University of Mississippi

eGrove

Electronic Theses and Dissertations | Graduate School

2013

Modeling Software Product Lines Using Feature Diagrams

Hazim Husain Shatnawi
University of Mississippi

Follow this and additional works at: https://egrove.olemiss.edu/etd

Part of the Engineering Commons

Recommended Citation

Shatnawi, Hazim Husain, "Modeling Software Product Lines Using Feature Diagrams" (2013). *Electronic Theses and Dissertations*. 447.
https://egrove.olemiss.edu/etd/447

This Thesis is brought to you for free and open access by the Graduate School at eGrove. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of eGrove. For more information, please contact egrove@olemiss.edu.

MODELING SOFTWARE PRODUCT LINES USING FEATURE DIAGRAMS

A Thesis
presented in partial fulfillment of requirement
for the degree of Master of Science in Engineering Science
in the Department of Computer and Information Science
The University of Mississippi

by

HAZIM SHATNAWI

APRIL 2013

Copyright Hazim Shatnawi 2013
ALL RIGHTS RESERVED

ABSTRACT

The leading strategies for systematic software reuse focus on reuse of domain knowledge. One such strategy is software product line engineering. This strategy selects a set of reusable software components that form the core around which software products in a domain are built. Feature modeling is a process that enables engineers to identify these core assets, in particular the common (e.g., shared) and variable features of products.

The focus of this thesis is to give an overview of the feature modeling process by introducing feature diagrams. Feature diagrams capture and represent common and variable properties (features) of the software products in a domain, focusing on properties that may vary, which are further used to produce different software products. We present practical examples that show how feature models are used to represent a set of valid composition of features (configurations), in which each configuration can be considered as a specification of a software system instantiated from a software product line.

DEDICATION

I dedicate this thesis to my father Husain Shatnawi, my mother Sana' Zuraiqi, and my sisters Soha and Hala for their endless love and support.

LIST OF ABBREVIATIONS AND SYMBOLS

CDMA	Code Division Multiple Access
COTS	Commercial Off-The-Shelf
DARE	Domain Analysis Reuse Environment
DSL	Domain Specific Languages
DSSA	Domain Specific Software Architecture
FODA	Feature-Oriented Domain Analysis
FORM	Feature-Oriented Reuse Method
GSM	Global System for Mobile Communications
IMSL	International Mathematics and Statistics Library
NAG	Numerical Algorithms Group
ODM	Organization Domain Modeling
OO	Object-Oriented
OOAD	Object-Oriented Analysis and Design
SDLC	System Development Lifecycle
SPL	Software Product Lines

TABLE OF CONTENTS

ABSTRACT ii

DEDICATION iii

LIST OF ABBREVIATIONS AND SYMBOLS iv

ACKNOWLEDGMENTS v

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTER 1 – INTRODUCTION 1

CHAPTER 2 – SOFTWARE REUSE 5

2.1 Introduction 6

2.2 Software Reuse Principles 8

2.2.1 Definition 8

2.2.2 Types of Software Artifacts for Reuse 9

2.2.3 Selecting Software Artifacts for Reuse 11

2.3 Software Reuse in System Development Lifecycle 14

2.4 Software Reuse Advantages 19

2.5 An Overview of Software Reuse Approaches and Techniques 21

2.5.1 Opportunistic Reuse 21

2.5.2 Reusable Software Libraries 22

2.5.2.1 Reusable Software Libraries Drawbacks 22

2.5.2.2 Third-Party Libraries 24

2.5.3 Planned Reuse 25
2.5.3.1 Domain Specific Languages 27
2.6 Summary 30
CHAPTER 3 – SOFTWARE PRODUCT LINES 32
3.1 Introduction 33
3.2 Domain Principles 35
3.2.1 Types of Domains 36
3.2.1.1 Domain as the Real World 36
3.2.1.2 Domain as Set of Systems 38
3.2.2 Domain Scoping 39
3.3 Software Product Lines 40
3.3.1 History of Software Product Lines 42
3.3.2 Software Product Lines Advantages 43
3.3.3 Software Product Lines Principles 46
3.4 Software Product Line Engineering 46
3.4.1 Domain Engineering 49
3.4.1.1 Domain Analysis 51
3.4.1.2 Domain Design 55
3.4.1.3 Domain Implementation 56
3.4.2 Application Engineering 57
3.5 Software Product Lines versus Other Software Methods in Software Reuse Research ... 59
3.5.1 Software Product Lines versus Single Systems Development 59

3.5.2 Software Product Lines versus Component-Based Software Engineering............... 62

3.6 Summary 65

CHAPTER 4 – DOMAIN ANALYSIS AND FEATURE MODELS 67

4.1 Introduction... 68

4.2 Feature-Oriented Domain Analysis (FODA) 69

4.2.1 FODA's Concepts and Activities.. 69

4.2.1.1 Context Analysis 70

4.2.1.2 Domain Modeling 71

4.3 Other Domain Analysis and Engineering Methods 73

4.3.1 Organization Domain Modeling (ODM) 74

4.3.2 Domain Specific Software Architecture (DSSA) 75

4.3.3 Domain Analysis and Reuse Environment (DARE)....................................... 77

4.4 Feature Models.. 80

4.4.1 Feature Definition 81

4.4.2 Feature Model Elements 82

4.5 Feature Diagrams 83

4.5.1 Concepts and Features in Feature Diagrams.. 84

4.5.2 Mandatory Features 86

4.5.3 Optional Features.. 88

4.5.4 Alternative (“one-off”) Features 90

4.5.5 OR (“more-off”) Features 94

4.5.6 Normalizing Feature Diagrams... 96

4.6 Summary 101

CHAPTER 5 – CASE STUDIES 102

5.1 A Software Product Line for a Video-Converter Program 103

5.1.1 Features Descriptions 110

5.1.2 Features References 113

5.1.3 Features Constraints and Dependencies 118

5.1.4 Possible Configurations for the Video-Converter Product Line 122

5.2 Feature Modeling versus Object-Oriented Modeling 125

5.2.1 A Software Product Line for Cellular Phones Using Feature Diagrams 125

5.2.2 Transforming a Cellular Phone Feature Diagram into a UML Class Diagram 128

5.3 Summary 133

CHAPTER 6 – CONCLUSION AND FUTURE WORK 135

6.1 Conclusion 135

6.2 Future Work 136

BIBLIOGRAPHY 137

VITA 148

LIST OF TABLES

TABLE 2.1: Common Domain Specific Languages and their corresponding problem areas 28

TABLE 5.1: Other types of feature dependencies used in feature diagrams.............................. 120

LIST OF FIGURES

FIGURE 1.1: Conceptual model of the system development lifecycle methodology 15

FIGURE 1.2: A system development lifecycle methodology that incorporates software reuse (from [KS91]) 16

FIGURE 3.1: Domain as the *Real World* (from [SCK+96]) 37

FIGURE 3.2: Domain as *Set of Systems* (from [SCK+96]) 38

FIGURE 3.3: History of software reuse from ad hoc to systematic (from [Nor08]) 43

FIGURE 3.4: Domain engineering and application engineering as two sub-processes of software product line practice (from [CN07]) 49

FIGURE 3.5: Domain engineering and application engineering processes (from [CE00]) 58

FIGURE 4.1: A feature diagram with root node *RN* as a concept node and two features, *FN1* and *FN2* 85

FIGURE 4.2: A feature diagram with atomic (*A1*) and composite (*M1*) mandatory features 86

FIGURE 4.3a: *ON* as an atomic optional feature 88

FIGURE 4.3b: *ON* as a composite optional feature 89

FIGURE 4.4: A feature diagram with three sets of alternative features 90

FIGURE 4.5: A feature diagram includes OR-features 94

FIGURE 4.6: A feature node that has a set of alternative features 97

FIGURE 4.7: A feature diagram with one optional alternative feature is normalized into a new and equivalent diagram with all optional alternative features 98

FIGURE 4.8a: A feature diagram with two optional OR-features is normalized into a new and equivalent diagram with all optional OR-features 98

FIGURE 4.8b: A feature diagram with two optional OR-features is normalized into a new and equivalent diagram with all optional OR-features and then normalized again into a new and equivalent diagram with all optional features 100

FIGURE 5.1: A feature diagram of a video-converter product line 105

FIGURE 5.2: An extension to the *Choices* feature that represents available input/output video files formats 114

FIGURE 5.3: An extension to the *Single-File Input/output* feature which represents available input/output video file formats for a video-converter program that supports only one video conversion at a time (Basic version) 116

FIGURE 5.4: An extension to the *Multi-File Input/output* feature that represents available input/output video file formats for a video-converter program that supports multiple video conversions at a time (Pro-version) 117

FIGURE 5.5: An extension to the *Merge* feature shows available formats that Pro-versions of the video-converter program support 118

FIGURE 5.6: A feature diagram of a video-converter product line with constraints 121

FIGURE 5.7: A feature diagram of a simple software product line for cellular phones 126

FIGURE 5.8: UML class diagram that shows one possible implementation of the cellular phone software 129

FIGURE 5.9: UML class diagram that shows another possible implementation of the cellular phone software using multiple inheritance mechanism 131

FIGURE 5.10: The extension point *Input-Device* with OR-features represented using a single inheritance mechanism 131

CHAPTER 1

INTRODUCTION

Over the last two decades, the software development process has been changing rapidly due to a large increase in the demand for computer software. We use software programs in almost every aspect of life. The previous ways of developing single systems from scratch are inadequate to meet the demands of this new environment [Pre05]. Therefore, the software industry has started to focus on reusing software to develop new products with less time and effort.

The techniques for software reuse can be categorized into two types: *opportunistic* and *systematic* reuse. Opportunistic reuse is the most common technique for software reuse; no specific technology is needed to perform it [Kru92], [KLM08]. An example of opportunistic reuse is reusable software libraries [MMM98], [Wen89], in which software companies can build libraries of software components that can be reused in a variety of new projects. One problem with reusable software libraries is when these libraries become large [Gac95], [Nei89]. The components must be classified carefully to enable developers to easily find what they need [Big94], [DF87]. [Nei89].

With the difficulties in applying opportunistic reuse when building complex projects, software companies started to shift their focus towards systematic reuse strategies.

Systematic software reuse strategies ([Par72], [LG84], [Kru92]) focus on reusing domain contents by classifying software components in software libraries according to their domains, where a domain represents an area that contains a set of related software systems/components sharing similar functionalities [CE00] [Nei84], [Big97], [Hen96]. This classification enables library users to find the desired components and retrieve them for reuse in new projects [Kru92].

One of the most efficient systematic reuse strategies is *Software Product Lines* (SPL), a technique that focuses on reusing a domain's contents to produce a collection of similar software systems from a shared set of reusable software assets [Nor02], [MYA+99], [CN01], [Kru06]. These reusable assets are developed specifically for reuse and classified according to their domains.

Software Product Line Engineering (SPLE) is an approach for developing lines of software products [PBL05]. A line of software products consists of a set of related software products that share common characteristics (features) in addition to variable parts that differentiate one software product from another. SPLE involves two sub-processes: *Domain Engineering*; a process that attempts to develop reusable software assets to be used in new projects, and *Application Engineering*; a process used to build individual systems from the reusable assets developed during the domain engineering [CE00], [PBL05], [Lin02], [BEG12], [WL99].

Developing reusable software assets in domain engineering is a task that goes through three phases: *Domain Analysis*, *Domain Design*, and *Domain Implementation* [Har02-A], [CE00]. The domain analysis phase defines the domain scope, identifies which software systems belong to that domain, and analyzes these systems to discover their common and variable features. If new requirements are needed, domain design and implementation phases are performed to develop new reusable software assets [Pri90], [KCH+90], [SCK+96], [CE00].

Identifying common and variable features among similar software systems in a domain is achieved by performing a commonality and variability analysis study for the software product line. The most efficient way to perform that study [CHW98] is by adopting feature modeling process that produces feature models [KCH+90]. A feature model is visually represented by a feature diagram, a graphical notation that represents all related software systems in a domain (a product line) in terms of features [DK02], [RBS+02], [KCH+90], [Bat05].

The main goal of this thesis is to show the importance of performing a feature modeling process to capture and model commonalties and variabilities of software systems in a product line. We will demonstrate how feature models play a key role in the development of software product lines by proposing examples that show how to perform feature modeling process on a set of software systems sharing common functionalities within a domain.

This thesis is structured as follows.

- Chapter 2 gives a general overview of the concept of reusing software and compares between ad hoc and systematic software reuse strategies.
- Chapter 3 introduces the software product lines technique and its sub-processes (domain and application engineering), explains the domain engineering phases and shows how they work in parallel with application engineering process, and compares software product lines with other systematic reuse approaches.
- Chapter 4 introduces Feature Oriented Domain Analysis (FODA), a domain analysis method best known for introducing the feature modeling process, briefly presents other domain analysis and engineering methods, and describes the feature modeling process in detail.
- Chapter 5 is divided into two sections; the first one is a case study that shows how to perform a complete feature analysis study on a software product line for a video-converter program. The second section introduces a product line for a cellular phone, represents it by a feature diagram and its equivalent object-oriented model using a UML class diagram, and illustrates the differences between the two approaches in modeling commonalties and variablities across software systems in a domain.

CHAPTER 2

SOFTWARE REUSE

A software product line process attempts to provide software companies with a systematic strategy that maximizes the benefit of reusing their already built software systems in new projects. The interest in this process emerged from software reuse research. This chapter introduces the concept of reusing software, presents different types of software artifacts that can be reused, and identifies which artifacts are more preferable to be parts of the reusing process. It also shows how to apply the software reuse process in traditional software engineering approaches to developing software applications. Finally, this chapter differentiates between ad hoc and systematic software reuse strategies by surveying different approaches and methodologies for software reuse that have been applied and discusses advantages and drawbacks of each approach.

2.1 Introduction

In software engineering research, the idea of reusing software is not new. Programmers have reused ideas, sections of code, design patterns, algorithms, and procedures in their work since the earliest days of software development, but they have done so in ad hoc and informal ways [Pre05].

In 1968 the participants at the *NATO Software Engineering Conference* discussed possible solutions to the emerging "*software crisis*." The increase in power of computers meant that government and industry sought to use computer software more widely. This wider use meant the need for more programs, larger and more complex programs, and programs whose failures could cause more damage. The previous processes for software development seemed inadequate to meet the demands of this new environment.

The NATO Software Engineering Conference addressed difficulties such as:

- Predicting software development costs (to budget accurately)
- Estimating software development time requirements (to schedule projects reliably)
- Managing large software projects
- Maintaining large code bases
- Building software to meet user requirements
- Overcoming shortages of capable software professionals
- Developing software that is of high quality (correct, reliable, and robust)

The conference articulated the concept of software engineering for the first time [NR68]. The idea of software reuse process was proposed as a development method to solve the software crises [Kru92], [KS91]. During the conference, Douglas McIlroy presented the concept of software reuse through his thesis "*Mass-produced software components*," in which he focused on the concept of basing software industry on reusable software components [1] and automated strategies for customizing components.

McIlroy suggested that building complex systems requires standard source-code components, which serve as building blocks to construct complex systems:

> *"… the software industry is weakly founded, and that one aspect of this weakness is the absence of a software components sub-industry… The most important characteristic of a software components industry is that it will offer families of routines for any given job"* [Mcl68].

Following McIlroy's work, several researchers have sought to define systematic methods for software reuse (e.g., [Par72] and [LG84]). As more complex systems were being built in more competitive time frames, the desire to reuse software became a necessity, and companies started to concentrate more funding on finding better techniques to develop products with less time and effort.

[1] Software components presented by McIlroy indicate libraries of mathematical routines.

2.2 Software Reuse Principles

2.2.1 Definition

Many definitions of software reuse appear in the literature. Charles Krueger, an author of a widely referenced survey on software reuse strategies titled *"Software Reuse,"* defined the process as [Kru92]:

> *"Software reuse is the process of creating software systems from existing software rather than building software systems from scratch"*

Mili et al. proposed another definition of software reuse [MMY+99]:

> It's a process " *whereby an institution defines a set of systematic operating procedures to produce, classify and retrieve software artifacts for the purpose of using them in its development activities"*

By reusing previously built software components in new systems, both the productivity of the software development process and the quality of the final product can be improved. Reducing the total number of new lines of code written leads to less documentation and testing. Increasing productivity leads to the reduction of both software development costs and development time, which are the primary factors that increase the return on investments for organizations.

2.2.2 Types of Software Artifacts for Reuse

The process of reusing software artifacts can be applied in any system lifecycle; it is not limited to reusing source code [GK05]. There are several ways to reuse software artifacts from the requirements, analysis and, design phases of existing software systems to build new software applications. Reusing artifacts can appear in one of the following forms.

- **Code Reuse**

  Programmers can reuse existing source code, executable code, libraries, functions and procedures, templates, packages, or even the code of an entire system.

- **Requirements Documents**

  By reviewing documents about the previous systems' properties, specifications, and operations, developers can find and exploit similarities in requirements among applications. Software engineers, system analysts, and stakeholders (e.g., customers and end-users) can all help in developing requirements documents.

- **Design Patterns**

  By examining design approaches adopted in previous software systems, developers can choose appropriate design strategies to build new software. They can collect these in their own catalogs of design patterns or adopt design patterns identified by others [GHJ+04].

Moreover, the software reuse process can be applied on software units with different sizes. Ian Summerville suggested that software units that are subject for reuse can vary from a single function to a whole application [Som10].

- **Application System Reuse**

  This approach reuses an entire software application by integrating it with another application without change or by building a new application that exploits similarities with an existing application. Perhaps the programmers can develop a family of applications that share a common architecture and core assets and use those common assets to configure different applications for different users. The software product line process is an example of application families, which is the main topic of the next chapter.

- **Component Reuse**[2]

  Software components provide a subset of the functionality for a particular system. Brown and Wallnau defined software components as [BW96]:

  > ”*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only***.**”

  Suppose a developer wants to build peer-to-peer chatting applications. There are likely components of these applications, such as the user interface and chat protocol, that are very similar from one version to another. These components are candidates for reuse among all the versions.

- **Object and Function Reuse**

  Objects and functions from previously built projects can be used in new systems. Areas in which a set of reusable classes, objects, and functions are available are called *Standard Software Libraries*.

[2] Reusing software components will be presented in the next chapter in Section 3.5.2.

An example is the "*C Mathematical Library*," which is a standard library for the C language that contains a group of functions that implement basic mathematical operations. Such functions are available for reuse in any newly developed systems. For instance, if a new application requires a square root function, the function's implementation is available in the *C Math library* as "*sqrt*" and can be readily linked with the application's code. By avoiding the need to re-implement the "*sqrt*" function, the developers save time and effort.

