FEATURE ORIENTED DOMAIN SPECIFIC LANGUAGE FOR DEPENDENCY INJECTION IN DYNAMIC SOFTWARE PRODUCT LINES

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

BY

ORÇUN DAYIBAŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2009

Approval of the thesis:

FEATURE ORIENTED DOMAIN SPECIFIC LANGUAGE FOR DEPENDENCY INJECTION IN DYNAMIC SOFTWARE PRODUCT LINES

submitted by **ORÇUN DAYIBAŞ** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

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

Prof. Dr. Müslim Bozyiğit ______________________
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Halit Oğuztüzün ______________________
Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Assoc. Prof. Dr. Ali Doğru ______________________
Computer Engineering Dept., METU

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

Assoc. Prof. Dr. Nihan Kesim Çiçekli ______________________
Computer Engineering Dept., METU

Asst. Prof. Dr. Erol Şahin ______________________
Computer Engineering Dept., METU

Dr. Bülent Mehmet Adak ______________________
Senior Software Engineer, Aselsan Inc.

Date: ______________________

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

Name, Last name : **ORÇUN DAYIBAŞ**

Signature :

ABSTRACT

FEATURE ORIENTED DOMAIN SPECIFIC LANGUAGE FOR DEPENDENCY INJECTION IN DYNAMIC SOFTWARE PRODUCT LINES

Dayıbaş, Orçun
M.Sc., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Halit Oğuztüzün

September 2009, 61 pages

Base commonality of the Software Product Line (SPL) Engineering processes is to analyze commonality and variability of the product family though, SPLE defines many various processes in different abstraction levels. In this thesis, a new approach to configure (according to requirements) components as building blocks of the architecture is proposed. The main objective of this approach is to support domain design and application design processes in SPL context. Configuring the products is made into a semi-automatic operation by defining a Domain Specific Language (DSL) which is built on top of domain and feature-component binding model notions. In order to accomplish this goal, dependencies of the components are extracted from the software by using the dependency injection method and these dependencies are made definable in CASE tools which are developed in this work.

Keywords: Software Product Line, Domain-specific Language, Variability Management, Feature-component binding, Dependency Injection.

ÖZ

DEVİNGEN YAZILIM ÜRÜN HATLARINDA BAĞIMLILIK İLETİMİ İÇİN YETENEK YÖNELİMLİ ALANA ÖZGÜ DİL

Dayıbaş, Orçun
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Halit Oğuztüzün

Eylül 2009, 61 sayfa

Yazılım Ürün Hattı (YÜH) mühendisliği, her ne kadar farklı seviyelerde çesitli süreçler tanımlasa da süreçlerin temel ortaklığı ilgili ürün ailesi üzerinde değişkenliklerin ve ortaklıkların çözümlenmesidir. Bu tez çalışmasında, mimari yapı taşları olarak yazılım bileşenlerinin, (gereksinimlere göre) yapılandırılması için yeni bir yaklaşım sunulmaktadır. Bu yaklaşımın temel amacı, YÜH bağlamında alan tasarımı ve uygulama tasarımı aşamalarının desteklenmesidir. Alan ve özellik-bileşen eşleme modellerindeki unsurlar üzerine kurulmuş bir alana özgü dil tanımı getirilerek, ürün yapılandırma işlemleri yarı otomatikleştirilmiştir. Bu amaca ulaşmak için bağımlılık iletimi yöntemi kullanılarak bileşenler arası bağımlılıklar yazılım dışından kurulabilir hale getirilmiş, çalışma kapsamında geliştirilen bilgisayar destekli yazılım mühendisliği (CASE) araçları ile de ilgili bağımlılıklar yapılandırılabilir hale gelmiştir.

Anahtar Kelimeler: Yazılım ürün hattı, Alana özgü dil, Değişkenlik yönetimi, Yetenek-bileşen eşlemesi, Bağımlılık iletimi.

ACKNOWLEDGEMENTS

I would like to express my special appreciation to my family for their unconditional support and understanding throughout this thesis work.

This study was partially supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under the BİDEB grant programme.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ v

ACKNOWLEDGEMENTS vi

TABLE OF CONTENTS vii

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF ABBREVIATIONS xii

CHAPTERS

1. INTRODUCTION 1
   - 1.1 Background 1
   - 1.2 Scope and Objectives 2
   - 1.3 Contributions 3
   - 1.4 Document Organization 4

2. SOFTWARE REUSE 5
   - 2.1 Overview 5
   - 2.2 Reuse In Software 6
   - 2.3 Domain Engineering Processes 10
     - 2.3.1 Feature-Oriented Domain Analysis (FODA) 10
     - 2.3.2 Organization Domain Modeling (ODM) 11
     - 2.3.3 Reuse-driven Software Engineering Business (RSEB) 13
     - 2.3.4 Feature-Oriented Reuse Method (FORM) 15
   - 2.4 Software Product Line Processes 16
     - 2.4.1 Product Line Software Engineering (PuLSE) 22
     - 2.4.2 KobrA 23
     - 2.4.3 Component-oriented Platform Architecting Method (CoPAM) 24
     - 2.4.4 Product Line ULM Based Software Engineering (PLUS) 25
   - 2.5 Discussion 26

3. VARIABILITY MANAGEMENT AND FEATURE MODELING 27
3.1 Variability Management 27
3.1.1 Variability Modeling 29
3.2 Feature Modeling 30
3.2.1 Feature Modeling Tools 33
3.3 Discussion 33
4. DEPENDENCY INJECTION FOR DYNAMIC SPL 35
4.1 Dynamic Software Product Lines 35
4.2 Dependency Injection 37
4.3 Kutulu DSL for Modeling 38
4.3.1 Meta-models of Language Elements 40
4.3.2 Kutulu CASE Tools 42
4.3.3 Using Kutulu Generator for Assembler Configuration 44
5. CASE STUDIES 48
5.1 MVC, Factory Design Patterns and Two Alternative Features 48
5.2 Sample Product of TADES SPL 50
6. CONCLUSION AND FUTURE WORK 56
REFERENCES 57

LIST OF TABLES

TABLES

Table 1 - Some Variability Mechanisms.. 14

Table 2 - Some DSPL features and dependency injection.. 38

LIST OF FIGURES

FIGURES

Figure 1 - Raising the level of reuse 7
Figure 2 - Customizability vs. Reuse Granularity... 7
Figure 3 - A framework for Software Reuse... 9
Figure 4 - ODM Process Tree 12
Figure 5 - Flow of artifacts in RSEB 13
Figure 6 - FORM Engineering Processes.. 15
Figure 7 - Software Product Line Engineering Framework...................................... 16
Figure 8 - Application design sub-process.. 20
Figure 9 - PuLSE Overview... 22
Figure 10 - UML-based Component Modeling in KobrA 23
Figure 11 - CoPAM as a method family.. 24
Figure 12 - Evolutionary Software Product Line Engineering Process 25
Figure 13 - Early and delayed variabilities 28
Figure 14 - Sample for Orthogonal Variability Modeling (OVM) 30
Figure 15 - Feature diagram notations 31
Figure 16 - De facto sample of the feature diagrams; a simple car............................ 31
Figure 17 - Sample feature configurations and their feature diagram 32
Figure 18 - Tree display and graph display of a model in pure::variant 33
Figure 19 - SPL vs. DSPL on variability management.. 36
Figure 20 - The dependencies for a Dependency Injector. 37
Figure 21 - Using Kutulu DSL in SPL Process 39
Figure 22 - Domain meta-model of Kutulu DSL... 41

Figure 23 - Feature-binding meta-model 42
Figure 24 - Implementation of Kutulu CASE Tools (Editors).. 43
Figure 25 - Implementation of Spring.NET Generator.. 44
Figure 26 - Spring.NET objects XML schema 45
Figure 27 - Sample model in Kutulu feature-component binding editor...................... 48
Figure 28 - Sample model in Kutulu domain editor 49
Figure 29 - Generated Spring.NET configuration.. 50
Figure 30 - Sample TADES application module 50
Figure 31 - TADES Log operations in Kutulu domain editor 51
Figure 32 - TADES Met operations in domain editor.. 52
Figure 33 - Sample part of TADES feature tree 52
Figure 34 - Feature bindings of log and met operations 53
Figure 35 - Generated configuration.. 54
Figure 36 - Configured TADES application 55
Figure 37 - Configuration change in TADES application 55

LIST OF ABBREVIATIONS

C2	Command & Control
CASE	Computer Aided Software Engineering
CMMI	Capability Maturity Model Integration
COTS	Commercial off-the-shelf
DARPA	Defense Advanced Research Projects Agency
DI	Dependency Injection
DSL	Domain Specific Language
DSPL	Dynamic Software Product Line
E-R	Entity-relationship
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
IDE	Integrated Development Environment
J2EE	Java Platform, Enterprise Edition
M2T	Model-to-Text
MVC	Model-View-Controller
NATO	North Atlantic Treaty Organization
OVM	Orthogonal Variability Modeling
RLF	Reuse Library Framework
SPL	Software Product Line
SPLE	Software Product Line Engineering
UML	Unified Modeling Language
VSL	Variability Specification Language

CHAPTER 1

INTRODUCTION

1.1 Background

Every software development project, in general, produces two types of software: custom software and standard software (COTS – Commercial off-the-shelf Software). Custom software is built on special order and tailored to the specific requirements of one customer. Standard software, on the other hand, is supposed to meet the needs of many customers. Once developed, standard software is produced in a mass production process by duplicating the distribution medium. The main problems of both types of software are: custom software causes high development costs and standard software is cheaper, but barely meets exactly the customers' requirements. Therefore, any effort to make custom software more economic or to make standard software more customized feature-rich is vastly important for software engineering discipline. Software product line engineering (SPLE) is one of the most promising efforts in that manner [1, 2].

Clements et al. define the term "Software Product Line (SPL)" as follows: "A software product line is a set of software-intensive systems that share a common, managed set of features satisfying 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" [1]. For the sake of establishing of understanding this term better, let us retrace the important parts of the definition: First of all, "set of software-intensive systems" remarks that the term covers more than just a methodology for single software/system development project; it bounds a family of software systems (or systems involving software as an essential ingredient) [1] and regulates relations among them. Second, "managed set of features" remarks that commonality among

the members of the family is feature-based and managed in that concept. “A particular market segment” expression remarks that SPL is mainly a market-driven concept. Czarnecki et al. define this aspect of the SPL as a main difference from “system family” concept [3]. Then, “common set of core assets” remarks that products of the SPL share common building blocks namely assets. In detail, SPLE deals with discovering these assets and managing their adaptability to specific product requirements. Last, but not least, “prescribed way” expression remarks that reuse in SPL is managed and enforced.

In the light of above definition, it can be deduced that the following aspects are crucial and indispensible parts of the SPL development project:

- Scoping: Drawing the boundaries of the concerning market (domain) and defining legacy or future product for this market.

- Domain Design: Designing the technical core assets in architectural perspective and defining data flow between them.

- Application Design: Designing the product according to individual requirements of it.

Section 2.4 is solely related to the SPL hence, there is no need for further explanation for the introduction section. In the frame of this thesis work, it has been suggested a supporting mechanism in order to make above aspects of SPLE methodology easier. Following sections refer to content of this frame and give more detailed information about that.

1.2 Scope and Objectives

In this study, a feature-oriented dependency injection method has been developed in order to manage building parts of the product (some or all of the common core assets). The main motivation is that externalized dependencies of dependency injection approach and data flow (relations) of the domain design in SPL context are

similar way of defining software architecture. It can be deduced that defining production plan (feature configuration) for a family member of the SPL (product) and mapping this plan to the reference architecture supports the application design process. Therefore, supporting these two processes with a robust strategy makes the SPL process easier to apply.

SPLE is relatively new area of subject and there is no any dominant methodology for developing a brand new SPL. Therefore, some of these methodologies are out of scope for this study. Feature-oriented approach for requirements and component-oriented approach as an architectural definition are seized in this work. As a consequence, SPLE methodologies which are able to adopt these approaches in principle can also be evaluated in the scope of thesis work.

1.3 Contributions

The main contribution of the study lies in defining a semi-automatic method for component binding as a foundation of variability management of SPL. In this approach, binding relations and specific attributes of the component defines specific feature of the product and injected dependencies makes the product be able to handle this requirement (feature). As previously mentioned, SPL is a strictly managed way of reuse and this study assists that notion by defining a DSL (named as “Kutulu”) to manage architectural reuse assets. Furthermore, externalizing dependencies (to inject declaratively) and defining a way of binding them after deployment also cause SPL to gain dynamic notion.

In the literature, there are some methods in order to make construction of the component based reuse infrastructure easier (E. Almeida et al. list some remarkable methods in [4]). Lots of CASE tools are also available for software architects which are considered to use these methods in their process. On the other hand, very few of them complete solution and traceable to the implementation level. These features can also be evaluated as other beneficial contribution of this study.

1.4 Document Organization

This thesis work includes five chapters. Chapter 2 is about the state-of-art on reuse in software intensive systems. SPLE that this work has been motivated from is also described in details in this chapter. Chapter 3 defines the variability management and feature modeling concepts. These definitions also introduce our approach in this work. Chapter 4 introduces dynamic software product line and dependency injection as an enabling method for DSPL. This chapter also locates the introduced approach in the big picture of the SPLE by mentioning our DSL definition and its usage with CASE tools. Case study of the proposed methods is implemented in the chapter 5 and all of the thesis work is concluded in chapter 5.

CHAPTER 2

SOFTWARE REUSE

2.1 Overview

The term “Software Crisis” is coined by F. L. Bauer at the first NATO Software Engineering Conference in 1968 [5]. This term is still popular and that case makes some people discuss that “Software Engineering” term is an oxymoron [6].

In 1957, “programmer” was not acceptable answer of “What is your profession?” question [7]. Today, this profession is very well known and widely accepted one. Furthermore, programming work is done in teams consisting of project managers, requirements analysts, software engineers, documentation experts, and programmers. So many professionals collaborate in an organized manner on a project but many basic problems about programming (or constructing software) still remain.

“Software Crisis” refers to the difficulty of writing correct, understandable, and verifiable computer programs. The roots of the crisis are complexity, expectations, and change. E. W. Dijkstra expressed his hope at the first NATO Software Engineering Conference [5]; “The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement”.

After nearly fifty years of progress, software development has remained as a craft and has yet to emerge into a science. Many beneficial programming practices and project management methodologies have been developed in this period of time. The most common characteristic of these practices and methodologies is reuse notion.

2.2 Reuse In Software

Every building block of the system demands some amount of resource in any kind of engineering area. Therefore, reusing pre-implemented (or designed) building blocks is fairly common in diverse engineering areas. In this respect, software is not an exception although software engineering is less mature than the most of the others [6].

There are different reusable assets in a software development project; source codes, executable components, requirements, test cases, etc… Using these building blocks for more than once brings many benefits:

- Shorter delivery time: One or more of the assets is not required to be implemented hence, development time decreases.

