# HIERARCHICAL VARIABILITY MANAGEMENT IN SOFTWARE PRODUCT LINES

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

BY

MEHMET EMRE ATASOY

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

OCTOBER 2013

Approval of the thesis:

**HIERACHICAL VARIABILITY MANAGEMENT IN SOFTWARE PRODUCT LINES**

Submitted by **MEHMET EMRE ATASOY** in partial fulfillment of the requirement 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. Adnan Yazıcı _____
Head of Department, **Computer Engineering Dept., METU**

Prof. Dr. Ali Hikmet Doğru _____
Supervisor, **Computer Engineering Dept., METU**

**Examining Committee Members:**

Assoc. Prof. Dr. Ahmet Coşar _____
Computer Engineering Dept., METU

Prof. Dr. Ali Hikmet Doğru _____
Computer Engineering Dept., METU

Assoc Prof. Dr.Pınar Karagöz _____
Computer Engineering Dept., METU

Halil Kolsuz (M.Sc) _____
Lead Design Engineer, Aselsan Inc.

Mert Burkay Çöteli (M.Sc) _____
Expert 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:** MEHMET EMRE ATASOY

**Signature :**

## ABSTRACT

### HIERACHICAL VARIABILITY MANAGEMENT IN SOFTWARE PRODUCT LINES

ATASOY, Mehmet Emre

M. Sc., Department of Computer Engineering
Supervisor: Prof. Dr. Ali Hikmet DOĞRU

October 2013, 53 pages

Software product lines (SPL) aim is to analyze commonality and variability of product family although SPLE describes much kind of processes in different abstraction levels. In this respect, numbers of variations are increasing for the types of products so that may result in increasing cost of the managing variability process. So that variability models is used to manage variabilities in software product lines. Representing solution space variability in an understandable way in software product line engineering is an important challenge. In this thesis, a new technique is offered to configure variabilities leading to hierarchical structure. The main issue of this approach is to divide variability model into two layers which are system engineering level variability and software engineering level variability. The new models subtract a balance between formalism's expressiveness and specific configurations of application. The products are configured by merging these variabilities which are defined in different layers. Dependencies between these two layers can be managed semi automatically using Case tools which are developed in this work.



**Keywords:** Software Product Line, Variability Management, Hierarchical Variability Management, Domain Variability, Application Variability

# ÖZ

## YAZILIM ÜRÜN HATTINDA AŞAMALI YETENEK YÖNETİMİ

ATASOY, Mehmet Emre

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Ali Hikmet Doğru

Ekim 2013, 53 Sayfa

Yazılım ürün hattı her ne kadar farklı soyutlama düzeylerinde birden çok süreç tanımlasa da amacı ürün ailesinin ortak ve değişkenliğini analiz etmektir. Bu bağlamda değişkenlik gösteren yazılımların artan varyasyonları yönetilmelerini zorlaştırmaktadır. Bu sebeple değişkenlik modelleri yazılım ürün hattında değişkenlik yönetimi için kullanılmaktadır. Bu tez çalışmasında, değişkenlikleri yönetmek için yeni bir yaklaşım olarak aşamalı değişkenlik yönetimi sunulmaktadır. Bu yaklaşımın asıl amacı, sistem ve yazılım mühendisliği seviyesi olmak üzere değişkenlik modelini iki aşamaya bölmektir. Yeni model biçimsel anlaşılırlık ve uygulamaya özel yapılandırmaya dengeli bir biçimde ayırır. Yazılım ürün hattından çıkacak ürünler bu iki katmanın birleştirilmesi sonucu ayarlanmasıyla ortaya çıkar. Bu iki katman arasındaki bağımlılıklar çalışma kapsamında geliştirilen bilgisayar destekli yazılım mühendisliği araçları (CASE) ile yarı otomatik olarak yönetilebilir.

**Anahtar Kelimeler:** Yazılım Ürün Hattı, Değişkenlik Yönetimi, Aşamalı Değişkenlik Yönetimi, Alan Değişkenliği, Uygulama Değişkenliği

*To My Parents and To My Sister*

**ACKNOWLEDGEMENTS**

I like to express my special gratitude to my supervisor Assoc. Prof. Dr. Ali Hikmet Doğru for his understanding, patience and supervision throughout this thesis. This thesis would not have been completed without his realistic, encouraging and constructive guidance.

I would like to thank to all my friends and colleagues for their understanding and continuous support during my thesis. Special thanks to my work team named as TADES Software Development.

Special thanks to Mert Burkay Çöteli, Halil Kolsuz, Necip Gürler and Orçun Dayıbaş.

I would like to thank to TÜBİTAK for the scholarship throughout the thesis.

Finally, I would like to thank my family - my sister, my mother and my father - for their love, trust, understanding and every kind of support not only throughout my thesis but also throughout my life.

**TABLE OF CONTENTS**

## LIST OF TABLES

## LIST OF FIGURES

FIGURES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| AD | Application Design |
| AE | Application Engineering |
| CASE | Computer Aided Software Engineering |
| COTS | Commercial off-the-shelf |
| CSCI | Computer Software Configuration Item |
| DE | Domain Engineering |
| DSL | Domain Specific Language |
| FB | Feature Based |
| IDE | Integrated Development Environment |
| METU | Middle East Technical University |
| MVC | Model-View-Controller |
| M2M | Model to Model |
| M2T | Model to Text |
| PDF | Probability Density Function |
| PV | Pure Variant |
| SC | Software Component |
| SP | Software Product |
| SPL | Software Product Line |
| SPLE | Software Product Line Engineering |
| SRS | Software Requirement Specification |
| TADES | Teknik Ateş Destek Sistemleri |
| UML | Unified Modeling Language |
| VP | Variation Point |
| XML | Extendible Markup Language |

# CHAPTER 1

# INTRODUCTION

In the industry, there are two types of software: tailor made software (custom) and COTS. Custom software is specially developed for some specific organization or people requirements. On the other hand, commercial off-the-shelf software meets the requirements of many customers. COTS software is developed for the mass market. A custom software development cost is high and in the contrast COTS software meets customer needs hardly. Because of the fact that customers' demands change from one person to another, companies have to provide different kinds of products. For instance, customers may want to browse on the internet and read newspapers. Thus, they do not want to pay money for unused product issues. This mass customization has to be satisfied by the industry [1]. On the other hand, making custom software more economic and making standard software more customized is important for software market. Due to the cost of individualized products, platform based development and mass customization have to be. In addition, features depending on the customer needs have to individualize the software products. Software product line (SPL) is one of the solution for this manner [2,3].

What is software a Software Product Line? Paul Clements et al. defines SPL as follows: "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"[2]. Let us retrace the important parts of this term in order to understand better. "Set of software-intensive systems" means that is managed set of features which satisfy the needs of specific product segment. It remarks more than a methodology for a single software project. It covers a family of software systems and manages relations between them. On the other hand, "managed set of features" means that common features among the product family are feature-based and managed in SPL concept. "Particular market" means that SPL is defined for market. This aspect is defined as main difference from "system family" concept [3]. "Common set of core assets" means that developed products using SPL share common building parts named as assets. These assets' adaptability is managed by SPL.

What Software Product Lines Are Not? Software product line could be mixed up with many approaches at first sight. These approaches may be any of the following[2]:

- Fortuitous Small-Grained Reuse [2]: It is not new idea and SPL absolutely involves reuse. In shortly, in SPL the reuse is made in planned way and profitable. Due to reusing all assets more than one system, they are optimized and designed accordingly.

- Single-System Development with Reuse [2]: The new system that seems similar to old one could be developed and many assets can be used from old system by borrowing and modifying as it necessary. This technique seems that it gives economic advantage of developing new system. But there are two different kinds of systems. SPL has major differences from this approach. First of all, SPL use assets which are developed explicitly for reuse.

- Component Based Development [2]: The definition is component based development is the selection of components from library or market in order to create products. SPL definitely involves component based development. In addition to component based development, SPL assembly the components in a prescribed way using well defined architecture.

- Reconfigurable Architecture [2]: Frameworks and reference architectures are developed to be reused many systems and reconfigured as requisite. But SPL architecture deals with the variation needed by the products in product line.

- Releases and Versions of Single Products [2]: Releases and versions are usually generated by software teams. These new versions use same architecture with old ones. They are tested and documented as older releases. In this concept, older versions don't have market potential. However, in SPL all versions of product have market potential and must be kept as a valuable member of family.

- Set of Technical Standards [2]: Technical standards and development limitations are defined for software engineers' choices by many organizations. For instance, software engineer must select between two log infrastructure. These standards must be defined for product family not for organization.

SPL offers techniques in order to reduce cost of development using reuse and variability management. Software product line engineering concerns reusability of software components mainly. There are three parts of engineering process in SPLE: Scope Engineering, Domain Engineering and Application Engineering. In Scope Engineering domain is analyzed and boundaries of product family is drawn[5]. In Domain Engineering phase common components are developed. These common components are used in Application Engineering phase.

Declaration about SPL is given chapter 2 so no more information is given for introduction chapter. In this thesis work, a new technique is offered in order to apply mentioned techniques of SPLE in a simple way.

In this thesis, feature modeling and variability management of TADES (it is the name of product line project in ASELSAN) developed in ASELSAN will be discussed. In this point, there is benefit to mention about stakeholders in ASELSAN. There are four major stakeholders that works during software production activities: project management team, system engineering team, software engineering team and test engineering team. Employers who work as a system engineer are responsible to decide the product developed by using product line and choose high-level features of this product. On the other hand, software engineers gather features from system engineers and comprise these features software-related features which aren't known by system engineer. These features lay on software-level. Software engineers comprise these two types of features and configure their applications. There isn't any feature model used in TADES at the moment. In literature, organizations use single feature model in order to manage variability in product line. Due to level of requirements difference between software engineers and system engineers and avoiding tiring system engineers with software features, a new technique will be introduced that differs feature modeling into two layers in this thesis. In contradistinction to literature, there will be described two different feature models. However, there will be necessity in order to merge system level features and software level features. This merge

activity can be manually or automatically. In addition, there will be given a different technique in order to manage code-level features which are mentioned in literature as using component reuse technique.

This thesis includes following chapters: Chapter 2 gives information about SPLE on literature. Information about variability management and feature modeling is given Chapter 3. These descriptions also present my approximation in this thesis. Chapter 4 specifies variability management, not as the classical method in literature entails. This chapter, information about SPL architecture in TADES will be given. Hierarchical variability management solution for TADES will be given in this chapter. Chapter 5 includes a review of the documented work and conclusion of the document.

# CHAPTER 2

# SOFTWARE PRODUCT LINE ENGINEERING

## 2.1 What is the Software Reuse?

Software reuse, also called code reuse. It is the meaning of using the existing software in order to build new one [6]. From the early years of programming, developers have always reused parts of codes, functions, components. Software reuse is a part of study in software engineering. Software reuse is the term that a part of program developed before is being used in another program which will be developed in future. It gives advantage to developers because it saves time and decrease cost by reducing nonessential work.

Software reuse isn't a new concept. There are many articles about software reuse in literature. The article defines software reuse as below[7]. Next paragraph is little part of the abstract:
*Software reuse is the process of creating software systems from existing software rather than building software systems from scratch. This simple yet powerful vision was introduced in 1968.Software reuse has, however, failed to become a standard software engineering practice. In an attempt to understand why, researchers have renewed their interest in software reuse and in the obstacles to implementing it.*

One of the examples of software reuse is the software libraries. Software library is created and developed by the decision of the programmer and used many times. Software library is the most common example of software reuse. Software library has many advantages like being well-test, implementing unusual cases and qualified. On the other hand, it has many disadvantages because it has inability to unveil the performance efficiency of software and it has learning and sustaining cost [8].

In software engineering another example of software reuse is design patterns. A design pattern is a general reusable solution to recurring problem in software design. Patterns consists best design experiences that the programmer should implement them in the software design phase [9].

Another example of software reuse is frameworks. Pieces of software are used by software programmers using frameworks. However, software frameworks are usually related to specific domain.

Last but the most important example of software reuse is systematic software reuse. This technique increases the productivity and improves quality of the software. Though it seems simple in notion, implementing successful is hard in practice. A reason for this difficulty is the context dependency of software reuse. These issues which are problematical are related to systematic software reuse[9]:

- Well-specified product vision is a necessary base for an SPL
- As a strategy for companies an evolutionary implementation should be selected
- SPL needs full management support and leaders who are sure for success.
- SPL needs suitable organizations

Reuse in software is developing concept. In history, functions are used as a reuse. Because of high prices of memory, callable procedures are used to save memory [11]. Using code parts is overly intensive and managing this kind of reuse is hard. With evolution of software engineering, abstraction enhanced and thin-grained reuse replaced with large-grained ones. Separately, objects, frameworks and domain models occur to handle reuse [11].

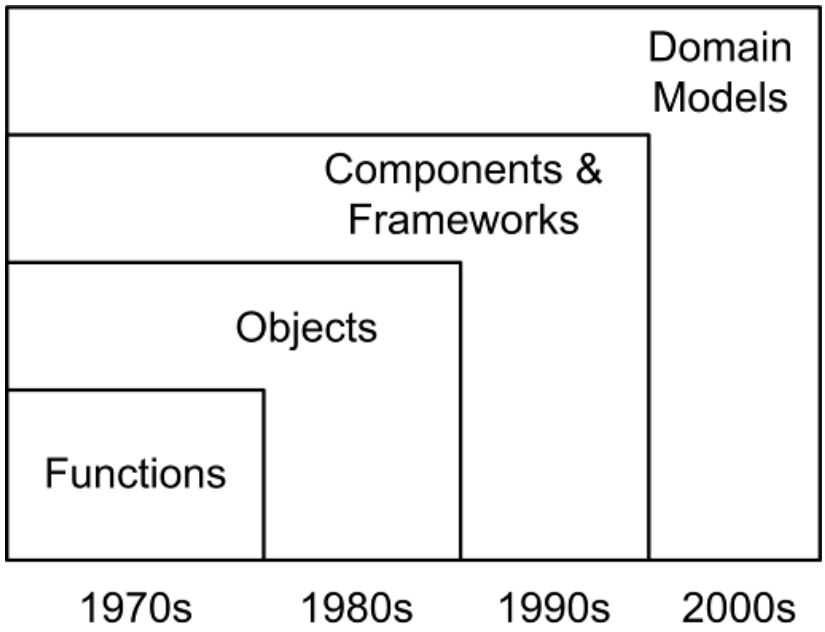


Figure 2.1 – Evolution of software reuse [2]

The chronological evolution of reuse in software development is seen on Figure 2.1. High level of abstraction usage brings many problems together. Most important problem of using abstraction level is customization.

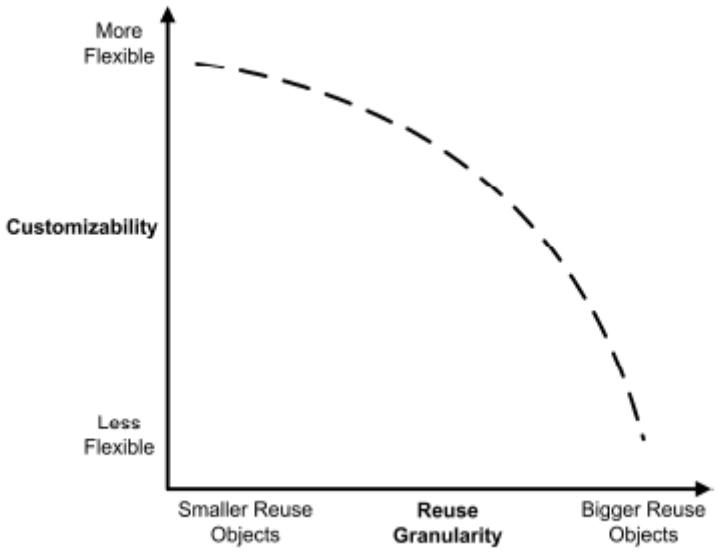


Figure 2.2 – Customization & Reusability

Software elements that consist a single well-described object, function or components of an application may be reused [12]. Using object reuse is less customizable than function reuse. Accordingly, object reuse is more customizable than component reuse. Obviously, it is required that before using software reuse detailed analysis must be evaluated to be succeeded. Ian Sommerville claims that software reuse has many advantages and disadvantages:

Advantages [12]:
1) Increased dependability
2) Reduced process risk
3) Effective use of specialists
4) Standards compliance
5) Accelerated development

Disadvantages [12]:
1) Increased maintenance costs
2) Lack of tool support
3) Not-invented-here syndrome
4) Creating and maintaining a component library
5) Finding, understanding and adapting reusable components

According to Wayne C. Lim, reuse types are categorized in two different classes: technical reuse and non-technical reuse [13]. In this study I will focus on technical reuse techniques. By the way, next section consists of some guidance software reuse techniques in literature and they are mainly related to technical reuse techniques according to defined classification.