- **Concept Reuse**

  Sometimes extracting code from an existing application and modifying it for reuse can take too much effort. An alternative option is to reuse the concept of the software systems' components. This can be achieved by building a conceptual model that describes and represents the structure and behavior of the software system or component in an abstract way without representing the implementation details. An example of such an approach is reusing design patterns, as mentioned previously.

2.2.3 Selecting Software Artifacts for Reuse

In addition to choosing *how* to reuse software, developers must choose *what* software to reuse. Selecting software artifacts to become a part of the reuse process is a challenging activity. Krueger suggested that achieving effective software reuse requires four basic steps [Kru92]:

1. **Abstraction**

   Developers must understand a software artifact's functionality in order to reuse it. The first step in preparing an artifact for reuse is to express it in terms of appropriate abstractions. An abstraction for an artifact is a description that hides unnecessary low-level details and emphasizes the important information that a developer wants.

2. **Specialization**

   Concrete implementations of software artifacts are difficult to reuse directly. To prepare an artifact for reuse, developers usually identify what aspects of the artifact are likely to vary from one use to another. They then build a generic version of it that enables the artifact to be specialized when reused. For example, the artifact may be defined in terms of parameters that can be given concrete values when reused.

3. **Selection**

   Developers must know which artifacts are candidates to become a parts of the reuse library.

4. **Integration**

   When selecting an artifact to be reusable, it's necessary for the developer to know how to integrate it with newly developed systems.

Software artifacts with the following characteristics are considered the most preferable software units for reuse [McC89], [Cyb96]:

- **Understandability and Correctness**

  Well-documented and understandable artifacts help those who want to reuse or modify them. Poorly documented artifacts are difficult to understand and hence difficult to reuse reliably.

- **Formalism**

  To validate a correctness property, it's better to formally describe reusable software artifacts with mathematical specifications. An example of a formal notation that is used to describe software entities is the *Z notation* [Znt00].

- **Adaptability**

  As mentioned before, software components are nearly independent software entities in the context of their functionalities. A well-designed software component exhibits the property known as *information hiding* [Par72]. It has a well-defined interface that does not change. The implementation details are hidden inside; these details may change from one implementation to another. Thus it is possible to substitute one implementation of the component for another without affecting other parts of the system. The more independent a reusable artifact is, the easier it is to integrate it with software applications. The concept of software components adaptabilities will be presented in the next chapter.

- **Ease of Change**

  In some cases, developers must change the specification of a reusable artifact in order to accommodate the requirements of new projects. Reuse libraries are more useful when they hold artifacts that are easy to modify.

- **Verifiability**

  Reusable software artifacts should pass the testing process conducted by their maintainers with no problems. Maintainers are software developers and end-users who might want to integrate the reusable artifact into their own systems.

2.3 Software Reuse in System Development Lifecycle

As previously pointed out, the cost of software development continues to rise as computer usage increases in various aspects of life. Many different software systems for many different areas must be built within competitive time frames. These factors, along with software developers' salaries, which ranked among the highest salaries in business, have all led to high costs for development of software systems. To reduce both effort and cost, companies have incorporated the software reuse process in their project development lifecycles.

In software companies, a software system is developed in phases, starting from an initial feasibility study for building the system through testing and maintenance. This work is performed through a process called the *System Development Lifecycle* (SDLC) [SDL08].

Figure 1.1 shows a conceptual model that represents the system development lifecycle activities. It shows that a system development lifecycle is a software development methodology that encompasses the overall process of bringing a software system to market. In the first phase, the developers perform a preliminary analysis that defines the organization's objectives and describes the costs and benefits of building the software project. The second phase, analysis and requirements, determines what the end-user requirements are and defines the project's functions and features. The design phase shows how the system should operate by designing the system's architecture to satisfy the requirements specified in the previous phase. In the implementation phase, the programmers write the code of the system and then test it to find errors and fix them. The last phase is the evaluation of the whole system, which is an ongoing process that occurs in all phases.

The evaluation phase differs from the testing phase in that software developers evaluate the system in terms of future needs. The evaluation phase includes activities such as studying the system's speed, reliability, robustness, and portability.

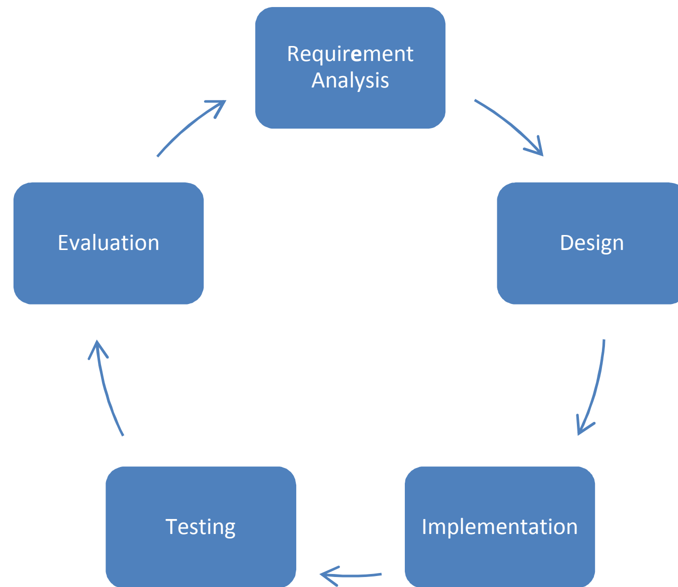

Figure 1.1: Conceptual model of the system development lifecycle methodology

In order to integrate software reuse with the system development process, SDLC must expand to support two kinds of software artifacts:

- Software artifacts to be developed from scratch
- Existing software artifacts for reuse

Following [KS91], Figure 1.2 shows a modified system development methodology that incorporates the software reuse process. The first phase indicates a new software reuse activity in which existing, previously-built software resources are classified to be reused in upcoming projects. Classification of reusable resources can be based on their types (e.g., software components, functions, or algorithms) and areas of functionality (e.g., database or networking domain) in order to organize the library of reusable software resources so they can be easily accessed and retrieved by the company's developers.

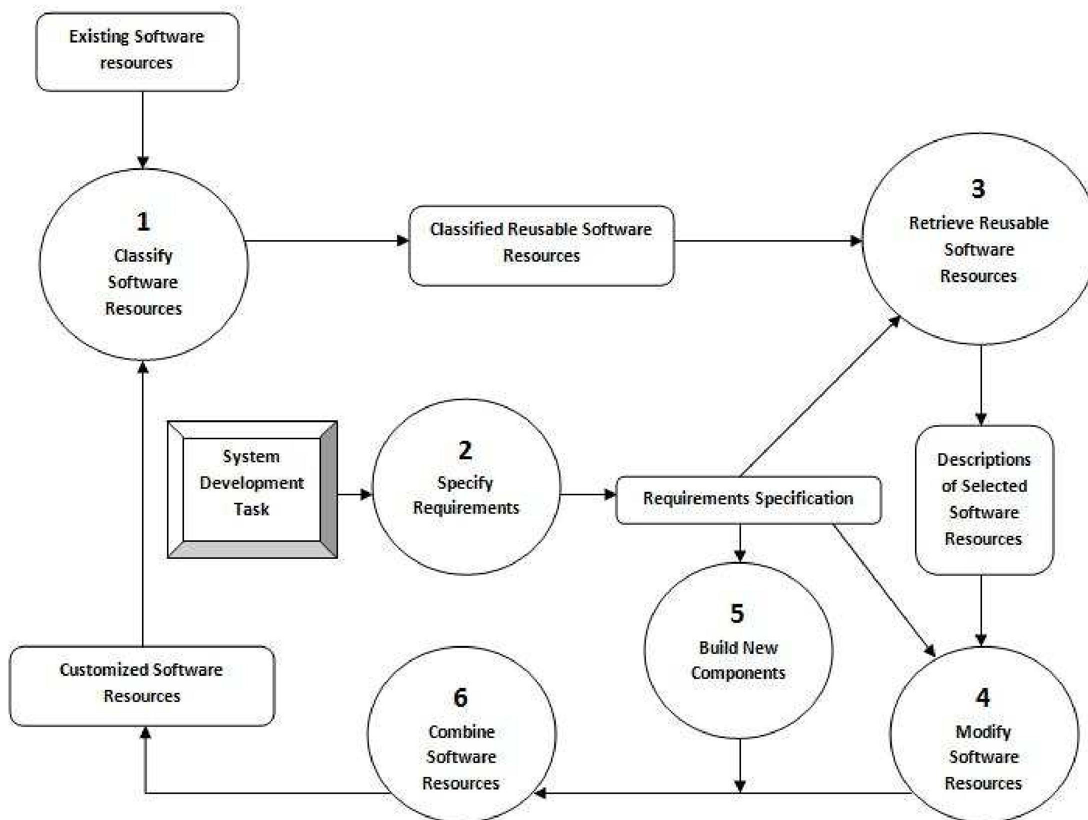

Figure 1.2: A system development lifecycle methodology that incorporates software reuse (from [KS91])

Classification of reusable resources is an ongoing activity. It should be applied every time a software resource is selected for possible reuse in order to classify and include it in the reusable library.

Following the classification activity, the task of the second phase is to specify the project's requirements and the business needs of the organization by performing a detailed analysis of the problem. The requirements specification activity is the result of the requirements analysis process, which provides a full description of the system's behavior. The requirements specification is performed using the traditional SDLC method regardless of whether the software artifacts are to be reused from existing resources or to be developed from scratch. The task of the third phase is to select the software resources needed to build the system from the reusable software resources classified in the first phase. The selection activity is based on the requirements specification process.

Because some reusable parts might not completely satisfy the requirements specification, the task of the fourth phase is to modify the reusable software resources to make them fit the requirements specification. This activity might be costly because the task of modifying software parts is not as trivial as "*cutting and filling*." It has been an issue for research to find better solutions; one promising approach is *Generative Programming*, which allows programmers to specify what they need in an abstract form and use generators to produce the desired system component. This method is intensively discussed in [CE00].

In some cases, modifying reusable software parts might require more time and effort than implementing such a part from scratch. If no reusable resources found in the library fit the requirements specification or if such resources are found but can only fit through a long process of modifications, then developers implement the new part from scratch during the fifth phase and include that new part in the library.

Newly developed software parts are combined with the existing reusable resources to construct the whole system during the final phase; newly customized parts from the previous phase are classified and added to the library of reusable resources.

Companies that incorporate software reuse in their system development processes face challenging issues such as choosing the appropriate reuse strategy to follow and having to expend more time and effort developing reusable software parts as those parts have to be built as generic parts. The more specific the software parts, the less flexibility there is in using them. Generalizing the functionality of reusable software parts allows the organization to use them in various software environments, but developing reusable parts with generic functionalities is more time consuming than building non-reusable parts.

On the other hand, building a software repository that consists of reusable software artifacts is often a profitable investment for companies in the long run, as those artifacts can be used in the organization's future projects, saving on both development cost and effort [BK91].

2.4 Software Reuse Advantages

Many software companies have derived benefits from introducing software reuse into their development processes. Developers do not need to expend as much effort to write new software applications. In addition to reduction in both applications' cost and time-to-market, studies have demonstrated that 40%-60% of source code is reusable between software applications; 60% of both design techniques and source code is reusable in business applications; 75% of application functions are common to more than one application, and only 15% of code written for applications is unique for specific applications [Sam97], [BS09]. It was reported in 1984 that approximately 50% of code in Toshiba's software products were reused [Mat84]. In the AT&T Corporation, the time-to-market period was reduced by almost a half when reuse strategies were applied by 40-90% [JGP12]. Another study shows that the US Department of Defense could save around $300 Million a year by increasing its level of reuse by only 1% [Ant93]. Several statistical studies on the benefits of applying software reuse in organizations are mentioned in [JGP12].

Several benefits can result from adopting software reuse [Som10], [Cyb96], [Sam97]. These include the following.

- **Cost reduction and earlier time-to-market**

  Reusing codes, ideas, and design patterns increase the company's *productivity* as developers will not have to build a whole system from scratch. Spending less time on developing software systems, leads to faster delivery of systems and yields overall savings on cost.

- **Increased dependability and product quality**

  As mentioned in Section 2.2, reusable artifacts follow a standard development lifecycle through specification, design, implementation, and testing phases. Every time a reusable artifact is selected for reuse, there is an opportunity to review it and fix any errors found. This increases the artifact's dependability and *reliability* and may yield a better quality product than single-use artifacts would.

- **Standards compliance**

  Standard components such as user interfaces can be built using reused software components. An example is *Menu Selection*; a standard Menu can be presented the same way in several applications, leading to improved dependability as users grow more familiar with the same menu format provided through different applications.

- **Improvement in documentation, testing, and maintenance**

  Software systems that are built using reusable software artifacts are often simpler and more abstract than others developed from scratch, as they are constructed of previously designed, implemented, tested, and documented reusable parts from previously developed projects. For that reason, newly developed systems using these reusable parts are easier to document, test, and maintain.

2.5 An Overview of Software Reuse Approaches and Techniques

Despite the many benefits that software reuse provides, there are challenging problems that might occur when using such a process. For instance, some requirements documents and design techniques used in building previous software systems might not be appropriate for developing new systems. This case may require effort and time to find the correct artifact for the current situation, understand it, retrieve it from the library, modify it, integrate it into the new system, and test it.

2.5.1 Opportunistic Reuse[3]

Most problems occur when *unplanned* reuse strategies are used for applying software reuse in a system's development. As pointed out at the beginning of this chapter, programmers have historically reused software in opportunistic or *ad hoc* ways. The concept of opportunistic reuse means randomly placing potentially reusable chunks of software (fully written, tested, and used in previous systems) into a library until an opportunity to use them arises.

Opportunistic reuse is the most common method for software reuse; no specific technology is needed to perform it. This is a low-cost strategy in comparison to other systematic reuse methods [KLM08].

[3] The term was presented by Charles Krueger [Kru92]. Some authors prefer the term "incidental reuse" [GE06] or ad hoc reuse.

In opportunistic reuse, programmers or designers are not only aware of getting the opportunity of reusing artifacts in libraries of reusable software parts but also in search for and retrieval of appropriate artifacts from different sources (e.g., reusable software libraries).

2.5.2 Reusable Software Libraries

"*Software libraries' most common application is software reuse*" [MMM98]. Software companies can build libraries of software artifacts that can be reused in a variety of new projects. Such libraries enable developers to assemble new software systems by combining operations already written and tested in previous systems.

Reusable software libraries contain software assets that developers can browse, retrieve, and utilize in their projects. These software assets include software parts such as source code, subroutines, classes, and documentation. In addition, reusable software libraries should also offer services such as accessing external storage, retrieving components from different application domains by interfacing to external programs, and binding software parts from the library [Wen89].

2.5.2.1 Reusable Software Libraries Drawbacks

Reusable software libraries are not considered a primary factor for successful software reuse [FBD+91]. Problems with reusable libraries occur when libraries grow large and complex. These libraries make it difficult to find and reuse appropriate components [Gac95].

To overcome these problems, software libraries need systematic methods to classify assets and to search the library [Nei89]. Therefore, opportunistic reuse might solve problems in small libraries, but it becomes unwieldy for large, complex libraries.

Through his extensive research on reusable libraries, James Neighbors [Nei89] described the problem as the *overall library problem* that increases all other problems. His suggestion was based on the assumption that reusable libraries should only contain small artifacts that are flexible, well-specified, and thus easy to understand (Neighbors referred to this as the flexibility and structural specification problem).

Although a library with many small artifacts decreases the flexibility and structural specification problem, it increases the classification and search problem. Neighbors suggested that if the software parts in the library are small, then the number of those parts in the library must be very large, which would be a hard task for classifying (e.g., according to its functionality or application domain) and thus searching for them. A library with a few large artifacts decreases the classification and search problem but also increases the flexibility and structural specification problem [Nei89].

In other words, problems in the utilization of reusable software libraries are ~~the~~ complexities in classification and search when general purpose repositories of reusable components grow larger. When a library contains a large number of components, the library's storage grows exponentially as it becomes harder to provide descriptions to artifacts, resulting in poor search-matching probabilities for locating reusable artifacts.

To support reuse effectively, software libraries must classify artifacts based on their functional descriptions and organize them for efficient search and retrieval. In addition, providing a good description for each artifact helps programmers to understand its functionality, thus increasing the likelihood it will be reused. If the reuse library is too cumbersome to use, programmers will just ignore it and develop all their code from scratch.

Some suggestions for handling the growth of reusable libraries are discussed in [Big94]. Another work dealing with the overall library problem presented by Neighbors is provided in [DF87].

2.5.2.2 Third-Party Libraries

Commercial-Off-The-Shelf Software (COTS), or third-party libraries, is an alternative to reusable software libraries. Third-party libraries are external libraries that offer well-designed and documented reusable software components along with tools for finding, linking, and loading them. Using third-party libraries can reduce development time and cost by offering pre-built artifacts that can be integrated into new systems.

According to [GE06], software artifacts offered through third-party libraries are fully documented, ready to use, generalized for use in various software environments, easy to access and integrate with applications, and often updated and improved regularly. An example of a third-party library is the *NAG Library* [NAG13], a software library of numerical analysis routines developed by the *Numerical Algorithms Group*. The NAG library provides a large collection of mathematical and statistical algorithms supporting different programming languages, such as C, C++, FORTRAN, .NET and MATLAB.

However, using third-party libraries does not guarantee a gain in productivity. One drawback of using such libraries is that using them restricts programmers to the library's methods and standards, which may not provide the exact functionality needed. Another disadvantage is the familiarization problem. Unlike libraries compiled from a company's already existing projects, using external third-party libraries may necessitate training courses to enable developers to familiarize themselves with a library's features and use.

Third-party libraries may be costly, both to acquire the license initially and keep up-to-date as new versions are released. The usage of third-party libraries and in real-world applications is discussed in [RDV12].

2.5.3 Planned Reuse

With the difficulties in applying opportunistic reuse, software companies have started to invest more resources in developing systematic software reuse strategies.

Through his extensive research on software reuse research, Charles Krueger presented eight different approaches to software reuse, where he used a taxonomy to describe and compare between those approaches. The taxonomy characterizes each approach in terms of its corresponding reusable artifacts and how those artifacts are abstracted, specialized, selected, and integrated (see Section 2.2.3) [Kru92].

As a result, Krueger demonstrated that the most preferable and successful approach for achieving the maximum benefits of reusing software components is by classifying and grouping reusable software components within well-defined and well-understood domains, and the approach of using libraries of reusable software artifacts without a good classification and retrieval schemas is considered among the least effective approaches for supporting software reuse.

One software reuse approach discussed by Krueger is the *Source Code Components* approach. This approach builds on a comprehensive scheme for classifying components according to their application domains. Before a component is inserted into the library, its behavior must be specified abstractly and classified according to its application domain. This classification enables library users to find the desired components and retrieve them for reuse in new projects.