- More reliability: Pre-used asset means it is tested before. It is expected to be more reliable than brand new implemented one. Therefore, overall reliability of the system increases, also.

- More standard implementation: If reuse increases, more standard implementation is also achieved by less effort. This also one of the main pivot points of the first two items.

Reuse in software is evolutionary concept. In early days, code snippet (functions) was the main object to be reused. In the earliest systems, memory was so expensive that it was often necessary to save memory by using callable functions [8]. Sharing code snippets is vastly labour-intensive work and it’s very difficult to manage this type of reuse. As time passed, abstraction increased and coarse-grained reuse objects have been substituted by fine-grained ones. Respectively, objects, components & frameworks, domain models took place in order to manage software reuse.

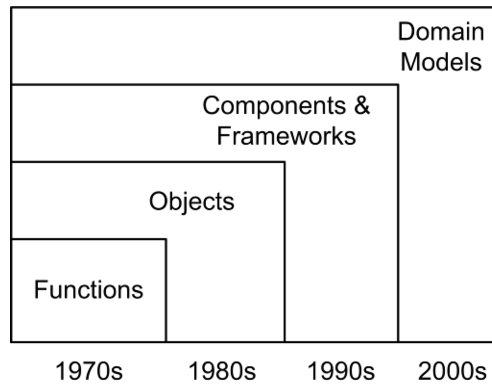

Figure 1 – Raising the level of reuse [8]

Figure 1 depicts the chronological order of the reuse abstractions. In this context, it can be deduced that abstraction constantly increases and reusable building blocks are getting bigger. Raising the level of abstraction brings some new problems and questions. Thus, there is no any commonly accepted level, today. For instance, object is still commonly used abstraction level in many projects. Using a higher abstraction level has a trade-off that has to be considered; customization.

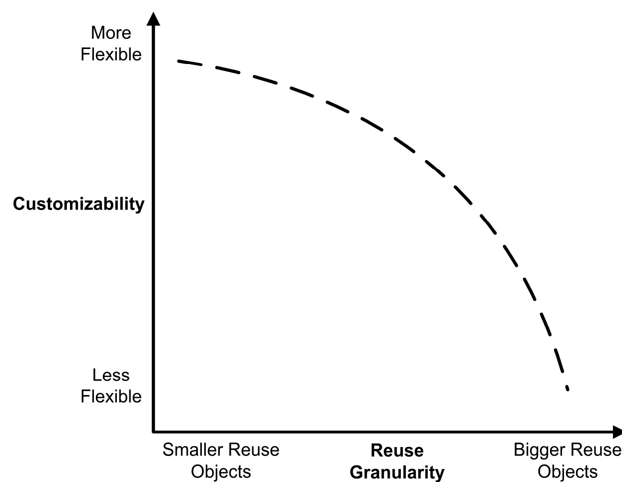

Figure 2 - Customizability vs. Reuse Granularity

For instance, using function-level reuse is more customizable than using object-level one and again, component-level reuse is less customizable than object-level one. It is obvious that detailed analysis is required in order to achieve goal; increasing reuse without concession in customization. A. Abran et al. describe software reuse as being a key factor in maintaining and improving productivity in the development of software [9]. They also state that effective software reuse requires a strategic vision that reflects the “unique power and requirements” of reuse techniques.

W.C. Lim, classifies reuse techniques as technical ones and non-technical ones [10]. As Figure 3 depicts, non-technical aspects are also as crucial as technical ones. On the other hand, in scope of this study, technical aspects (processes and tools) of software reuse are focused on. Following sections cover some pioneering software reuse techniques and they are mostly technical according to above classification.

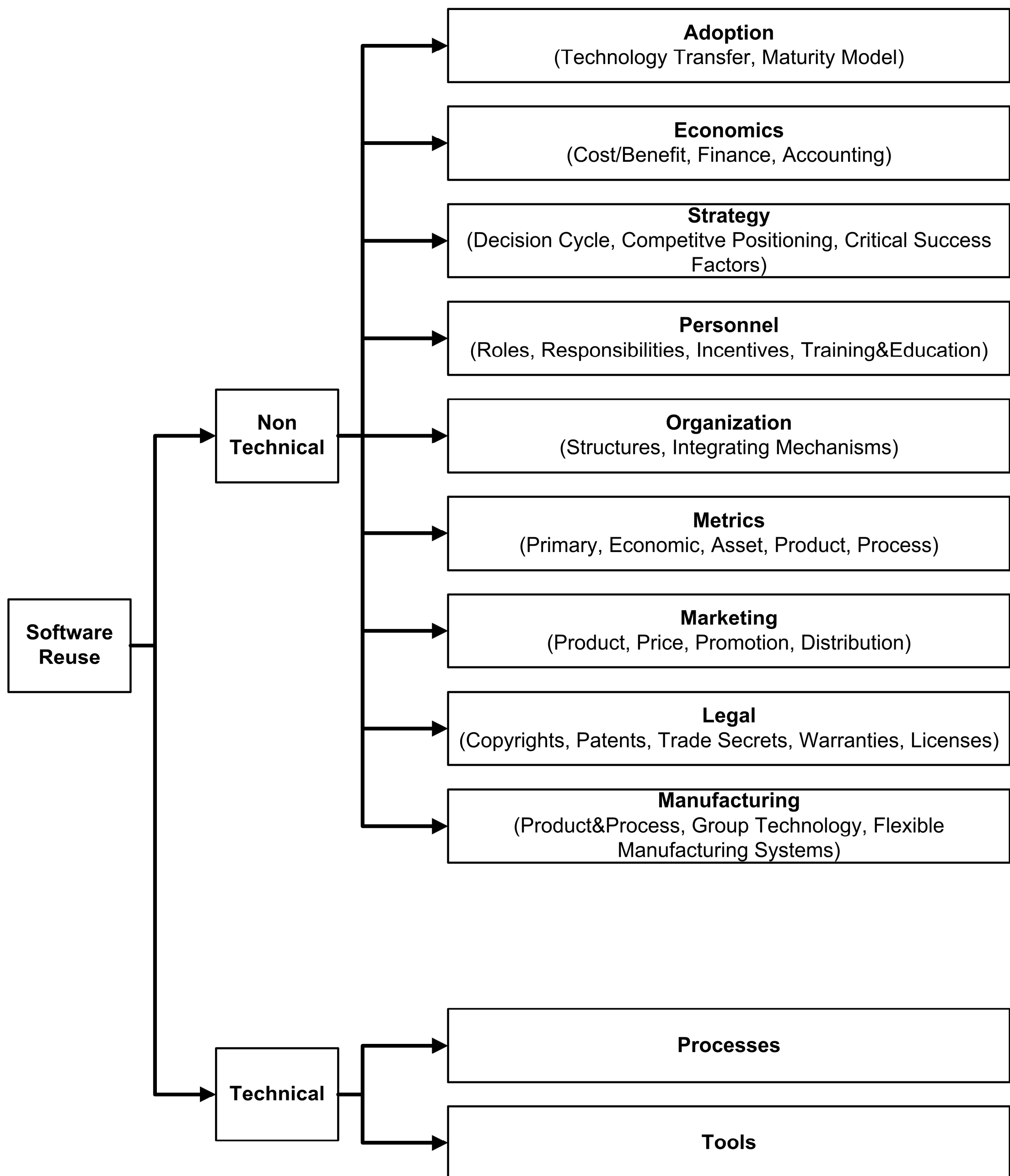

Figure 3 – A framework for Software Reuse (adapted from [10])

2.3 Domain Engineering Processes

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, as well as providing an adequate means for reusing these assets when building new systems [3]. Domain engineering is one of the first mature reuse efforts and it is still an accepted approach, today. Below techniques fulfil the domain engineering frame with their own methods.

2.3.1 Feature-Oriented Domain Analysis (FODA)

Domain engineering has several steps but its first step is "domain analysis". FODA is a domain analysis method that focuses on the features of the domain elements (systems). It identifies three phases [11]:

- Context Analysis: The context analysis defines the context of the domain like its scope, constrains and relations between the other domains. In other words, it defines boundaries of the domain.

- Domain Modeling: Creating domain model. Produced domain model is assembly of the other elements; E-R model, feature model, functional model and domain terminology dictionary.

- Architecture Modeling: Creating architectural elements in order to solve problems which are defined in the preceding step.

Context analysis is documented in a context model. This model includes some structure diagrams and data-flow diagrams of target domain. If there exists any related higher level domains or sub-domains, relations of them and relations of in-domain structures are defined in this model. K.C. Kang et al. state that the diagrams of the context model can be informal block diagrams [11].

Scoped domain in context analysis phase is analyzed in order to identify commonalities and variabilities in domain modeling phase. As stated previously, feature model is one of the outputs of this phase. Feature is a kind of abstraction to deal with requirements. Therefore feature model depicts the implemented requirements of the domain in a hierarchical structure. E-R diagram is also one of models of the domain modeling phase. E-R diagram depicts the domain entities and their inter-relations.

After the domain modeling, the first implementation related phase takes place, namely, architectural modeling. In this phase, outputs of the previous phases are represented in developer point of view. Main activity of architectural modeling phase is mapping domain model to architectural structures. Consequently, domain analysis ends up with this phase and there is adequate information in order to start development of the identified domain assets (reusable components and libraries).

2.3.2 Organization Domain Modeling (ODM)

Organization Domain Modeling (ODM) had been developed in the frame of DARPA (Defense Advanced Research Project Agency) STARS (Software Technology For Adaptable, Reliable Systems) program [12]. ODM was not one of the main objectives of the STARS program. It raised from the design process of "Reuse Library Framework – RLF", a knowledge-based reuse support environment [3] [13]. ODM is chiefly developed by Mark Simos and it dates back to late of 80's. ODM has been applied on several projects from companies like Hewlett-Packard, Lockheed Martin, Rolls-Royce [14]. Thus, it is very mature domain engineering method.

ODM assumes that there is a complex structure of interrelationships between other projects, stakeholders and economic objectives of organizations. Therefore, these relationships must be recognized and be clearly exposed by analyzing domain of the concerning organization.

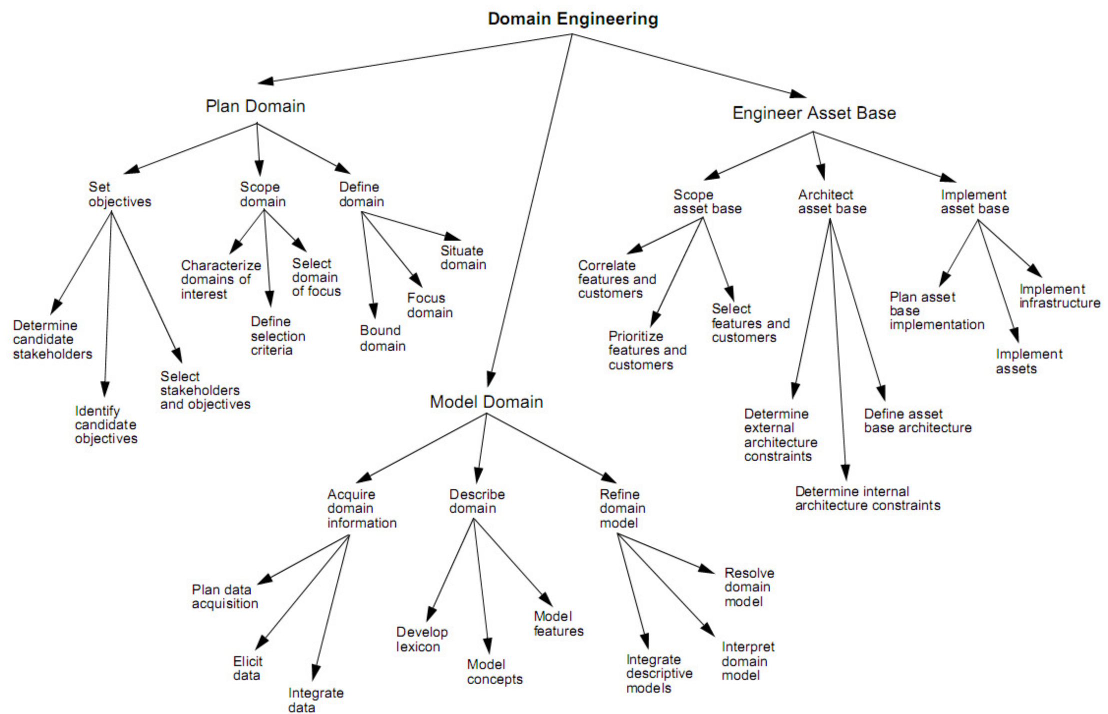

Figure 4 - ODM Process Tree [14]

M. Simos et al. describe ODM process in three main phases (see Figure 4): Plan Domain, Model Domain and Engineer Asset Base [14]:

- Plan Domain: This phase consists of defining and scoping domain. Firstly, objectives of the process are defined and a boundary of the domain (scope) is determined. Scoping domain is very crucial for the process. In single system, scope is shaped by the requirements which are usually are given. On the other hand, scoping a domain means we have to consider multiple application contexts. Finally, domain is also defined in this phase according to previous steps.

- Model Domain: Firstly, information acquisition task is planned and domain information is collected from concerning persons or objects (legacy systems, literature about the domain, etc…). Domain is described after adequate

information is collected. This step covers textual descriptions and/or visual descriptions. Finally, defined domain is revised and refined. The main objective of this phase is to produce a domain model to describe common and variant features of the systems within the domain.

- Engineering Asset Base: This phase is the only step which is related to implementation. In this phase asset base is scoped, architected and implemented. Asset base has to have constraints on configurations of features that supported for the selected (scoped) customer settings.

ODM is comprehensive domain engineering method and it is adaptable to specific organization needs. It gives special importance to integrate organizational and technical aspects of the domain engineering.

2.3.3 Reuse-driven Software Engineering Business (RSEB)

Reuse-driven Software Engineering Business (RSEB) is a model-driven large scale reuse method [15]. Although it was first introduced in 1997, it is based on practices which are conducted by Hewlett-Packard and Intecs corporations and date back to 1988.