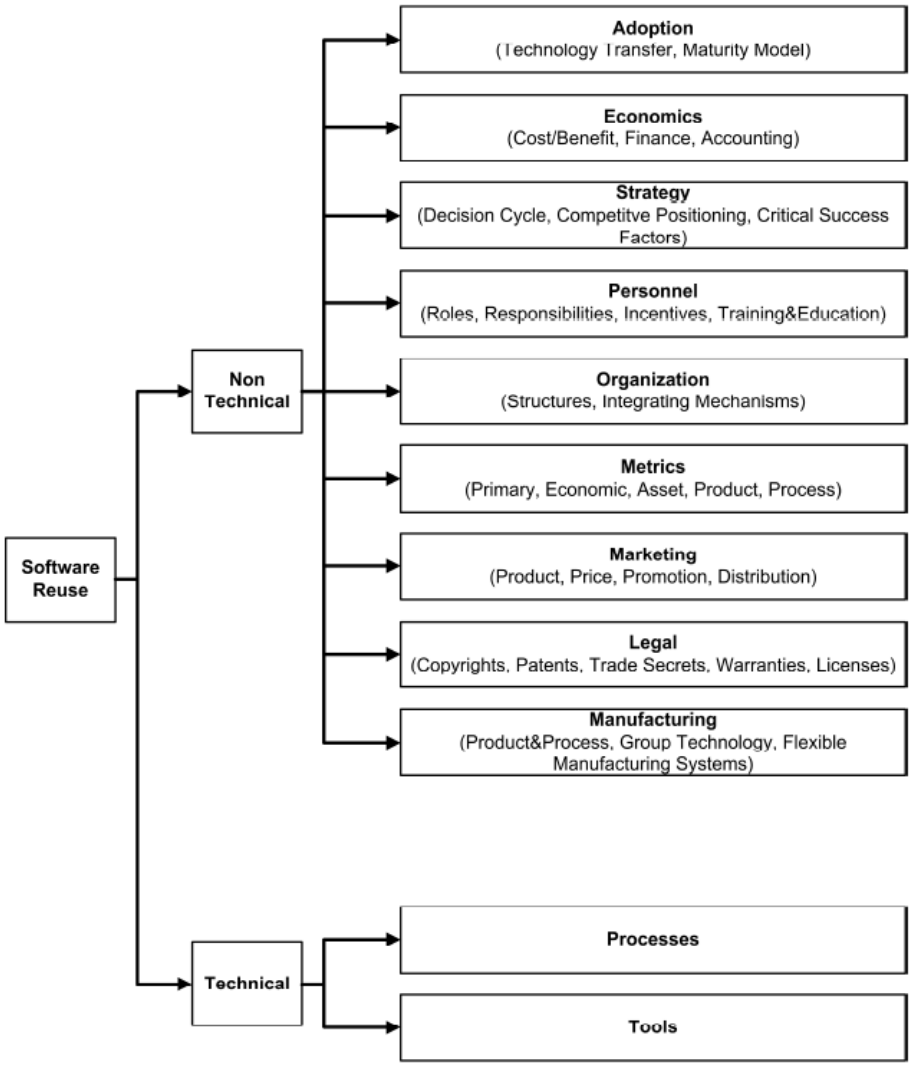


Figure 2.3 – Classification of Software Reuse [13]

## 2.2 Domain Engineering Overview

Domain Engineering is the process in order to produce new software systems by reusing domain knowledge. Many companies produce applications in only limited domains. They are continuously likely systems in variations of same domain in order to satisfy customer requirements. Rather than producing new systems, they are using managed components of previous systems in the domain in order to produce new one.

The main purpose of domain engineering is to enhance the quality of software products owing to reuse of software components [6]. Domain engineering claim that most of developed software products are not new products, whereas they are the variant of older ones in the same domain [6].

Domain engineering is possessed of three fundamental phases: domain analysis, domain

design and domain implementation [16]. In these phases reusable components and configurable requirements are developed[17].

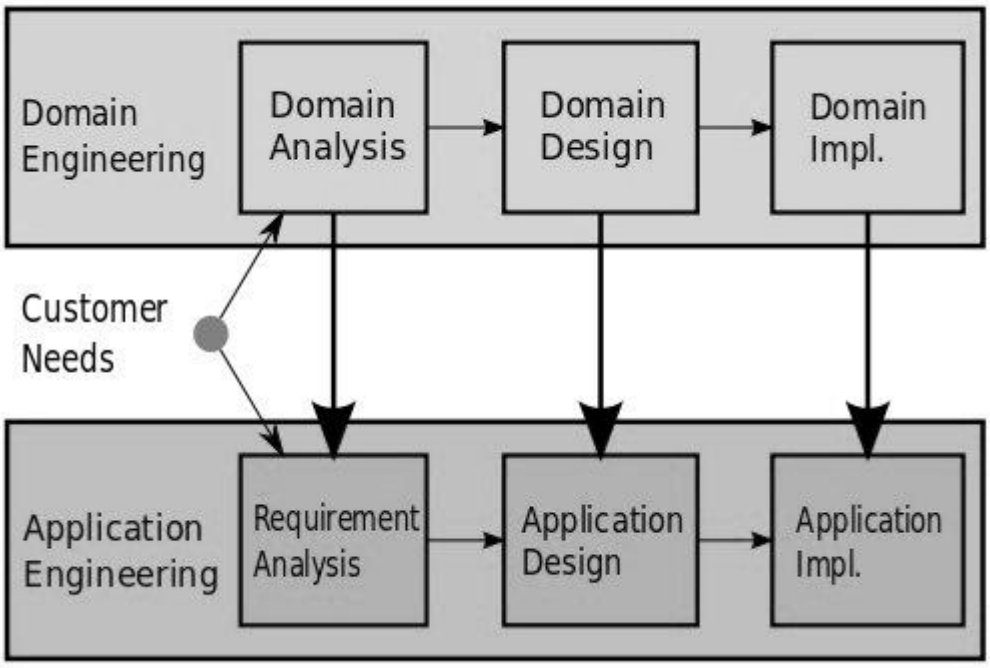


Figure 2.4 – Domain Engineering Phases

Domain engineering is one of the most advanced efforts in software reuse and it is still used nowadays. Following sections contain domain engineering examples and methods.

### 2.3 ODM (Organization Domain Modeling)

ODM has been developed by Mark Simons to systemize domain analyses method in terms of a core domain modeling life cycle for STARS (Software Technology For Adaptable, Reliable Systems) program [15].Early years, STARS developed RLF (Reuse Library Framework) which originated ODM and used as a domain modeling tool for software reuse[14].

Because there was a gap in the domain engineering area, ODM has been developed. Domain analysis was a mainly technical modeling problem in domain engineering area. Small-scale domain engineering projects were managed by people negotiating the side of issues in planning and managing their work. But they didn't have full support of already defined infrastructure in organization. ODM method attempts to meet this need and initializes pilot projects into developing reuse programs [14].

ODM has been applied many kind of companies. Some of these are: Lockheed Martin Corporation, Hewlett-Packard (HP) , Logicon Corporation (on behalf of the U.S. Navy Program Executive Office for Cruise Missiles and Unmanned Aerial Vehicles) and the Rolls-Royce University Technology Centre[14].

ODM claims that there is a complicated relation between other projects, stakeholders and economical purposes of companies. By analyzing domain, these relations must be analyzed clearly according to ODM.
ODM process model is organized hierarchically. ODM describes process life cycle in three phases (as shown in Figure 2.5): Plan Domain, Model Domain and Engineer Asset Base.

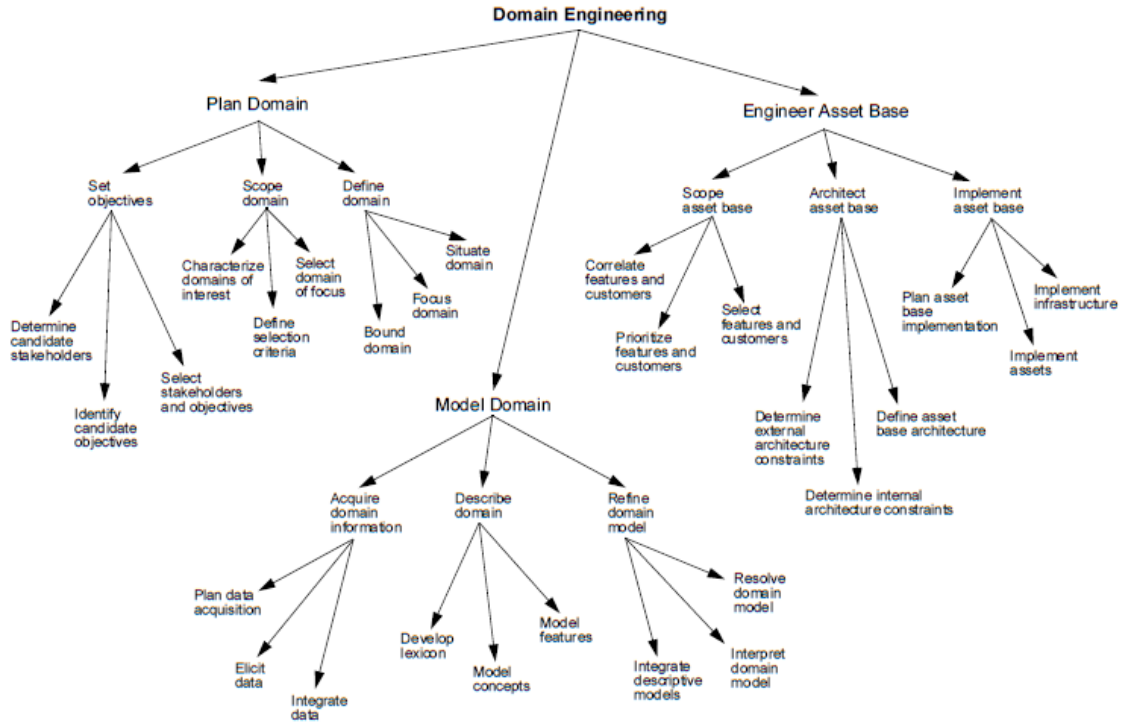


Figure 2.5 – Process Tree [14]

The main goal of plan domain phase is to define objectives and scope of a domain through organizational needs. Scoping a domain is very important for the process and is usually constituted beside requirements. Moreover, defining a scope of domain required to think about multiple application contexts. Another key task of this phase is acquiring commitment of other stakeholders. In a model domain phase, domain model is developed for the selected domain. Common and variant features are defined in this phase. Engineering asset base phase consists the process of implementation according to defined domain. In this base, architecture is defined and implementation of this architecture developed.

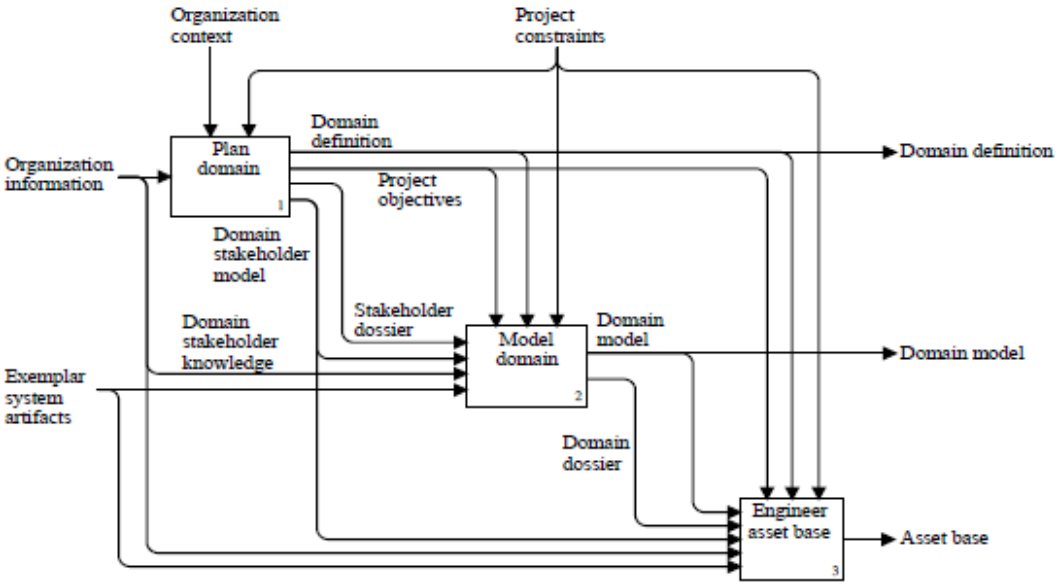


Figure 2.6 - Domain Engineering Phases of ODM

ODM is enough comprehensive to cover the variability and can be used in specific company's needs. It integrates organizational and technical issues of DE.

### 2.4 FODA (Feature Oriented Domain Analysis)

Feature Oriented Domain Analysis (FODA) method introduced feature concept of commonality and variability between products in a product line. FODA sets a method in order to perform a domain analysis and defines the products of domain analysis process. The main goal of FODA is to perform a domain analysis [19].

According to FODA, huge and complicated systems required an obvious comprehension of requested system features and the capabilities of system in order to implement these features. Domain Analysis is the investigation of software systems to define commonality, features and abilities of related software systems [19].

FODA describes domain analyses in three basic phases:

1) Context analysis: This phase consists of describing the scope of domain. The extent of a domain for analysis is defined by domain analyst in this phase.
2) Domain modeling: Domain analyst creates a domain model by using information and products of the context analysis phase.
3) Architecture modeling: By using the outputs from preceding step domain analyst creates produces architecture models of domain. Software engineers, domain experts and requirements experts should review this model.

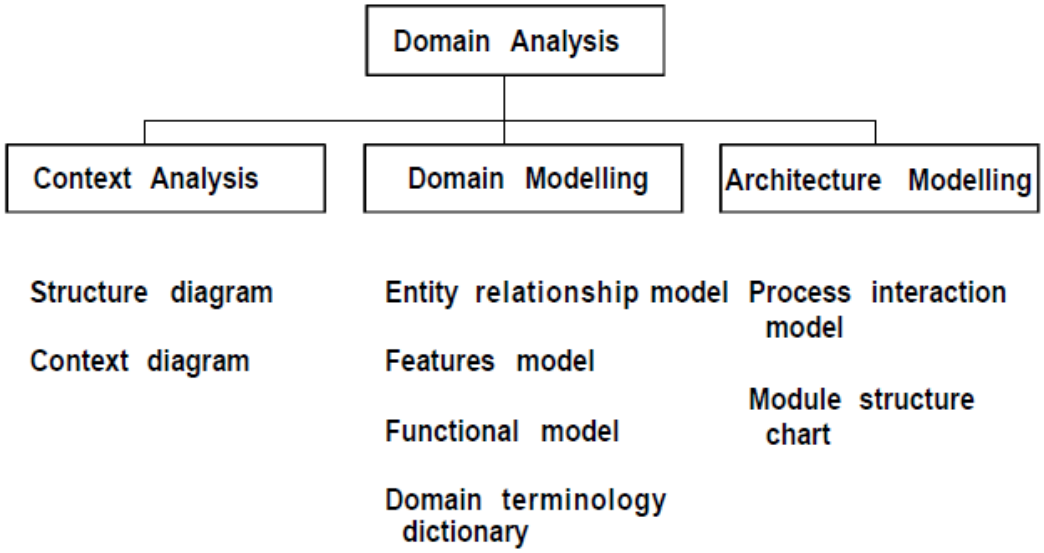


Figure 2.7 – Products of FODA Domain Analyses' Phases

Context analyses have outputs which are context and structure diagrams. Structure diagrams and data-flow diagrams are defined in a context model. If the target domain is related to higher, lower and peer-level domains, a structure and context diagram of context model are defined as an informal block diagram by Kang [19].