Examples of large libraries with excellent classification schemas are the *Reuse Software Library* (RSL) from IBM Corporation [PY93] and the *International Mathematics and Statistics Library* (IMSL). IMSL contains two manuals, [SL03a] and [SL03b], devoting approximately 1200 pages to classify and describe around 900 mathematical routines; each routine is classified by its abstract computational or analytical capabilities [Kru92].

Describing software components is usually a process performed using a specification language; formal languages (e.g., *Z notation* [Znt00]) can express and describe *what* software systems/components do at a higher level of abstraction than executable programming languages.

Since many software systems are developed with informal requirements specifications and ad hoc design and implementation, formal specification languages can improve the systems' development process by enabling statement of mathematically precise requirements and design specifications [MS90]. This results in building software systems that are formally specified through all of their development phases and thus, formally specified software artifacts that can be part of developing future systems (through matching the formalism attribute presented in Section 2.2.2).

According to Krueger, the most promising approach to reusing software components is by classifying them according to their application domains (areas of functionalities).

> *"The most successful reusable component systems, such as the IMSL math library, rely on concise abstractions from a particular application domain. One-word abstraction specifications such as Sine often allow a software developer to go directly from an informal requirement to a fully implemented and tested source code component"* [Kru92].

General abstraction techniques, some of which are discussed in [GZP94], are required with other software components that require more detailed abstraction specifications.

2.5.3.1 Domain Specific Languages

Besides using reusable component systems such as the IMSL library, using specialized languages called *Domain Specific Languages* (DSLs) can be very effective [Nei89], [Cyb96].

General Purpose Languages (GPLs) are programming languages developed to support the design and development for a wide variety of application domains and are used generally across a range of a problem space. Typical examples of GPLs are C++ and Java languages.

Unlike GPLs, Domain Specific Languages (DSLs) are restricted to a relatively narrow problem domain. By taking advantage of the features of its domain, a DSL can be highly expressive. Deursen et al. [DKV00] defined DSLs as:

> "*A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain*"

A problem domain (also called an application domain) is a field of endeavor (e.g., business, biology, or physics), and DSLs are languages developed to simplify writing programs limited and dedicated to that field. Inspired by [MHS05], Table 2.1 shows examples of well-known DSLs and their corresponding application domains.

DSL	Application Domain
Unix shell scripts	Data organization
LATEX	Typesetting
GraphViz DOT	Graph drawing
SQL	Database queries
HTML	Hypertext web pages
ColdFusion	Data-driven websites
VHDL	Hardware design
Excel	Spreadsheets
YACC	Language grammar
MATLAB	Technical computing

Table 2.1: Common Domain Specific Languages and their corresponding problem areas

As mentioned above, DSLs are languages that incorporate domain concepts, which allow getting benefits of specific properties in a particular domain through offering formal reusable artifacts classified based on that domain [Hen96], [DKV00]. Reusable artifacts that can be reused through DSLs are language grammars, source code, software designs, and domain abstractions [MHS05]. A DSL's syntax and semantics conform to a specific area of functionality (application domain), sometimes incorporating preexisting domain specific notations.

As shown in Table 2.1, *MATLAB* is an example of a domain specific language that provides users with numerical computations. In *MATLAB*, problems and solutions are expressed in familiar notations according to their areas of functionalities (e.g., signal measurement or spectral analysis).

Krueger [Kru92] indicated that the most successful reusable components systems rely on concise abstractions from a particular application domain with components/functions described in familiar notations specific to their application domains.

MATLAB is a typical example of what Krueger proposed, as mathematical functions are expressed in familiar and easy to understand notations, mostly in one-word abstractions with complex implementation details hidden from users. For example, any electrical engineer who is not familiar with general purpose programming languages such as C++ can write code in *MATLAB* to compute a *Fast Fourier Transform* (*FFT*) function; a function can transform a time-based representation into a frequency domain representation using the built-in *FFT* algorithm by just using the name of the function as *FFT(x),* where *x* represents a vector or a matrix. There is no need to understand how the function works.

This is an appealing property of DSLs since many users are not very familiar with programming issues and only care about working at the level of the domain concepts.

Moreover, a function name itself indicates what that function does. *FFT* is a one-word abstraction that indicates the *Fast Fourier transform* function in the *Signal processing application domain* through the *Transforms* section[4].

DSLs are considered specification languages as well as programming languages with highly specified libraries of software artifacts classified based on their application domains.

A key to reuse is *domain content* (e.g., a set of reusable software components that capture the detailed knowledge specific to a problem domain). Strategies that focus on reusing domain knowledge are the leading strategies adopted by software companies to incorporate systematic software reuse into their development processes [Hen96], [Big97], [Nei84].

2.6 Summary

This chapter introduced software reuse, the process of creating software applications from existing software instead of constructing all parts from scratch. The first section presented the history of the process first mentioned in Douglas McIlroy's thesis "*Mass-produced software components*" as a development methodology to solve the software crisis. The second section defined the process, described several types of software artifacts that can be included in the reuse process, and explained how a software artifact can be selected for reuse.

---

[4] *FFT* function is located in the Transforms section; a section in the MATLAB functions in Signal processing toolbox. For more information, please visit http://www.mathworks.com/help/signal/functionlist.html

The third section explained how software reuse can be involved in a system development process by explaining how reuse activities can be added to the traditional phases of the system development lifecycle.

The fourth section listed several advantages of software reuse and provided statistical evidence of the benefits of reusing software in organizations.

The last section distinguished between opportunistic and planned reuse strategies. The first part of the section introduced the concept of opportunistic reuse, the most common approach being used in software companies. The second part of the section introduced planned reuse approaches, which are formal and systematic reuse strategies that require huge investments for organizations but which are profitable in the long run by systematically developing reusable core assets for future projects.

Domain Specific strategies that focus on reusing domain knowledge are the main topic of this thesis. The next chapter will introduce *Software Product Lines*, one of the most promising *planned-reuse* strategies that focus on reusing domain knowledge for developing new software systems.

CHAPTER 3

SOFTWARE PRODUCT LINES

Chapter 2 identified two types of software reuse: *opportunistic* and *planned*. This chapter focuses on planned reuse in which organizations use systematic strategies to reuse their previously written and maintained projects in order to develop future projects efficiently. We introduce *Software Product Lines*; a technique that focuses on reusing domain's content to produce a collection of similar software applications from a shared set of reusable software assets. This chapter defines the term *Domain* and its types in the context of software engineering; presents the *domain scoping* process for identifying software systems and components related to a problem area; reviews the evolution of software product lines; outlines software product line practices; and introduces *domain engineering* and *domain analysis* processes used in the software product line process for analyzing related software systems to build a shared set of reusable software assets. Finally, we compare software product lines with other approaches used in the software reuse field.

3.1 Introduction

As mentioned in Chapter 2, a key to reuse is domain content [Big97]. Software companies can develop reusable software assets that capture the detailed knowledge specific to a certain domain area. Reusing domain knowledge is the leading strategy for achieving effective software reuse in systems development.

Consider a software company that wants to build software management systems for libraries. Instead of developing different applications from scratch for different libraries, and rather than randomly selecting some previously built artifacts to use, the company can apply a systematic reuse strategy that manages the process of reusing already implemented projects to develop new systems.

The company can create a software library that contains the company's already-implemented software assets (e.g., components, functions, algorithms, and design patterns), classify them based on their functionalities (networking domain, database, etc.), and then start developing new artifacts specifically for reuse as components in future systems.

To capture and classify existing software parts, developers can analyze already implemented systems to identify which software parts are common among them and which are different. Common software parts can be incorporated into future projects in the same area while variable parts can be customized to yield different software products.

To develop reusable artifacts from scratch, the company must analyze its operations and determine what types of systems it develops, what software parts comprise those systems, and which parts would be sufficiently common across those systems to justify the costs of developing them as reusable software parts.

After capturing already existing software resources and developing new reusable software parts, the company can construct a product line of software management systems for libraries (e.g., a family of related software systems) which are developed to provide solutions for the same problem area (e.g., managing libraries). The company can then combine reusable artifacts from this product line with variable software artifacts to yield different software applications for different clients.

Areas of interest are called *domains* [SCK+96]. The process of developing reusable software artifacts for a domain is called *domain engineering* [CE00]. The process of analyzing related software systems to identify and capture common and variable characteristics between them is called *domain analysis* [Pri90].

The *Software Product Lines* approach uses both domain analysis and domain engineering processes to build software systems that share common functionalities to provide solutions for specific domain areas (banks, libraries, etc.) instead of creating different systems one by one from scratch.

3.2 Domain Principles

Before introducing software product lines or domain engineering and analysis processes, it is necessary to first define the term *domain*, which is the key to successful domain specific strategies applied in the software reuse field.

The term *domain* has different meanings in different contexts. The *Oxford Dictionary* [OxD12] defines a *domain* as:

"A specified sphere of activity or knowledge."

In mathematics, one definition of the term *domain* is a set of values for which a function is defined. In biology, a domain is a top-level taxonomic subdivision in the classification of organisms. In information technology, a domain can carry different meanings according to Computer Science's branches (*Domain Name System* in networking, *Data Domain* in databases, etc.). To summarize, the term *domain* indicates an area of knowledge consisting of a set of shared concepts and terminologies specific to that area.

In the field of software engineering, *domain* is characterized as a field of endeavor that represents knowledge by defining a set of entities for any software program, where the entities are usually common requirements, vocabularies, and functionalities gathered and built to solve problems specific for that field.

3.2.1 Types of Domain

In their research for developing the *Organization Domain Modeling* (ODM) method[5], which is one of the most popular methods in the domain engineering process, Simos et al. [SCK+96] distinguish between two types of *domain*:

- *Domain* as the *Real World*
- *Domain* as *Set of Systems*

3.2.1.1 Domain as the *Real World*

Figure 3.1 depicts the *Domain as the Real World* approach. Some researchers use this term to describe an area in the real world about which software developers have sufficient knowledge to develop software systems. The type of knowledge that software developers have about the domain in real world is *System Knowledge* about proper programming languages, design techniques, and other system requirements in order to construct a software application that serves that domain area [SCK+96].

Domain as the real world is used in the Object Oriented (OO) software engineering, Artificial Intelligence (AI), Expert Systems (ES) and Knowledge Engineering (KE) communities [SCK+96] and [CE00]. Consider a banking system as an example of a domain in the real world. While software developers have the *system knowledge* [SCK+96] about the banking domain, other stakeholders (e.g., domain experts, knowledge engineers, and end-users) have the *domain knowledge* [SCK+96] on how work is completed in the banking system.

[5] ODM method is presented in the next chapter

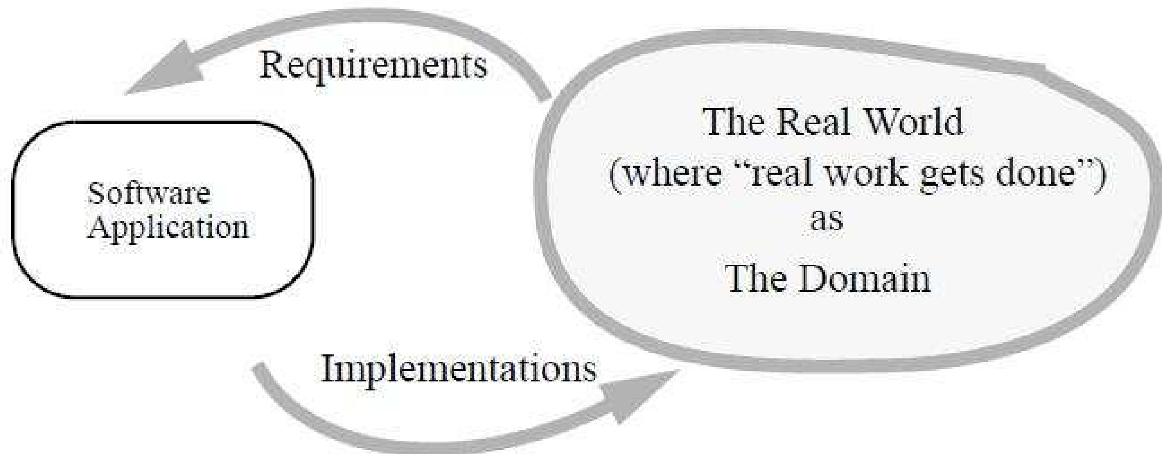

Figure 3.1: Domain as the *Real World* (from [SCK+96])

In the banking system domain, knowledge engineers and AI researchers collect information on how the banking system works (e.g., types of bank accounts, transactions) by interviewing domain experts. They then interpret the collected information and create a domain description that can guide software developers to construct the desired software system [SCK+96].

Domain descriptions can be performed through OO methods and techniques, such as the *Unified Modeling Language* (UML), to describe the bank domain's scenarios, requirements, and elements through models such as class diagrams and use-case diagrams. Domain descriptions structure the basis for implementing OO software systems to support real world domains.

3.2.1.2 Domain as *Set of Systems*

The other type of domain is *Domain as Set of Systems*. As shown in Figure 3.2, unlike the domain as the real world where a domain represents an area in which software systems are deployed, domain as a set of systems represents the software applications themselves.

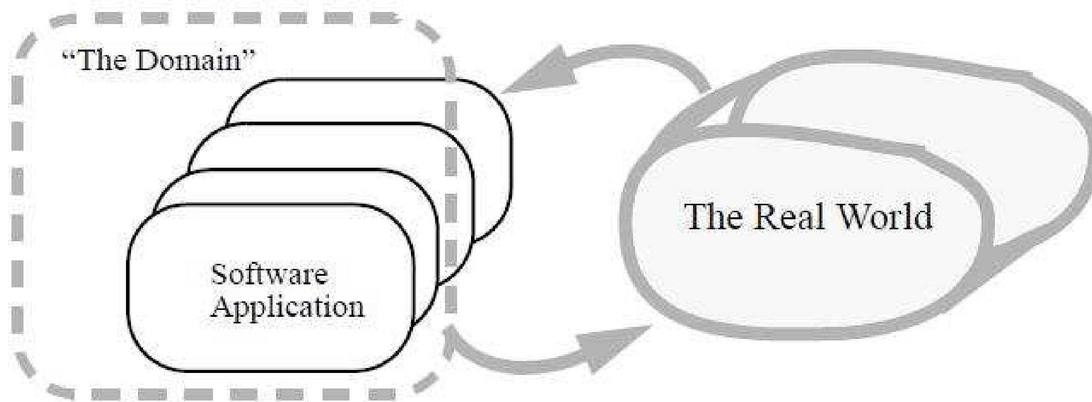

Figure 3.2: Domain as *Set of Systems* (from [SCK+96])

Domain as a set of systems describes a set of related software systems (a family of systems) that share common functionalities specific to a problem area. The software reuse community defines this type of domain using the domain engineering and domain analysis processes.

In the domain as a set of systems approach, knowledge engineers and AI researchers who collect information to build domain as the real world descriptions are called *domain analysts*[6].

[6] See Section 3.4.1.1 for further explanation

A *domain analyst* is a person who has specialized knowledge of a domain and has analyzed the target family of software systems. The domain analysts describe the requirement so that software developers implement software systems as members of that family [Nei89].

In the banking example, a domain analyst is the person who has experience in investigating the settings and requirements for banking systems. As knowledge engineers and AI researchers in the domain as the real world approach provide software developers with domain descriptions using UML, which includes actors and other elements used in use-case diagrams and class diagrams, domain analysts in the domain as a set of systems approach provide domain descriptions that include features, domain entities and their attributes, procedures, and relationships between software systems' components [SCK+96]. Domain descriptions in the domain as a set of systems type are usually performed using domain modeling techniques.

3.2.2 Domain Scoping

Where domain as a set of systems indicates a set of related software systems that share common properties, *domain scoping* is the activity of determining which software systems/components belong to a certain domain [CE00].

A complete software system may include several domains. For example, a software system may contain a *Database Domain*, *Data Structures Domain*, *Network Domain*, and *User-Interface Domain*. The scope of each domain focuses on a single functional area; there may be many functional areas implemented in the system. Therefore, Simos et al. [SCK+96] suggest two kinds of domain scope:

- **Horizontal scope** is a domain that contains *whole systems* that provide IT solutions for real world problems such as banking systems and library management systems

- **Vertical scope** is a domain that contains parts of systems that may appear in many different kinds of software systems such as database and networking systems

Simos et al. [SCK+96] propose different techniques to determine the scope of a domain, for example: *Innovative* versus *Native* domains. The Innovative domain technique is a software product line approach in which a software company defines a domain when it recognizes that it has a set of existing systems that share common functionalities, a fact that was previously unnoticed. The company can leverage commonalties among systems by making them reusable core assets available for constructing future projects.

Native domains are already recognizable as domains. A software company can determine which software systems/components fit in an already existing domain, or it can use the innovative domain technique to split a large domain into smaller and more focused domains that precisely scope certain areas of functionalities.

3.3 Software Product Lines

One of the most effective reuse approaches that focuses on reusing domain knowledge and has been adopted by organizations as a systematic reuse strategy is creating *Software Product Lines* (SPLs). Using software product lines is a leading strategy in systematic software reuse [Nor02] and is considered by many researchers the dominant approach currently applied in organizations [MYA+99].

Product Lining - as a general term - is an approach to producing a group of related products that share common features for sale. The notion of software product line is similar. It is a strategy based on understanding and capturing knowledge about a family of related software systems in a certain domain area to emphasize and discover common and variable parts; the common parts across the family are used to build a software platform [VH01], which serves as a baseline for all systems in the family while allowing variations among the family members in order to yield different products. Clements and Northrop define software product line as follows [CN01]:

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

Software core assets are not only restricted to source code fragments or previously implemented algorithms, but they also include reusable software components, requirement documents and specifications, design patterns, software architectures, and documentation.

According to [CN07], developing software systems using software product lines is similar to the activity of assembling parts together rather than constructing each part from scratch; the activity is a matter of integration rather than programming. In software companies that adopt the software product lines approach, each software system is developed by utilizing reusable software artifacts from the companies' common core assets, and each component has a built-in variability to yield different products. The process of assembling the collection of components is based on the product line's core architecture constructed using domain design techniques explained later in this chapter.

3.3.1 History of Software Product Lines

The basic concept of a software product line is not new; it was presented by David Parnas in 1976 as follows [Par76]:

> *"A set of programs constitutes a product line whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual members."*

Although the idea was presented in the mid-1970's, the actual construction of software product lines started in the early 2000s when software companies shifted their focuses on systematic reuse strategies [Nor08]. As mentioned in the previous chapter, software reuse practices have been adopted since the earliest days of programming, but in an ad hoc manner without following planned reuse methods. Since the 1950s, programmers have used source code fragments, subroutines, and other general software components gathered from already built systems and stored in reusable software libraries. Later, organizations started to shift their focus into more systematic reuse strategies, especially with the emerging field of software product line techniques. Inspired from [Nor08], Figure 3.3 shows the history of software reuse practices with software product line as the most recent reuse technique.