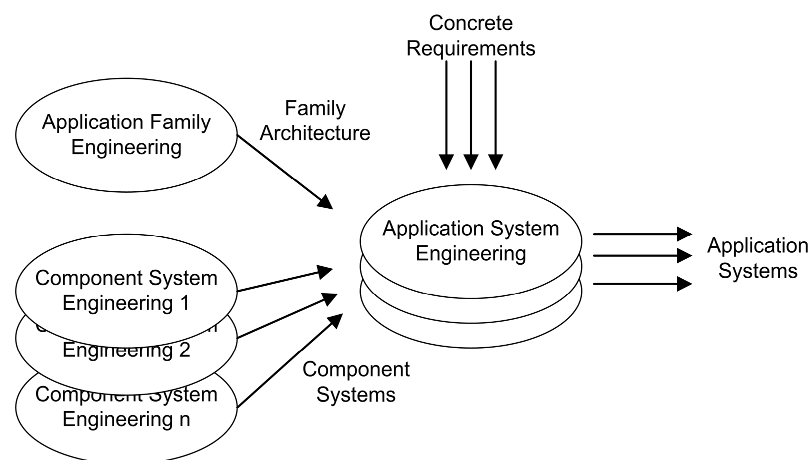

Figure 5 - Flow of artifacts in RSEB [3]

RSEB is based on the UML notation and it brings object-oriented notion to the domain engineering. Similar to Unified Process [16], RSEB is an iterative and use-case-centric method [3]. RSEB splits domain engineering into two sub-processes; "Application Family Engineering" that develops a layered architecture and "Component System Engineering" that develops systems of reusable components. In RSEB, application engineering is called "Application System Engineering" that develops selected applications.

RSEB also emphasizes modelling variability by using extended UML notation. A variation point identifies one or more locations at which the variation will occur [15]. Variation points are implemented in more concrete models using different variability mechanisms. Examples of the variability mechanisms are summarized in Table 1.

Table 1 - Some Variability Mechanisms ([3], [15])

Mechanism	Type of Variation Point	Type of Variant	Use Particularly When
Inheritance	Virtual Operation	Subclass or Subtype	Specializing and adding selected operations, while keeping others
Extensions	Extension Point	Extension	Attaching several variants at each variation point at the same time
Uses	Use Point	Use Case	Reusing abstract use case to create a specialized use case
Configuration	Configuration Item Slot	Configuration Item	Choosing alternative functions and implementations
Parameters	Parameter	Bound Parameter	Selecting between alternative features
Template Instantiation	Template Parameter	Template Instance	Doing type adaption or selecting alternative pieces of code
Generation	Parameter or Language Script	Bound Parameter or Expression	Doing large-scale creation of one or more types or classes from a problem-specific language

In RSEB, variability is expressed at the highest level in the form of variation points (especially in use cases), which are then implemented in other models using various variability mechanisms [3].

Griss et al. report that use case based approach is insufficient in practice [17]. They have summarized their experience which is in telecom domain. They have also introduced new method, namely FeatuRSEB, in order to handle shortcomings of the RSEB in [17].

2.3.4 Feature-Oriented Reuse Method (FORM)

FORM (Feature-Oriented Reuse Method) is a systematic method that focuses on capturing commonalities and differences of applications in a domain in terms of features and using the analysis results to develop domain architectures and components [18]. FORM extends FODA (see Section 2.3.1) by comprising design and implementation processes. It mainly deals with how to build domain architectures and reusable components.

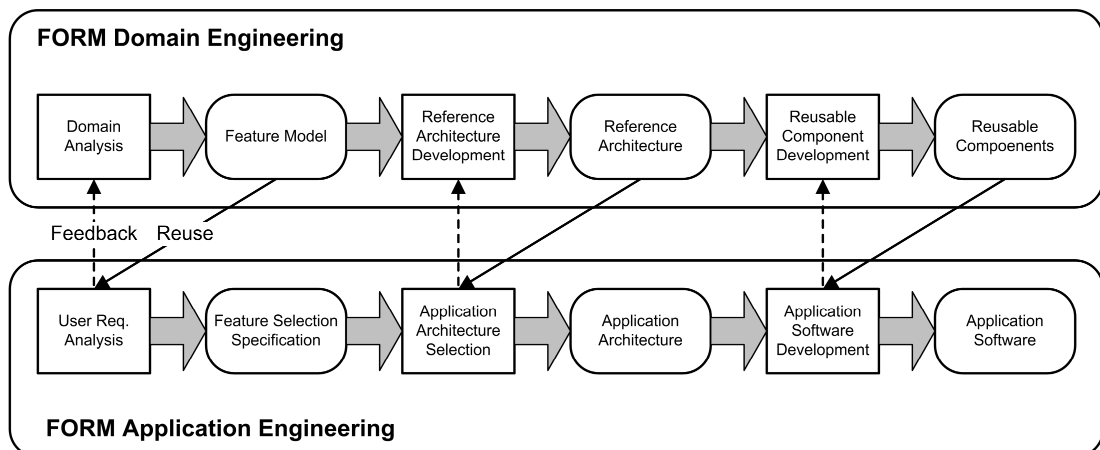

Figure 6 - FORM Engineering Processes (adopted from [18])

FORM also has two separate processes: Domain engineering and application engineering (see Figure 6). Domain engineering process consists of activities for analyzing systems in a domain and creating reference architectures and reusable components based on the analysis results [18]. The reference architectures and

reusable components are expected to accommodate the differences as well as the commonalities of the systems in the domain. The application engineering process consists of activities for developing applications using the artefacts created in domain engineering [18].

2.4 Software Product Line Processes

Until 1998, the software reuse processes were only related to domain engineering issues [4]. Later, Software Product Line approach has been introduced [1]. As previously mentioned, Software Product Line is a method for developing software applications with mass customization. SPL defines processes to facilitate the development of a family of products in a pre-defined market more economically. SPL relies on a fundamental distinction between development for reuse (Domain engineering) and development with reuse (Application engineering) [19].

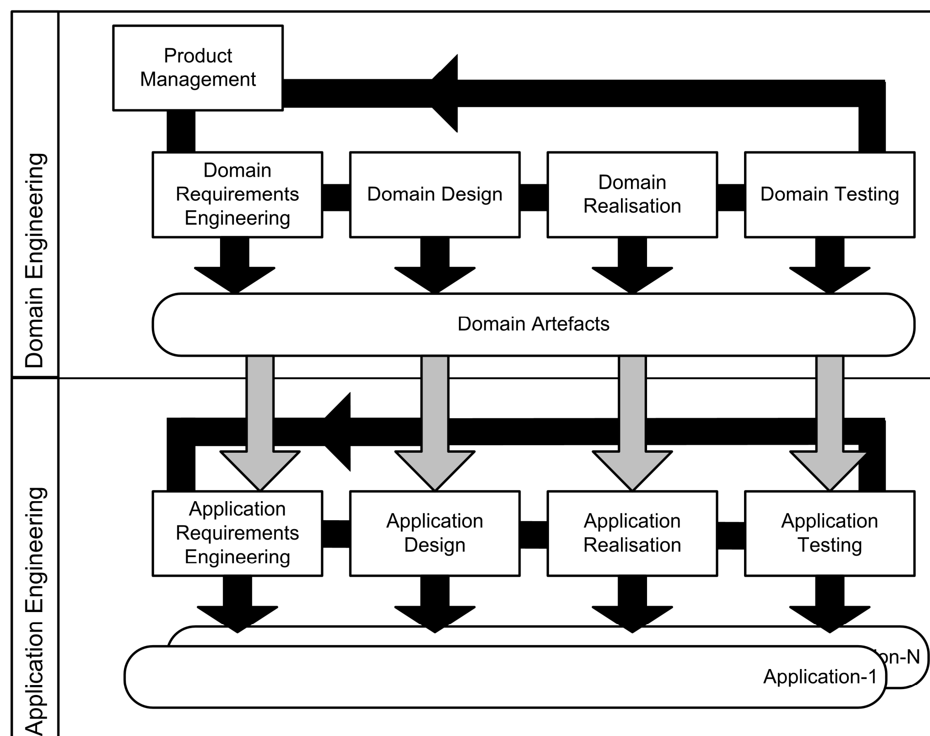

Figure 7 - Software Product Line Engineering Framework (adapted from [19])

Product management deals with the economic aspects of the software product line and in particular with the market strategy. Its main concern is the management of the product portfolio of the company or business unit [19]. This phase is very crucial for the other sub-processes of SPL. According to J. Bayer et al. [17], domain engineering approaches have three shortcomings;

- Misguided scoping of application area
- Lack of operational guidance
- Overstressed focus on organizational issues

In SPL context, the first two issues are mainly handled in this phase. Thus, this case also conveys the importance of the product management phase.

The domain requirements engineering sub-process encompasses all activities for eliciting and documenting the common and variable requirements of the product line [19]. Domain requirements engineering differs from requirements engineering for single systems because of following points [19]:

- The requirements are analysed to identify those that are common to all applications and those that are specific for particular applications (i.e. that differ among several applications).
- The possible choices with regard to requirements are explicitly documented in the variability model, which is an abstraction of the variability of the domain requirements.
- Based on the input from product management, domain requirements engineering anticipates prospective changes in requirements, such as laws, standards, technology changes, and market needs for future applications.

The domain requirements engineering sub-process defines requirements and their variabilities explicitly. These assets are inputs of the domain design sub-process. The domain design sub-process encompasses all activities for defining the reference architecture of the product line. The reference architecture provides a common, high-level structure for all product line applications [19]. Although variabilities are exposed in the previous sub-process (domain requirements engineering), they are all external variabilities (visible to end user). Therefore, internal variabilities (technical and non-visible to end user) are firstly exposed in the domain design sub-process.

The domain design activities produce reference architecture (product line architecture) as an output. This architecture is a roadmap for the succeeding sub-process, namely, domain realisation.

The domain realisation sub-process encompasses building of the previously defined architecture (reference architecture). Acquisition of the each the asset can be one of the make/buy/mine/commission options [20]:

- Make: the asset is built in-house. The main advantage of this option is the level of control that it implicates. The organisation has full control over the specification and implementation of the asset, limited only by its own capabilities. This is especially valuable when the assets are distinguishing for the product line, for example because they enable an innovative feature.

- Buy: the asset is bought as an off-the-shelf product (COTS). Commodity assets that are readily available in the market are often cheaper when bought from others. Examples are not only operating systems (e.g. Windows, Linux), middleware software (e.g. J2EE, .NET), but also development tools and processes (e.g. RUP, CMMI).

- Mine: the asset is reused from an existing system within the company. In this case, the organisation opens its lumber room and searches its existing systems for an asset to be used. Their freedom is limited by the range of assets that are

available, and how easy it is to adapt them to the platform. Especially if the system being mined has reached end-of-life or is out of use, getting high-quality assets out of it can take significant reverse engineering effort. In some cases, an application-specific asset can be taken and turned into a common asset. If the application engineering process responsible for the asset is still running, this can be relatively easy.

- Commission: the asset is assigned to be built by a third party. The asset's specifications are created in-house, but it is left to a third party to implement it. This may well create a gap between the ones who make the specifications and the ones who implement it. This implicates the risk with it that the implementation does not meet the original intention of the asset. This risk should be addressed properly, e.g. by putting effort in creating very high quality specifications, or installing extra communication mechanisms and short feedback loops.

Domain testing is responsible for the validation and verification of reusable components. Domain testing tests the components against their specification, i.e. requirements, architecture, and design artefacts. In addition, domain testing develops reusable test artefacts to reduce the effort for application testing. The input for domain testing comprises domain requirements, the reference architecture, component and interface designs, and the implemented reusable components. The output encompasses the test results of the tests performed in domain testing as well as reusable test artefacts [19].

Up to this point, all sub-processes of the domain engineering process have been explained. The other process of the SPL is application engineering and it has also respective sub-processes like the ones of domain engineering. For many aspects, the application engineering process resembles a single software development process. On the other hand, it uses domain engineering outputs in its each respective sub-process. This case differs the application engineering from a single software development process.

Application requirements are used to define a particular product. Therefore, the application requirements engineering sub-process involves definition activities of these requirements. Application requirements engineering differs from requirements engineering for single systems for the following reasons [19]: Firstly, requirements elicitation is based on the communication of the available commonality and variability of the software product line. Most of the requirements are not elicited anew, but are derived from the domain requirements. Secondly, during elicitation, deltas between application requirements and domain requirements must be detected, evaluated with regard to the required adaptation effort, and documented suitably. If the required adaptation effort is known early, trade-off decisions concerning the application requirements are possible to reduce the effort and to increase the amount of domain artefact reuse.

Application design sub-process is the process that deals with derivation of the application design from reference architecture. Variation points of the reference architecture are bound with the variants of concerning application. The requirements of the application are satisfied by resulting architecture; the application architecture.

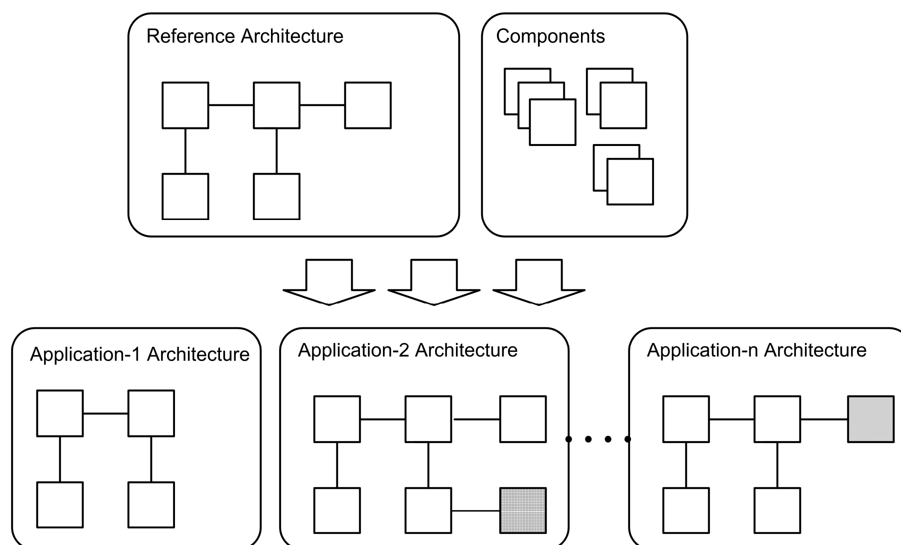

Figure 8 - Application design sub-process

The application realisation sub-process creates the considered application. The main concerns are the selection and configuration of reusable software components as well as the realisation of application-specific assets. Reusable and application-specific assets are assembled to form the application [19]. Reusable components have to have some binding mechanisms for their internal variabilities. Hence implementation of the application also covers configuring these components. F. Van Der Linden et al. state that making configuration method more uniform is less error-prone, and specialised configuration components or tool support are also options to do this more efficiently [20].

The application test sub-process is the final (for a single iteration) process of the application engineering. It is the process of ensuring that the product has specified features and quality. According to [19], the major differences from single-system testing are:

- Many test artefacts are not created anew, but are derived from the platform. Where necessary, variability is bound by selecting the appropriate variants.

- Application testing performs additional tests in order to detect defective configurations and to ensure that exactly the specified variants have been bound.

- To determine the achieved test coverage, application testing must take into account the reused common and variable parts of the application as well as newly developed application-specific parts.

Following sections briefly summarizes some leading SPL methods. Each one of these methods fulfils SPLE framework with their own approaches.

2.4.1 *Product Line Software Engineering (PuLSE)*

PuLSE is developed by Fraunhofer Institute for Experimental Software Engineering (IESE). PuLSE defines four different deployment phases and components that are consistently used in these phases.