One of outputs of domain modeling phase is feature model. Feature model consists of features which are the attributes of a system. These features affect directly end-users. Therefore feature model is an abstraction level which is created by using system requirements. A simple example of feature model is given in Figure-2.8[19].

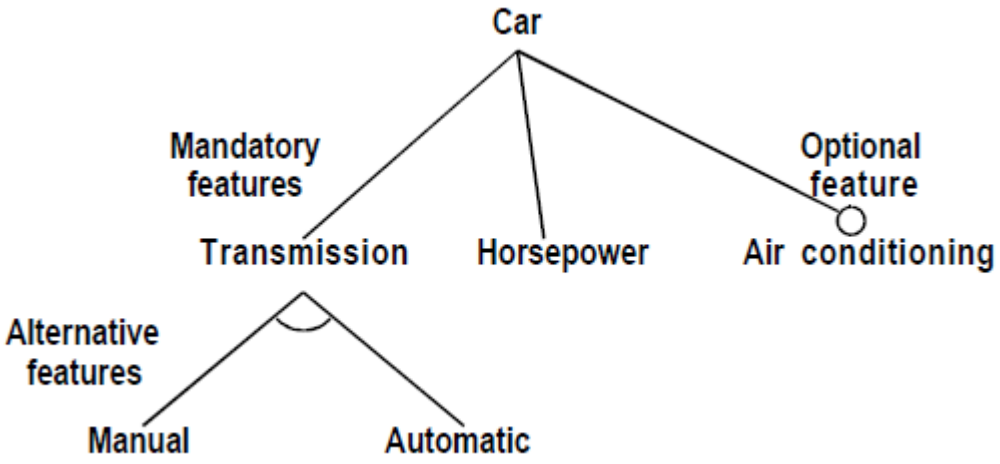


Figure2. 8 - Features of a Car [19]

Implementation activities start in architecture modeling phase by using outputs of domain modeling phase. Analyst maps a domain model to architectural model.

Architecture model consists of a high-level design of the applications in a domain. An architectural model represents domain model as a view of developer. While creating architectural model, possible changes occurring after the problems and technology are considered. At the end of this phase, domain analysis produces information about reusable assets which will be used in development [19].

### 2.5 FORM (Feature-Oriented Reuse Method)

FORM (Feature-Oriented Reuse Method) describes a systematic method which seeks and manages commonalities and differences of products in a domain in terms of features and using the analysis result to build up domain architectures and components[20]. FORM is extended from FODA which is defined before. FORM includes design and implementation activities in addition to FODA. FORM describes how the feature model is used to produce domain architectures and components for reuse.

FORM engineering process consists of two engineering processes: domain engineering and application engineering.

Domain engineering process includes activities in order to analyze systems, point out commonalities, and develop reference architecture and reusable assets from analysis results [20]. Domain engineering process' outputs are given to application engineering process as shown in Figure 2.9.

Application engineering process contains works for producing applications using the inputs produced in the domain engineering process. Requirement analysis and application development are done in this phase. It shouldn't be forgotten that the reference architecture and reusable artifacts are assumed to consist the differences in addition to commonalities of the systems in domain [20].

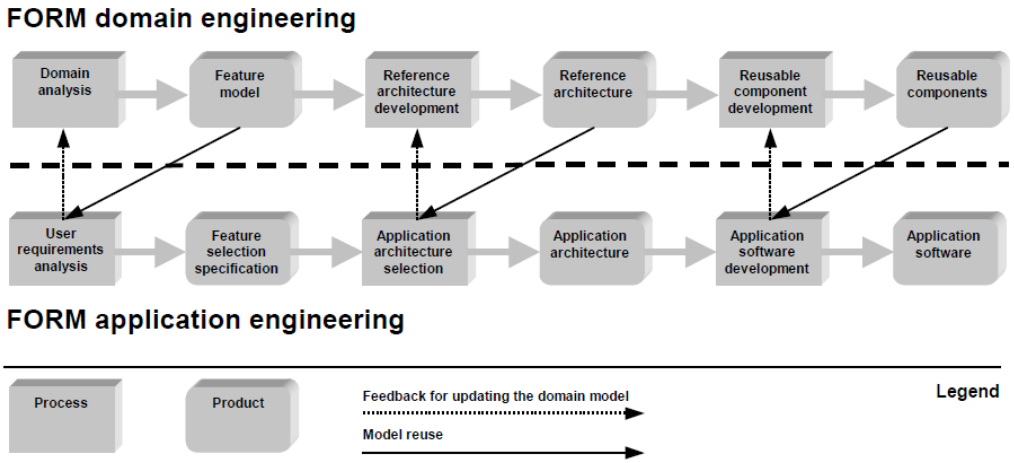


Figure 2.9 – FORM engineering process

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

FAST (Family-Oriented Abstraction, Specification, and Translation) is a development activity for producing software in a family-oriented way. There are two-main parts according to FAST in product line process. First step provides core assets consisting of the environment for developing each product. Second step consists of producing other assets which are used to other products belonging to the product family [21].

FAST process can be defined and used in a prescribed way that called PASTA (Process and Artifact State Transition Abstraction) model. PASTA model defines rules to obey during FAST process. PASTA describes instructions to follow. However, it has support to make individual choices during FAST process. Its aim is to make software reuse easier [21].

FAST process is divided to sub-process:

1) Qualify domain: Exposing and identifying families worthy of investment. An economic perspective used to analyze software family.

2) Domain engineering: Making investigation in order to produce products as a family member. This process defines which parts of the products are common and which parts of products differ from each other.

3) Application engineering: Products defined from family are produced quickly [21].

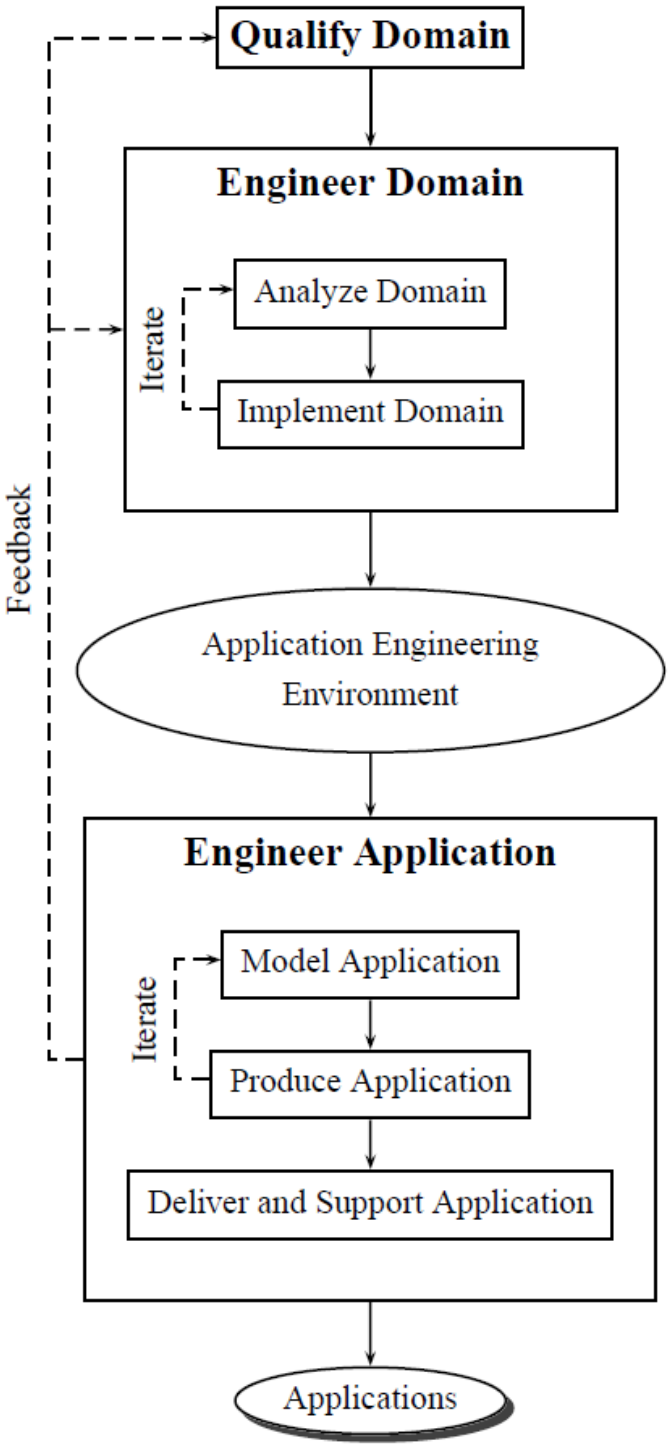


Figure 10 – FAST process pattern [21]

### 2.7 RSEB (Reuse-driven Software Engineering Business)

RSEB is a software engineering method based on object reuse. This method was defined for simplifying development of reusable software artifacts. RSEB's main intention is based on variability modeling and managing traceability between all links connecting representation of variability models. Analysis, design and implementation variability models are defined in RSEB. RSEB is an iterative and use-case-centric method as Unified [22].

RSEB has divided engineering methods for reuse into two phases: "Domain Engineering" and "Application Engineering". In addition to this separation, RSEB defines "Domain Engineering" in two sub-processes which are "Application Family Engineering" and "Component System Engineering".

1) Application family engineering: This process produce layered architecture. This process contains steps : analyzing requirements, robustness analyses, design, implementation and test.

2) Component system engineering: This process focus on developing systems of reusable components. This process contains steps: capturing requirements taking into consideration variability, robustness analyses, design, implementation, test, and packaging.

Another definition of application engineering in RSEB is "Application System Engineering". This process includes implementation of system.

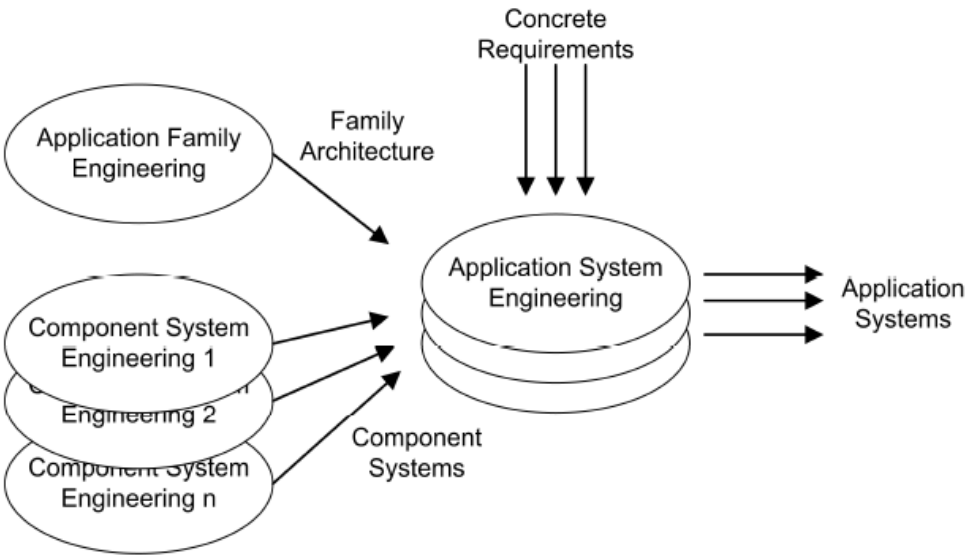


Figure 11 – RSEB process

As mention above, key ideas in RSEB are based on modeling variability by using extended UML notation [22]. RSEB defines variation points which are implemented in separated variability mechanisms.

| 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 |

Figure 2.12 – Example of Variability Mechanisms in RSEB [22]

Although RSEB emphasis variability, it doesn't consists domain scoping and feature modeling. Moreover, RSEB doesn't define a method to develop an asset and it doesn't have feature models as needed. Variability is defined at the highest level in the form of variation points, which are then implemented in other models using variability mechanisms [22].

RSEB is found inefficient in practice by Griss so that they have also represented improved method named as FeatuRSEB to overcome insufficient points of RSEB [25].

### 2.8 About Software Product Line

As i mentioned in previous sections, the reuse in software engineering is related to domain engineering process until 1998[26]. Afterwards, SPL engineering has been presented. SPL produces products with a mass customization which is the large-scale development of products according to individual customers' requirements. The software product line engineering paradigm is separated into two processes: production for reuse and production with reuse. In other words, they are domain engineering and application engineering respectively [27].

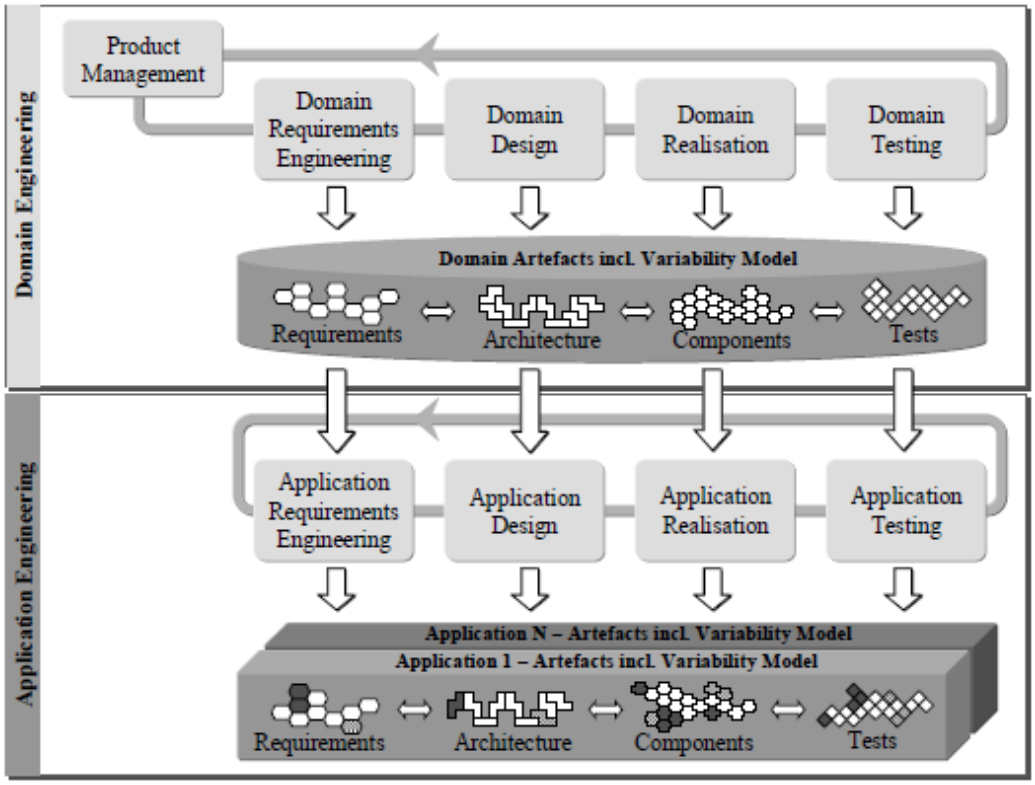


Figure 2.13 – SPL Engineering Framework [27]

First of all, domain engineering process intents to define commonality and variability. Secondly, domain engineering process' goal is to define set of applications for which SPL is planned. Another goal is to define reusable components. As figure 2.13 shown, domain engineering's starting point is the product management.