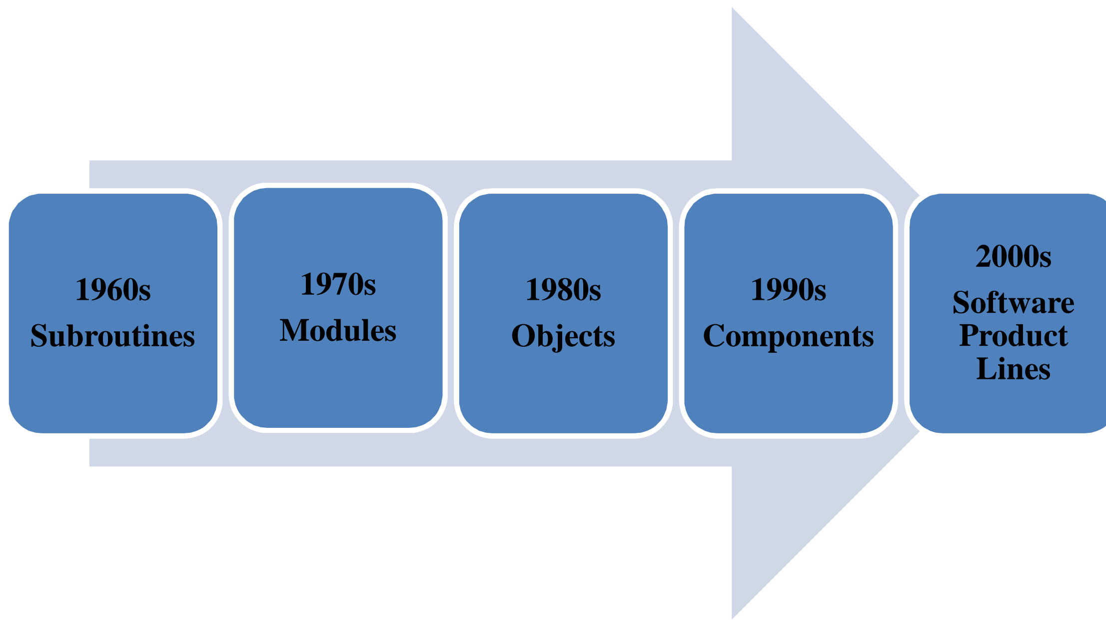

Figure 3.3: History of software reuse from ad hoc to systematic (from [Nor08])

3.3.2 Software Product Lines Advantages

The most important motivation for adopting software product lines is to leverage a set of reusable core assets from a specific problem area to develop future software systems. The cost of developing such reusable assets is distributed across all the systems in the family. In the development of single systems, each system requires its own production plan [CE00], which is basically the system's development lifecycle task. By using software product lines, the development tasks that include domain analysis, design, and implementation[7] are spread across all members of a software family. Therefore, both the cost and effort of the development process are amortized across all systems in a software product line instead of each system requiring a complete development process.

[7] Domain analysis, design, and implementation are the primary phases of Domain Engineering process (see Section 3.4.1)

Charles Krueger refers to the advantages of using software product lines as:

> *"tactical engineering benefits that are often large enough to have an impact well beyond the borders of the engineering department, offering strategic competitive benefits to the way that a company conducts its business which can be translated into a very powerful set of strategic business benefits."* [Kru06].

The advantages of software product lines are similar to the general software reuse process advantages presented in the previous chapter with one exception. Software product lines extend the reuse process to cover all aspects of the software development lifecycle. According to [CN01] and [Kru06] strategic business benefits that software product lines offer are:

- Improvement in product quality
- Reduction in development and maintenance effort
- Faster time-to-market and time-to-revenue
- Reduction in development costs and risks
- Higher customer satisfaction
- Help in finding and erasing redundant implementations

The software product line approach has been successfully applied in organizations in several application domains. Numerous case-studies are presented in literature, including the two summarized below:

- ***Cummins Inc***., the largest producer of diesel technology in the world as of 2012 with more than 20 product groups and over 1000 separate engine applications. *Cummins* adopted a software product line strategy for its real-time embedded diesel engine controls. Within a few years, the result was a reduction in the product cycle time from 250 person-months to a few person-months[8]. The building and integration time was reduced from almost a year to one week; productivity increased with higher customer satisfaction. The company's quality goals surpassed its expectations, which enabled the company to dominate the industrial diesel engine market [Nor02], [Dag00].
- ***Nokia**,* one of the world's leading corporations for manufacturing mobile telephones. The company manufactures different phone series with variable features such as screens, keyboards, and interfaces that support 58 languages for around 130 countries [Nor02]. Due to variability among the company's products, *Nokia* adopted a product line to create a common set of core assets for both software and hardware parts, and used the variations between products to customize and produce different products. This approach allows *Nokia* to produce more than 30 different phone models each year, where it previously produced only 4 to 10 models [Cle10].

Other case studies of the impact of adopting software product lines are presented in [CN01].

[8] Person-Month is a measurement of a person's effort working full-time on a specific task for one month. For example, if a project requires four months to be finished with four full-time employees working on it, the person-month is calculated as: $4 \times 4 = 16$ person-month effort.

3.3.3 Software Product Line Principles

As already mentioned, the goal of adopting software product lines in organizations is to construct software systems related to a particular domain area with the least amount of effort while maintaining high product quality.

The process that delivers reusable software artifacts that can be used to instantiate software products in a certain domain is called the Software Product Line Engineering (SPLE). It consists of two main parts: *Domain Engineering* and *Application Engineering* [PBL05].

3.4 Software Product Line Engineering

Software Product Line Engineering (SPLE) is an approach for developing lines of software products. A line of software products consists of a set of related software products that share common characteristics (e.g., components, requirements, and architectural designs) in addition to variable parts that differentiate one software product from another. In [PBL05], the SPLE process is defined as:

> *"Software Product Line Engineering is a paradigm to develop software applications (software intensive systems and software products) using platforms and mass customization."*

Following the definition, software product line engineering supports both intensive systems and stand-alone systems. Intensive systems are embedded systems in which a software system interacts with other software systems or hardware devices (e.g., a bank ATM). Developing both stand-alone and embedded systems through software product line engineering requires combining *mass customization* with platforms [PBL05].

Joseph Pine [Pin93] defines mass customization as:

> *"A large-scale production of goods tailored to individual customers' needs."*

Large-scale production indicates producing large quantities of end-products through an assembly line process, a process in which interchangeable components are assembled to produce a complete product. An example is an assembly line used for a particular car series. A car is manufactured by assembling interchangeable components (e.g., engines, body, and wheels) specific to that type of car. An assembly line enables each car to be built more quickly than doing so by hand.

Mass customization attempts to exploit mass production to keep the costs low but still to enable individual products to be customized for individual customers. Software product line engineering requires the developers to manage the variability among a set of related systems in a specific domain area so that they can build different systems to satisfy different customers' orders.

Following [ML97], platforms are defined as:

> *"A set of subsystems and interfaces that form a common structured from which a stream of derivative products can be efficiently developed and produced."*

In the software product line engineering process, developers must plan for reuse by building core assets designed specifically for reuse in a specific domain area. These reusable assets comprise a platform, a baseline to develop software systems.

The combination of *common platforms* (managing commonalties) and *mass customization* (managing variabilities) represents the objective of software product line engineering process [PBL05].

In the literature, many researchers (e.g., [PBL05], [Lin02], [CE00], [BEG12], and [WL99]) agree that the process of software product line engineering can be divided into two sub-processes: *Domain Engineering* and *Application Engineering*.

- Building a common platform (reusable software assets) represents the first process. (*Domain Engineering*)
- Building customer specific applications (mass customization) represents the second process (*Application Engineering*)

Figure 3.4 shows the "*Three Essential Activities for Software Product Lines*" as described by Clements and Northrop [CN07]. It depicts the relationship between the domain and application engineering sub-processes in terms of managing the development of both core assets and customer specific applications.

Managers must define the business goals, control the risks involved in the family-oriented development, analyze the potential market for products in the family, and keep the work within budget and one time.

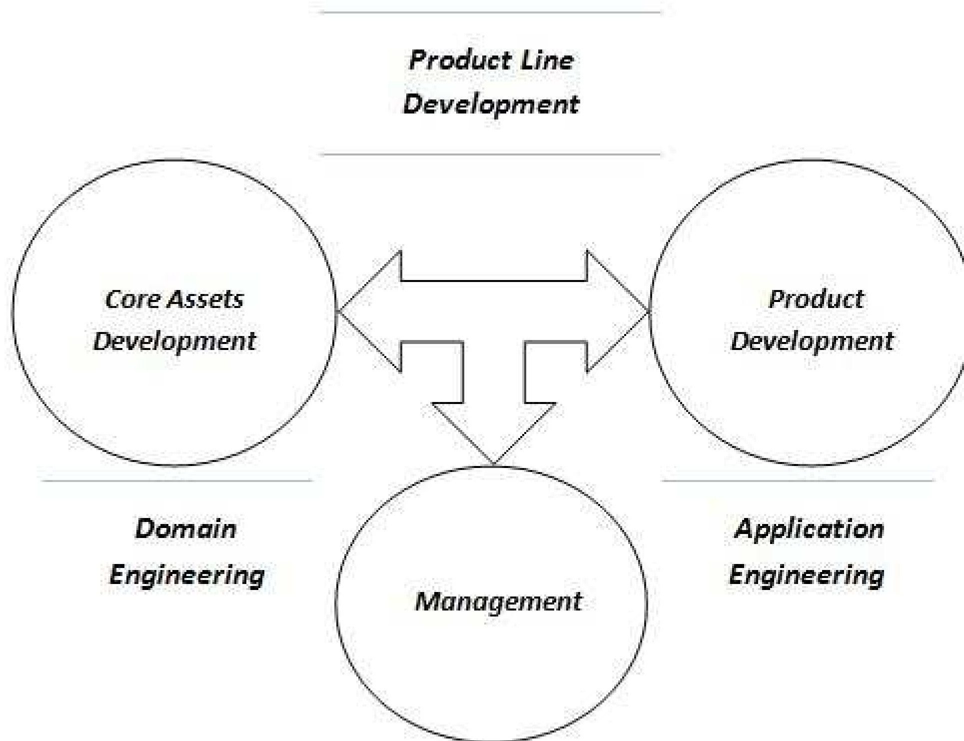

Figure 3.4: Domain engineering and application engineering as two sub-processes of software product line practice (from [CN07])

3.4.1 Domain Engineering

As defined in Section 3.2, domains are areas that group a particular set of systems or parts of systems together. Most software systems are developed to perform tasks related to some business areas (e.g., banking systems) and can be categorized based on their areas of functionality (e.g., Database systems). Business areas and areas of functionalities represent domains where a set of systems are grouped and dedicated for finding solutions specific to those areas.

Software systems deployed to provide solutions in a specific domain share common features and have similar development lifecycles. A software company that has built several software systems within a domain can capture its knowledge about the domain and use it to develop a family of systems.

The concept of domain engineering is similar; it captures the domain knowledge acquired from building several similar software systems in the form of reusable software assets and uses those assets for developing new software systems within the domain. This speeds the production process.

In the literature, several definitions of the domain engineering process are proposed. All describe the process as a method that systematically reuses domain knowledge [PBL05], [KCH+90], [Har02-A]. Czarnecki and Eisenecker [CE00] define the process of domain engineering as:

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

As mentioned in the previous section, the software product line engineering process consists of domain engineering as the process of developing reusable core assets and application engineering as the process of developing individual products. Domain engineering consists of three main phases: *Domain Analysis*, *Domain Design*, and *Domain Implementation* [CE00]. The following sections introduce each phase. They give more attention to *domain analysis*, which is the main topic of this thesis.

3.4.1.1 Domain Analysis

Domain Analysis (also called *Product Line Analysis*) is the process of analyzing a set of related software systems in a domain to find out what common features they share and what variable features differentiate them from each other. The term *domain analysis* was coined by James Neighbors in his PhD thesis "*Software Construction Using Components*" [Nei80] which explored the concepts of a family of software applications developed within an area of functionalities in 1980.

Domain analysis process originated from software reuse research [DK02]. It was proposed as a method for identifying and gathering information from a set of software systems that share some common features; it then organizes and represents that information in a meaningful way in order to reuse it again for building new software systems [Pri90].

There are two main purposes for performing the process of domain analysis according to [CE00] and [Pri90]:

- Selecting and defining the domain scope
- Acquiring appropriate domain knowledge by collecting information related to the domain scope and integrating it into the domain model

Selecting and defining domain scope requires performing business and risk analysis to select a domain that satisfies the company's objectives and goals. Information related to a specific domain is collected from different sources such as existing systems, stakeholders, domain experts and analysts, textbooks and standard documents, and experiments and prototypes.

Domain analysis is performed by domain analysts. As pointed out in Section 3.2.1.2, domain analysts collect information about a certain domain area in order to provide domain descriptions for software developers. James Neighbors [Nei89] defines a domain analyst as:

> "*A person who examines the needs and requirements of a collection of systems which seems similar.*"

James Neighbors insists that this work can only be performed successfully by a domain analyst who has the experience of building many software systems for different customers in the same domain. In contrast, a system analyst has the experience of developing different software systems in different domains, instead of being restricted to a specific problem area. Therefore, domain analysts have the domain knowledge appropriate to provide software developers with domain descriptions since their work focuses on related systems in a certain domain.

The domain analysis phase produces a *domain model*, a conceptual model that captures the ideas of the problem domain by representing common and variable features of software family members in the domain.

The domain modeling process produces several models and consists of several activities. These activities involve defining the domain scope and vocabulary, describing the domain concepts and their corresponding attributes, identifying relationships among the features and concepts, and describing the dependencies and constraints among the variable features. The literature varies in how it identifies the domain model's elements. However, the following are the most commonly identified elements of the domain modeling process:

- ***Domain scoping:*** As mentioned in Section 3.2.2, domain scoping is the activity of determining the boundaries of a domain by finding out which systems and components belong to it. Domain scoping also gives examples of software applications already in the domain and outside the domain, gives rules of inclusion and exclusion in order to find out which software systems belong to the domain, and determines external systems that may interact with the domain [SCK+96], [CE00].
- ***Domain lexicon:*** A domain lexicon is a data dictionary that defines the domain vocabulary and terminology to make the communications easier between programmers and stakeholders. The domain information can be obtained from different sources such as textbooks, already existing system designs and applications, surveys and articles written by domain experts, and documented requirements and manuals [KCH+90], [SCK+96].
- ***Commonality and variability analysis:*** Commonality is list of assumptions that is true for all members in the family, while variability is a list of assumptions that is true only for some members in the family [CHW98]. This activity is performed by domain analysts using feature modeling techniques, which are presented in the following chapters.
- ***Notations:*** Notations are used to visually represent domain concepts and their attributes, relationships between concepts**,** and dependencies between them. This activity is performed using a wide variety of well-known models. Kang et al. [KCH+90] present several models used in domain analysis. Briefly, these are:
  - *Feature models*: The fourth element of domain engineering process (described in detail in the next chapter).

- *Context models*: Usually used by requirements analysts to determine if a particular application ordered by a customer is within the domain boundary for which products related to the domain are available.
- *Data flow-models*: Usually explain how data is processed and show how data-flows between the selected domain and other domains and how they communicate.
- *Entity-Relationship models*: Usually used by requirements analysts to gain knowledge about a domain's entities and their inter-relationships.
- *Architectural models*: Usually used by domain designers for designing software applications.

- ***Feature modeling:*** The most important output of the domain analysis phase is the feature model. A feature model is a representation of all related software applications (family of systems) in a domain. In the context of a software product line, the term *Feature* is defined as "a *characteristic of the system that is visible to the end user*." [KCH+90]. In a product line, each member is defined by a set of features that differentiate it from other members in the family of products within the domain. Features in feature models are represented at the highest level of abstraction. Users prefer to understand a particular product in the form of user-visible aspects, avoiding the complex details of internal functions that other models, such as data-flow diagram, show. Once a particular application is defined by domain scoping, requirements analysts use feature models to discuss the application's features with users.

Since most of the work in the domain analysis phase is performed during the domain modeling process, some authors prefer to call this phase the *domain modeling* phase [Har02-A]. Domain modeling using feature models will be presented in the following chapters.

3.4.1.2 Domain Design

The second phase of domain engineering is domain design. This is the activity of mapping the configurable requirements and the domain model generated from the domain analysis phase to technical solutions. This can be performed by creating a common product line architecture, which provides a common, high-level structure for the family of software systems in the target domain [CE00].

Domain design involves the selection of the architectural style [Har02-A] that specifies the design rules for software products in the product line. The designed architecture is represented using view models such as a *4+1 Model* [Krc95] for architectural representation and modeling software components. The domain architects design the common architecture to determine how requirements are reflected in the architecture and specify how the software components will be connected, as well as relationships, interactions, and dependencies among them. It also shows how external (user-visual) variability between software members are represented [PBL05], and how products are configured.

Following [CE00], a production plan should be developed after developing the product line architecture. A production plan illustrates how concrete software applications can be assembled using the common architecture and components in the domain, provides systems interfaces for customers to order concrete systems, and describes the process of custom development for managing customers' requests.

Concrete software applications can be assembled manually or automatically, where software applications can be entirely generated using techniques such as generative programming through generator tools [CE00].

3.4.1.3 Domain Implementation

The third and final phase of domain engineering is implementing the reusable software components and their interfaces in addition to the generic architecture designed during the domain design phase [CE00]. All of those implemented artifacts are used in the *application engineering* process for building concrete systems. Tools such as generators for automatic components assembly or domain specific languages are used to implement components. As shown in Figure 3.5, if products created for customers need additional features based on customers' requests, then custom development has to be carried out with implementation tools.

3.4.2 Application Engineering

The second part of software product line engineering is the application engineering process, which involves building concrete systems based on the results produced from the domain engineering process [CE00]. Similar to domain engineering, the application engineering process, as shown in Figure 3.5, consists of three main phases: *Requirements Analysis*, *Product Configuration* and *Integration and Testing*. They operate in parallel with domain engineering phases.

In Figure 3.5, customer needs are reviewed and analyzed during the requirements analysis phase of application engineering, while during the domain analysis phase of domain engineering, domain knowledge is gathered and the domain model is produced. The domain model represents the key concepts of the problem domain as well as attributes, roles, relationships, and constraints among the entities of software systems in the domain. Features from the existing domain model generated in the domain analysis phase are selected as requirements when matching customers' orders**.** Domain engineering provides domain models for a family of applications, but the requirements analysis phase in application engineering process analyzes the requirements for the concrete systems requested by users.

Requirements in the application engineering process are presented as features that, in addition to the common domain architecture, are used to build a product configuration. The domain design phase produces the common product line architecture that specifies the design rules for products in which different product configurations produce different products in the corresponding product line.