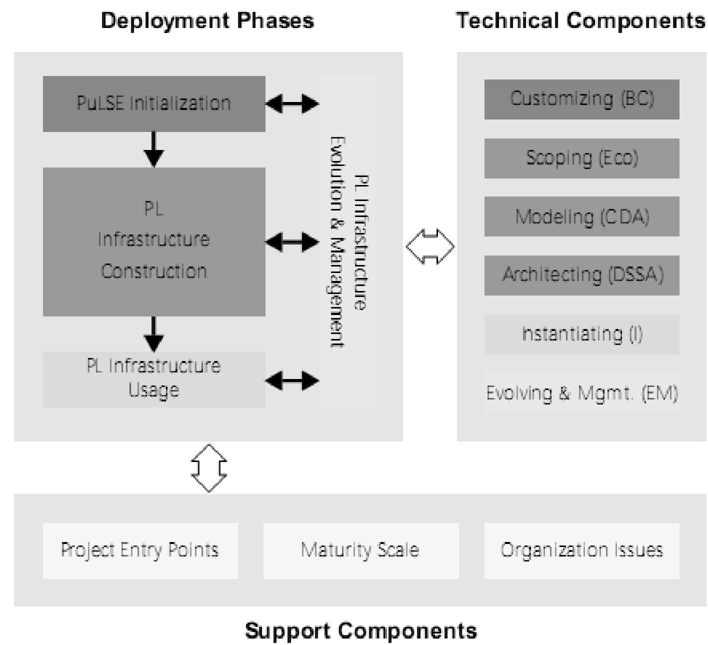

Figure 9 - PuLSE Overview [21]

Deployment phases are the main processes of the PuLSE. J. Bayer et al. define them like below [21]:

- Initialization: baseline the enterprise and customize PuLSE as a result.
- Infrastructure Construction: scope, model and architect the product line infrastructure.
- Infrastructure Usage: use the infrastructure to create product line members.

- Evolution and Management: evolve the infrastructure over time and manage it.

Technical components are the main building blocks of the deployment phases. They involve technical know-how that makes possible these phases like how to scope, how to model, how to develop, etc... Support components are guidelines which are used during deployment of the SPL.

PuLSE is very comprehensive method to implement SPL. It has very well documented processes and CASE tool supports (e.g. DIVERSITY/CDA which is developed to support modeling phase of the PuLSE).

2.4.2 KobrA

KobrA is customized version of the PuLSE process. It is component-based software engineering approach and its product line aspects are based on the PuLSE [22]. KobrA uses UML models in order to identify components.

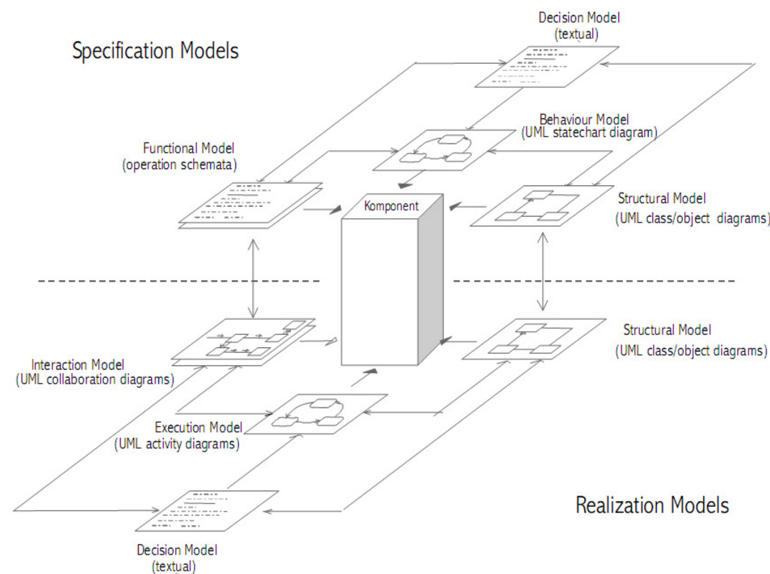

Figure 10 - UML-based Component Modeling in KobrA [22]

A component (komponent in Figure 9) is described in two distinct groups. First, specification models cover external features of a component that are visible to other components. These features can be seen as requirements of any given component. Other group (realization models) covers internal structures of the component and it can be seen as architecture of the component.

2.4.3 Component-oriented Platform Architecting Method (CoPAM)

P. America et al. define CoPAM as a method family that involves methods in order to share know-how among the developers of various product families and their family engineering methods.

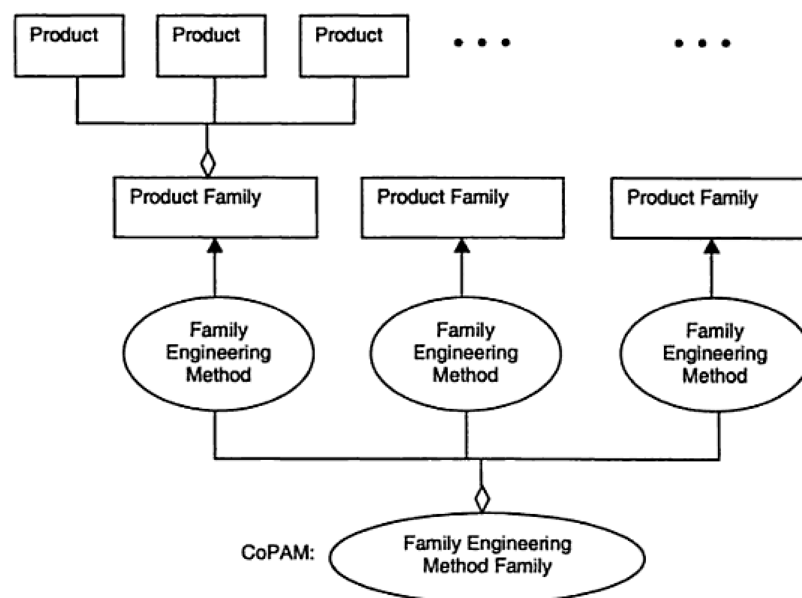

Figure 11 - CoPAM as a method family [23]

For each product family, the CoPAM approach advocates the development of a specific family engineering method from the method family, in a step called method engineering [4]. CoPAM has two main sub-processes; platform engineering and

product engineering. The first process deals with developing reusable components and the second one deals with developing products using these reusable components.

2.4.4 *Product Line ULM Based Software Engineering (PLUS)*

PLUS [24] was introduced by H. Gomaa in 2004. PLUS method can be seen as an extended form of the UML-based single system development methods (e.g. Comet which is also introduced by H. Gomaa).

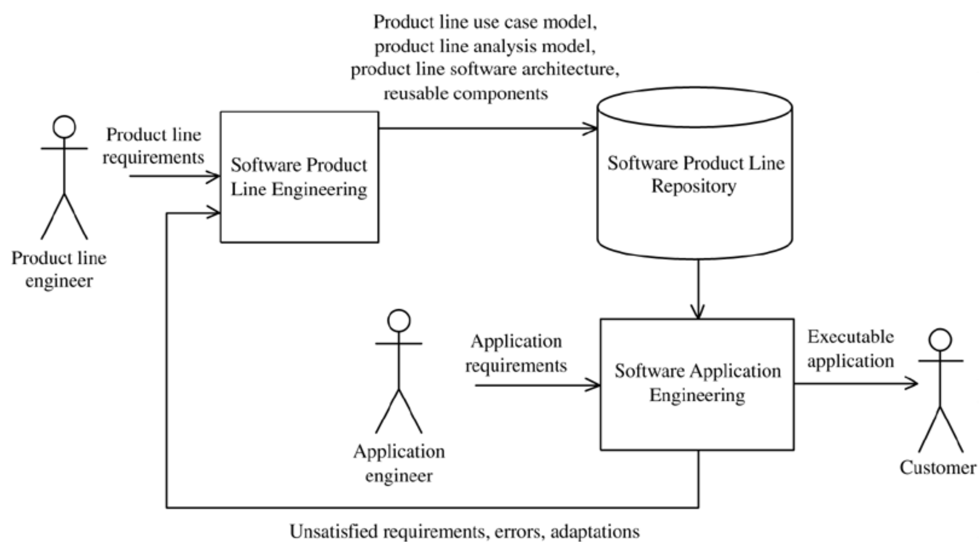

Figure 12 - Evolutionary Software Product Line Engineering Process [24]

PLUS follows a process model which is called as ESPLEP (Evolutionary Software Product Line Engineering Process). This model is adoptable to RUP (Rational Unified Process) and Spiral software development models. It can be deduced that ESPLEP has been influenced mainly from RUP. PLUS also uses UML extension mechanisms (stereotype, constraint, tag) in order to support product line modeling.

2.5 Discussion

In this chapter, notion of the software reuse has been discussed. Two fundamental reuse processes have been handled. Firstly, domain engineering processes and secondly, software product line processes have been mentioned. For the sake of brevity, some pioneering methods of these processes are also mentioned in this chapter.

In this thesis work, feature-oriented approach of FODA, CASE tool supported approach of the PuLSE and component-oriented architectural approach of the KobrA have been adopted. Albeit, the work is directly coupled none of these processes, all of them some how inspired it. For instance, meta-model of defined DSL has "feature" element that stands for encapsulated requirement(s). Architectural building blocks are also abstracted as "component" in this DSL.

CHAPTER 3

VARIABILITY MANAGEMENT AND FEATURE MODELING

3.1 Variability Management

Variability management is one of the fundamental principles of the SPLE. K. Pohl et al. define "variability" as follows: "To facilitate mass customization the platform must provide the means to satisfy different stakeholder requirements. For this purpose the concept of variability is introduced in the platform. As a consequence of applying this concept, the artefacts that can differ in the applications of the product line are modeled using variability" [19]. This definition clearly depicts the difference of the SPLE approach from other reuse methodologies. The applications are considered as variations of common theme in SPLE [20]. These variations are located (variation points) on the artefacts in prescribed way. The variants (possible values of the variation points) are also a managed set of artefacts. Therefore, adapting the common theme is under the control of variation management process.

Variabilities are divided into two separate class; internal and external ones. This classification is based on user perception and user visible ones are named as external where non-visible ones are named as internal variabilities. K. Pohl et al. defines this classification as follows; "As external variability is visible to customers, they can choose the variants they need. This can happen either directly or indirectly. In the former case, customers decide for each variant whether they need it or not. In the latter case product management selects the variants thereby defining a set of different applications among which the customers can choose. The two cases can also be combined, i.e. product management defines a set of applications but only binds a part of the external variability. Thus the customers are able to decide about the unbound variants themselves… All decisions that concern defining and resolving internal

variability are within the responsibility of the stakeholders representing the provider of a software product line. The customer does not have to take internal variability into account when deciding about variants." [19]. External variability is used to satisfy user needs and internal variability is used to enable this kind of features in the product.

Another concern of the variability management is when it is decided that some variant instance is used, known as the binding time. Variability binding can be handled at different phases. Generally, these phases are compile time, link time and run time. Any given SPL may use all or subset of these binding time options. Variabilities can be named as early or delayed, relatively (see Figure 13).

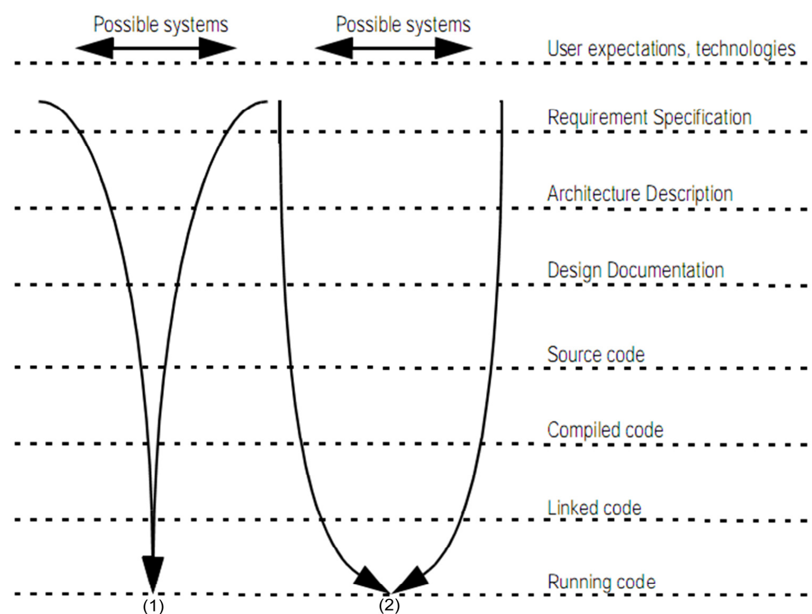

Figure 13 – Early (1) and delayed (2) variabilities [25]

Managing differences between products of the SPL is main purpose of the variability management and this objective is one of the aspects that differs the SPL from the single software development process. In fact, software variability is very different

from the SPL variability. SPL handles commonalities among the products (its members) hence variable artefact of the products may be common. In this case, this artefact is a commonality for the SPL although it is software variability in its products.

Variability exists throughout the whole process of the SPL. Specifically, domain engineering and application engineering processes handle variability as follows [26]:

- In domain engineering process, variability is identified, designed and implemented for reuse. It includes: variability identification, variability analysis & design and variability implementation.

- In application engineering process, the variability in product line is tailored and configured while developing a new member of family products. It includes: variability customization, variability configuration and variability binding.

Modeling is a kind of way to represent and share complex information in more abstract form. In this respect, modeling variabilities is very helpful and wise aspect of the variability management process.

3.1.1 Variability Modeling

As stated in the previous sections, variability management is distributed over the other sub-processes of the SPL. Therefore, it is demanded a shared information model. Modeling variability is very important for the SPL. It depicts variability power of the SPL and this is the base of many important decisions like making a new product, member of the SPL (scoping).

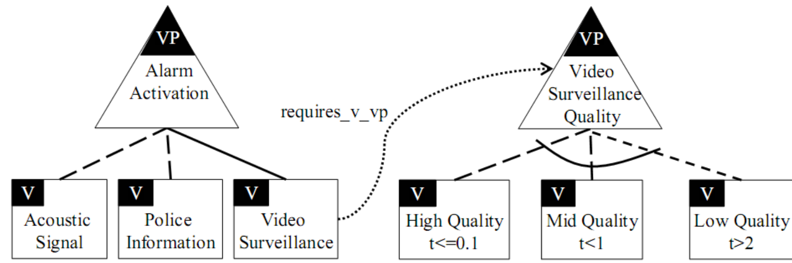

Figure 14 – Sample for Orthogonal Variability Modeling (OVM) [19]

There are some methods in order to model variability like Variability Specification Language / VSL [27], Decision-Oriented Variability Modelling / DoVML [28], UML extensions [29] [30], Orthogonal Variability Modeling / OVM [19]. Some of these methods are related only a subset of the sub-processes of the SPL, and some of them are generic methods. Furthermore, none of these methods are more widely accepted than the feature diagrams (feature modeling) in order to model requirements.