Economical manners of SPL and market strategy are considered in product management phase. Its main goal is to manage product family of the organization. Product management uses the company goals which are defined by senior management as an input and it produces a product roadmap which consists of common and variable features using this input [27].

The common and variable requirements of product family are elected and documented in the domain requirements engineering phase. This phase uses product roadmap that is produced in product management phase as an input and produces reusable requirements which are textual and model-based and especially the variability model of product family as an output [27]. Requirements engineering in domain requirements engineering is different from requirements engineering for individual systems [27]:

- The requirements are defined without thinking commonality in single systems in contrast domain requirements engineering.

- Variability model which is an abstraction of variability of the domain requirements isn't defined in requirements engineering for single systems.

- Domain requirements engineering expects changes in requirements for future applications. Laws, standards, technology and market needs may result in changes in software development.

The reference architecture which provides a common and high level design structure for all product line applications is defined in the domain design sub process. Domain design uses domain requirements and variability model as input and produces reference architecture and refactored variability model which consists internal variabilities. Design for individual applications is different from domain design [27]:

- Configuration mechanisms are integrated in order to manage variability into the reference architecture.

- Reference architectures' design in domain design has flexibility in order to overcome future changes.

- Common rules and standards are defined in domain design in order to develop particular applications.

- Reusabilty is considered at the component level domain design in order to be developed and tested in application engineering phase.

The detailed-design and the implementation of reusable software components are made in the domain realization phase. The domain realization phase uses reference architecture as an input and produces detailed design and implementation assets as an output. Domain realization is different from the realization of individual systems [27]:

- Domain realization includes loosely coupled and configurable assets. In addition, it doesn't consist of a running application.

- Each components and interfaces are produced by considering reusability. Different applications are supported for reuse.

- Configuration mechanisms are defined in domain realization in order to manage variability of SPL.

Validation and verification of reusable components are made in domain testing phase. The components produced in domain realization are tested according to requirements, architecture and design artifacts in this phase. Domain testing includes producing reusable testing artifacts to decrease the cost of application testing. Domain testing uses requirements, reference architecture, components design and reusable component as an input and it produces test results of reusable components. Domain testing is different from individual system testing [27]:

- Domain testing doesn't test executable components or running applications. Actually, these executable components and running applications are defined by product management. However, these components are tested in application testing phase.

- Domain testing approaches different testing strategies in order to test integrated

components which have variable parts.

Until this point of this section, domain engineering sub-processes is described. Another process of SPL is application engineering and it has also sub-processes. Application engineering intends to reuse as much as possible of the domain assets when producing an application from product line. Application engineering process uses domain engineering process' outputs as an input and produces applications. Application engineering has also sub-processes like domain engineering [27].

The application requirements specification activities are made in application requirements engineering sub-process phase. This phase combines the domain requirements with the main features of application in order to use as an input. The requirements specification for specific product is produced. Requirements engineering in application requirements engineering is different from requirements engineering for individual systems [27]:

- Most of application requirements aren't defined anew, but are inherited from domain requirements in application requirements engineering.

- During definition of application requirements, detection of differences deltas between application requirements and domain requirements are made and analyzed in order to decrease the effort and to improve the amount of domain component reuse.

The application architecture is produced in application design (AD) phase which uses reference architecture in order to create application architecture. Application design sub-process uses reference architecture and application requirements as an input and produces application architecture as an output. Design for individual applications is different from application design [27]:

- Application design is based on reference architecture and doesn't develop the application architecture in a random way.

- Application design must obey the rules described in the reference architecture.

- Application design considers the implementation effort for each requirement and may not accept modifications that would require similar effort as for developing the application in a random way.

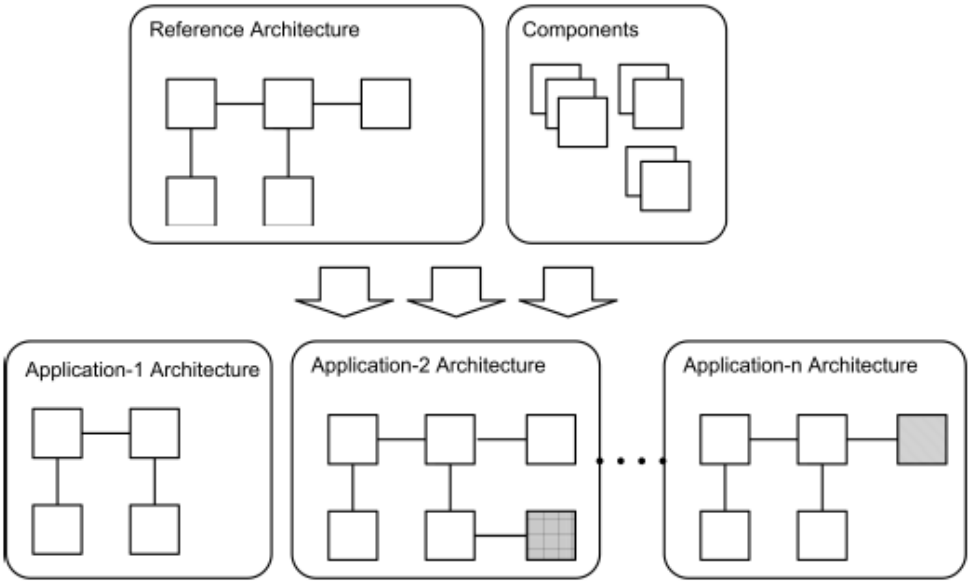


Figure 2.14 – AD sub-process

The considered application is implemented in the application realization sub-process phase. The decision of which software components are used and how these components are configured is made in this phase. This sub-process uses the application architecture and reusable assets as an input and produces a running application. Application realization is different from the realization of individual systems [27]:

- Many components aren't implemented anew because they are derived from binding variability.

- Application realization must obey the domain realization structure in order to reuse components and interfaces. Application specific components are variants of artifacts which are developed domain realization.

Validation and verification of application specific components are made in application testing sub-process phase. Test artifacts from domain testing and the implemented application is used as an input for this sub-process. On the other hand, test reports for the application components are produced as an output in this phase. Application testing is different from individual system testing [27]:

- Many test components aren't implemented anew because they are derived from binding variability.

- Additional tests must be defined in order to find out configuration bugs and to be sure that the variant bound is combined correctly.

- Application testing must consider the reused common and variable components of the application as well as newly implemented application-specific parts in order to analyze test coverage.

Next sections, the examples of software product line methods and the information about them will be given:

- PuLSE (Product Line Software Engineering)
- KobrA
- CoPAM (Component-oriented Platform Architecting Method)
- PLUS (Product Line UML Based Software Engineering)

### 2.9 PuLSE

The PuLSE is developed by the Faunhofer Institute for Experimental Software Engineering (IESE) in Germany in order to overcome problems of methods for managing product lines based on domain engineering [28]. The PuLSE method consists of three main elements which are the deployment phases, the technical components, and the support components.

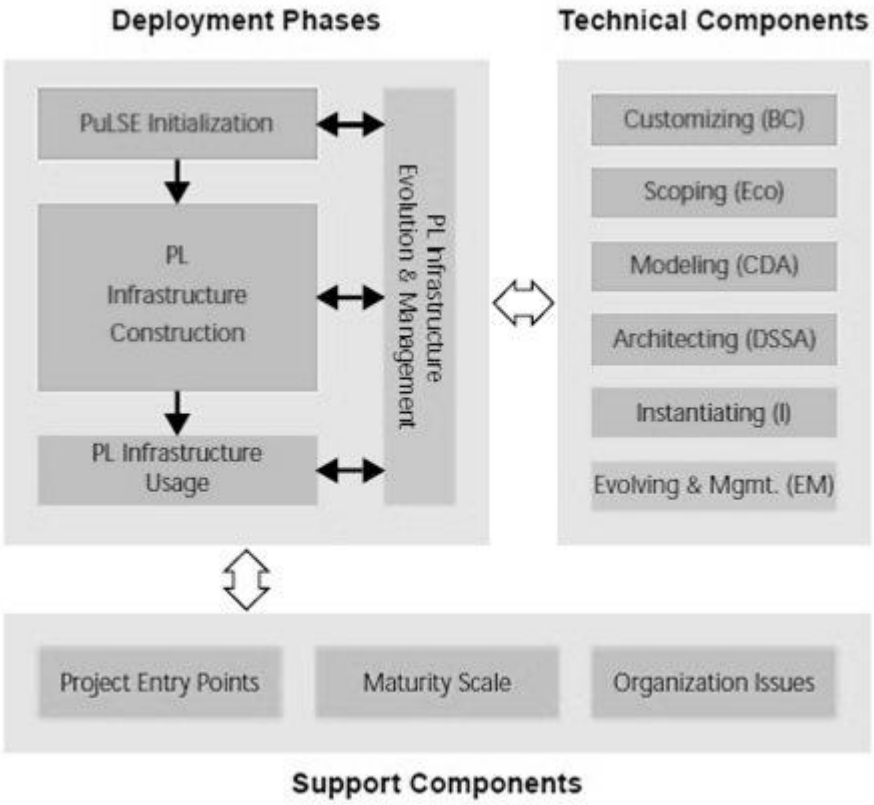


Figure 2.15 – PuLSE Overview [28]

The deployment phases are considered as logical stages which a SPL introduced. There are four deployment phases in PuLSE :

- Initialization: creation of approach and configuring PuLSE

- Infrastructure Construction: scoping boundary , modeling and defining architect of the product line structure

- Infrastructure Usage: usage of the infrastructure in order to develop PL members.

- Evolution and Management: evolving the infrastructure in course of time and managing it

Technical components are providers of technical knowledge in order to operate the PL development. These components help customizing, scoping, modeling, architecting, evolve and manage.

The support components consist of packages which are guidelines during development of SPL. These components are project entry points, maturity scale and organization issues.

PuLSE is a customizable method in order to develop SPL. Moreover, it has advantages because of being well-documented and having tool supports.

### 2.10 KobrA

KobrA is defined by customizing the PuLSE process. The Kobra consists of improved software technologies, PL development, component based software development, frameworks, architecture-centric inspections, quality modeling, and process modeling. KobrA describes the framework and application engineering activities using UML which brings an advantage together [29].

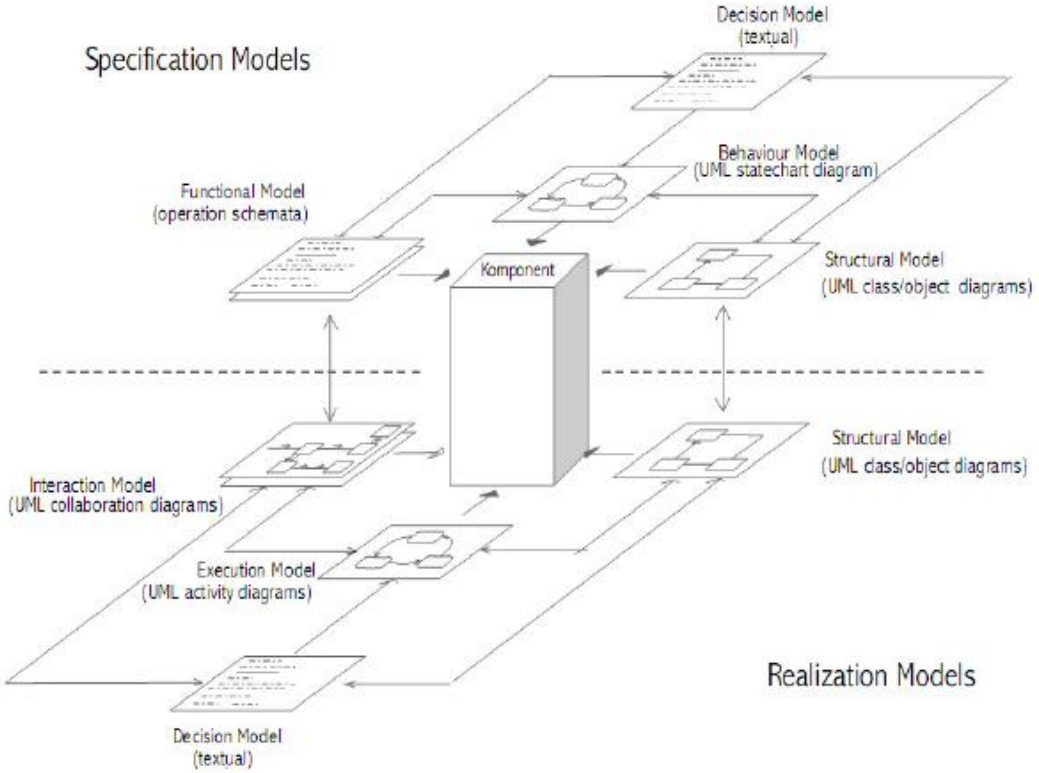


Figure 2.16 – Component specification in KobrA [29]

Component specification is defined in two different groups: specification models and realization models. The specification models consist of external definitions of a component which are visible to outside of model. Realization models consist of internal elements of component and it is considered as a design of the component.

### 2.11 CoPAM

CoPAM is defined as a technique in order to share experiences between the developers who works in different product families. In addition, they also share their family engineering methods.

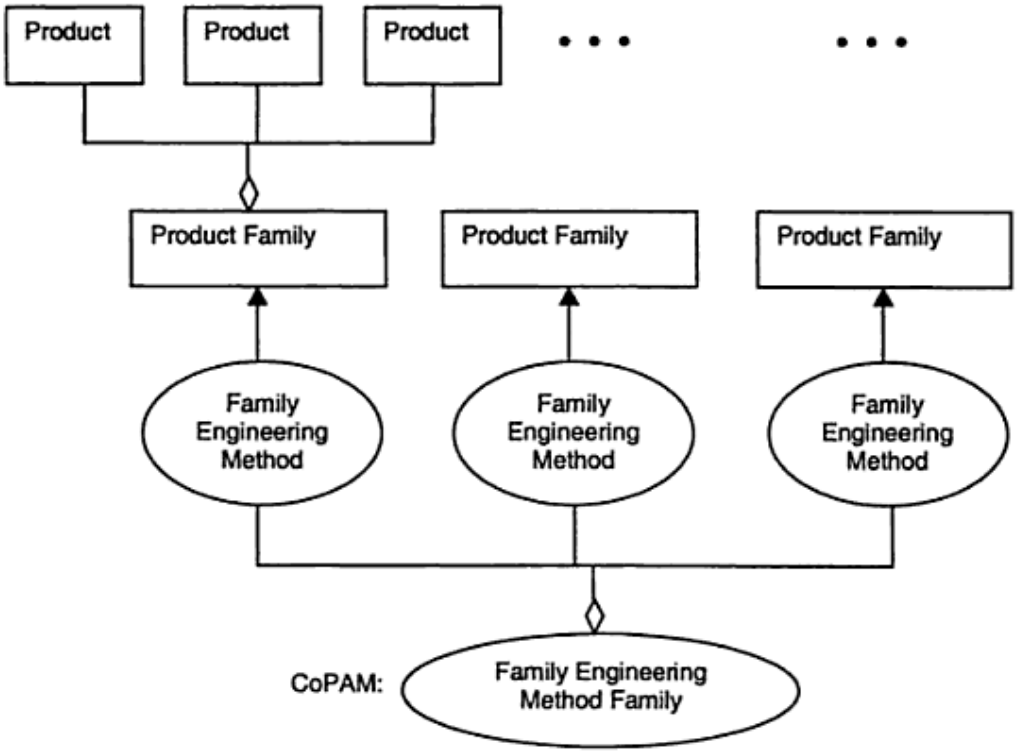


Figure 2.17 – Method of CoPAM [30]

Platform engineering and product engineering sub-processes are defined in CoPAM. Platform engineering sub-process handles producing reusable components. On the other hand, product engineering sub-process handles developing products using reusable components which are developed in platform engineering sub-process.