Organizations can also take advantage of existing domain models and requirements if their features match customer requests, and if applications are generated based on existing reusable components. Otherwise, if new requirements do not match existing domain models, a custom development is required and the entire process is repeated.

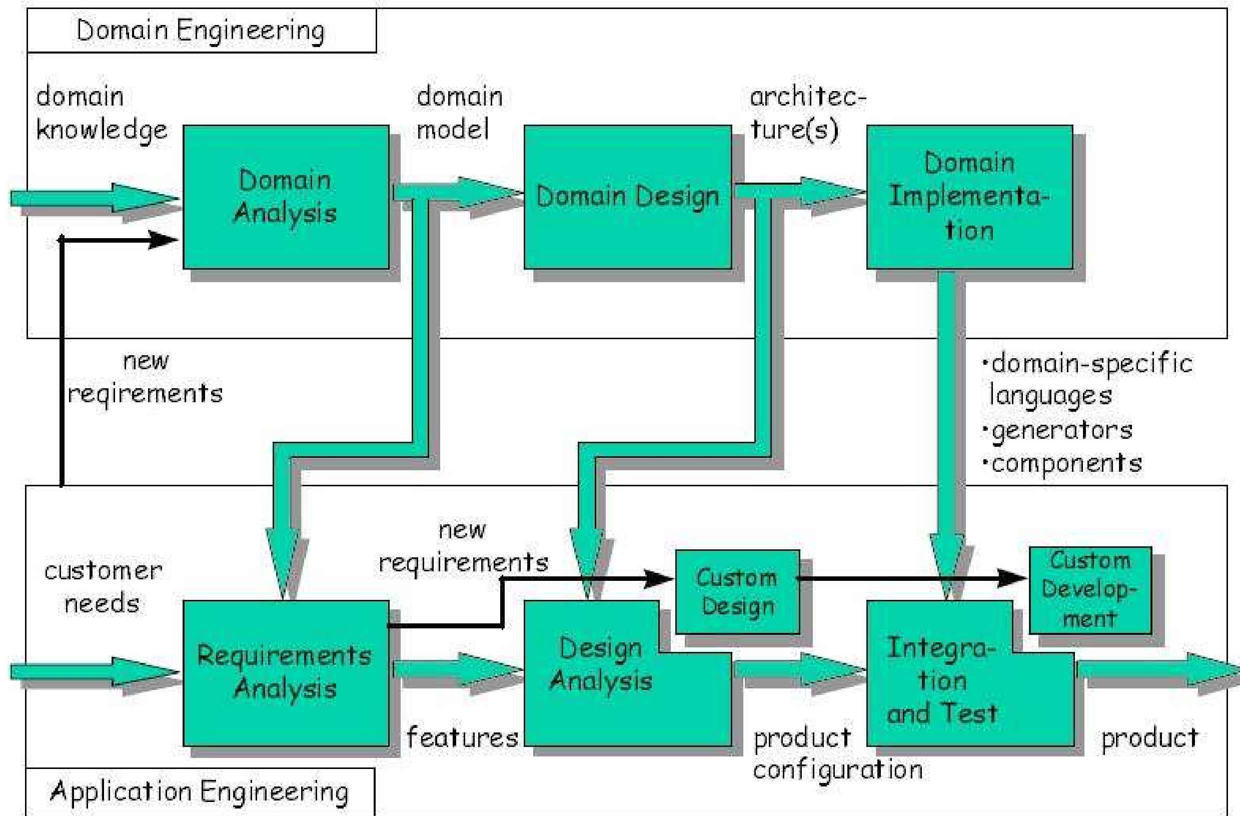

Figure 3.5: Domain engineering and application engineering processes (from [CE00])

In summary, domain engineering considers all possible software systems in the domain while application engineering uses the results of domain engineering to create concrete systems based on customers' requirements.

3.5 Software Product Lines versus Other Software Methods in Software Reuse Research

As pointed out in the previous chapter, there are several approaches and methods used in software reuse research. For example, Charles Krueger presents eight categories of software reuse approaches in [Kru92] and distinguishes between opportunistic and planned (systematic) reuse strategies. As a conclusion of his survey, Krueger indicates that the most effective approaches that focus on reusing domain knowledge are being used, especially with the emerging field of software product lines which focuses on reusing domain knowledge.

The first part of this section briefly discusses the main differences between software product lines and methods that support single system developments (one-of-a-kind production [CE99-A]), such as *Object Oriented Analysis and Design* (OOAD) methods. The second part introduces *Component-Based Software Engineering* (CBSE), a process for managing reusable software components, and whose basic ideas might conflict with the concept of software product lines.

3.5.1 Software Product Lines versus Single Systems Development

Single systems development is the process of developing software systems individually. When developing software systems that share common features with those that already exist, software developers usually use some of the same software parts (e.g., requirements specification, design patterns and code fragments) that they used on previous projects, modify those parts if necessary, and integrate them into new systems.

Although a software developer can reuse and modify software parts from a previously implemented project, the new project will still be different than the old one; for example, each project would has its own development and maintenance cycle yielding two different products with no common base between them [CN07]. Software companies use a similar approach for developing single systems with a reuse category by using *Object Oriented Analysis and Design* methods.

In software engineering, Object Oriented Analysis and Design (OOAD) is an approach that models software systems as interacting objects so that the behavior of a software system is the result of its objects collaborating with others. OOAD is a design strategy; it's different than *Object Oriented Programming* (OOP) as OOAD is not restricted to any specific implementation languages. In the context of software reuse, though, the OOAD approach supports the reuse process by providing object oriented abstractions such as inheritance - a way to reuse code by specializing classes for specific software applications; however, it's still not considered as an effective and efficient approach for software reuse as software product lines.

In their definitive analysis on software product lines and domain engineering process, Czarnecki and Eisenecker [CE00], [CE99-A], and [CE99-B] listed several deficiencies of OOAD methods in comparison to the domain engineering process in the context of strategic software reuse. Some of their points are summarized below.

- ***Planning for reuse.*** In software product line engineering, there is a distinction between domain engineering (development for reuse) and application engineering (development with reuse).

The scope of domain engineering is the system family within a domain that allows the development of reusable software assets to be used in the application engineering process. The OOAD approach is close to the application engineering process, in which the process of developing reusable assets from the start (domain engineering process) is not included because the scope of development for reuse in the OOAD approach is a single system.

- ***Domain scoping*.** Since OOAD methods focus on single systems development without building reusable core assets, the OOAD approach lacks a domain scoping activity used in the domain engineering process. In software product lines, the domain is defined during the domain scoping activity to determine which systems belong to the domain and which do not, in addition to identifying stakeholders' goals. Therefore, the domain engineering process focuses on satisfying both current and future potential customers with systems related to the domain.
- ***Modeling variability*.** The most important problem Czarnecki and Eisenecker mention about OOAD methods is their inadequate modeling of variability. The weakness of OOAD modeling techniques in comparison with software product lines is the lack of an abstract and concise model of commonalty and variability. In the OOAD approach, object oriented modeling mechanisms, such as classification, aggregation, and inheritance, are used to support modeling of variabilities. The same variable feature might be implemented using different variability mechanisms through models such as UML class diagrams. In contrast, the domain engineering process uses feature models to represent common and variable features across different software systems in a certain domain area.

Feature models are more abstract, concise, and represent the concept (a system or a product) and its features in the highest level of abstraction. Another weakness of OOAD methods is that when representing variation points using a UML class diagram, a decision should be made from the start of drawing the diagram to select what implementation mechanism is used to represent a variability (e.g., inheritance, aggregation, and classification). The next chapter will present feature modeling in detail. A comparison between feature models and UML diagrams is conducted through the final Chapter to show the differences in modeling variabilities between OOAD and software product lines.

In conclusion, software product line engineering reuses assets that are developed explicitly for reuse through the domain engineering process, a process that the OOAD approach does not support. In product line engineering, each product is built from the reusable core assets in addition to other variable parts for differentiating each product from another. A product line is viewed and maintained as a whole, not as a set of multiple products that are viewed and maintained separately, as in most of the OOAD techniques.

3.5.2 Software Product Lines versus Component-Based Software Engineering

As presented in the previous chapter, a software component provides a subset of functionality for a particular system.

Due to the growth in computer power as software systems are deployed in all aspects of life, software companies started to focus on constructing complex projects by assembling them from reusable software parts or components, rather than starting from scratch. *Component-Based Software Engineering* (CBSE) is a reuse-based approach that focuses on providing an assembling technique for integrating independent software components into constructing complex projects.

According to [Pre05], the basic idea behind CBSE is following Fred Brooks's suggestion on "*Buying versus Building*" software [Bro87]:

"The most radical possible solution of constructing software is not to construct it at all."

Reusable software components used in the CBSE approach are usually *Commercial Off-The-Shelf* (COTS) components[9]. After establishing a system's requirements following the common system development requirements and analysis techniques, the CBSE approach aims to develop a generic architecture to set up the system's structure. During the implementation phase, instead of developing the system's elements from scratch, software developers check the requirements to find out what subsets of functionalities (components) will properly fit for composition, rather than following the construction process. If some system requirements cannot be implemented using COTS components, software developers modify or, in some cases, and delete those requirements to avoid custom implementation in order to save both cost and time [Pre05].

[9] Discussed in Section 2.5.2.2

Although the process of component-based development mostly depends on buying independent software components, custom developments are required in some cases: for example, in the case of system requirements that are necessary and cannot be modified or deleted to match COTS components or if no COTS components are available to cover a subset of functionality for a particular system. Therefore, software developers follow conventional software engineering methods to develop new software components to meet a system's requirements.

The main difference between the standard CBSE approach and software product lines is that in SPLs all software components are specified through the product line architecture [CN07], which includes both the common and variable components designed and developed during the domain engineering process and provides the application engineering process with those components required to generate concrete applications. In the CBSE approach, system requirements and architecture define software components (COTS or reusable components developed in-house) and the design rules for them, while software developers are responsible for adapting those components to fit properly with the system's architecture and [CN07]. If the components are incompatible with the design rules determined through the architecture, they will be either modified or discarded and replaced by proper components [Pre05], which is basically following the cutting and filling procedure [CE00]. This drawback clearly explains why CBSE is not considered an effective approach in the context of strategic software reuse as software product lines. SPLs specify software components based on the architecture and the production plan which guarantee that each component follows the design rules provided by the architecture.

Furthermore, software product lines provide built-in variability techniques (e.g., feature modeling) in the components in order to use them in specific concrete systems. On the other hand, the CBSE approach not only lacks such methods, but also lacks the technical and management aspects[10] that are primary features of software product lines [CN07].

3.6 Summary

Throughout this chapter, we defined a *domain* area in software engineering as a field of endeavor that represents knowledge by defining a set of entities for any software program. We introduced two types of domain: domain as *The Real World*, which is an approach that describes an area in the real world, such as a banking system, in which software programs are built to provide solutions for it, and domain as *Set of Systems*, which is an approach that represents the idea of software product lines by describing a set of related software systems that share common functionalities specific for a domain. We introduced the domain scoping activity of determining which software systems/components belong to a certain domain area.

The second section presented the software product line process. First, we defined the process; then we surveyed the history of the process and listed strategic business benefits that the process offers along with examples of organizations that have adopted software product lines in their projects.

[10] See Figure 3.4 in Section 3.4

The third section introduced software product line engineering, which is an approach for developing lines of software products through domain and application engineering sub-processes. We explained that domain engineering works as a process of developing reusable core assets by capturing the domain knowledge acquired from building a number of software systems that share similar characteristics and requirements in a certain domain area. We discussed three phases of domain engineering process: analysis, design, and implementation domains and explained how domain engineering phases function in parallel with the application engineering process for developing concrete applications from the shared set of software assets, while also managing variable software parts to produce a different software product based on customers' requirements.

The fourth and last section conducted two comparisons. The first comparison illustrated the differences between software product lines as a method of supporting system family development with single system development methods, such as object oriented analysis and design approach. The second comparison presented the component-based development approach and discussed the main differences between adopting it as a software reuse method with software product line engineering process.

CHAPTER 4

DOMAIN ANALYSIS AND FEATURE MODELS

When we introduced software product line engineering in the previous chapter, we mentioned feature modeling as the software product line's most important property. Feature modeling provides software product lines with properties such as the ability to represent variability in software systems in an abstract way. Such properties make software product lines a more appealing strategy for software companies than other approaches to systematic software reuse. The feature modeling process produces feature models, a key output from the domain analysis phase. While software product line engineering is a strategy based on understanding and capturing domain knowledge about a set of related software systems (a family of systems), feature modeling is used to specify and model those systems in terms of features. A feature model captures the results of domain analysis and distinguishes among members of a product line by representing their common and variable features [CK05]. Managing and representing common and variable features form the basis for configuring systems in a product line [BNG+12]. Feature models provide software developers with abstract representations that specify all possible configurations of software family members, with each configuration representing a fully formalized specification of a software product in the product line.

4.1 Introduction

Domain analysis is the first phase of domain engineering process. It analyzes a set of similar software systems to identify, capture, and organize information used in developing those systems in order to build reusable software artifacts.

The domain analysis phase generates domain models. Domain models systematically exploit commonality and variability across systems in a domain using the feature modeling process. It helps software developers to develop reusable components as core assets. These can then be used to create new systems that share common assets but enable variations in the features present in individual software products. The process of developing reusable core assets for a family of systems is the domain engineering process (see Section 3.4.1); the process for developing individual systems that use those reusable core assets is the application engineering process (see Section 3.4.2) [CHE05].

Since domain analysis was first introduced by James Neighbors [Nei80], several researchers have proposed methods to enhance its modeling techniques.

The next section introduces *Feature Oriented Domain Analysis* (FODA), a domain analysis method best known for introducing the feature modeling process. The third section briefly describes other domain analysis and engineering methods.

The fourth and the fifth sections describe the feature modeling process. They define features in the context of domain analysis and engineering and explain the key elements of feature models expressed using feature diagrams.

4.2 Feature Oriented Domain Analysis (FODA)

FODA is a well-known domain analysis method that was developed by the *Software Engineering Institute* (SEI) of Carnegie-Mellon University. In 1990, Kang et al. [KCH+90] described the FODA concepts and introduced feature modeling. Feature models use *feature diagrams* to model common and variable features for a set of software systems in a domain.

FODA has contributed significantly to the development of the domain engineering process [CE00]. A number of other methods in domain analysis and engineering research (e.g., Feature Oriented Reuse Method (FORM) [KKL+98]) build on or extend and improve FODA's feature modeling techniques (e.g., [CE00], [ESJ11], [RBS+02], [GFD98], and [CK05]).

4.2.1 FODA's Concepts and Activities

The primary objective of the FODA method is to capture and represent the commonalties and variabilties found in related software systems in a domain. It represents the user-visible features using feature models. Most of FODA's activities are also similar to domain analysis's phases (see Section 3.4.1.1).

Kang et al. [KCH+90] defined FODA as having three phases: context analysis, domain modeling, and architectural modeling.

Subsequently, the architectural modeling phase was absorbed into the domain design phase of domain engineering [CE00], which takes the domain model (as an output of the domain analysis phase) and creates a generic architecture for the product line. The following two sections describe FODA's two phases: context analysis and domain modeling phases.

4.2.1.1 Context Analysis

As we discussed in the previous chapter, a prerequisite for creating a domain model is performing context analysis (also called the domain scoping activity). Context analysis is the process of defining the scope of a domain and determining its boundaries by identifying which systems or parts of systems belong to the domain. Defining the scope of a domain requires performing business and risk analysis to choose a domain that meets a company's goals [CE00]. Context analysis is usually performed by a domain analyst who interacts with stakeholders (e.g., domain experts, users) for the purpose of identifying the sources of domain knowledge, collects information from those sources to describe domain contents to define the domain's boundaries, and establishes the scope of the domain [SCK+96].

Domain knowledge is collected from sources such as existing systems, stakeholders (e.g., users, domain experts, and domain analysts), textbooks and standard documents. The results of context analysis are documented in a context model [PC91]. The context model is visually represented by a context diagram, which is a data-flow diagram that shows data-flow between the candidate domain and entities that the domain interacts with. More about FODA's context models is discussed in [PC91] and [KZ96].

4.2.1.2 Domain Modeling

After defining the domain scope in the context analysis phase, the domain modeling phase aims to determine and model common and variable features that characterize software applications in the domain [KZ96]. The process helps to understand software application behaviors and functionalities by systematically modeling their objects, functions, relationships, and dependencies within the domain.

Following [KZ96], FODA's domain modeling process involves four activities: domain dictionary, information analysis, feature analysis, and operational analysis.

- ***Domain Dictionary***

This activity, also called *domain lexicon* (see Section 3.4.1.1), is the activity that defines the domain terminologies (e.g., textual definitions and descriptions of software systems' features within the domain). In doing so, it makes communication easier between software developers and stakeholders by providing stakeholders clear descriptions of features, entities' terms, and acronyms. Sources for domain terminologies include requirements documents, stakeholders (e.g., discussions with domain analysts and experts), textbooks, surveys, and articles written by domain experts [KCH+90], [SCK+96].

- ***Information Analysis***

Information analysis is the activity that defines the data requirements and domain knowledge necessary to build applications in the domain. It explicitly represents them in the form of domain entities and their attributes, relationships, and dependencies.

According to [KZ96], domain knowledge refers to contextual information that gets lost after development or information that is deeply embedded in the software. Analysts represent domain entities using notations such as entity-relationship models[11]. The resulting information model (similar to concept model in domain analysis process) is then used by requirements analysts and domain designers to make sure that the domain's data abstraction and decompositions are used appropriately for developing software applications [KZ96].

- ***Feature Analysis***

In FODA, features are user-visible or characteristics of the domain [KCH+90]. Features in feature models represent the attributes of a concept (a family of systems in a domain) that stakeholders (e.g., customers, end-users) rely on to select products from a product line. The goal of feature analysis is to capture the customers' knowledge of the capabilities of software applications within the domain and to represent system features in a meaningful way. Features of a family of software applications in the domain are represented using a feature model, a graphical notation that depicts all possible configurations of a system product line having those features. We will present this activity in greater detail in Section 4.3.

- ***Operational Analysis***

The operational analysis activity shows how applications within a domain work by representing the behavioral and functional relationships between entities in the information model (the output of the information analysis activity) and features from the feature model (the output of the feature analysis activity).

[11] Some authors refer to information analysis activity as Entity-Relationship modeling ([KCH+90] and [PC91])

The difference between feature models and operational models is that feature models represent a software application's common and variable features in an abstract way so that is easy for end-users to understand. Operational models, on the other hand, focus on describing the specifications of software applications' functionalities and behaviors in terms of applications' inputs, outputs, and internal data [KCH+90].

The *Feature-Oriented Reuse Method* (FORM) is a systematic domain engineering method that extends FODA to support architectural design and object-oriented components. FORM attempts to capture the common and variable characteristics in terms of features for a set of related software systems within a domain. The result helps analysts to define domain artifacts (through the domain engineering process) used in developing applications [KKL+98].