3.2 Feature Modeling

A feature model is a hierarchically arranged set of features. Relationships between a parent (or compound) feature and its child features (or sub-features) are categorized as follows [31]:

- And — all sub-features must be selected,
- Alternative — only one sub-feature can be selected,
- Or — one or more can be selected,
- Mandatory — features that required, and
- Optional — features that are optional.

Feature diagrams are graphical representations of the feature models hence, they also involve all contents of the feature model and they represent these contents graphically. A. Metzger et al. define feature diagrams are used in many diverse purposes and their notations also vary [32]. Figure 15 shows the common graphical notations of the feature diagrams.

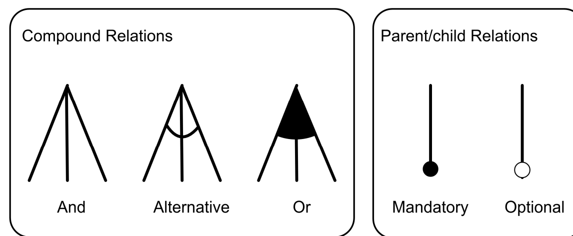

Figure 15 - Feature diagram notations

The feature diagram of the any given SPL is a way to specify product derivations of this SPL. In this respect, a valid feature configuration which is a set of selected features defines a product of SPL (whether its member or not). It's also stated that the products of the SPL are a set of bound variabilities and commonalties of this SPL. Therefore, it is also possible to use feature models to manage variabilities.

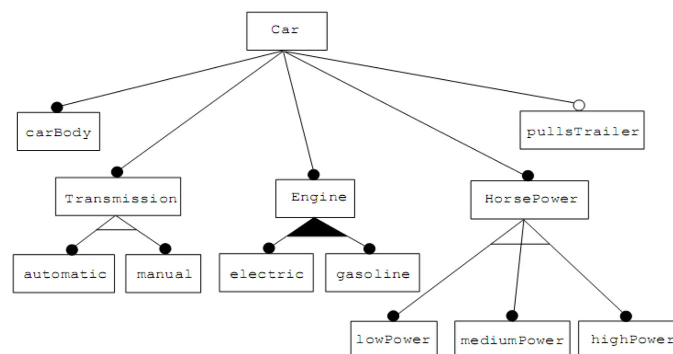

Figure 16 - De facto sample of the feature diagrams; a simple car [33]

According to the above feature diagram (Figure 16): Any given car has to have a body, transmission, engine and horse power but it is optional that the car pulls a trailer. Transmission of the car is automatic or manual. Its engine works with gasoline or electric or the both of them (hybrid). Its horse power is one of these classifications; low, medium, high.

Above paragraph is a textual definition of the sample feature diagram. In this respect, it does not define any specific configuration instance of the feature model. Possible options have to be selected in order to get a valid feature configuration that defines a product. For instance, again according to the diagram in Figure 16, it is a valid feature configuration: carBody, manual, electric, gasoline, mediumPower, pullsTrailer.

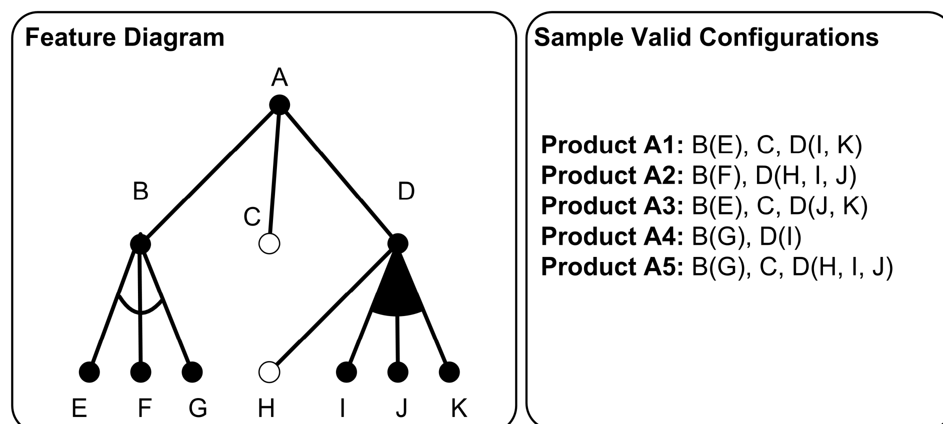

Figure 17 - Sample feature configurations and their feature diagram

It is possible to derivate many distinct feature configurations from even tender feature diagrams. It may become cumbersome task to handle these diagrams. In this respect, D. Batory proposed a grammar for feature models [31] in order to handle more comprehensive models. Figure 17 depicts feature diagram and some sample configurations that obey to the model which is defined by this feature diagram. It can

be easily deduced that supporting feature modeling activity with a tool is a wise choice.

3.2.1 Feature Modeling Tools

Feature modelling is important activity and its usage is very common in diverse application areas (not only software development). Thus, there are many feature modelling CASE tools which are ready to use; pure::variant from Pure Systems [34], fmp from University of Waterloo [35], Captain Feature [36], FeatureIDE from University of Magdeburg [37].

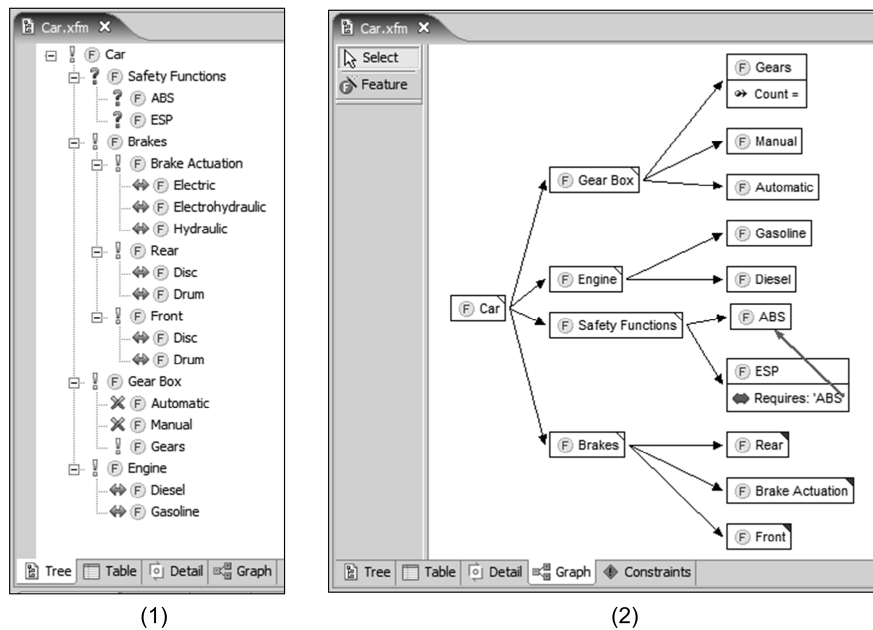

Figure 18 - Tree display(1) and graph display(2) of a model in pure::variant

3.3 Discussion

Variability management is very crucial for any given SPL process. In this respect, this management activity is determinative part of the general SPL process; well managed variabilities contribute to make the SPL more effective.

Modeling is an indispensable part of the variability management. Therefore it is also crucial for the SPL process. Feature modeling is a very suitable way to model variabilities, and it is widely accepted by the SPL community. Kutulu DSL does not cover feature models but it covers feature-component bindings. Thus, it is assumed that one analyzed feature-to-feature dependencies in the previous phases of the SPL process and final form of them is a resolved feature set (valid feature configuration; see Figure 17). It has been defined in more detail in following chapter.

CHAPTER 4

DEPENDENCY INJECTION FOR DYNAMIC SPL

4.1 Dynamic Software Product Lines

Software engineers are faced with increasing pressure to deliver high-quality software more economically. It is also desired that the delivered software has high degree of adaptability. In this respect, software reuse has evolved from reuse of code snippets to large-scale software product lines. Recently, Dynamic Software Product Line (DSPL) extended the SPL with dynamic features.

S. Hallsteinsen et al. state that any given DSPL has subset of the following features [38]:

- Dynamic variability (configuration and binding at runtime)
- Changes binding several times during its lifetime
- Variation points change during runtime; variation point addition (by extending one variation point)
- Deals with unexpected changes (in some limited way)
- Deals with changes by users, such as functional or quality requirements
- Context awareness (optional) and situation awareness
- Autonomic or self-adaptive properties (optional)
- Automatic decision making (optional)

- Individual environment/context situation instead of a “market”.

The product of a DSPL is capable of adapting to diverse operating environments after deployment. This is very important capability, especially for the closed run-time environments that do not allow remote access and easy configuration changes. This situation is not exceptional in domains like defence. Therefore, SPL development efforts for this type of domain (or market) have to consider dynamic notions. However, a more adaptable system requires more configurations to be supported and each new configuration item increases the complexity of variability management exponentially [39]. Accordingly, we have to set up some mechanisms in order to handle variability and these mechanisms have to support our management processes.

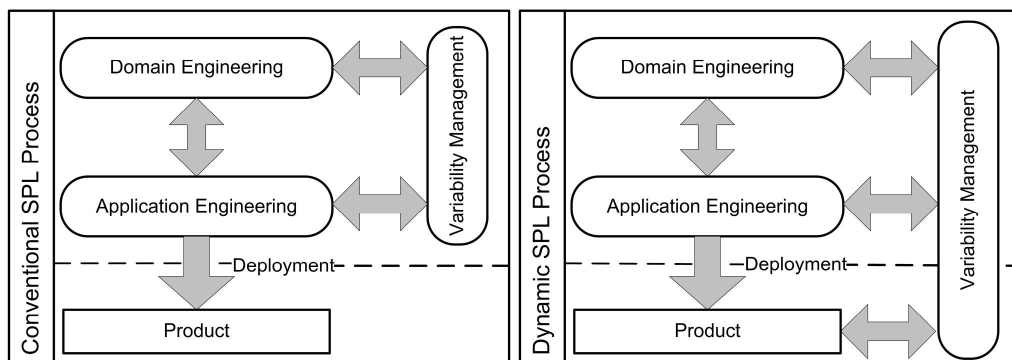

Figure 19 - SPL vs. DSPL on variability management

Although traditional SPL engineering recognizes that variation points are bound at different stages of development, and possibly also at runtime, it typically binds variation points before delivery of the software [38]. In contrast, DSPL typically binds variation points at run-time.

Binding time of the variations is a common and determinative property of DSPLs. It is generally very hard to classify any given SPL as a dynamic or not because DSPL is an extension to SPL process. As previously mentioned, some dynamic properties

may be included by the SPL and more of these properties make the SPL in question more dynamic.

4.2 Dependency Injection

Dependency Injection (DI) is a style of object configuration in which an object's fields are set by an external entity. In other words, objects are configured by an external entity. Dependency injection is an alternative to having the object configure itself (setting its own fields with self created instances). Using this style provides dependency agnostic (actually, it is only depended to interface definition) code and externalized dependencies. Some empirical studies state that dependency injection is not widely used because it is not common part of the software design courses [40]. Although DI has not deserved value recently, it is increasing its popularity in software development area, nowadays.

The following example DI implementation is taken from [41] and it shows the basic idea of DI. In naïve form, MovieLister creates MovieFinderImpl and uses this object via MovieFinder interface. Therefore, it is dependent on both of these two classes. On the other hand, using DI style changes these dependencies (see Figure 20). In this case, external entity (Assembler) injects appropriate implementation (MovieFinderImpl) into the MovieLister object. Then, it uses this injected object via MovieFinder interface. It only depends on this interface, not its implementation.

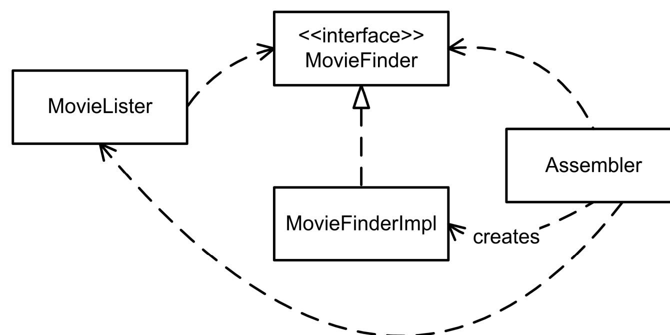

Figure 20 - The dependencies for a Dependency Injector [41].

DI is a very appropriate style for implementing a DSPL. An external entity (assembler) can easily manage dependencies on behalf of the actual product and this entity encapsulates all component configurations. The assembler injects all dependencies at start-up time or later execution time of the product. Thus, DI can be regarded as an enabling style for the dynamic product line architecture in this thesis work.

Table 2 shows some DSPL features and respective DI contributions. Albeit, this list is very appropriate to be extended, even these items prove that DI is a suitable style for DSPL.

Table 2 - Some DSPL features and dependency injection

DSPL Feature	How DI contributes to enable this?
Dynamic variability; configuration and binding at runtime	Assembler can configure and bind the variation points at runtime.
Changes binding several times during its lifetime	Assembler can have an external method which binds variation points and this method can be invoked several times.
Deals with unexpected changes (in some limited way)	In the exceptional cases, assembler can bind the variation point with pre-defined variant.
Deals with changes by users, such as functional or quality requirements	Configuration of the assembler can be visible to user (for instance options screen in GUI) hence; binding scheme can be changed by users.
Autonomic or self-adaptive properties	Status information (situation) can be input for the assembler. Therefore it analyzes this input and decides the bindings.
Automatic decision making	

4.3 Kutulu DSL for Modeling

Modeling architectural artefacts by defining traceability links to the concerning requirements is not an easy task. Therefore, supporting modeling process is a wise choice. In this respect, by using DSL, designing reference architecture with pre-defined components (domain design) and then for each feature, constructing traceability links between features and components (called the feature-component binding) provide solid foundation to expose dependencies. DSL (as the term

suggests) is a language dedicated to a particular domain. Therefore, any given DSL is a mediator which is used for expressing a problem in its domain.

In the frame of this work, a visual DSL has been introduced in order to support domain and application design processes of the SPLE, namely “Kutulu DSL”. The main objective of the Kutulu DSL is to be able to express features and components of the SPL (step 1-2 refer to this expression activity in Figure 21) in accordance to a common meta-model. These expressions are usable for an external entity which is employed to set up dependencies.

In previous sections, fundamental SPLE processes and how they can attain dynamism with dependency injection have been explained. It is clear that in order to employ an external entity (assembler) to inject dependencies we have to provide enough information to this entity. According to our method this information can be captured by the Kutulu DSL during domain design. Later, it can be used in application design (see Figure 21, step 2).