### 2.12 PLUS

PLUS method was introduced as an advanced method of UML-based individual system development methods by H.Gomma[31].

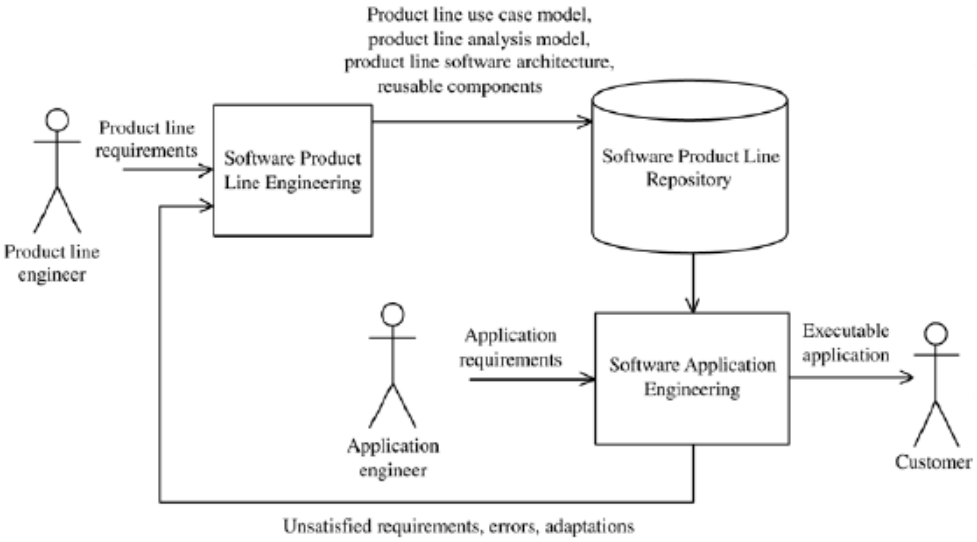


Figure 2.18 – Evolutionary SPL Process

PLUS is an evolutionary software product line process which is named as ESPLEP. On the other hand, PLUS is related to rational unified process. UML stereotype, constraint and tag are used in PLUS in order to support product line modeling.

# CHAPTER 3

# BACKGROUND

## 3.1 Variability Management

In order to reuse in large systems software product lines are developed. Variability can be described as follows: "In order to simplify mass customization the product must meet different stakeholders' needs. For this intention the variability term is presented for product. As a result of variability concept, the artifacts that can cause the differences in the applications of the product line are modeled using variability"[1]. As realized from this meaning, software product line is a different approach from other reuse techniques. The applications are different varieties and configured items of SPL. The artifacts in SPL contain variation points which are defined in a common described way. Definition of new variation points are controlled by variation management process. In short, variation points are the all possible subset of variability subject in SPL. In order to clarify variability management the meaning of variability object and subject should be given. A variability subject is the definition of variability in real world. A variability object is crated using variability subject. For instance, the color of the car is variability subject. Variability objects for this subject yellow, black, and white. Variant is a definition in order to represent variability object. By the way, if automotive company produces black and white cars, then only the variants black and white cars are defined. Other variability objects are not taken into consideration as variants for the automotive company[1].

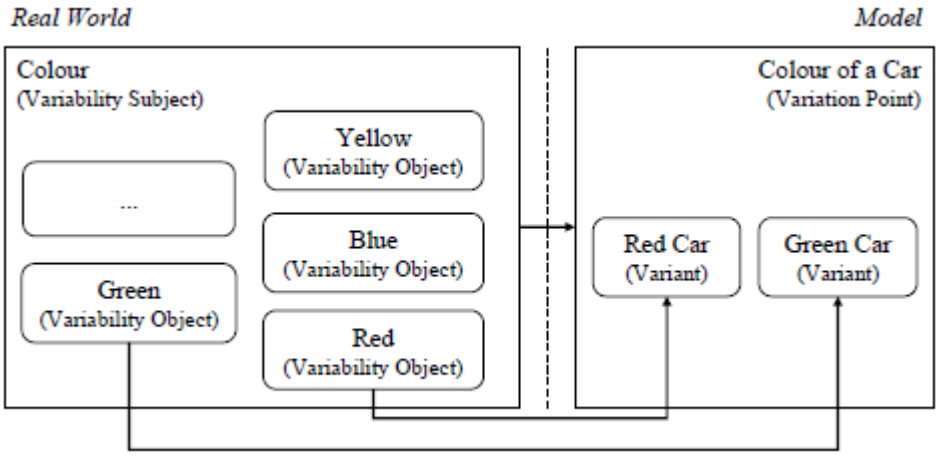


Figure 3.1 – Variability subject, object and variant[1]

In order to define variability in SPL variation points and variants which are identified in a systematical way help us. Definition of variation points and variants are made in three steps:

1) Identification of the variability subject which is the instance of real world.

2) Definition of variation point in the context of SPL

3) Definition of variants which are derived from variant objects

Variability and commonality are two major terms in software product line. Commonality indicates features that are used each application produced from product line. Variability is a distinguishing part of products developed from product line. Variability is used to customize applications. For instance, user interface of home automation system offers customers choice to select language. That feature is common for each home automation system. User interface language is used as commonality in this example. On the other hand, users of a home automation system can choose the language on installation when user interface language is used as variability [1].

In product development, there are stakeholders who have different line of sight from each other. Customer requests applications modified to their individual requirements. This causes that customers have knowledge of a little part of the variability of a product line. Otherwise, variability is an internal part for the organizations and major anxiety for them is to develop software product line. This causes that the variability is defined in two groups: Internal Variability and External Variability [1].

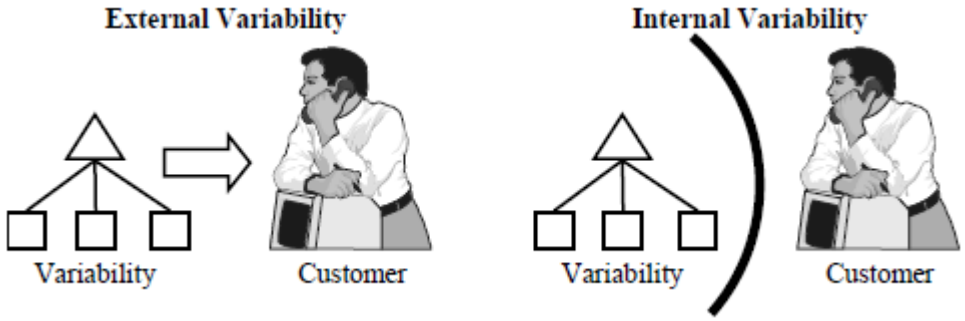


Figure 3.2 – Internal and External Variability

Internal variability is visible to developer and hidden from customer. Customer doesn't have to take into consideration when choosing the variants. On the other hand, customers have opinion about external variability which is visible to them. They can choose variants of domain artifacts. As an example, the network protocol of a home automation system which works in two modes as high bandwidth or error correction is selected by developers. On the other hand, three electronic door identification systems can be chosen by customer. Customer decides to select electronic system between three systems: electronic key, card, and fingerprint scanners.

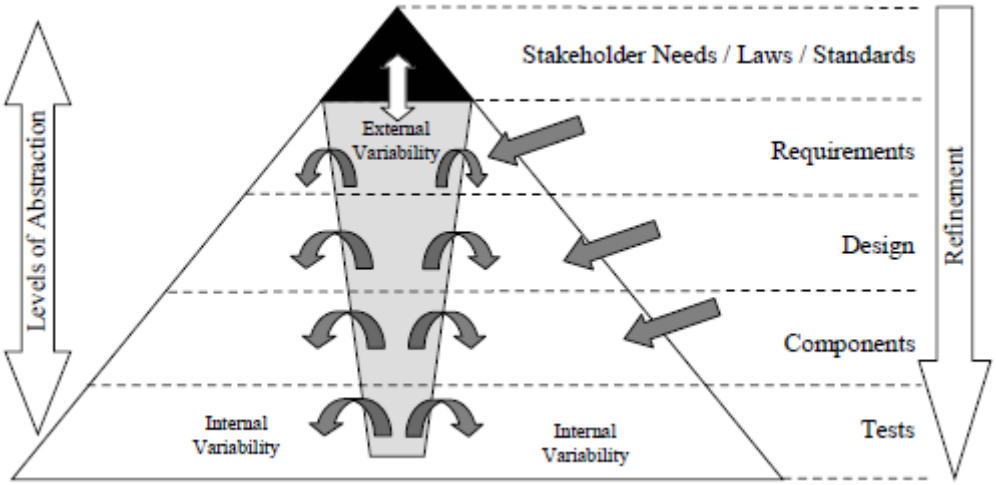


Figure 3.3 – Variability Pyramid[1]

As shown figure 3.3, external variability decreases and internal variability increases while decreasing level of variability abstraction from higher to lower level.

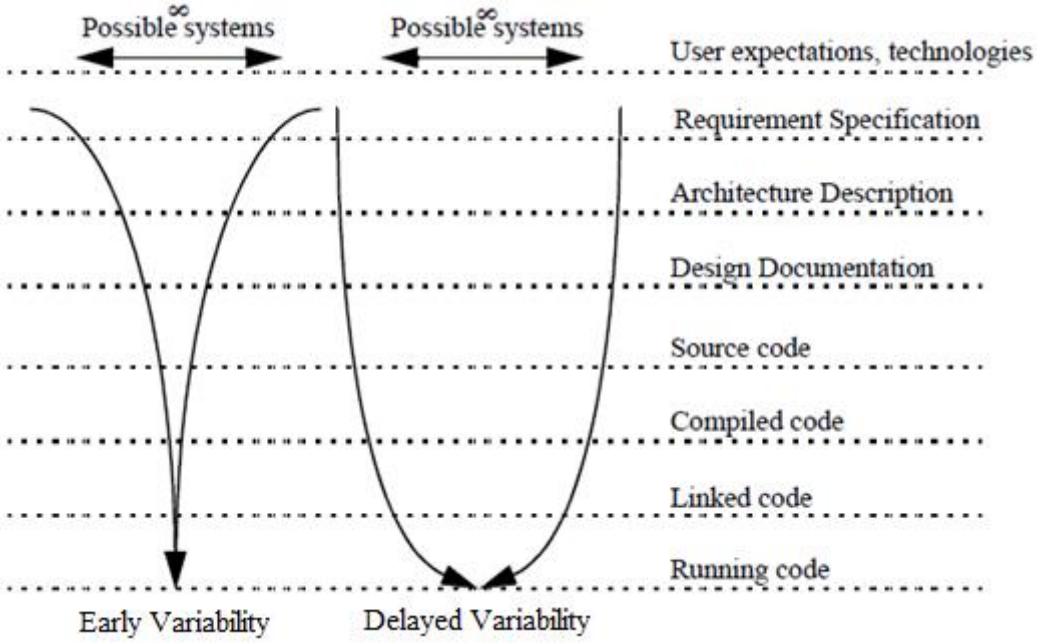


Figure 3.4 – Variability management life cycles

Another anxiety of variability management is the decision to choose the point in the development cycle. It is crucial because it has effect to optimize overall business intentions. This variability binding phases are generally: run time, link time and compile time. Variabilities can be described as early or delayed variability [32].

### 3.2 Feature Modeling

Feature modeling gathering popularity among researchers is described a key prerequisite for software product line engineering. Feature modeling is the action which is defining visible features and properties of an application in a defined domain and collecting them into a model named as feature model [33].

Relationships between a parent feature and its child features are categorized as [34]:

- Mandatory : selection of child feature is required
- Optional: selection of child feature is optional
- Or : at least one of sub-features must be selected
- Xor (alternative) : one of the sub-features must be selected
- And : all sub-features must be selected

Besides these notations, cross-tree constraints are allowed. Most used cross-tree notations are:

- A requires B : If A selected then B must be selected
- A excludes B : If A selected then B must be discarded

Graphical presentation of feature model is feature diagram which has different notations and used for different purposes [34].

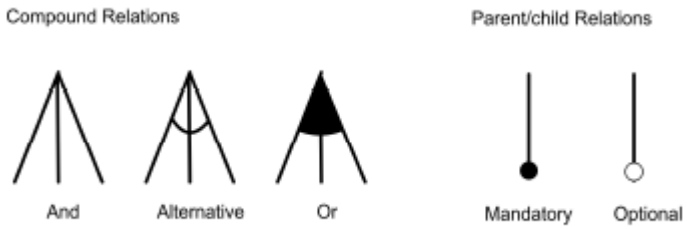


Figure 3.5 – Feature Diagram Elements

The feature diagram is a method in order to produce products deriving features from SPL. Configuration consisting of selected features validation is made by using this feature model. In addition, feature models are used to manage variabilities and commonalities.

The following figure shows the feature model of the collaboration system. Different collaboration systems can be derived from this sample model by simply selecting from a feature model and without touching the code. In this way, we can derive several products with different functionality, such as instant messaging or forum functionality.

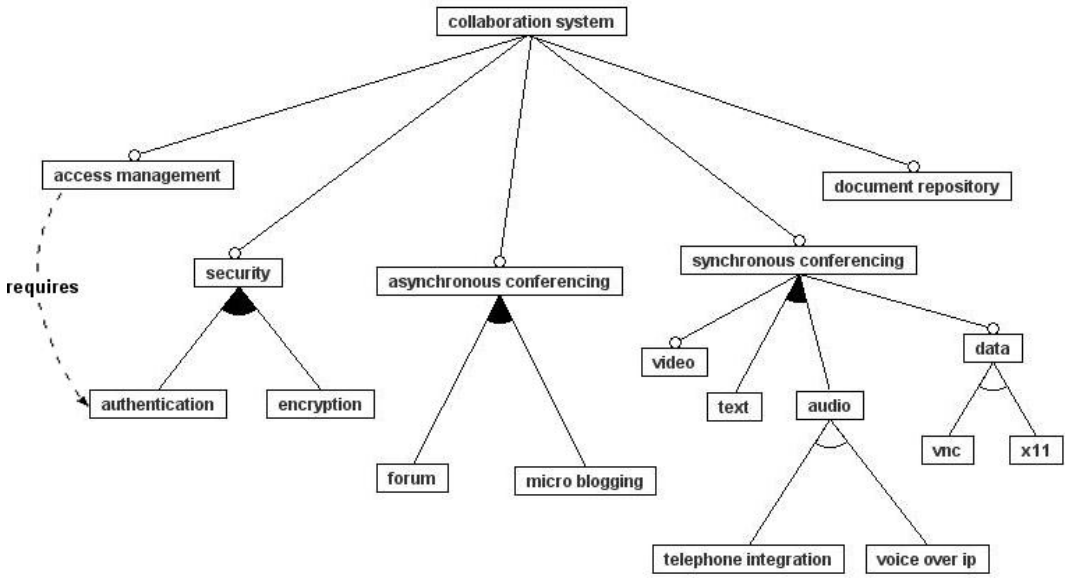


Figure 3.6 – Feature modeling example

There are many different configurations which can be selected from feature diagram. Selecting valid feature configuration from feature model describes a product in product domain. For example, according to figure 3.6, it is a valid configuration: access management, authentication, forum asynchronous conferencing, document repository.

Derivation of different kind of products from feature diagram may cause managing this diagram. A grammar is offered by D.Batory to manage more complex models and in figure 3.7 an example to this grammar is shown.