FODA represents common and variable features of systems using feature diagrams, which bridge the gap between the requirement and design phases. FORM shows how to implement reusable components from the FODA feature models [Mat04]. A full description of the FORM method is presented in [KKL+98].

4.3 Other Domain Analysis and Engineering Methods

In addition to FODA, several domain analysis and domain engineering methods exist. This section briefly describes three other popular software reuse methods.

4.3.1 Organization Domain Modeling (ODM)

Organization Domain Modeling (ODM) is a systematic approach to domain analysis and engineering processes developed by Mark Simos at *Organon Motives Inc.* [Sim95]. ODM is a configurable method that can be integrated with other software engineering processes [AR07]. The method consists of three phases: *Plan Domain*, *Model Domain*, and *Engineer Asset base*.

- The *Plan Domain* phase aims to define the scope of a domain for the target project that satisfies both the project's objectives and the project stakeholders interests [AR07]. This phase matches FODA's context analysis phase presented in Section 4.2.1.1.
- The *Model Domain* phase gathers and documents domain information and generates a domain model. It describes systems in the domain using the domain definition provided by the plan domain phase. The domain model describes the common and variable features of systems in the domain to show the possible configurations of individual systems [SCK+96]. This phase matches FODA's domain modeling phase presented in Section 4.2.1.2.
- The *Engineer Assets Base* phase generates the architecture for the software systems in the domain and implements the reusable assets [SCK+96].

Although ODM shares some common aspects with the FODA method, ODM provides several unique features [CE00]. In comparison to domain engineering process discussed in Chapter 3, the first and second phases of ODM typically are domain analysis work (Section 3.4.1.1), while the final phase of ODM matches domain design (Section 3.4.1.2) and domain implementation (Section 3.4.1.3) works. For more detail about ODM method, a complete guidebook for its most recent version (ODM version 2.0) is presented in [SCK+96].

4.3.2 Domain Specific Software Architecture (DSSA)

Domain Specific Software Architecture (DSSA) is a domain engineering method developed by the *Advanced Research Projects Agency* (ARPA), an agency of the US Department of Defense [Tra95]. The DSSA method aims to build a generic architecture for software systems in a domain that reflects the systems' requirements, provides a common, high-level structure for systems in the target domain, and specifies the design rules to create the final architecture for each system. Hayes-Roth [HaR94] defines the method as:

> "*An assemblage of software components, specialized for a particular type of task (domain), generalized for effective use across that domain, composed in a standardized structure (topology effective for building successful applications.*"

According to [Tra95], the DSSA approach consists of three phases: *domain model*, *reference requirement*, and *reference architecture*.

- The *Domain model* phase analyzes an application domain to produce a domain model. The domain descriptions in this model specify the systems' behaviors so that developers can develop software applications effectively within the domain. The domain model phase also creates a domain dictionary to define the domain's terminology and its entities and the relationships among them using modeling techniques such as Entity-Relationship diagrams, context diagrams, and object models. The domain model also provides scenarios that describe the dataflow and control flow among the domain's entities [Tra95]. The domain model phase for DSSA is equivalent to the domain dictionary and information analysis phases for the FODA method and model domain for the ODM method.
- The *Reference Requirements* phase defines the functional and nonfunctional requirements for software systems in a domain. Functional requirements (also called the application architecture of a system) include the functions and tasks that a system does. Those functions and tasks specify the behavior of that system (e.g., data manipulations). Nonfunctional requirements (also called the technical architecture of a system) are global constraints on a software system, such as performance, reliability, robustness, and portability.
- In the final phase *(Reference Architecture)*, the domain architect uses the output of the reference requirements phase to design reusable and extensible reference architectures for systems in the domain. This phase corresponds to the domain design phase of the domain engineering process discussed in Section 3.4.1.2. The reference architecture phase consists of four activities [Tra95]:

- An *Architecture schema* that describes the software components in a DSSA.
- A *Reference architecture model* that represents the components' dependencies, relationships and constraints and provides descriptions for their interfaces.
- A *Configuration decision tree* for instantiating software products from the reference architecture. This activity corresponds to FODA's feature diagrams and is discussed in the following section.
- *Configuration Rational and Constraints* that specify rules and constraints used by the application developer when building software applications. This activity corresponds to feature diagrams' constraints, a key element for constructing a full feature model. Feature diagrams' constraints are presented in the next chapter.

A comprehensive explanation of DSSA's concepts and phases can be found in [Tra95] and [TC92]. An example of applying the DSSA method to projects and building reference architecture for an application domain is presented in [HPL+95].

4.3.3 Domain Analysis and Reuse Environment (DARE)

Domain Analysis and Reuse Environment (DARE) was jointly developed by *Reuse Inc.* and the *Software Engineering Guild* as a domain analysis method and CASE[12] tool that provides automated support the domain analysis process, as well as in the FODA and ODM methods [FDF98].

[12] Computer-Aided Software Engineering (CASE) tool is a computer-based product that is used to automate various aspects of software development lifecycles (e.g., diagramming and documentations tools).

According to [DFG92] and [FDF98], the primary goal of DARE is to create a generic architecture that describes the architectural elements (common and variable) and their relationships and dependencies for a set of related software systems in a domain. The concept of DARE is to extract and record domain information from resources, such as code and documents (domain lexicon[13]), acquire domain knowledge from domain experts, and store all information and knowledge captured during the domain analysis process in a *domain book*.

DARE's domain book is a database that provides a full specification of the domain. It contains products created by DARE's tool collection. These work-products include all domain resources captured during the building of domain lexicon process, a domain vocabulary extracted from domain descriptions using lexical analysis tools, architecture and code analysis, a domain model generated from domain lexicon process, and a generic architecture for the domain's systems and reusable components [FDF98].

Similar to the software product line engineering process, DARE is used to create a generic architecture that describes architectural elements and their relationships for systems within the domain. Designing a generic architecture requires a commonality and variability analysis for domain elements and their relationships; common features are used to create the structure of the domain's architecture while accommodating variable features to instantiate different system products [FDF98]. More detail information about the DARE development phases, work-products, and tools collection is presented in [FDF98] and [DFF95].

[13] See Section 3.4.1.1

Other domain analysis and engineering methods and approaches described in the literature include:

- The *Feature Oriented Abstraction, Specification and Translation* (FAST) for defining domain engineering and application engineering processes [Har02-B].
- The *Product Line Software Engineering* (PuLSE) for domain engineering and product line practices [BFK+99].
- *DARCO*, the first approach to domain engineering research developed by James Neighbors and presented in his PhD dissertation "*Software Construction Using Components*" [Nei80].

The literature documents a few methods that attempt to combine domain engineering with *Object Oriented Analysis and Design* (OOAD) methods in order to provide OOAD methods with techniques that support software reuse and model systems' variability (see Section 3.5.1). These include *Object Oriented Role Analysis and Modeling* (OOram) [RWL96] and Sherlock [SVV+00], both of which are domain analysis and engineering approaches that use object-oriented modeling techniques, such as use-case and class diagrams, for modeling common and variable features for a family of systems.

4.4 Feature Models

As discussed in the previous chapter, software product line engineering is a software engineering paradigm that enables software companies to develop a software product from a common set of reusable core assets instead of developing each product separately from scratch.

To develop reusable core assets for use in future systems, however, software product line engineering must manage the commonalities and variabilities across all products in a product line within a domain perspective. After determining the commonalities and variabilities, artifacts that share the most common features across products are chosen as *candidates* to be reused as core assets.

Other artifacts that are not parts of the specified commonality will not be considered as parts of the reusable core assets, but instead will be stored in the company's reusable library as *variation points*. Variable artifacts will be used to create different products that satisfy different customer's needs by using the application engineering process.

As a simple example, consider a company that manufactures cellular phone products. An example of a common core asset across cellular phone products would be a *speaker* or a *microphone* since it is mandatory for each phone to have a speaker or a microphone. In comparison, a *Bluetooth* service supporting a cellular phone is optional since many types of cellular phones do not support the Bluetooth feature. Thus, the Bluetooth artifact is considered a variable feature that would not be part of the common core assets. It would instead, be stored in the company's reusable library, which would be available to developers who wish to include it with phone products that support the Bluetooth service.

In order to determine which artifacts are candidates for the product line's reusable core assets and which are considered variable, feature models are used. Feature models support software product line engineering process by representing similarities and differences for a set of related products in a domain.

4.4.1 Feature Definition

What is meant by the term *feature*? The Oxford Dictionary [OxD12] defines a feature as "*a distinctive attribute or aspect of something*". In the context of software engineering, a feature's definition is similar; it represents a characteristic of a software system or a product in a product line. In FODA (where the concept of feature modeling was first introduced), features are defined as: "*User-visible aspects or characteristics of the domain"* [KCH+90].

In the cellular phone example, a company might provide a cellular phone with 16, 32, or 64GB capacity. In FODA, a cellular phone's capacity is considered a feature (characteristic) of the cellular phone product and a user-visible aspect of the product since the customer can understand its functionality and select it when ordering the phone.

For a more precise definition in the context of product line engineering, [BSR04] defines a feature as "*a product characteristic that is used in distinguishing programs within a family of related programs*". Features are considered "*a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements*" [CZZ+05].

Feature models visually represent features as user-visible aspects of products in a product line [KCH+90].

4.4.2 Feature Model Elements

Stakeholders view features as a product's characteristics that define and distinguish between members in a product line. As described in Section 4.2.1.2, features of a family of software systems are defined, documented, and represented using FODA's feature models.

The concept of feature models was first introduced by FODA to define and represent all software systems in a product line in terms of features. A feature model is not just a single model. Kang et al. [KCH+90] defines a complete FODA feature model to consist of the following elements:

- *Feature diagram*: A feature diagram is a graphical notation that represents features in a hierarchy, declaring each feature as *mandatory*, *alternative*, or *optional*.
- *Feature descriptions*: A feature description is a short semantic description for each feature. In FODA, each feature is described through "a *feature definition form*". A feature definition form includes the following [KCH+90]:
  - ✓ Feature name (*Name: <standard feature name>*)
  - ✓ Textual description of a feature (*Description: <textual description of the feature>*)
  - ✓ A hierarchal structure of a feature that indicates what a certain feature consists of (*Consists Of <featurenames> [{optional | alternative}]*) (Hierarchal structures of features can also be presented graphically.)

- ✓ Feature's source that indicates from which source a certain feature is derived (*Source: <information source>*) (Examples of such sources are textbooks, current systems, and domain experts)
- ✓ Feature's relationships with other features that show if a feature is mutually exclusive with other features (*Mutually Exclusive With: <feature names>]*) or mandatory (*[Mandatory With: <feature names>]*)

- Composition rules for features
- Rationales for features

This chapter will focus on the feature diagram concepts and feature relations. Composition rules and rationales for features will be explained in the next chapter which will present complete feature models examples.

4.5 Feature Diagrams

A feature diagram is a graphical notation that visually represents a feature model, the most important output product of performing domain analysis for product lines. Feature diagrams form the basis for configuring systems by visually representing all possible configurations for a software product line [DK02], [RBS+02]. Feature models capture and represent the common and variable properties among the products in a domain, focusing on properties that may vary. These variable properties are used to produce different software products.

In the cellular phone example, any cellular phone would definitely have characteristics that vary (e.g., phone color). By changing some of the phone's characteristics, a different phone product can be produced. Characteristics of a product that can be changed to produce a different product are called *variation points*.

A phone's color, screen type (e.g., touch-screen or keyboard), or memory capacity are all considered as variation points of the phone product. A list of options that are available for each variation point is called a product variant (e.g., a phone's screen is considered as a variation point for the phone product, while the screen type whether it's a touch-screen or a keyboard is considered as a product variant).

While each variation point's configuration yields different products in a product line, feature diagrams aim to model those variation points, along with their corresponding product variants, to give stakeholders (domain designers, developers and end-users) an abstract view of all possible configurations. The following subsections introduce the principles of feature diagrams.

4.5.1 Concepts and Features in Feature Diagrams

Feature diagrams, as presented in [KCH+90] and [CE00], are graphical notations that visually represent feature models in forms of tree-like diagrams (tree of features). All feature diagrams start with a node at the top of the diagram as a *root node*. Root nodes are called *concepts* and each concept represents a certain domain or a complete product line [BHS+04].

To show how a concept node is visually represented, consider the feature diagram shown in Figure 4.1. The root node of the feature diagram is *RN*, which represents a concept node. The concept node *RN* has two characteristics: *FN1* and *FN2*. *FN1* and *FN2* are called feature nodes of the concept *RN*. As already mentioned, a feature diagram views features in a hierarchal structure that shows a *Parent/Child* type relationship between nodes. In Figure 4.1, *RN* is the parent node of *FN* and *FN1* is the parent node of *FN2* while *FN1* and *FN2* are child nodes of *RN* and *FN1*, respectively.

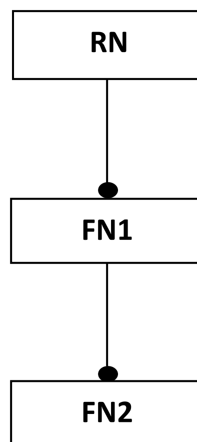

Figure 4.1: A feature diagram with root node *RN* as a concept node and two features, *FN1* and *FN2*

In feature diagrams, features are categorized by their hierarchy relations. Each feature has exactly one hierarchy relation that specifies the feature's category. In the literature, the main feature's categories provided in the majority of feature diagrams' representations include: *mandatory*, *alternative*, *optional*, *AND*, and *OR* features [CE00], [KCH+90], [Bat05], [RBS+02].

4.5.2 Mandatory Features

A *mandatory feature* is a feature that is always included in the final product of a product line. This means that a mandatory feature is the feature that appears in all possible configurations of a certain product or system. A mandatory feature can be a parent or child feature. A child feature is considered mandatory if, and only if, its parent is mandatory and included in the final product. A mandatory feature is indicated by a black circle on top of the node in the feature diagram. Consider the feature diagram shown in Figure 4.2. Features *A1*, *M1*, *M2*, and *M3* are all mandatory features since each feature is marked with a black circle.

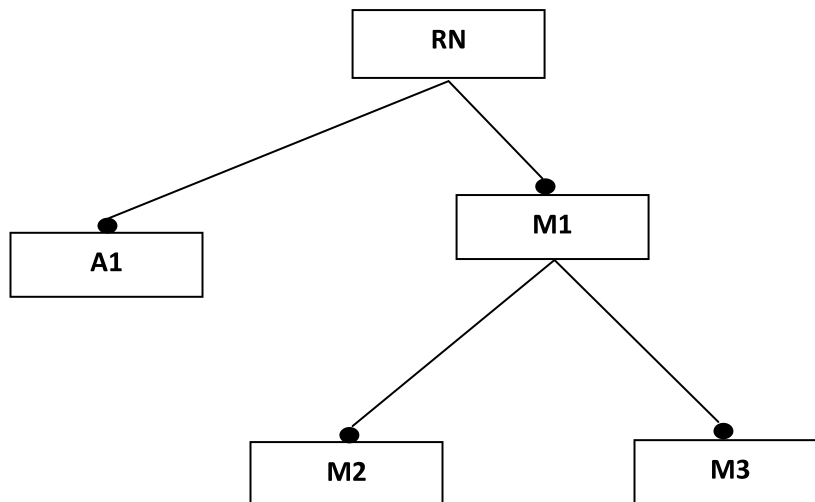

Figure 4.2: A feature diagram with atomic (*A1*) and composite (*M1*) mandatory features

The mandatory feature *A1* is called an *atomic feature* [DK02]. Atomic features are features that can't be subdivided into other features. Thus, an atomic feature has no child features. The opposite of atomic features are *composite features* [DK02]. The mandatory feature *M1* is a composite feature since it has two child features, *M2* and *M3*.

In the cellular phone example, a microphone and a screen are considered as primary features for all cellular phones and thus mandatory features in a cellular phone product line. An example of a composite mandatory feature is an *input device* in a phone that supports both a touch-screen and a keyboard. The input device is a composite feature that has two mandatory touch-screen and keyboard as child nodes.

A description of an instance of the concept node (called *concept instance*) is a set of feature nodes that will always include the concept node in addition to some or all other feature nodes.

Since mandatory features are included in all possible configurations of a product, a mandatory feature will always appear in the description of the concept instance.

Following [CE00], a feature *instance* is described as:

> *"… by adding the concept node to the feature set, by traversing the diagram starting at the root, and depending on the type of the visited node, including it in the set or not."*

In Figure 4.2, since all features are mandatory (parents and children), they are all included in any instance description of the concept node *RN*. Every instance of the concept node *RN* has the mandatory features, *A1* and *M1*; since every instance of the concept node *RN* has the mandatory feature *M1*, which is the parent of the mandatory features *M2* and *M3*, every instance of the concept node *RN* will have *M2* and *M3* features. Thus, every instance of the concept node *RN* will be described by the set {*RN, A1, M1, M2, M3*}, which is the only possible configuration available considering that the diagram has no variation points (e.g., optional features, *OR* features, *AND* features, or Alternative features).

It is obvious that a concept node of any feature diagram is considered a mandatory feature because it represents the entire product line.

4.5.3 Optional Features

An *optional feature* is a feature that may or may not be selected in the final product of a product line or in the instance description of a concept node. The optional feature might be included if its parent is included. If its parent is optional and not included, then the optional feature will not be included. An optional feature is recognized in feature diagrams by a white circle on top of the node. There are two cases of optional features' positions in a feature diagram. The first case is as shown in Figure 4.3a.

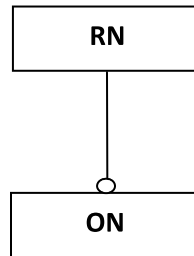

Figure 4.3a: *ON* as an atomic optional feature

Figure 4.3a shows an optional feature *ON* which is also an atomic feature. The parent node of *ON* is the concept feature *RN*. *RN* is a mandatory feature that is always included in the final product.

When the parent of an optional feature is included in the final product, the optional feature may or may not be included based on the product's configuration that is selected by the designer or the customer. The instance description of the concept node *RN* will be {*RN, ON*} or {*RN*}.

The second case is shown in Figure 4.3b. In this case, the optional feature is a composite feature that has mandatory feature *MN* as a child feature. This is an example of a mandatory feature that might not be selected in the instance description of the concept because it is a child of an optional node that may or may not be included. In the instance description of the concept, the *RN* node is always included, while the mandatory feature *MN* is dependent on the selection of its parent node *ON*. If *ON* is selected, then *MN* will definitely be selected. Therefore, the instance description of the concept will be {*RN, ON, MN*} or just the concept node {*RN*}.