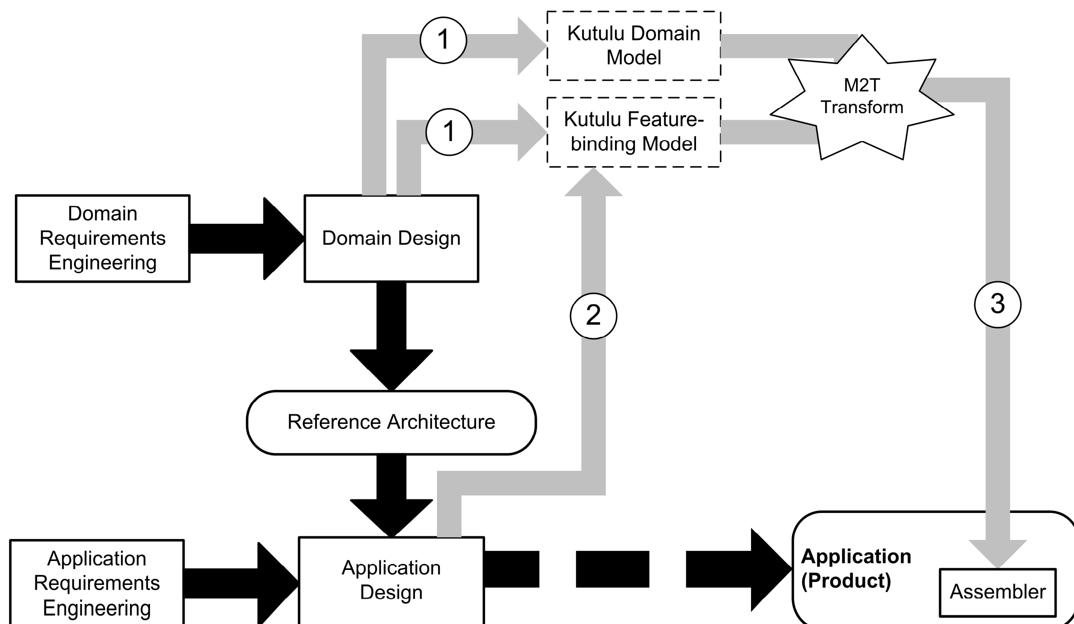

Figure 21 - Using Kutulu DSL in SPL Process

In Figure 21, black arrows refer to original SPLE processes and gray ones refer to Kutulu extensions. Respectively, reference architecture and feature-component bindings are defined in DSL (see Figure 21, step 1). Application design is expressed by activating/deactivating features on model (see Figure 21, step 2). Identified dependencies are transformed into a configuration for assembler (see Figure 21, step 3). Assembler uses these configurations at run time.

As mentioned in section 2.4, requirements which are analyzed in the preceding process (domain requirements engineering), are input of the domain design process and product line architecture (reference architecture) is the main output of it. According to Kutulu approach, abstract requirements (features) have to be mapped to components in this phase and these mappings have to be expressed in the Kutulu DSL (specifically, the feature-component binding model). Inter-relations of the components are also important information sources to decide dependencies. Hence, modeling components and their relations (Domain model) in terms of Kutulu DSL are also crucial to apply our method. After these steps, identified dependencies can be injected by the DI assembler at run time. Although, it is possible to implement a custom-made assembler, using a widely accepted assembler (COTS) is more reliable way. In this respect, a separate transformation is defined. It exports the configuration that can be processed by a DI assembler (e.g. Spring, Spring.NET, Google Guice).

4.3.1 Meta-models of Language Elements

Kutulu DSL is based on two meta-models. The first one is the domain meta-model which is used to model components and their inter-relations. Each domain model is composed of the component and relation definitions. Component definitions simply refer to implementation assets, whereas relation definitions refer to different situations according to their types. There are three types of relations in the DSL. Assume that there is a relation between two different components; A and B.

- Uses Relation: A calls a method of B and B is used by A.

- Creates Relation: A calls the constructor method of B (this is the case where A is the factory of B).

- Event Relation: A publishes an event which is related to B.

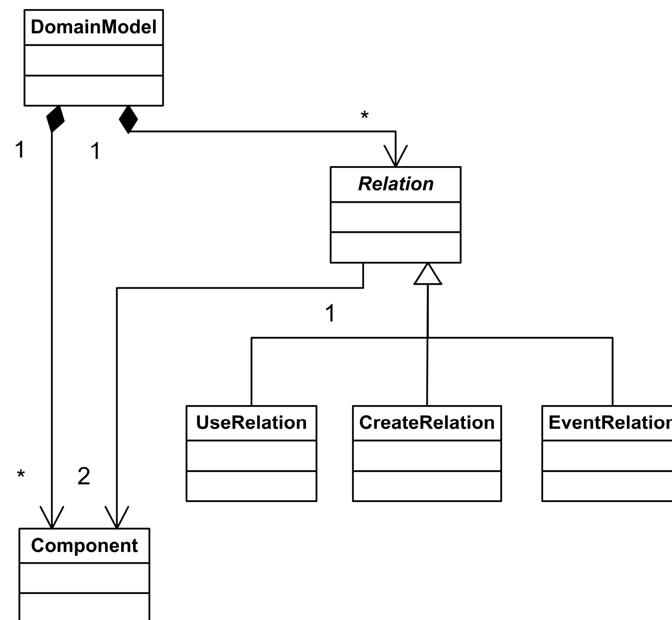

Figure 22 - Domain meta-model of Kutulu DSL

The other meta-model component of the language is the feature-binding meta-model which enables the modeling of feature-component bindings. Each feature is related to some components and enabling any given feature for the product means binding some properties of its related components and activating them in the product. Thus, our language has to cover feature, component and property as essential concepts. As we stated in section 2.3, properties can be concrete value, reference value. Moreover, heterogeneous collections (uniform or mixed type of elements) of these values are also valid property instances. Therefore, list type property also must take place in language meta-model. Below diagram depicts the feature-binding meta-model of the Kutulu DSL.

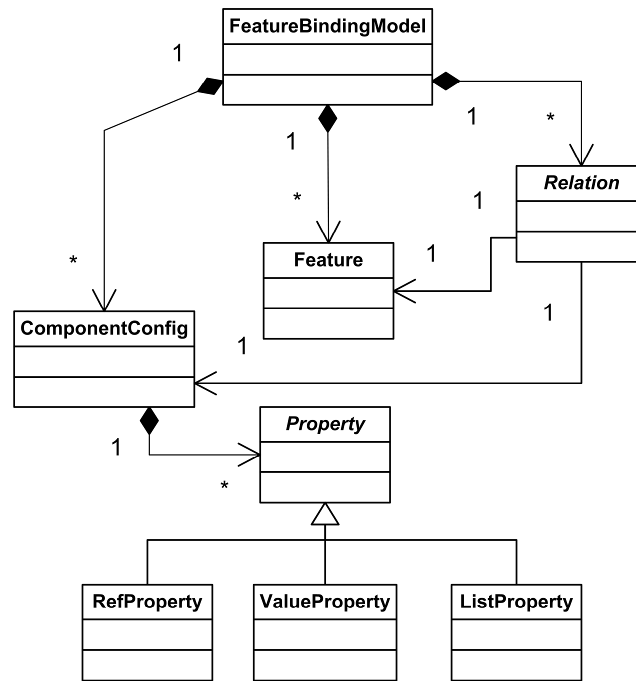

Figure 23 - Feature-binding meta-model

4.3.2 Kutulu CASE Tools

Presently, there are two modeling tools in Kutulu project: the domain editor and the feature-binding editor. As explained in the previous sections, there are also two main information sources in domain design process, namely, the domain and feature-binding models. These models can be composed by using concerned editors. Thus, dependencies are readily expressed in our DSL. The third tool transforms (see Figure 21, step 3) DSL expressions into a configuration for the assembler.

Eclipse is a very suitable platform to develop graphical/textual editors (e.g. IDEs). Therefore, there is no need to implement the editors from scratch. In this frame, Kutulu CASE tools had been built on top of Eclipse platform. In this platform, GMF is a framework which is used for implementing graphical editors [42]. Both of the Kutulu editors (domain modeling and feature-component binding) are GMF-based editors. GMF uses EMF meta-models to construct a graphical editor. Therefore, meta-model has to be defined as EMF model in order to implement GMF-based

editor. Then, the editor is generated by using some other manual configurations. On the other hand, there exists a tool that handles these manual configurations in order to make editor development easier. EuGENia (part of the Epsilon project [43]) is a tool that automatically generates these manual configurations according to the defined annotations in the EMF model. Therefore, it is also possible to directly generate the editors from meta-model with some annotations.

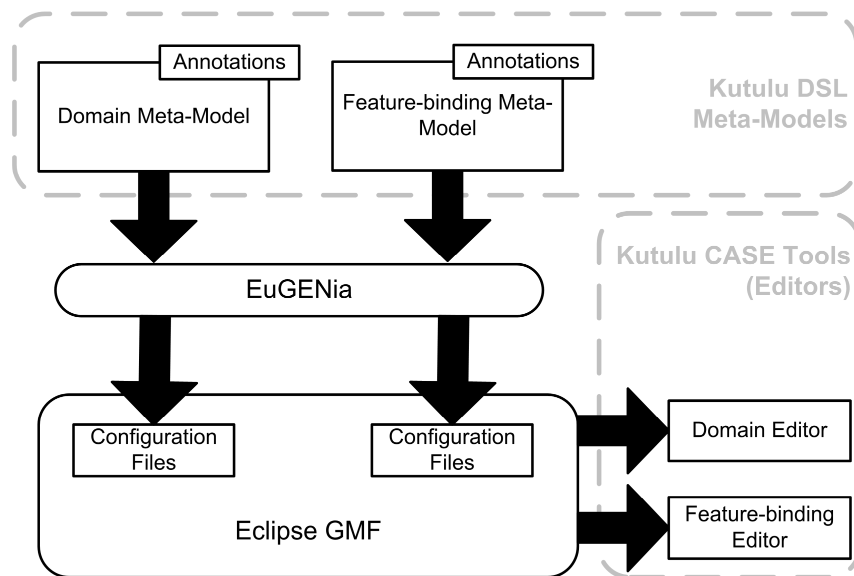

Figure 24 - Implementation of Kutulu CASE Tools (Editors)

The main objective of these tools is to generate assembler configuration (see step 3 in Figure 21). Therefore, crucial last step of the process is M2T transformation which is handled by another tool called Kutulu generator. Generation is coupled with the assembler hence; the generator is implemented separately for the specific assembler.

In the present work, a simple generator has been developed for Spring.NET DI assembler as a proof of concept. This generator is implemented as an EGL (Epsilon Generation language) script which is interpreted by EGL generator (part of the Epsilon project) [44].

4.3.3 Using Kutulu Generator for Assembler Configuration

Spring.NET is one of the leading application frameworks for the .NET application development platform. Spring.NET provides comprehensive infrastructural support for developing .NET applications. It allows one to remove incidental complexity when using the base class libraries facilitate best practices, such as test driven development, easy practices [45].

Spring.NET has many different modules and each of these modules has many crucial features. Spring.Core is one of these modules and it is used to configure applications using dependency injection. The design of Spring.NET is based on the Java version of the Spring Framework [46], which is used in many different applications world wide.

Kutulu generator transforms user-defined dependencies into the form of configuration which is understandable by DI assembler, in this case Spring.NET assembler.