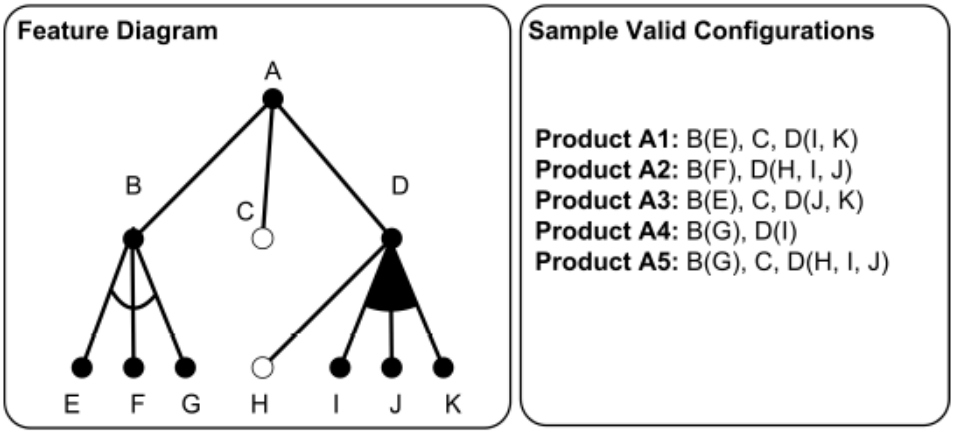


Figure 3.7 – An example for feature configuration

Organizations generally use feature modeling tools in order to manage feature variability. There are many tools which can be used directly. One of them is pure::variant developed by Pure Systems. With pure::variants a tool for variant management of product line based

software development is available. Pure::variant is the tool to outline and manage efficiently all parts of software products with their components, restrictions and terms of usage. With this set of information and with the continuous tool support throughout the entire software configuration process valid solution are created automatically from the chosen features [35].

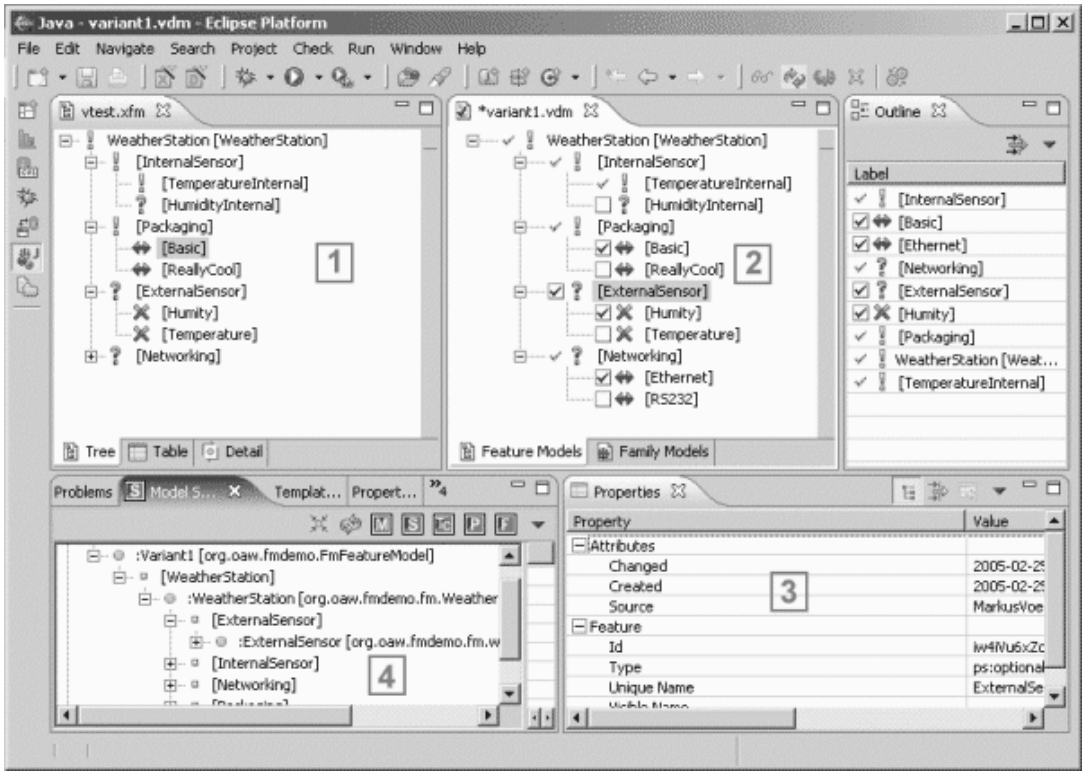

Figure 3.8 - A feature modeling example from pure::variant

Part 1 in the figure 3.8 shows the definition of a feature model. It shows the hierarchy features in a tree, the various icons define how legal subsets of these features (variants) can be defined. In part 2, such a variant is defined, the tool verifies that the variant is the consistent with the constraints defined in part 1. Absolutely, we can define any number of valid variants. Part 3 shows the properties the currently selected feature. Part 4 shows one aspect of the integration with architecture.

# CHAPTER 4

# EXPERIMENTAL BACKROUND AND CASE STUDIES

## 4.1 Dependency Injection, DSL and Model Driven Engineering

Dependency injection is a technique of object configuration which makes possible to change object dependencies at run-time [36]. Dependency injection is used to load real objects in application. Dependency injection intends to make possible selection among multiple implementation choices of a given dependency interface at runtime.

Dependency injection separated into different forms:

- Constructor Injection
- Setter Injection
- Interface Injection

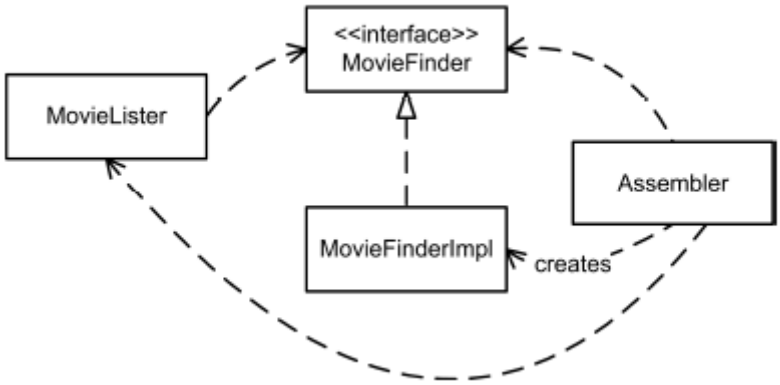


Figure 4.1 – Dependency Injection Design Pattern [37]

Dynamic injection which helps to manage dependencies and configurations of product is usually used in product line projects. Using dependency injection products are customized according to features at start-up. TADES which is a name of SPL project in ASELSAN uses dynamic injection in order to manage variabilities.

A domain specific language (DSL) is a kind of programming language which has a specific grammar rules and solves particular problem for a domain. DSL is used for special purposes. It is becoming more popular by increasing use of domain specific

modeling. DSL can be a visual diagramming language like a Microsoft Visualization Framework or Generic Eclipse Modeling System, or textual languages. DSL shouldn't be confused with scripting languages. DSL doesn't contain low-level functions for operating system and it doesn't compile to byte code. In model driven engineering DSL is used widely in software development. In this thesis, a DSL for hierarchical variability management will be introduced.

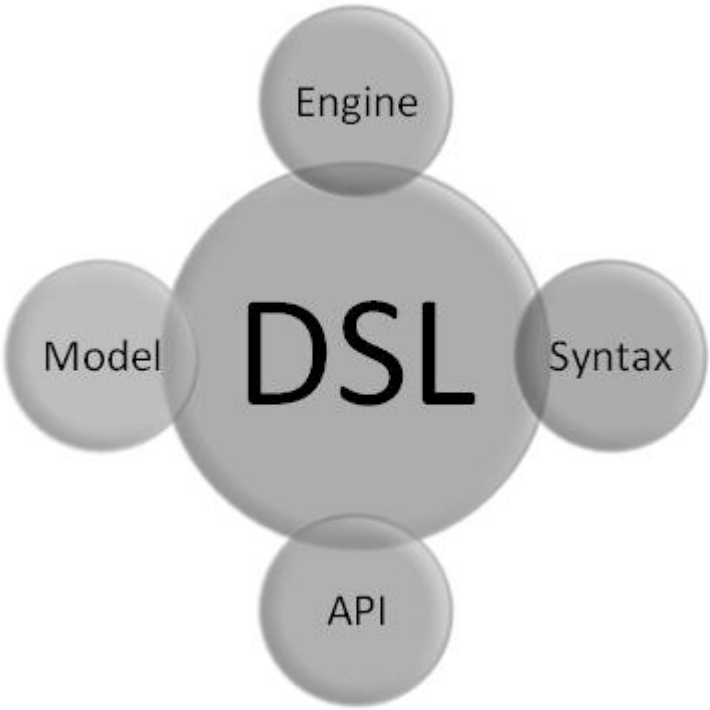


Figure 4.2 – Parts of DSL

Model-driven engineering (MDE) is a software development methodology. MDE consists of abstract representation of the knowledge rather than classical programming concept. MDE intends to improve productivity by using standard models and make easy the process of design by using design patterns. It also increases interaction between team members by using standard terminologies.

The model-driven architecture approach describes system functionality using DSL. The object management group holds the trademark on MDA. OMG focuses on creating code from abstract, class diagrams and separating design from architecture [38].

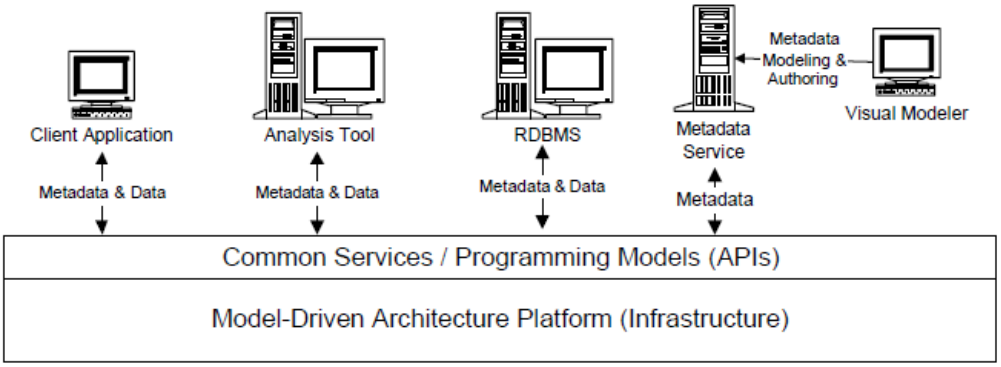


Figure 4.3 – MDA realization example

### 4.2 TADES SPL

TADES is a name of SPL project which is developed in ASELSAN Inc. This projects intents to produce technical fire support command and control products in software market [39]. TADES started with defining reference architecture and it has common design. It also uses design patterns which are usually known. This project is commercially confidentially so that some particular information will be given in this thesis.

TADES consists of many CSCI (Computer Software Configuration Items) which uses common design pattern named MVC (Model View Controller). As shown in Figure 4.4, TADES has different layers which are GUI Managers, Business Managers and Views. GUI managers are responsible to control GUI (Graphical User Interface). It doesn't know any logical information about operation which is passed to business layers. Business layers controls the logical and persistency operations. It doesn't know any information about views. Views which can be identified by their name are used to define graphical layers.

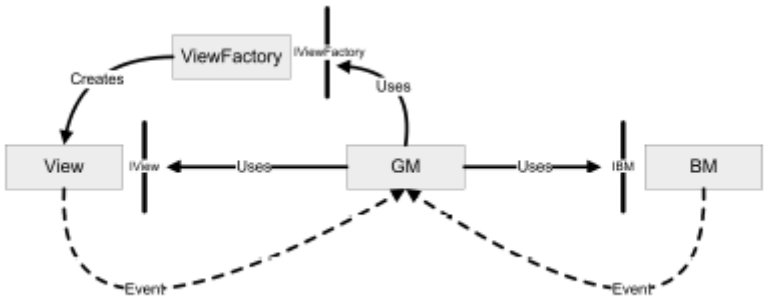


Figure 4.4 – Common MVC pattern for TADES CSCI's

In TADES, objects of these layers are created by using spring framework via dependency injection [9]. Each CSCI has different kind of features in TADES. Some of CSCI is dependent to another CSCI because of using another CSCI's feature. Spring framework also helps to manage these relationships between CSCI's. An example of defining relationships between CSCI is given in Figure 4.5.

```
<object id="objBMUnitManager" type="Tr.Com.Aselsan.TADES.FireSuppor
  <property name="UnitPersistencyManager" ref="objPMUnitManager"></
  <property name="PMSystemUnitManager" ref="objPMUnitManager"></pro
  <property name="BMMasterUnitListManager" ref="objBMMasterUnitList
  <property name="BMMessageManager" ref="objBMMessageManager"></pro
  <property name="BFGViolationBusinessManager" ref="objBMBFGViolati
  <property name="SCUCanSend" value="true"/>
  <property name="PrintReportFactory" ref="objBMUnitManagerPrintRep
  <property name="SCUCanPrint" value="true"/>
</object>

<object id="objBMUnitManagerPrintReportFactory" type="Tr.Com.Aselsa

<object id="objPMUnitManager" type="Tr.Com.Aselsan.TADES.FireSuppor
  <property name="CommonPersistencyManager" ref="objCommonPersister
  <property name="PersistencyBusinessManager" ref="objBasePersister
</object>

<object id="objGMUnitManager" type="Tr.Com.Aselsan.TADES.FireSuppor
  <property name="UnitBusinessManager" ref="objBMUnitManager"></pro
  <property name="ViewFactory" ref="objUnitViewFactory"></property>
  <property name="MasterUnitListGUIManager" ref="objGMMasterUnitLis
  <property name="BMMasterUnitListManager" ref="objBMMasterUnitList
  <property name="GISOperator" ref="objGMGISManager"/>
  <property name="GMPrintManager" ref="objGMPrintManager"/>
  <property name ="LocationProviders">
    <list element-type="Tr.Com.Aselsan.TADES.Interfaces.ILocationPr
      <ref local="objGMCoordinateConverter"/>
      <ref local="objGMIntersection"/>
      <ref local="objGMResection"/>
      <ref local="objGMTravers"/>
      <ref local="objGMShift"/>
      <ref local="objGMKnownPointManager"/>
      <ref local="objBMGPSDeviceManger"/>
      <ref local="objGMHKBSDeviceManger"/>
      <ref local="objGMGISManager"/>
    </list>
  </property>
</object>

<object name="objUnitViewFactory" type="Tr.Com.Aselsan.TADES.FireSu
```

Figure 4.5 – TADES dependency management using spring

Above figure, UnitManager and MessageManager are two CSCI defined in TADES. UnitManager consists of features about system unit and MessageManager controls communication layers and interfaces. One of feature of UnitManager is to give ability to user sending system information to another unit. However, UnitManager doesn't have communication ability in order to satisfy demand. In contrast, in TADES MessageManager controls communication operations so that UnitManager must use MessageManager in order to communicate with other units. As seen, one of UnitManager's feature is whether it can communicates or not.

Below figure depicts the part of the TADES relationships. This part consists the components that deal with system unit operations.