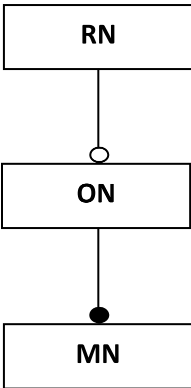

Figure 4.3b: *ON* as a composite optional feature

An example of an optional feature in the cellular phone product line is a Bluetooth. If we assume that each phone product that supports a camera must contain a flash, then a possible composite optional feature can be a camera as an optional feature with a flash as a mandatory feature. For a better understanding, the next chapter presents two feature diagram examples for a video-convertor program and cellular phone product line.

4.5.4 Alternative ("one-of") Features

The *alternative feature* (also called exclusive "one-of" choice [DK02]) indicates a set of features in which only one feature is selected from the set and included in the final description of the concept (if the parent of the set is also included). As shown in Figure 4.4, alternative features are graphically represented by an arc or a line that joins the alternative features' edges forming a triangle-looking shape.

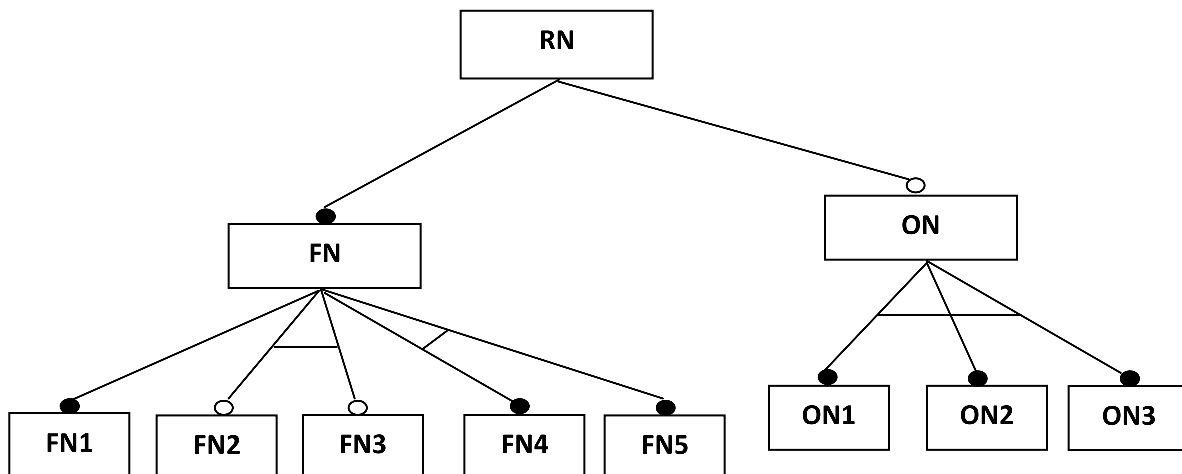

Figure 4.4: A feature diagram with three sets of alternative features

The feature diagram in Figure 4.4 shows concept node *RN* that has two child features: *FN* as a mandatory feature and *ON* as an optional feature. The *FN* feature has one atomic mandatory feature, *FN1*, and two sets of alternative sub-features {*FN2, FN3*} and {*FN4, FN5*}. *FN1* will be selected since it's a mandatory feature that has a mandatory parent *FN,* which also has a mandatory parent *RN*. The first alternative sub-features set, *FN2* and *FN3*, contain two optional features. If an alternative set contains only optional features, the multiplicity of the set will be (0…1) [RBS+02], which indicates that no, or at most, one feature should be chosen.

The other set of alternative sub-features is {*FN4, FN5*}, in which both are mandatory features. Since their parent *FN* is mandatory, exactly one feature should be chosen from the set. The second child feature of the concept node is the optional feature *ON* that has one set of alternative sub-features containing three mandatory features *ON1, ON2*, and *ON3*. Although only one feature will be selected from the set, the selected feature might not be included in the instance description because its parent *ON* is an optional node. Notice that the features grouped in a set are called *sub-features*, if the set is linked with a regular feature apart from the concept node.

Before presenting all possible instances of this feature diagram, it is useful to show how to calculate the number of possible instances (configurations) that a feature diagram can generate.

Consider the optional node *ON*; it has a set of alternative mandatory sub-features from which at least one feature should be selected. Thus, the number of possible instances from the *ON* node is {*ON, ON1*}, {*ON, ON2*}, {*ON, ON3*}; or it is possible that none are chosen since the parent node *ON* is optional and might not be included.

Therefore, the number of possible configurations of the *ON* node is **4**. The second part of the feature diagram is the mandatory node *FN* that has three child features. The first child is the mandatory feature *FN1* that has a mandatory parent *FN1*; this means that the feature will appear in all instance descriptions (**1** configuration). The second child is the set of alternative optional sub-features {*FN2, FN3*} that indicates a (0…1) set multiplicity. This means that *FN2*, *FN3*, or none are chosen from the set (**3** possible configurations). The third child is a set of alternative mandatory sub-features {*FN4, FN5*}, in which exactly one feature must be chosen from the set: either *FN4* or *FN5* (**2** possible configurations). Multiplying the possible configurations of each node in the feature diagram produces all possible instances for the entire diagram. In this case there are **24** possible instances: **4** (*ON*) × **1** (*FN1*) × **3** (*FN2, FN3*) × **2** (*FN4, FN5*) = **24**.

The 24 possible instances are:

{*RN,FN,FN1,FN4*},{*RN,FN,FN1,FN4,ON,ON1*},{*RN,FN,FN1,FN4,ON,ON2*},{*RN,FN,FN1,FN4,ON,ON3*},{*RN,FN,FN1,FN5*},{*RN,FN,FN1,FN5,ON,ON1*},{*RN,FN,FN1,FN5,ON,ON2*},{*RN, FN,FN1,FN5,ON,ON3*},{*RN,FN,FN1,FN2,FN*},
{*RN,FN,FN1,FN2,FN4,ON,ON1*},{*RN,FN,FN1,FN2,FN4,ON,ON2*},
{*RN,FN,FN1,FN2,FN4,ON,ON3*},{*RN,FN,FN1,FN2,FN5*},
{*RN,FN,FN1,FN2,FN5,ON,ON1*},{*RN,FN,FN1,FN2,FN5,ON,ON2*},
{*RN,FN,FN1,FN2,FN5,ON,ON3*},{*RN,FN,FN1,FN3,FN4*},
{*RN,FN,FN1,FN3,FN4,ON,ON1*},{*RN,FN,FN1,FN3,FN4,ON,ON2*},
{*RN,FN,FN1,FN3,FN4,ON,ON3*},{*RN,FN,FN1,FN3,FN5*},
{*RN,FN,FN1,FN3,FN5,ON,ON1*},{*RN,FN,FN1,FN3,FN5,ON,ON2*},
{*RN,FN,FN1,FN3,FN5,ON,ON3*}

Besides capturing and modeling common and variable requirements of software product lines, feature models provide stakeholders with the total number of possible configurations for the entire product line. Since a feature model represents a complete product line, it represents the product line's features in a concise and abstract way that make it much easier to know how many products can be instantiated from a product line, rather than other modeling techniques, such as object-oriented models (e.g., UML class diagrams).

As pointed out in Section 3.5.1, object-oriented modeling techniques provide mechanisms called *object-oriented abstractions*, such as classification, aggregation, association, and inheritance to support modeling variability in software systems using modeling techniques such as UML class diagrams. Variable features can be represented in class diagrams using different variability mechanisms. For example, one variable feature can be represented using association while another variable can be represented using inheritance or aggregation. Different variability mechanisms used to represent variation points result in a complex model that is difficult for stakeholders to understand (e.g., users). Thus makes it difficult to anticipate the number of possible configurations for a product line or a family of systems.

For a better understanding of the difference between feature models and object-oriented modeling techniques, an example of a feature diagram will be presented in the next chapter with its equivalent object-oriented model using a UML class diagram to illustrate differences.

4.5.5 OR ("more-of") Features

The *OR-feature* — also called non-exclusive ("one-of") choice [DK02] — is a set of features from which a non-empty set is selected to be included in the final description of the concept when the parent of the set is also included in the description. The OR-feature was first presented by Czarnecki [Cza98] as an enhancement to FODA's feature models. As shown in Figure 4.5, OR features are graphically represented by a black-fill arc or line that joins the OR-features' edges forming a black triangle-looking shape.

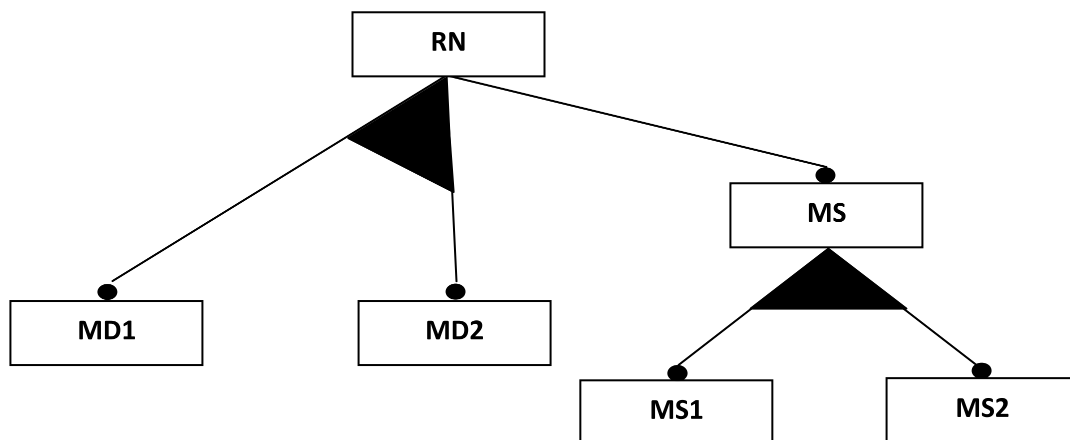

Figure 4.5: A feature diagram includes OR-features

The feature diagram in Figure 4.5 shows a concept node *RN* that has two child features: The first is a set of OR-features {*MD1, MD2*}, and the second is the mandatory feature *MS* that has a set of OR-sub-features {*MS1, MS2*}.

An OR-feature set generally indicates a multiplicity of (1…*), which indicates that at least one feature must be selected from the set if, and only if, the parent of the set is included in the final description of the concept. Since the root node *RN* will be included in all descriptions, *MD1, MD2*, or both of them will be included as well.

The same condition is applied to the OR-sub-feature set {*MS1, MS2*} since it has a mandatory feature *MS* acting as a parent node; it will be included in all descriptions because it has the concept node as a parent. Thus, *MS1, MS2*, or both will be included in the description. The number of possible configurations for the feature diagram is **3** possible configurations for the OR-feature set {*MD1, MD2*} × **1** configuration for the mandatory feature *MS* × **3** possible configurations for the OR-sub-feature set {*MS1*, *MS2*}.

The result is **9** possible configurations, which are:

{*RN,MD1,MS,MS1*}, {*RN,MD1,MS,MS2*}, {*RN,MD1,MS,MS1,MS2*}

{*RN,MD2,MS,MS1*}, {*RN,MD2,MS,MS2*}, {*RN,MD2,MS,MS1,MS2*}

{*RN,MD1,MD2,MS,MS1*},{*RN,MD1,MD2,MS,MS2*},

{*RN,MD1,MD2,MS,MS1,MS2*}.

4.5.6 Normalizing Feature Diagrams

Domain analysis researchers have enhanced FODA's feature models by normalization techniques that simplify feature diagrams to overcome the ambiguity in semantics and the redundancies that occur in certain cases. In the literature, there are several works on normalizing feature diagrams (e.g., [DK02], [RBS+02], and [LLC04]). Most of these take the initial work proposed by Czarnecki and Eisenecker [CE00] as a base.

A normalized feature diagram is a simplified version of the original one; it is *equivalent* to the original diagram in that both generate the same possible instances. According to Lengyel et al. [LLC04], a normalization process "*doesn't change FD's topology, but updates the feature attributes.*"

An example of a feature diagram with ambiguous semantics is shown in Figure 4.6. The figure shows a feature node that has a set of alternative features (sub-features if the node is the concept node) *ON*, *MN1*, *MN2*, and *MN3*. It shows *ON* as an optional feature while the rest are mandatory features.

As pointed out in Section 4.5.4, an alternative set of features indicates that *exactly* one feature is selected from the set under two conditions:

1. If the parent of the set is included in the description of the concept
2. If all features in the set are mandatory features

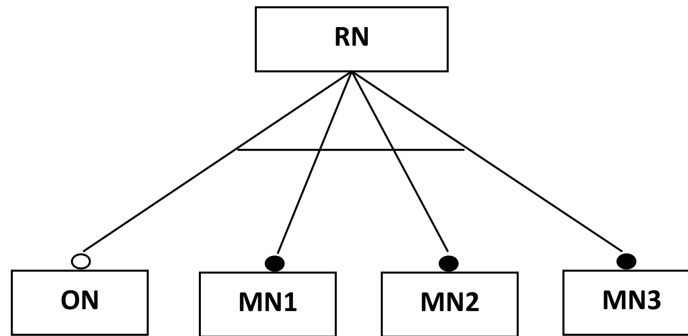

Figure 4.6: A feature node that has a set of alternative features

What about an optional feature in a set? As shown in Figure 4.6, three features are mandatory and one is optional. If a set of alternative features contains one or more optional features, then the second condition is not met, and the set's multiplicity changes from exactly one selected feature (1) to at most one feature (0…1), even if the rest of features in the set are mandatory features.

The set's multiplicity changes if there is an optional feature in a set of alternative features because two decisions are possible. The first decision is selecting the optional feature from the set, while the second is deciding whether or not to include the optional feature in the description of the concept. If the second decision indicates that the optional feature will not be included in the description, then having an empty set is one possible solution out of five possible solutions that can be produced from the feature diagram ({*ON*}, {*MN1*}, {*MN2*}, {*MN3*}, { }).

Therefore, having an optional feature in the set would have the same effect as if all the features in the set were optional. In both cases, the same possible instances are generated and both might produce an empty set. The normalization process is shown in Figure 4.7.

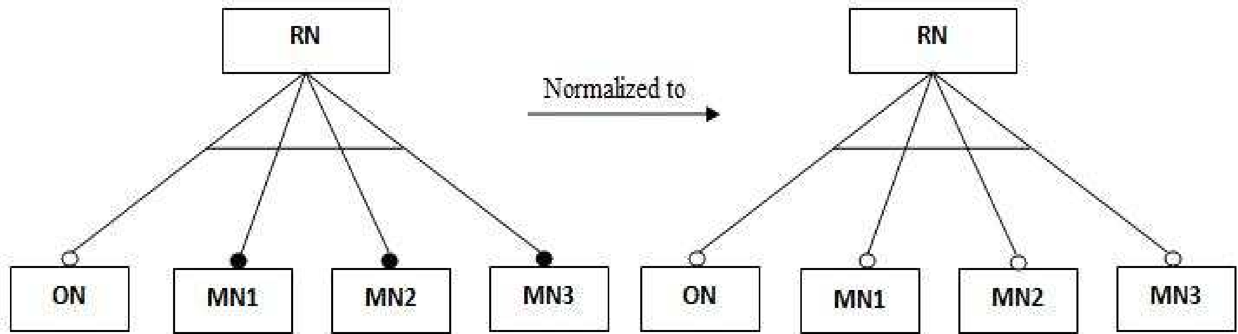

Figure 4.7: A feature diagram with one optional alternative feature is normalized into a new and equivalent diagram with all optional alternative features

As shown in Figure 4.7, both diagrams are equal in that the same possible instances generated from either. The normalized diagram is simpler and does not have the ambiguity in semantics that the original one does. In addition, OR-feature sets can also be normalized using the same normalizing process applied to alternative feature sets. Figure 4.8a shows a feature diagram with OR-features set on the left-hand-side of the figure and its normalized version on the right-hand-side.

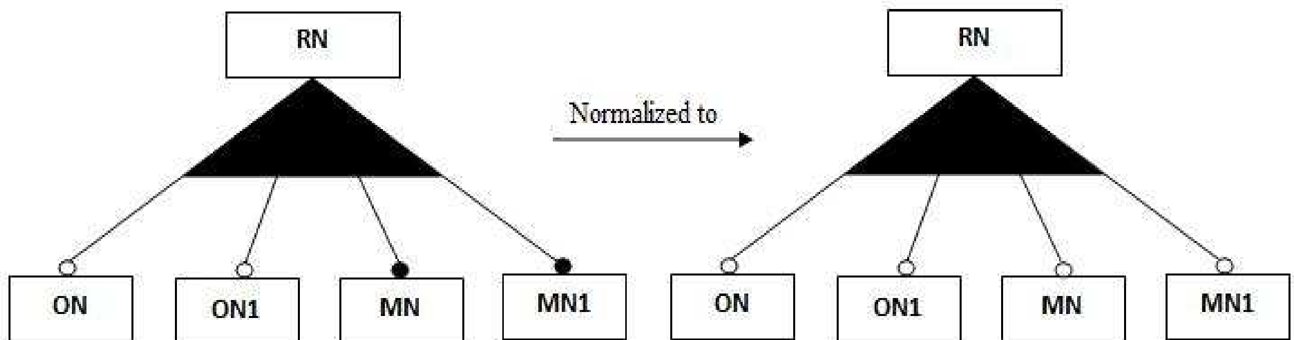

Figure 4.8a: A feature diagram with two optional OR-features is normalized into a new and equivalent diagram with all optional OR-features

The multiplicity of an OR-feature set is basically *one-to-many* (1…*) under two conditions that are similar to alternative features' conditions:

1. If the parent of the set is included in the description of the concept
2. If all features in the set are mandatory features

In Figure 4.8a, the set of OR-features contains two optional features *ON*, *ON1*, and two mandatory features *MN*, *MN1*. Since the set contains 2 optional features, the multiplicity changes from (1…*) to (0…*), indicating that an empty set is a possible choice of the total possible instances generated from the diagram. Therefore, having an optional feature in the OR-set has the same effect as if all features in the set were optional. In both cases, an empty set from each case might be produced and the possible instances generated from the two diagrams are identical: {*ON*}, {*ON1*}, {*MN*}, {*MN1*}, {*ON, ON1*}, {*ON, MN*}, {*ON, MN1*}, {*ON1, MN*}, {*ON1, MN1*}, {*MN, MN1*}, {}.

By viewing the normalized feature diagram in Figure 4.8a and its corresponding outputs, a choice ranges from choosing an empty set to choosing all of the optional features. This state has the same effect that the parent node would have if RN was a parent of optional features and not a parent of a set of optional OR-features, considering that the OR-feature in this case is unnecessary. Therefore, the normalized diagram shown in Figure 4.8a can be normalized again to replace the OR-feature set with optional features as shown in Figure 4.8b.