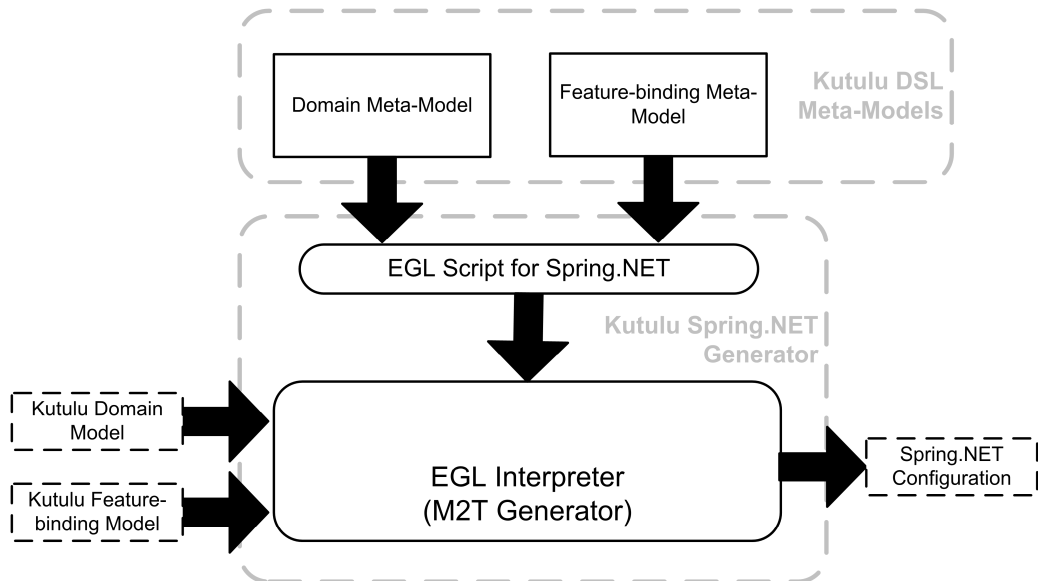

Figure 25 - Implementation of Spring.NET Generator

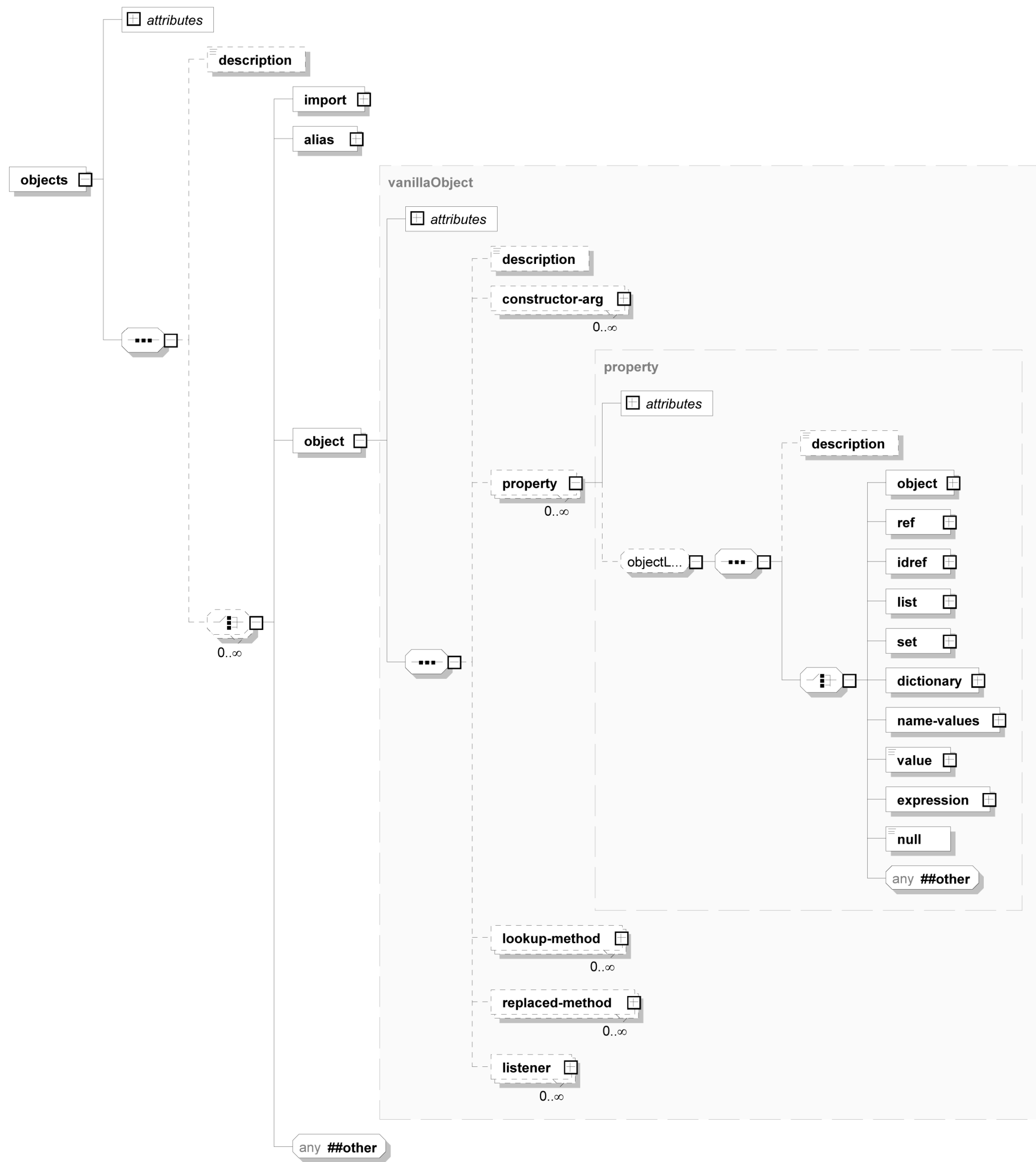

Figure 26 - Spring.NET objects XML schema

Figure 26 shows the XML schema of the Spring.NET objects. In this schema, there are many different information sources to be used in dependency injection operation by Spring.Core (Container / Assembler). Important information sources in “objects” section of the schema are explained below:

- For each (container handled) object of the application there is a “object” element. This element has to have “id”, “type” attributes and it may has “property” elements.
    - “id” attribute: Alias name of the object which is valid within the scope of the configuration file. All other objects that reference the concerning object use this alias name.
    - “type” attribute: Compound information of the object. It holds both class name of the object and its namespace information (implementation details of the object).
    - “property” element(s): All variable properties of the object are bound via these elements.
- For each property of the object there is a “property” element. There are three types of properties; reference, value and list properties. “property” element has “name” attribute which is independent from its type. “reference” type property has “ref” attribute, “value” type property has “value” attribute and “list” property has “list” element.
    - “name” attribute: Name of the property which is both valid in the configuration file and in the implementation class.
    - “ref” attribute: Reference value of the property. This value must be a valid alias name of the object in the related configuration file.

- “value” attribute: Concrete value of the property. This value is evaluated as a primitive type.

- “list” element: If the property type is “list”, there must be a multiple value in it. Each member of this list may be a list (recursively) or one of the other two types (reference and value).

- For each list type property of the object, there is a “list” element. This element has “element-type” attribute that defines the type of the members. It encapsulates its members as inner elements.

Above elements are only small part of the configuration schema but they are adequate for the common object bindings. Kutulu generator maps these elements of the simplified schema to the elements of DSL meta-model definitions (see section 4.3.1).

CHAPTER 5

CASE STUDIES

5.1 MVC, Factory Design Patterns and Two Alternative Features

Assume that, we are implementing an SPL for Fire Support Command and Control systems. The requirement analysis revealed that our SPL must support two different features as display option: UMPC (Ultra Mobile Personal Computer) display, which is suitable for tiny screens, and PC display, which is suitable for regular screens. Furthermore, we require our products to handle variability at run-time. During domain design, it is agreed that MVC (Model-View-Controller) and Factory design patterns will be used. As a sample implementation, we modelled "Target module" of the system which shows a list of targets in the final product.

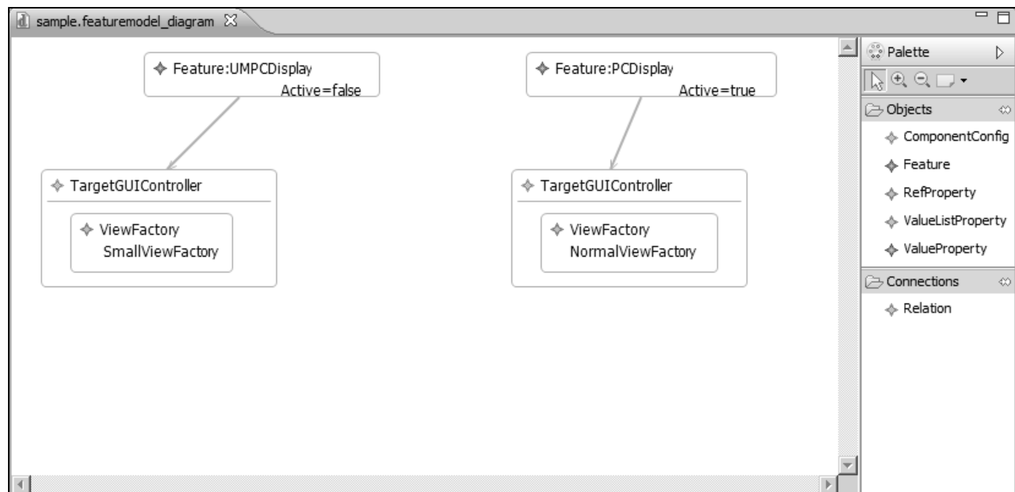

Figure 27 - Sample model in Kutulu feature-component binding editor

According to our simple feature-component binding definition, UMPCDisplay feature sets ViewFactory property of TargetGUIController component to SmallViewFactory reference value and PCDisplay feature sets same property to NormalViewFactory reference value.

During the application design process, the developer uses the feature-component binding editor in order to customize the particular product by selecting the activity status of the features. Current model (Figure 27) proves that PCDisplay feature is active for the product.

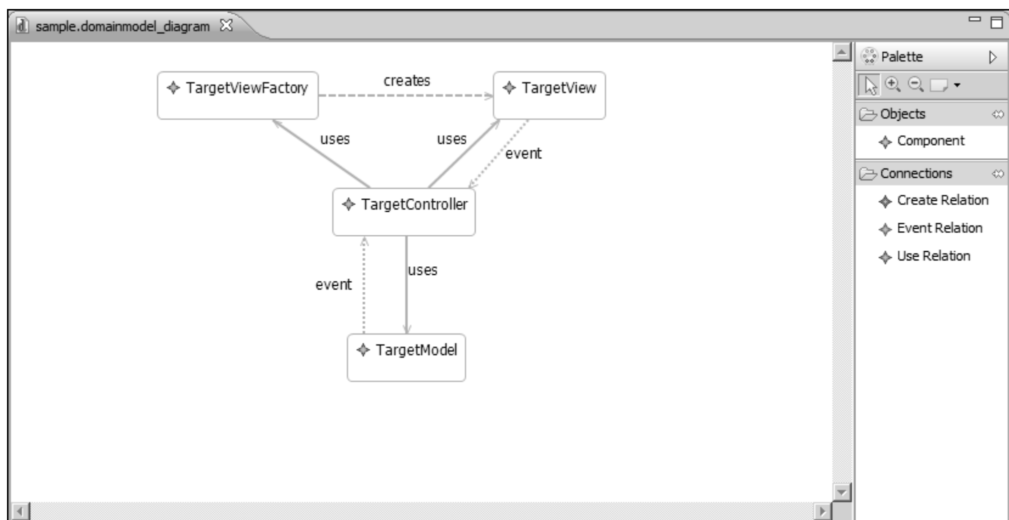

Figure 28 - Sample model in Kutulu domain editor

Figure 28 illustrates the implementation of the MVC design pattern for Target module. Note that there is a factory that provides platform independence to controller component.

As stated in section 4.2, we provide the notion of dynamism by using DI. We decided to use Spring.NET as DI assembler for this project. In this case, we have to run Kutulu transform tool to generate configuration for this assembler (currently,

Spring.NET is only DI assembler which is supported by Kutulu transform tool). Following configuration is generated for above DSL definitions.

```
<?xml version="1.0" encoding="utf-8" ?>
<objects xmlns="http://www.springframework.net" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" default-autowire="byName">
    <description>Application Context (Generated by Kutulu)</description>

    <object id="objTargetGUIController" type="TargetMng.GUI.TargetController, TargetHandler">
        <property name="ViewFactory" ref="objNormalViewFactory"/>
        <property name="Model" ref="objTargetModel" />
    </object>
    <object id="objNormalViewFactory" type="TargetMng.GUI.NormalViewFactory, TargetHandler" />
    <object id="objTargetModel" type="TargetMng.BL.TargetModel, TargetHandler" />
</objects>
```

Figure 29 - Generated Spring.NET configuration

5.2 Sample Product of TADES SPL

TADES is a SPL which is developed by Aselsan Inc. [47] in order to satisfy demands of the technical fire support C2 software market. TADES defines tiered architecture for applications and it also uses some very well known design patterns (e.g. Abstract Factory, MVC) in its reference architecture. Full details of the reference architecture are commercially confidential hence it has been partially analyzed in this work.

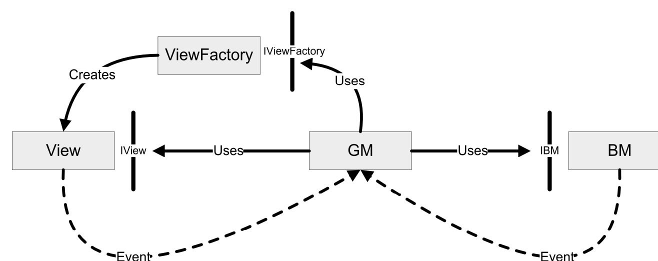

Figure 30 - Sample TADES application module

Typical architectural view of the TADES application module is depicted in Figure 30. All presentation logic is encapsulated in a view component and creation of this component is managed by a factory (ViewFactory in Figure 30). GUI Manager (GM in Figure 30) manages all GUI related operations and forwards model updates to view and vice versa. Business Manager (BM in Figure 30) encapsulates all business logic. There may be lots of other architectural relations like that pattern application and these relations are also needed to be expressed in Kutulu domain editor (see Figure 21, step 1).

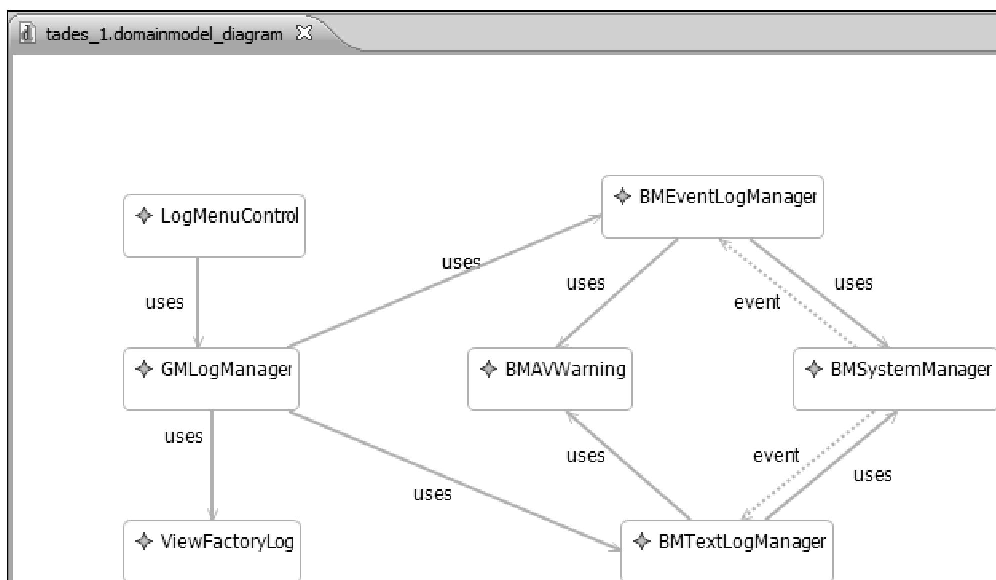

Figure 31 - TADES Log operations in Kutulu domain editor

Figure 31 depicts the part of the TADES domain model. Shown part covers the log operation components and other components that have relationship to them. Another part of the domain model is also listed below. This part includes the components that deal with meteorological report operations (relation labels are omitted for the simplicity all relations are "uses").

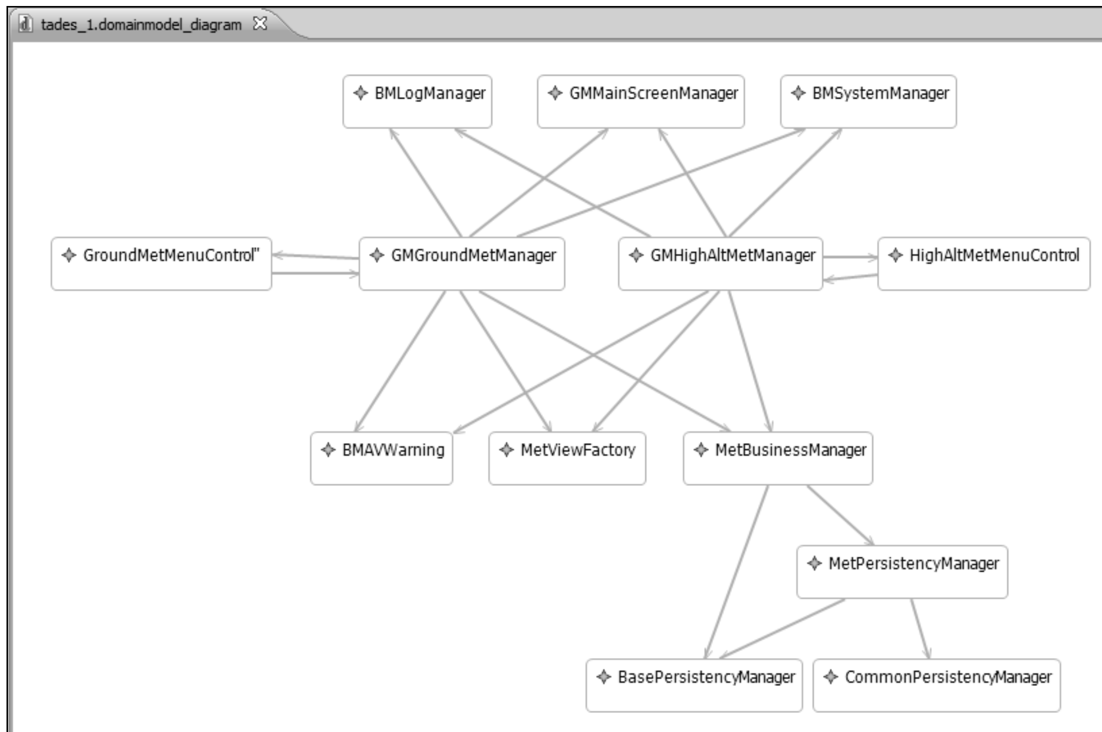

Figure 32 - TADES Met operations in domain editor

Below feature tree is part of the actual TADES feature tree. As stated previously, full tree is out of this work's scope, due to the commercial confidentiality. On the other hand this partial tree is also adequate to prove the concept.