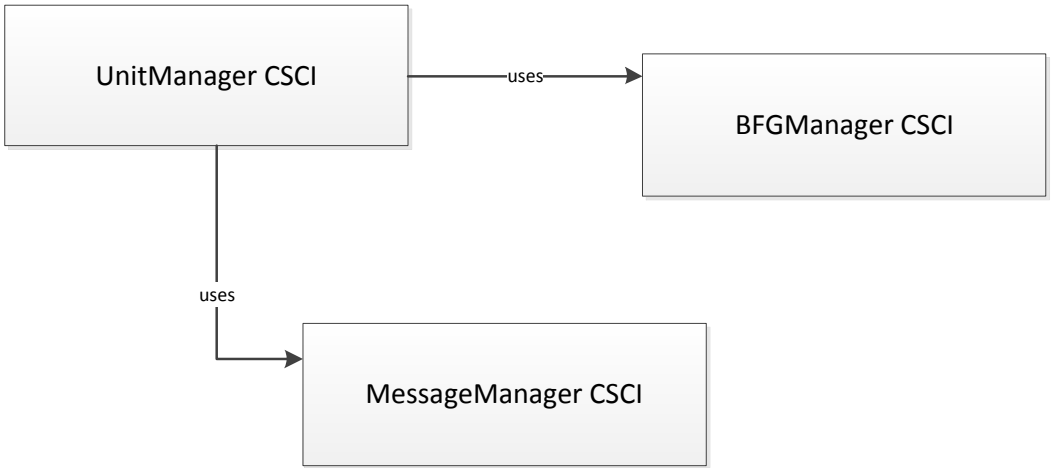


Figure 4.6 – Part of UnitManager dependencies

Another example to TADES CSCI is MetManager which deals with meteorological operations. As shown below feature, MetManager has dependency to UnitManager in order to retrieve system unit information.

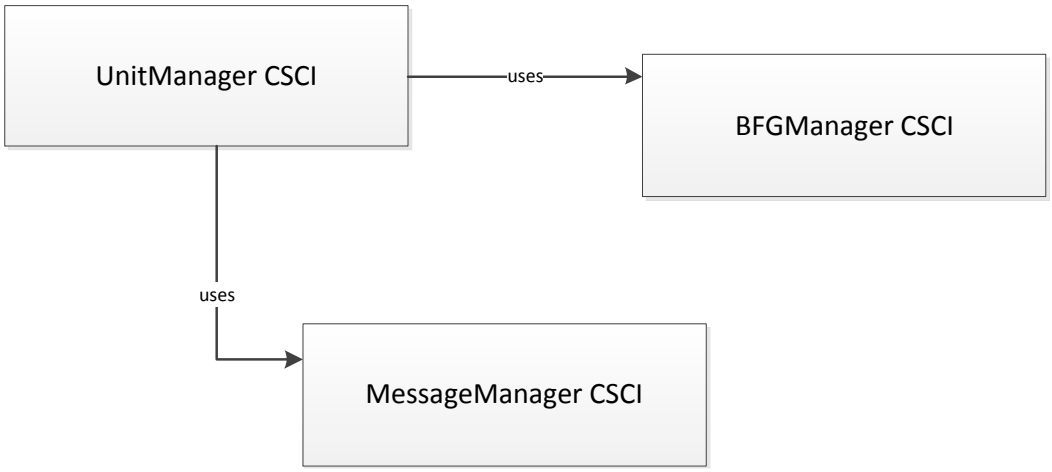


Figure 4.7 – Part of MetManager dependencies

Quick information about TADES architecture has been given. TADES also has feature tree in logical. Below figure includes the part of TADES feature tree.

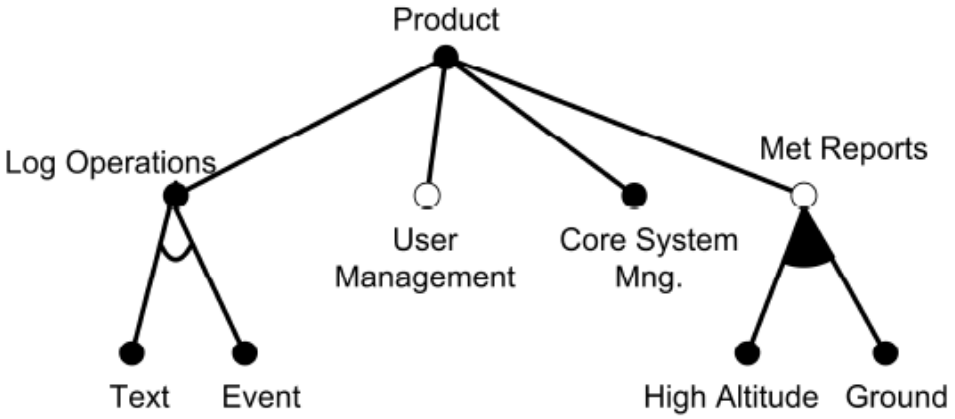


Figure 4.8 – Little part of FT

The structure of TADES configuration items which is described using spring.net has been introduced. At this point, it is beneficial to give quick information about Spring.Net. Spring.Net is an open source application framework that makes building enterprise .Net applications easier. Spring.Net consists of different kind of elements in order to handle dependency injection. In Figure 4.9, elements of Sprint.Net are given in XML schema.

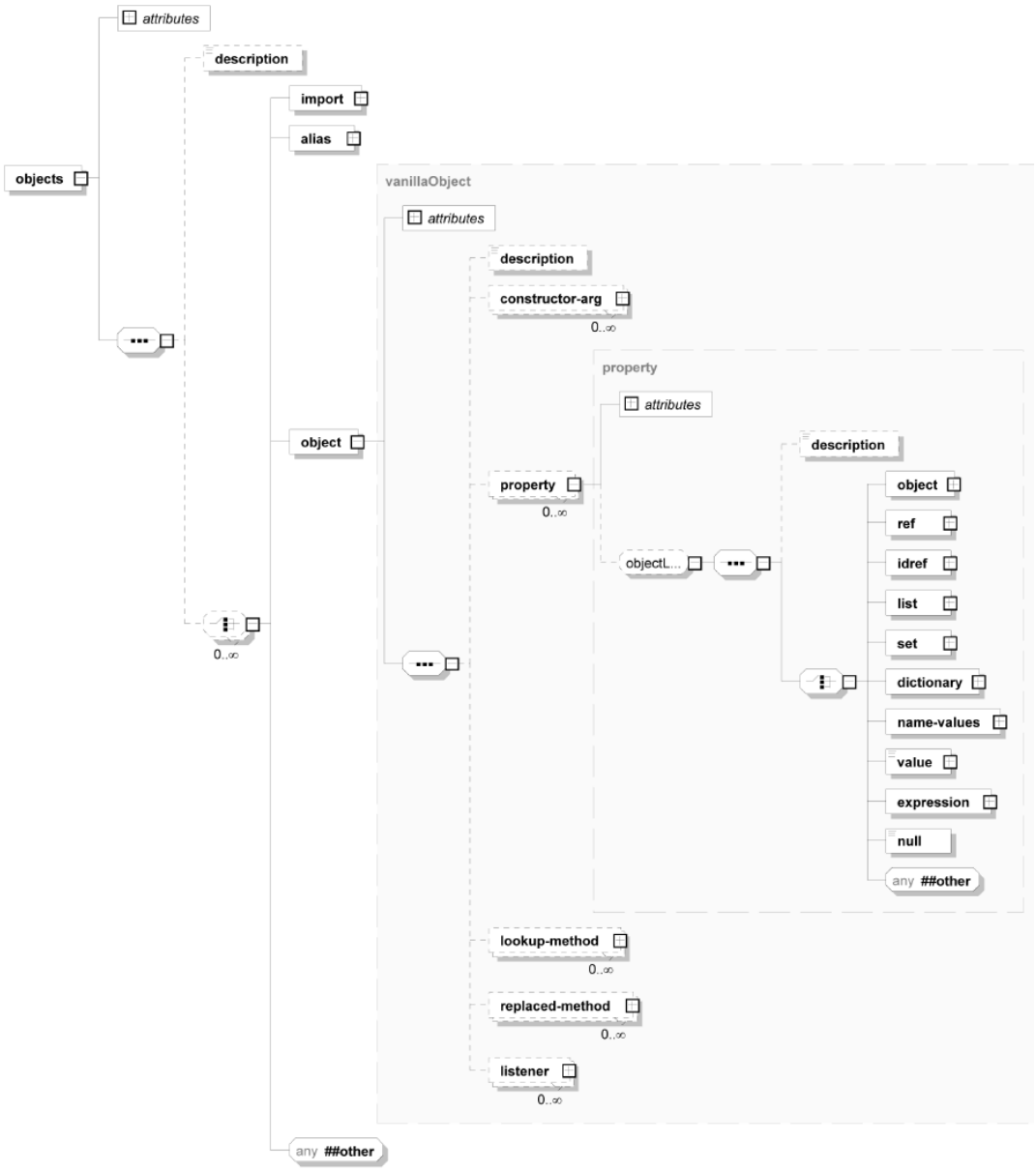


Figure 4.9 – Spring.Net elements

In Spring.Net there is an object element which has id, type attributes and property elements. Id attribute defines the alias name of the object which can be used by other objects in order to make reference. Type attribute includes class name and namespace information of object. Property elements are the properties of object. Property differs into three different types: reference, value and list properties. Property has name, ref, value and list elements. Name describes the name of property. Ref attribute is the reference to another object id. Value attribute is a primitive type and contains concrete value. List element includes multiple properties.

As seen below paragraphs, managing variability and software configuration on TADES is complicated. Besides software complexity, there is an organizational problem for SPL in

Aselsan Inc. There is another stakeholder named as system engineer beside software engineer in Aselsan Inc. These employers also have role to specify software products but limited-level. In order to manage this variability and commonality a solution will be given next section.

### 4.3 Solution

First of all, I used pure::variants feature modeling tool in order to model features on TADES. Pure::Variants as mentioned before based on DSL and model-driven architecture. Due to commercial restriction, short information about model will be given.

There are two different feature trees in our mechanism because of different stakeholders working in separated teams. Before explaining these feature models, I will explain what are these stakeholders and what are they responsible for.

System Engineer: There is a team consisting of system engineer in Aselsan Inc. These people are responsible for designing system which comprises both software and hardware. These people don't know software detailed. Moreover, they must have upper level knowledge about software. These people have interaction between customer, hardware engineers, project managers, test engineers and software engineers. While they are selecting cable of system, on the other side they choose software components. As a result, they don't have time in order to know software in detailed level but they must design software.

Software Engineer: There is a team which includes different people from system engineers in Aselsan Inc. These people are responsible for only software design, nothing else. They have interaction between test engineers and system engineers. In order to produce a product from TADES SPL, these people must get system requirements from system engineers and create software requirements according to technical requirements.

As mentioned before, in TADES it is complicated to manage variability by using XML editor because TADES isn't modeled by any feature modeling tools. There is huge necessity to manage variability in TADES. By using TADES SPL, 13 products are developed and 5 new products will be produced.

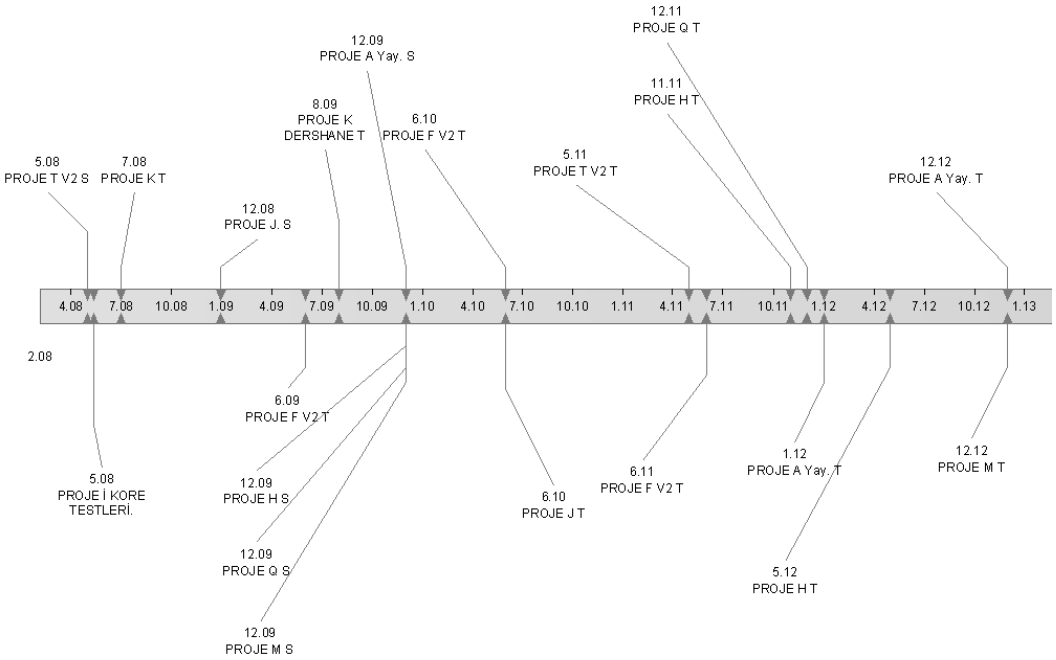


Figure 4.10 – Part of TADES product schedule

Above figure clearly depicts that TADES schedule is time constricted. By starting to develop our products, we have seen that many bugs are based on configuration and variability management. It is normal because feature modeling in TADES is made by manually. Another remarkable point is that managing variability takes much time from both system engineer and software engineer. They must configure the application variability together by sitting in front of same computer because some of features are known by system engineers and some of them are known by software engineers. There are two shortcomings in our SPL in order to solve:

- Managing variability by tool automatically
- Differing system engineers and software engineers variability levels

In literature, feature tree is used in order to manage variability. There is different tool support in order to model variability nowadays. These tools help to define feature and use DSL in order to remain consistency. As mentioned before, TADES CSCI's are configured via configuration files. Because my major goal is configuring software components by using this tool, I must select one of feature modeling tools which enables extending and storing in feature model extra information. It is required to store extra information in feature models which will be used in model to configuration file phase.

System engineer's and software engineer's level of abstraction details is different. For example, system engineer decides whether the product has log management ability or not. System engineer doesn't care what kind of log mechanism will be used in product. At this point, software engineer decides log mechanism according to technical requirements.

While producing application using TADES SPL there may be developed newer CSCI's which are specific to application. These newer components are also be designed according to TADES reference architecture.

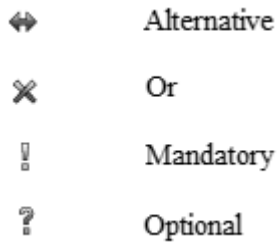


Figure 4.11 – PV notations

Above figure depicts PV notations. It helps to design feature tree and give understanding of following feature trees.

System features are defined in a specific model according to my approach. This tree is named as system level feature tree. In addition, software engineers have another feature tree named as software level feature tree which includes more detailed features and configurations for software. For instance, log management is a feature of TADES SPL. Moreover log-management has sub-features according to level of abstraction. In my approach, I divide these sub-features into two layers.

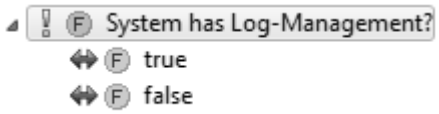


Figure 4.12 – System level feature tree

As shown in above figure, system level feature tree is clear to understand for system engineers. On the other side this tree doesn't comprise any information about software configuration. It is designed to specific system engineer's knowledge.

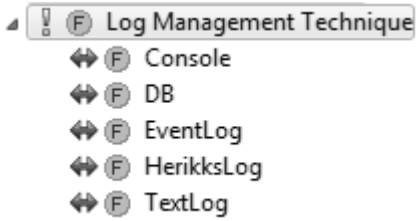


Figure 4.13 – Software level feature tree

Figure 4.13 depicts the software level feature tree. But there must be constraint and restriction between these two separated trees. If system engineer selects log management feature, then software should select one of sub-features of log management. It can be managed by semi-automatically or manually.

