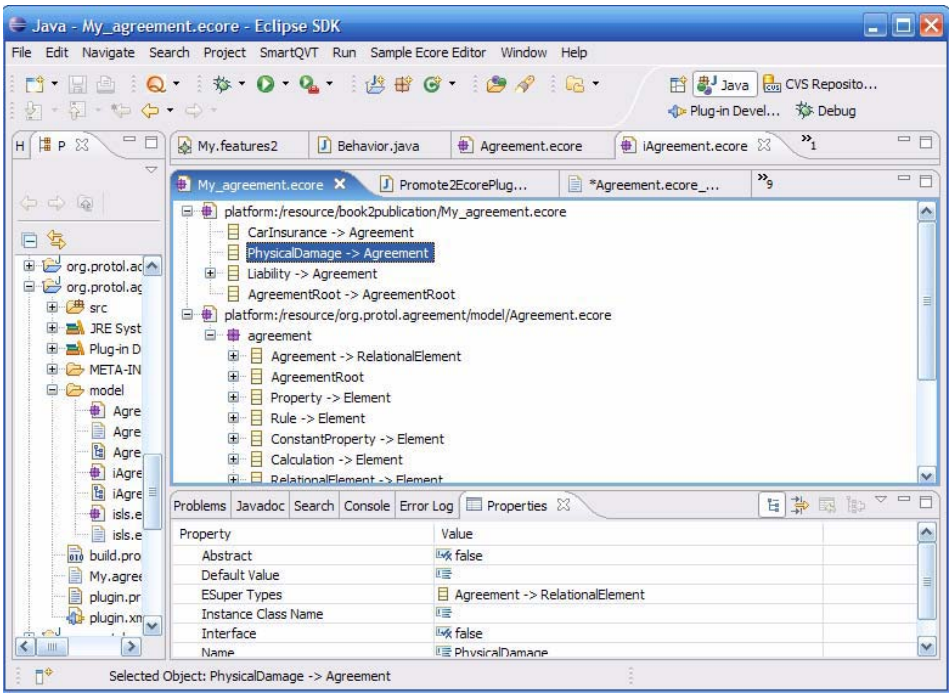

Figure 130: Inspecting the metamodel of stage N+1 that was obtained by promoting the insurance car agreement model of stage N

According to our approach, at each stage the *specialization* metamodel is used to generate the *instantiation* model (that is an ecore model) using the *instantiation transformation*. Figure 129 presents the instantiation model that resulted from the instantiation transformation on the agreement metamodel.

From the models created with the domain-specific modeling environment of each stage it is possible to generate the next specialization metamodel. Figure 130 presents the specialization metamodel that results from the specialization promotion transformation applied on the insurance car agreement model presented in Figure 128. The resulting specialization metamodel can be used to generate the domain-specific modeling environment for the next stage. This process can be repeated for each required modeling stage.

.