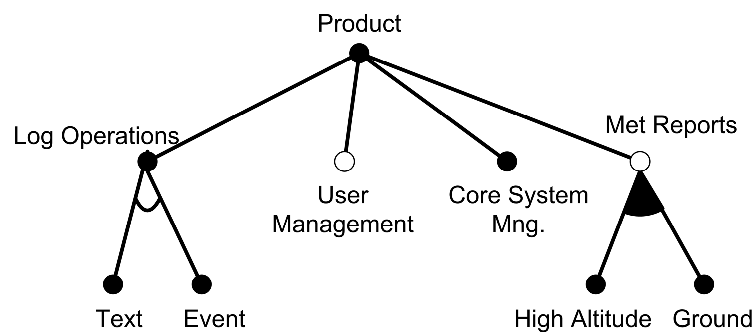

Figure 33 - Sample part of TADES feature tree

Assume that there exists simple application that manages fire support meteorological reports in the scope of the TADES SPL. Application engineer has defined this product as a valid feature configuration of the feature tree (Figure 33). It is assumed that the defined product uses event log as log registry and there is no need for user management (authentication, registration, etc…). As meteorological reports, it supports both high altitude and ground met reports.

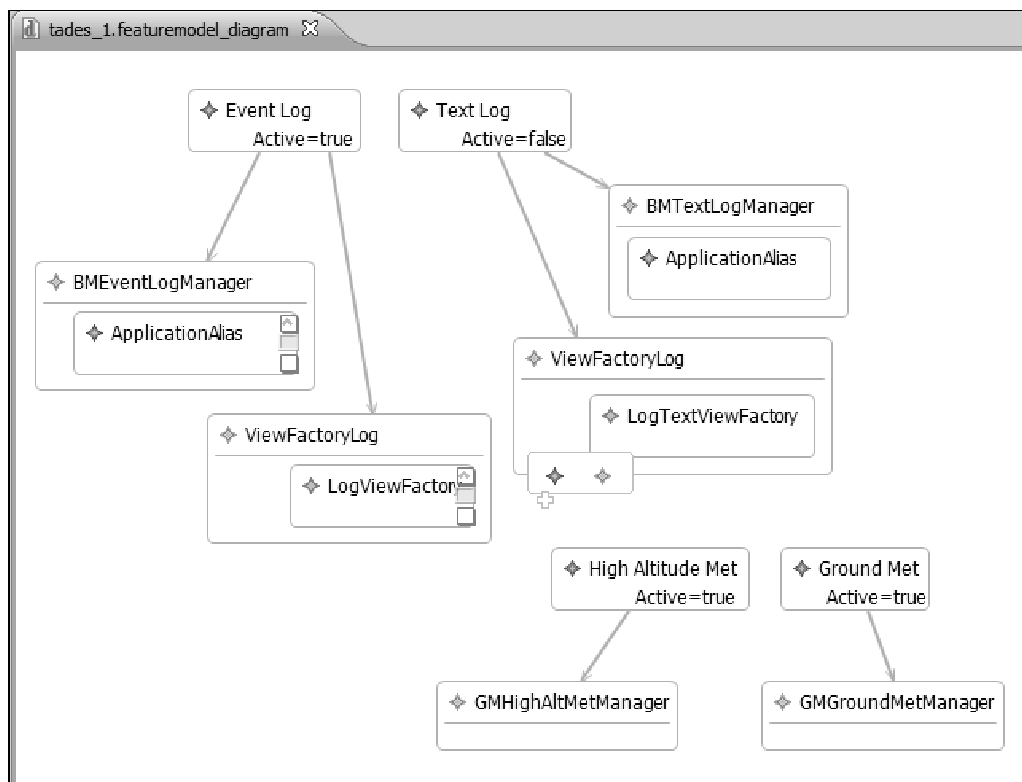

Figure 34 – Feature bindings of log and met operations

Up to this point, domain design phase activities have been completed. Domain and feature-binding models are ready to be used in application design. Application engineer is able to define product configuration by using feature-binding editor (see

Figrue 34). In this editor, each feature has an activity flag and application engineer activates demanded features by using these flags (see Figure 21, step 2).

Finally, Kutulu Spring.NET generator gets defined two model file and generates below configuration file (see Figure 21, step 3).

```
84    <object id="objBMLogManager" type="Tr.Com.Aselsan.TADES.ApplicationFramework.LogManager.BMEventLogManager,
      AppFWBusinessManagers">
85      <property name="ApplicationAlias" value="TADES" />
86      <property name="BMAVWarningManager" ref="objBMAVWarning" />
87      <property name="BMSystemManager" ref="objBMSystemManager" />
88      <property name="Configurator" ref="objBMConfigurationManager" />
89    </object>
90
91    <object id="objGMLogManager" type="Tr.Com.Aselsan.TADES.ApplicationFramework.LogManager.GMLogManager,
      AppFWGUIManagers">
92      <property name="MainScreenManager" ref="objGMMainScreenManager" />
93      <property name="LogBusinessManager" ref="objBMLogManager" />
94      <property name="ViewFactory" ref="objViewFactoryLog" />
95      <property name="BMSystemManager" ref="objBMSystemManager" />
96    </object>
97
98    <object id="objViewFactoryLog" type="Tr.Com.Aselsan.TADES.ApplicationFramework.LogManager.LogViewFactory,
      AppFWGUIManagers" />
99
100   <object id="objLogMenuControl" type="Tr.Com.Aselsan.TADES.ApplicationFramework.LogManager.MenuListControl,
      AppFWGUIManagers">
101     <property name="Controller" ref="objGMLogManager" />
102   </object>
```

Figure 35 - Generated configuration

As previously mentioned, Spring.NET Core component uses generated configuration file hence; Figure 36 shows the part of the sample product screen. Two menu items of “met reports” proves that the both feature is active and main data grid is also shows event log entries of application.

Figure 36 - Configured TADES application

Assume that the product configuration has been changed. Thus, there is no need for ground met report and all logs will be stored in text file. Application engineer activates appropriate features and re-generates the configuration. Application reboots and changed configuration applies to the product (see Figure 37).

Figure 37 – Configuration change in TADES application

CHAPTER 6

CONCLUSION AND FUTURE WORK

This thesis presents a contribution about dynamic software product line approach. The main motivation is to find a declarative way to realize the DSPL concept. This study demonstrates that inversion of control is a very appropriate way to implement DSPL. Furthermore, it also contributes to conventional SPL approach.

Firstly, architectural DSL definitions (domain model) are transported to a running application by a CASE tool. This extension makes domain design process easier for the designers, especially in posterior cycles of domain engineering. Secondly, Kutulu DSL and its CASE tools provide semi-automatic architectural variability management. An application engineer only activates the required features to customize the reference architecture in application design process.

DI style is applied in order to bring dynamism to the products of the SPL. Presently, the Kutulu generator has limited support for DI assemblers (currently only one) and it is planned to be extended.

Although our case studies only exhibit start-up time product customization, it is also possible to apply more sophisticated run time variability bindings. Altering assembler configurations according to the run time environment parameters is one possible way to implement adaptive assemblers. Custom-made assemblers can be implemented as a future work. In this case, required extensions to the DSL must be considered to work with these assemblers more effectively.

REFERENCES

[1] P.Clements, L. Northrop, “Software Product Lines: Practices and Patterns”, Addison Wesley, 2001

[2] Jan Bosch, “Design and Use of Software Architectures”, Addison-Wesley, ACM Press Books, 2000

[3] Krysztof Czarnecki, Ulrich Eisenecker, “Generative Programming: Methods, Tools, and Applications”, Addison-Wesley Professional, 2000

[4] Eduardo Santana de Almeida, et al. “C.R.U.I.S.E. - Component Reuse In Software Engineering”, CESAR e-books, 2007

[5] F.L. Bauer et al., “Report on Software Engineering Conference”, Garmich, Germany, NATO Science Committee, 1968

[6] “Software Engineering” an Oxymoron?, http://squab.no-ip.com/collab/uploads/61/IsSoftwareEngineeringAnOxymoron.pdf, last visited on August 2009

[7] Edsger W.Dijkstra, “The Humble Programmer”, ACM Annual Conference, Boston, 1972

[8] A.Kleppe, J.Warmer,W. Bast, “MDA Explained: Model-driven Achitecture: Practice and Promises”, Addison Wesley, 2004

[9] Alain Abran et al., “SWEBOK – Guide to the Software Engineering Body of Knowledge”, IEEE, 2004

[10] Wayne C. Lim, "Managing Software Reuse", Prentice Hall PTR, 2004

[11] Kyo Kang et al., "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI-90-TR-021, Software Engineering Institute - Carnegie Mellon, 1990

[12] M. A. Simos, “Organization Domain Modeling: A Tailorable, Extensible Framework for Domain Engineering”. Proceedings of the 4th International Conference on Software Reuse (ICSR ’96), pp. 230 - 232

[13] “Reusability Library Framework AdaKNET and AdaTAU Design Report. Technical Report”, PAO D4705-CV-880601-1, Unisys Defense Systems, System Development Group, Paoli, Pennsylvania, 1988

[14] M. Simos, 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

[15] Ivar Jacobson, I. Jacobson, M. Griss,"Software Reuse: Architecture, Process And Organization For Business Success", Addison-wesley Professional, 1997

[16] “Rational Unified Process – Best practices for Software Development Teams – Rational Software Whitepaper”, Rational Software, 2001

[17] M. L. Griss, J. Favaro, M. D’Allessandro, “Integrating Feature Modeling with the RSEB”, Proceeedings of Fifth International Conference on Software Reuse, IEEE Computer Society Press, 1998

[18] Kyo C. Kang et al., “FORM: A feature-oriented reuse method with domain-specific reference architectures”, Ann. Softw. Eng., Vol. 5, pp. 143-168, 1998

[19] K. Pohl, G. Böckle, F. Van Der Linden, “Software Product Line Engineering: Foundations, Principles and Techniques”, Springer, 2005

[20] F. Van Der Linden, K. Schmid, E. Rommes, “Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering”, Springer, 2007

[21] Joachim Bayer et al., “PuLSE: A Methodology to Develop Software Product Lines”, Proceedings of the symposium on Software reusability, ACM, pp. 122 - 131, 1999

[22] C. Atkinson et al., “Component-Based Software Engineering: The KobrA Approach”, Proceedings of the First Software Product Line Conference, 2000

[23] Pierre America et al., "CoPAM: a compact-oriented platform architecting method family for product family engineering", Proceedings of the first conference on Software product lines : experience and research directions: experience and research directions, pp. 167-180, 2000

[24] Hassan Gomaa, “Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures”, Addison Wesley Longman Publishing Co. Inc, 2004

[25] Jilles van Gurp, “Variability in Software System The Key to Software Reuse”, Licentiate thesis, Department of Software Engineering and Computer Science - Blekinge Institute of Technology, Karlskrona, 2000

[26] Jianhong Ma, Runhua Tan, "Handling Variability in Mass Customization of Software Family", International Federation for Information Processing (IFIP), Volume 207, Knowledge Enterprise: Intelligent Strategies In Product Design, Manufacturing, and Management, Springer, Boston, pp. 996-1001, 2006

[27] M. Becker, “Towards a General Model of Variability in Product Families”, Proceedings of the First Workshop on Software Variability Management, Groningen, February 2003

[28] Deepak Dhungana, Paul Grünbacher, “Understanding Decision-Oriented Variability Modelling”, First Workshop on Analyses of Software Product Lines, in collocation with the 12th International Software Product Line Conference, Limerick, 2008

[29] H. Gomaa, M.E. Shin, “Multiple-View Meta-Modeling of Software Product Lines”, 8th International Conference on Engineering of Complex Computer Systems (ICECCS 2002), IEEE Computer Society, pp. 238-246, 2002

[30] M. Clauss, “Modeling variability with UML”, Young Researchers Workshop Part of the Third International Symposium on Generative and Component-Based Software Engineering (GCSE 2001), Erfurt, 2001

[31] D. Batory, “Feature Models, Grammars, and Propositional Formulas”, Proceedings of Software Product Line Conference (SPLC), 2005

[32] Andreas Metzger, Patrick Heymans, "Comparing Feature Diagram Examples Found in the Research Literature", Technical Report, Software Systems Engineering University of Duisburg-Essen, 2007

[33] Arie van Deursen, Paul Klint, "Domain-Specific Language Design Requires Feature Descriptions", Journal of Computing and Information Technology, 2001

[34] pure::variants, http://www.pure-systems.com/pure_variants.49.0.html, last visited on August 2009

[35] Feature Modeling Plug-in, http://gsd.uwaterloo.ca/projects/fmp-plugin/, last visited on August 2009

[36] Captain Feature, http://sourceforge.net/projects/captainfeature/, last visited on August 2009

[37] FeatureIDE, http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/, last visited on Agust 2009

[38] S. Hallsteinsen, M. Hinchey, Sooyong Park, K. Schmid, "Dynamic Software Product Lines", IEEE Computer Society Computer Magazine Volume: 41 Issue: 4, pp. 93-95, 2008

[39] Andrew Hunt, David Thomas, "The Pragmatic Programmer: From Journeyman to Master", Addison-Wesley Professional, 1999

[40] Hong Yul Yang, Ewan Tempero, Hayden Melton, “An Empirical Study into Use of Dependency Injection in Java”, 19th Australian Conference on Software Engineering, 2008

[41] Inversion of Control Containers and the Dependency Injection pattern, http://martinfowler.com/articles/injection.html, last visited on August 2009

[42] Graphical Modeling Framework, http://www.eclipse.org/modeling/gmf/, last visited on August 2009

[43] The Epsilon Book, http://www.eclipse.org/gmt/epsilon/doc/book/, last visited on August 2009

[44] L. M. Rose, R. F. Paige, D. S. Kolovos, F. A. Polack, "The Epsilon Generation Language", Lecture Notes In Computer Science; Vol. 5095, Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications, Berlin, pp. 1-16, 2008

[45] Spring.NET, http://www.springframework.net, last visited on August 2009.

[46] Spring Source, http://www.springsource.org, last visited on August 2009.

[47] Aselsan Inc., http://www.aselsan.com.tr, last visited on August 2009.