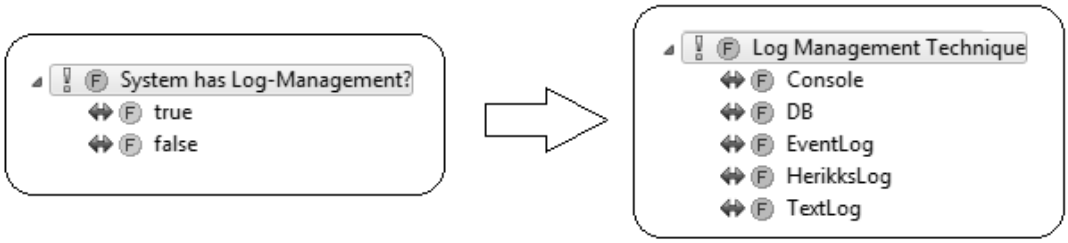


Figure 4.14 – Transition between feature levels

In my approach software level feature tree also comprises ingredients of software configuration. By adding configurations into feature tree gives an advantage as simplicity and managing configurations at single point. Figure 4.15 denotes the software configuration features.

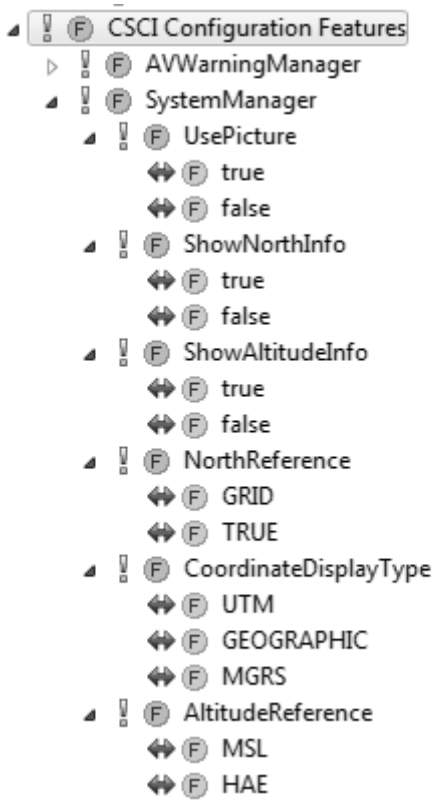


Figure 4.15 – Software configuration features

After selecting features and configurations both system level and software level features, configuration files are generated which configures TADES CSCI. At this phase, I used java script files which can be written inside to PV. Eclipse runs the script files and produces configuration files. Part of script file is given following figure.

```
//iterate over all models of variant
var modelIter = models.iterator();
while (modelIter.hasNext())
{
    var model = modelIter.next();

    /*writeline(indent + INDENT, fw,'');
    writeline(indent + INDENT, fw,'');
    writeline(indent + INDENT, fw,'');
    writeline(indent + INDENT, fw,'ITERATING: ' + model.getName());
   */

    var rootid = model.getElementsRootID();
    var element = model.getElementWithID(rootid);

    //writeline(indent + INDENT, fw,'ROOT: ' + element.getVName());

    var elementIter = ModelLogic.getOrderedChildren(element).iterato
    //writeline(indent + INDENT, fw,'iter');
    while (elementIter.hasNext())
    {
        var e = elementIter.next();

        //writeline(indent + INDENT, fw,'VNAme:' + e.getVName());
        if(e.getVName() == COMPONENT_NAME)
        {
```

Figure 4.16 – Sample part of the configuration generator script

As mentioned before, Spring.Net Core component uses generated file. Following figure depicts the sample of product configuration file. Log management has been chosen in type of Herikss and log-management is active.

```
<!-- LogManager -->
<object id="objGMLogManager" type="Tr.Com.Aselsan.TADES.ApplicationFramework.
        LogManager.GMHerikksLogManager, AppFWGUIManagers">
  <property name="ViewFactory" ref="objViewFactoryLog" />
</object>

<object id="objViewFactoryLog" type="Tr.Com.Aselsan.TADES.
        ApplicationFramework.LogManager.HerikksLogViewFactory, AppFWGUIManagers" >
  <property name="LogProcessor" ref="objGMLogManager"/>
</object>
<!-- End Of LogManager -->
<!-- ***************************************************************************
```

Figure 4.17 – Generated TADES configuration

Assuming product configuration has been changed in time. Thus, there is no need to change configuration files textually. Software and system engineer select appropriate features and configuration files are regenerated. Application restarts and changed features utilized to the product.

### 4.4 Assesment and Evaluation

Hierarchical Variability Management Process Algorithm:

1. Choose system level features
2. Feature selection validation
3. M2M transformation
4. Definition of must features based on system level features
5. Choose software level features
6. Feature selection validation
7. M2T transformation
8. Generation of configuration items

First of all, I would like to summarize hierarchical variability management life cycle. System engineer chooses system-level features from system-level-feature-tree as shown in next figure.

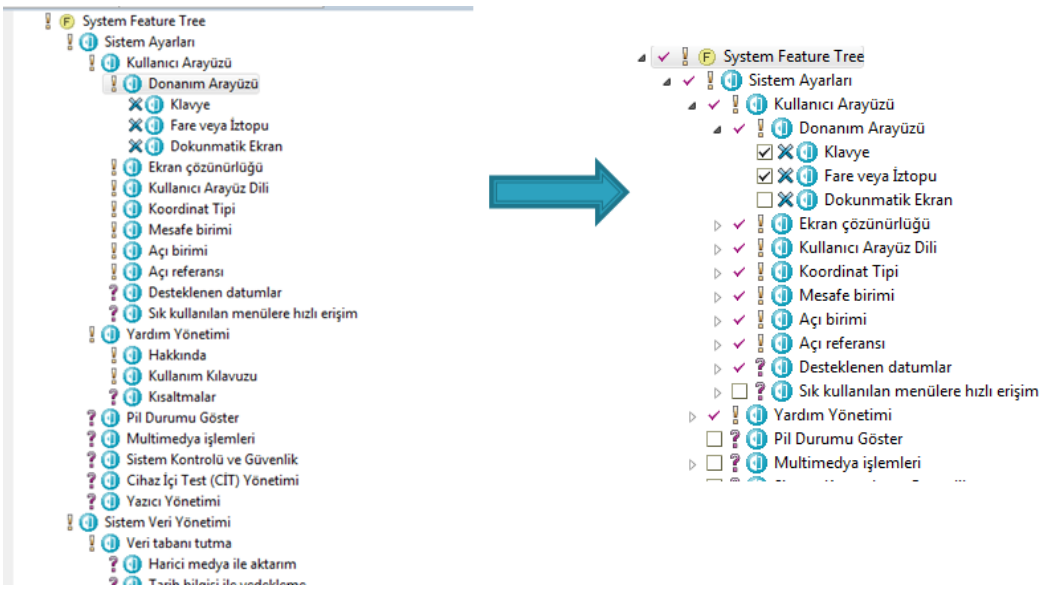


Figure 4.18 – System level feature selection

Next, M2M transformation is made by defining some constraints which lays between software and system level feature trees. As shown in Figure 4.19, the features which are selected in system-level are given as an input to software level feature tree.

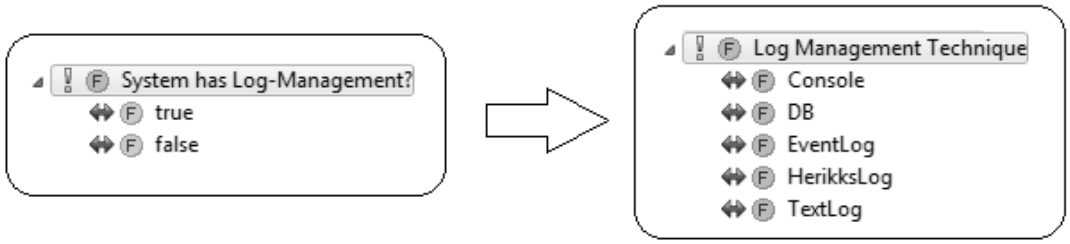


Figure 4.19 – M2M transformation between two layers

After M2M transformation, software engineer selects features from his own feature model. System level features and constraints are given as an input to software level feature tree.

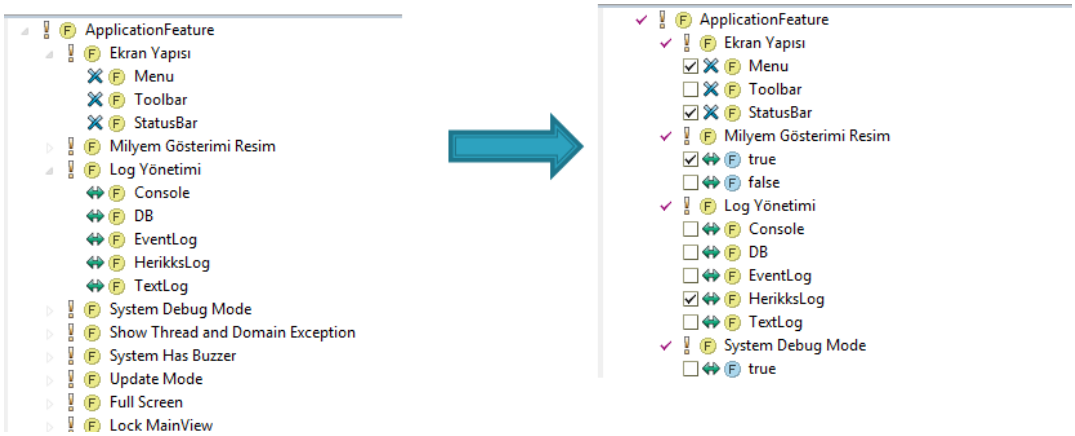


Figure 4.20 – Software level feature tree

After all, M2T transformation is made by prescribed scripts. Software engineer runs the scripts and generates configuration files. These configuration files are deployed into application and product is configured according to feature models.

Advantages of hierarchical variability management according to classical way:

- Using different levels of feature models provides ability to different stakeholders in order to work individually.
- Provides level of abstraction between system and software engineers.
- Managng software configuration by tool is time saving and reliable.
- Multiple layer feature model is extendible for other stakeholders. It can be applied test level in future.

Some metrics from our SPL projects from ASELSAN is given following parts. In Table 1, the lines of codes count of products which are produced using TADES SPL are given.

Table 4.1 – Lines of code counts of products

| Project Name | Lines of Code Count |
|---|---|
| Project J | 134.535 |
| Project K | 98.994 |
| Project Q | 225.988 |
| Project M | 88.903 |
| Project F | 155.934 |
| Project T | 156.099 |
| Project A | 235.985 |

Before my offering technique configuration files are edited manually. So it causes bugs while editing configuration files. Next table shows number of bugs which are caused of editing configuration files manually. After using my technique, number of bugs count are vanished.

Table 4.2 – Number of bugs which are caused of editing configuration files manually

| Project Name | Number of bugs |
|---|---|
| Project J | 15 |
| Project K | 12 |
| Project Q | 29 |
| Project M | 13 |
| Project F | 24 |
| Project T | 19 |
| Project A | 35 |

Product specific features must be defined in software level feature tree in order to apply my offering technique. Next table shows feature defining duration.

Table 4.3 – Duration of defining product specific features

| Project Name | Feature tree definition time (min) |
|---|---|
| Project J | 125 |
| Project K | 134 |
| Project Q | 239 |
| Project M | 89 |
| Project F | 200 |
| Project T | 84 |
| Project A | 327 |

**Evaluation Result:**

After this technique is used, number of bugs are decreased beause of predefined rules and consistent feature trees. Validation and verification mechanism lays under these feature tress. System engineer and software engineer can work individually with this technique. They must work together before this technique .

# CHAPTER 5

# CONCLUSION AND FUTURE WORK

This thesis has a contribution about variability management in software product line. The main motivation is to find an effective technique to manage variability in the industry. This thesis presents that hierarchical variability management is a very appropriate technique to manage variability in SPL. In addition, this technique offers more effective software production process.

First of all, my technique obtains details of abstraction between system engineer and software engineer. They shouldn't know their level of details about application features. Another advantage is the exhibition of the interfaces between software and system engineers. Using this technique and feature models, the number of bugs based on software configuration will be reduced because of configuring software items automatically or semi-automatically. Using traditional variability models instead of hierarchical variability models can cause complexity owing to managing big pieces of features. However, complexity brings errors and bugs with itself while configuring software. Otherwise, this technique matches the process of the companies and software production life cycle. In clearly, system engineer produces system design documents which consist of system design specifications. This document is an input to the software engineer as system-level-feature model. System-level-feature model can be associated with this document. Software engineers produce software requirements specification documents using system design documents as an input. On the other side, SRS (system requirements specification) document can be associated with software-level-feature model. At last but most importantly, using hierarchical variability management saves time. In traditional technique, system engineer and software engineer have to select software features together. It means time loss of two engineers because of being had to work together. However, my technique makes possible to work individual. A disadvantage of using hierarchical variability management is modifying feature models. In traditional variability management, updating feature model is simply because it contains single model. On the other side, my technique contains two feature models and dependencies among them. If we think that modifications in software product lines become rarely, time losses can be ignored.

Although my technique offers a way to improve software variability management, it is also possible to apply this technique into test engineering phase. In this way, hierarchical variability management can be used between all software development stakeholders. Test-level-feature model can be implemented as a future work.

## REFERENCES


[1] Pohl, K.; B¨ockle, G.; “Software Product Line Engineering”, Springer, Berlin, 2005, pp. 257-284, 355-370, 2005.

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

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

[4] Pressman, R. S.; “Software engineering: a practitioner's approach (sixth edition)”, 2005, pp. 1-64, 495-502.

[5] Tommi Myllymaki, Variability Management in Software Product Lines, 3.12.2001

[6] Frakes, W.B. and Kyo Kang, (2005), "Software Reuse Research: Status and Future", IEEE Transactions on Software Engineering, 31(7), July, pp. 529-536.

[7] Charles W. Krueger, Software Reuse, ACM Computing Surveys, Volume 24 Issue 2, June 1992

[8] "Code reuse". DocForge. Retrieved 15 December 2009.

[9] Introduction to spring framework, http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/overview.html, last visited on June 2013

[10] Champman, M; Van der Merwe, Alta (2008), Contemplating Systematic Software Reuse in a Small Project-centric Company, Saicsit 2008, South Africa

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

[12] Ian Sommerville, Software Engineering 7th Edition, 2004

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

[14] Lockheed Martin Tactical Defense Systems 9255 Wellington Road Manassas, VA 22110-4121,Organization Domain Modeling (ODM) Guidebook, Informal Technical Report for STARS, STARS-VC-A025/001/00, 1996

[15] 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

[16] Falbo, Ricardo de Almedia; Guizzardi, Giancarlo; Duarte, Katia Cristina (2002). "An Ontological Approach to Domain Engineering". Proceedings of the 14th international conference on Software engineering and knowledge engineering

[17] Harsu, Maarit (December 2002). A Survey on Domain Engineering (Report). Institute of Software Systems, Tampere University of Technology. pp. 26. ISBN 9789521509322.

[18] X. Ferré S. Vegas, An Evaluation of Domain Analysis Methods, Facultad de Informática –Universidad Politécnica de Madrid

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

[20] 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

[21] Weiss D., Lai C., Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley, 1999

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

[23] 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

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

[25] 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

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

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

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

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

[30] 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

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

[32] 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

[33] Kwanwoo Lee, Kyo C. Kang and Leajoon Lee, “Concepts and Guidelines of Feature Modeling for Product Line Software Engineering”, Department of Computer Science and Engineering, Pohang University of Science and Technology, Korea

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

[35] pure::variants, http://www.pure-systems.com/pure_variants.49.0.html, last visited on July 2013

[36] Niko Schwarz, Mircea Lungu, Oscar Nierstrasz, “Seuss: Decoupling responsibilities from static methods for fine-grained configurability”, Journal of Object Technology, Volume 11, no. 1 (April 2012), pp. 3:1-23

[37] Inversion of Control Containers and the Dependency Injection pattern, http://martinfowler.com/articles/injection.html, last visited on July 2013

[38] John D. Poole, Model-Driven Architecture: Vision, Standards And Emerging Technologies, April 2001

[39] Aselsan Inc., http://www.aselsan.com.tr, last visited on August 2013.