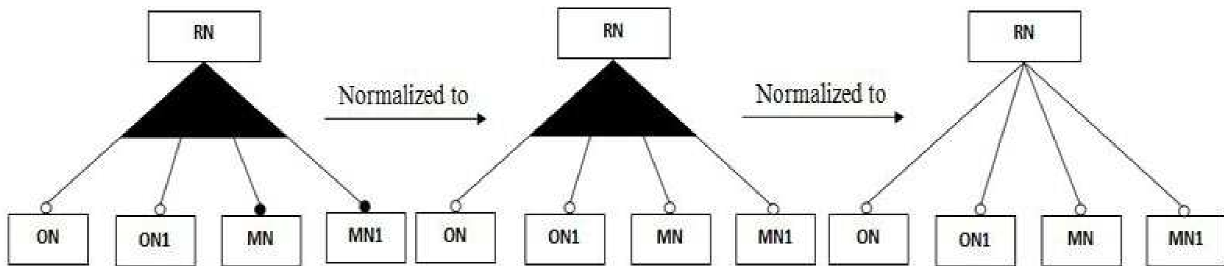

Figure 4.8b: A feature diagram with two optional OR-features is normalized into a new and equivalent diagram with all optional OR-features and normalized again to a new and equivalent diagram with all optional features

It is worth taking into consideration that the second normalization in Figure 4.8b is not possible to apply to alternative features (replace an alternative features set that contains one or more optional feature with all optional features) due to differences in multiplicities. The multiplicity of a set of alternative features that has one or more optional features changes from 1 to (0…1), indicating that at most one feature can be selected. The multiplicity (0…1) clearly prevents the second normalization from happening since the OR-features set has a multiplicity changed from (1…*) to (0…*), which is identical to the multiplicity of the second normalized diagram that has four optional features with a multiplicity of (0…*) as well.

4.6 Summary

Throughout this chapter, we presented FODA, one of the most popular methods in domain analysis research that introduced feature modeling techniques for domain engineering process. The second section surveyed other domain analysis and domain engineering methods that provided automated support to various activities in domain analysis, and extended FODA's abilities to accommodate domain design and implementation phases. The final section introduced feature models and their key elements. We focused on feature diagrams that visually represent feature models with different types of features. We then illustrated how feature diagrams can be simplified through normalization rules that are in addition to OR-features type are extensions of FODA's basic feature models.

In the next chapter, we will present practical examples of feature models that show how to perform domain and feature analysis required for constructing software product lines. We will show how feature models define and represent a set of valid compositions of features (configurations), in which each configuration can be considered as a specification of a software system generated from a product line.

CHAPTER 5

CASE STUDIES

The objective of software product line engineering is to support the development of software products within an application domain by organizing the commonalties and variabilities between these products in a systematic way. Domain analysis provides a detailed analysis of the application domain that allows one to discover these common and variable features to support designing and implementing reusable software assets through the domain engineering process. The results of domain analyses are documented, and represented by domain models that are used to manage common and variable requirements of software systems in a domain. The results are also used to form the basis for configuring systems. During the configuration phase, a software developer selects available features to create a software system that is composed of common features while variable features are used to yield different individual products. The outcome is a configuration that describes a particular product in the product line.

The previous chapter introduced feature models and their various feature types. This chapter presents two feature diagram examples.

The first diagram example is a part of a complete feature model that represents a software product line for a video-converter program. Through this example, we discuss the elements of feature models, including a graphical representation of the product line using a feature diagram, feature descriptions, and constraints that specify relationships and dependencies between features. They also show how feature models define and represent a set of valid compositions of features (configurations), where each configuration can be considered as a specification of a software system generated from a product line. The second feature diagram example represents a software product line for cellular phones. We use this diagram to compare software product lines and Object-Oriented Analysis and Design (OOAD) methods in the context of software reuse and modeling variable characteristics of software systems by transforming the diagram into a UML class diagram to illustrate the differences between them.

5.1 A Software Product Line for a Video-Converter Program

Imagine a small software company with limited resources started a project for developing commercial video-converter software. The idea was previously designed and implemented through one of the company's developers when he developed a video-converter program for personal use in the past. The company consists of three other software developers who plan to refine the video-converter program in order to qualify it as a commercial product.

Based on their market studies, the developers decide to present their product with different features in order to satisfy the various customers' needs as opposed to offering the same product for all customers.

The reason behind this is to achieve higher customer satisfaction by offering a product with wide varieties of features as well as to increase the company's revenues. This is logical because a program version that supports special features is more expensive for the customer than a program supporting basic features is. In order to achieve their goals with respect to their limited resources, the team decided to build a system family based on the system that has been enhanced and refined.

As explained in previous chapters, designing a system family or a product line starts with a detailed analysis of the application domain that performs domain and feature analysis and plans for reusability based on common requirements. In the current example, after defining domain requirements, the team perform domain and feature analysis and identify common requirements that will be present in all members of the video-converter product line. They also identify variable requirements that will be present in only some members of the product line. They document the result of feature and domain analysis (commonality and variability) using a feature diagram.

Figure 5.1 shows the feature diagram of the video-converter program's product line. The feature diagram starts with a root node *Video Converter* called the concept node that represents a family of video-converter systems that share common structures and behaviors.

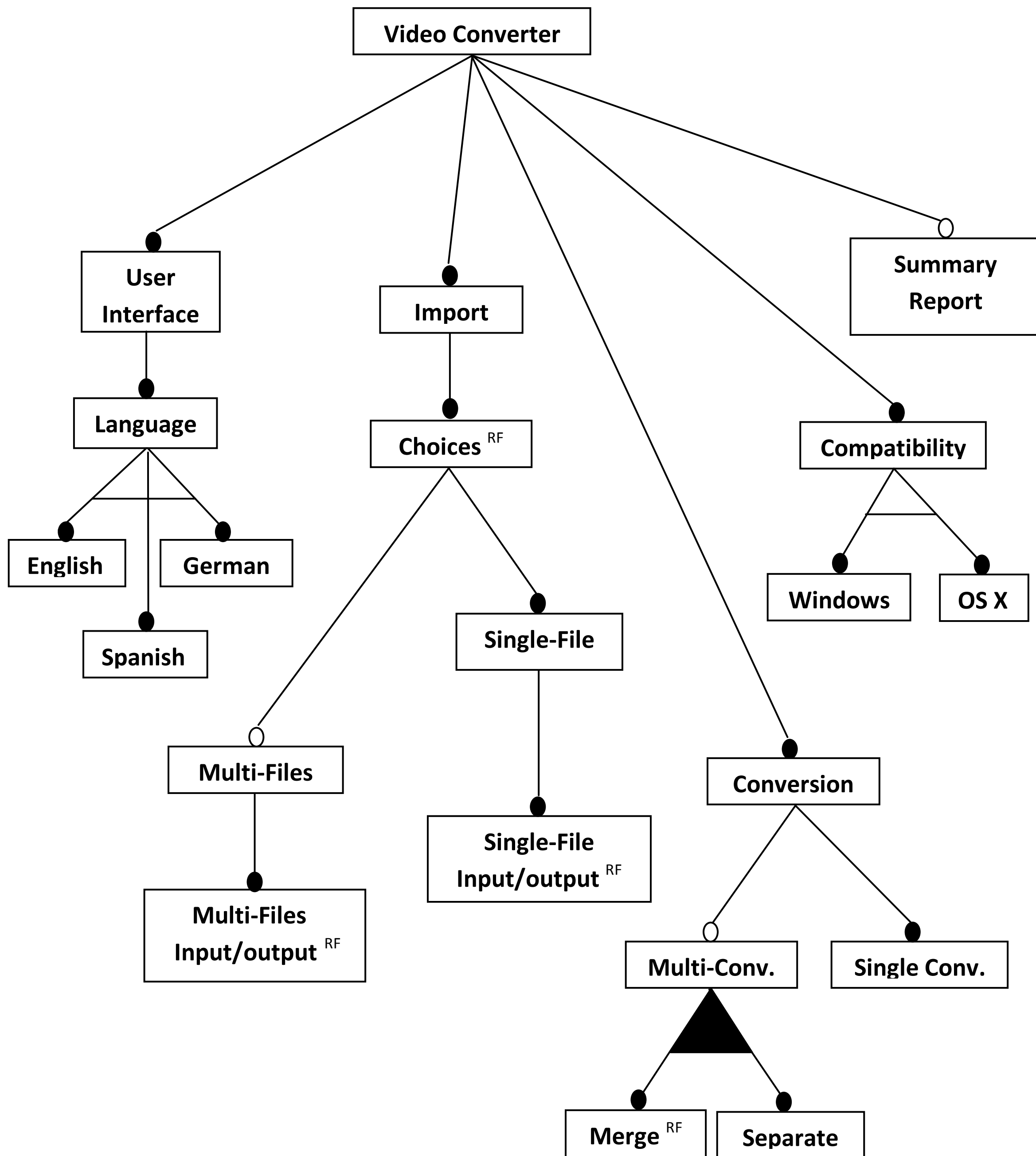

Figure 5.1: A feature diagram of a video-converter product line

As shown in Figure 5.1, functionalities specified as common to all systems in the video-converter product line are basic features that include a user interface, the process of selecting and converting a video file, and the program's compatibility with operating systems. Each video-converter program generated from the product line must contain these common features.

Functionalities specified in the requirements engineering phase as variation points are represented in feature diagrams by optional features, OR-features, and alternative features, which indicate requirements that are presented in only some members of the product line.

According to [CE00], a feature in a feature diagram is considered a *variation point* if it has at least one direct variable sub-feature or feature. Based on that, the first variation point in the video-converter diagram (if viewed from left to right) is the *Language* feature since it has three direct sub-features (*English*, *Spanish*, and *German*), where all are alternative features. *Language* is a mandatory feature and a child of the *User Interface* feature which is also a mandatory feature. Therefore, both *User Interface* and *Language* features are included in the final description of the concept, which means that they will both be included in every possible configuration of the video-converter product line.

Following [CE00], any feature that has all direct sub-features or features as alternative features is called a *dimension* feature. Thus, the *Language* feature is a dimension feature, and each program instantiated from the video-converter product line should support only one language: *English*, *Spanish*, or *German*.

The second variation point is the mandatory feature *Choices* since it has one variable sub-feature (*Multi-Files*). The *Choices* feature is a child of the mandatory feature *Import*, and thus both will be included in all possible configurations of the product line.

Notice that a variation point is different from a variable feature. The variation point is any feature that has at least one direct variable sub-feature or feature and, therefore, it can be a mandatory node or even the concept node. Any program instantiated from the video-converter product line should support the primary feature *Import*, which is the selection of the input video file for conversion.

The *Choices* feature indicates two features that a product line supports. The first feature is selecting a single video file for conversion through *Single-File*, a mandatory feature that indicates that every possible program generated from the product line should support converting one video file. The second child of the *Choice* feature is the optional feature *Multi-File*; a variable feature that may or may not be included in the final description of the concept.

The company decided to have two versions for profitable reasons: the first one is the basic version that supports converting only one video file at a time. The second version is a Pro-version that will support multiple conversions at a time and sell for a higher price than the basic version. Therefore, the product line analyst represents *Multi-File* as a variable feature that will be present in only some members of the product line based on the customer's requests. Both *Single-File* and *Multi-Files* features have mandatory sub-features *Single-File, Input/output,* and *Multi-Files Input/output*, respectively. We discuss these features in the next section.

Notice that we can remove the *Choices* feature and attach both *Single-File* and *Multi-Files* features to *Import*. The reason why we created the *Choices* feature is because it indicates other features that illustrate different file formats for the imported source video file. We explain this further in the next section.

The third variation point is the *Conversion* feature since it has one variable sub-feature (*Multi-Conv.*). *Conversion* is a mandatory feature, a child of the concept node and thus a common feature to all software products in the video-converter product line. The *Conversion* feature has two sub-features. The first one is the *Single-Conv.* sub-feature that represents the conversion process of the video-file selected through *Import* → *Choices* → *Single-File* features.

Single-Conv. is a mandatory sub-feature and a child for the mandatory feature *Conversion* and therefore will be included in all possible configurations of the product line. *Single-Conv.* is a commonality across all family members as it is a primary feature in the basic version of the video-converter program.

The second sub-feature of *Conversion* is the optional feature *Multi-Conv*. The company decided to include this feature in the video-converter's Pro-version that represents the conversion process for multiple video-files at the same time selected through *Import* → *Choices* → *Multi-Files*. The company offers two features that come along with the *Multi-Conv.* feature: *Merge* and *Separate*. These sub-features are combined through an OR-feature set. If a customer purchases the Pro-version that allows multiple selections and conversions of video files at the same time, he or she would have three choices since the OR-feature set indicates a multiplicity of (1…*). The first choice is to save the converted video files separately; the second choice is to merge all of them into one single file; and the third choice is to combine *Merge* and *Separate* features to get separate converted files in addition to one more file that combines all of the converted files together.

Notice that all three choices of the OR-feature might not be included in the final description of the concept since the OR-feature set is a child of the optional feature *Multi-Conv.* that will be selected if a customer orders the Pro-version of the program. A feature in which all direct sub-features are OR-features is called an *extension point* [CE00].

The fourth variation point of the video-converter product line is the *Compatibility* feature: a mandatory feature and a child of the concept node that will be included in every possible configuration of the product line. *Compatibility* is a dimension variation point since it has two direct sub-features, *Windows* and *OS X* as alternative features. *Compatibility* indicates two different video-converter versions. The first is a Windows version that supports the *Windows* operating systems; the second is a Mac version that supports the *OS X* operating systems. The user will have the choice to select either the *Windows* or *OS X* features as the multiplicity of the alternative feature set indicates exactly one choice. Selecting Windows or Mac versions has no interference with purchasing the basic or the Pro-version of the program. If there is interference, it will be explained separately through constraints, features descriptions, and dependencies between features since a complete feature model consists of a feature diagram and other information that explains other aspects that cannot be expressed in feature diagrams [KCH+90].

The last feature in the video-converter product line is the optional feature *Summary Report*, which represents a file generated at the end of the conversion process that provides customers with detailed information such as output duration time, frame size, video bit-rate, and location. *Summary Report* is optional and, thus, a variable feature that can only be included upon a customer’s requests.

The following sections illustrate features descriptions, features references, dependencies and constraints between features, and possible configurations for the video-converter product line.

5.1.1 Features Descriptions

We can consider what we have discussed and explained in the previous pages as a detailed description of the video-converter product line. The description would be more formal and abstract if it was a description written by domain and requirements analysts who provide software developers with high level descriptions of related systems in a domain (see Section 3.2.1). A feature description is necessary for both developers and stakeholders (e.g., end-users). According to [CSP+92]:

> *"The features model is the chief means of communication between the end-users and the developers of new applications. Features are meaningful to the end users and can assist the requirements analysts in the derivation of a system specification that will provide the desired capabilities. The features model provides them with a complete and consistent view of the domain."*

Therefore, feature descriptions are used to eliminate ambiguities that may occur when some requirements are represented in feature diagrams using acronyms that could confuse developers or stakeholders. For example, consider the *Multi-Conv.* feature in Figure 5.1. The feature name is derived from *Multiple Conversion* name, yet we used *Multi-Conv.* as an abbreviation to save space to represent other features in the diagram.

Although the meaning of *Multi-Conv.* might be clear to users and developers and we could have easily written its full name in the diagram, other feature diagrams that represent real applications are too complex and large to use full feature names. Therefore, it is almost impossible to do so. Using acronyms is a necessity.

There are several techniques for providing descriptions to domains and product lines. One of these techniques is the *feature definition form* provided by the FODA method (see Section 4.4.2). Since our feature diagram is simple, we applied the first three elements of FODA's form to the video-converter's features in the following distinctive way. Section 5.1.2 discusses the relationships between features.

Feature Name	Video-Converter
Feature Type	Concept
Feature Path	Root Node
Feature Description	Video-Converter product line. (A family of systems)

Feature Name	User Interface
Feature Type	Mandatory
Feature Path	Video-Converter → User Interface
Feature Description	Represents the Program's Interface.

Feature Name	Language
Feature Type	Mandatory and Variation Point
Feature Path	Video-Converter → User Interface → Language
Feature Description	Provides the Video- Converter program with three languages.

Feature Name	English
Feature Type	Mandatory Alternative
Feature Path	Video-Converter → User Interface → Language → English
Feature Description	Provides English language for the Video-Converter program.

Feature Name	Spanish
Feature Type	Mandatory Alternative
Feature Path	Video-Converter → User Interface → Language → Spanish
Feature Description	Provides Spanish language for the Video-Converter program.

Feature Name	German
Feature Type	Mandatory Alternative
Feature Path	Video-Converter → User Interface → Language → German
Feature Description	Provides German language for the Video-Converter program.

Feature Name	Import
Feature Type	Mandatory
Feature Path	Video-Converter → Import
Feature Description	Selecting input video file to convert.

Feature Name	Choices
Feature Type	Mandatory
Feature Path	Video-Converter → Import → Choices
Feature Description	Indicates two selection choices: Single or Multiple files. Also shows all video formats that the program supports. (Explained in the next section)

Feature Name	Single-File
Feature Type	Mandatory
Feature Path	Video-Converter → Import → Choices → Single-File
Feature Description	Selecting only one video file to convert.

Feature Name	Multi-Files
Feature Type	Optional
Feature Path	Video-Converter → Import → Choices → Multi-File
Feature Description	Selecting multiple video files to convert.

Feature Name	Single-File Input/output
Feature Type	Mandatory
Feature Path	Video-Converter → Import → Choices → Single-File → Single-File Input/output
Feature Description	Shows available formats for the selected video file. Program's basic versions support all source and output formats expressed in the feature's extension. (Explained in the next section)

Feature Name	Multi-Files Input/output
Feature Type	Mandatory
Feature Path	Video-Converter → Import → Choices → Multi-Files → Multi-Files Input/output
Feature Description	Shows available formats for the selected video files. Program's Pro-versions support all source and output formats expressed in the feature's extension. (Explained in the next section)

Feature Name	Conversion
Feature Type	Mandatory and Variation Point
Feature Path	Video-Converter → Conversion
Feature Description	Indicates the conversion process. Algorithms and codec used are hidden since feature diagrams focus on representing program's features in abstract ways that concentrate only on concepts levels rather than the actual procedures.

Feature Name	Single-Conv.
Feature Type	Mandatory
Feature Path	Video-Converter → Conversion → Single-Conv.
Feature Description	The conversion process of a single video file.

Feature Name	Multi-Conv.
Feature Type	Optional and extension point
Feature Path	Video-Converter → Conversion → Multi-Conv.
Feature Description	The conversion process of multiple video files at the same time.

Feature Name	Merge
Feature Type	Mandatory OR-feature
Feature Path	Video-Converter → Conversion → Multi-Conv. → Merge
Feature Description	Merging the converted video files into one video file. (Explained in the next section)

Feature Name	Separate.
Feature Type	Mandatory OR-Feature
Feature Path	Video-Converter → Conversion → Multi-Conv. → Separate
Feature Description	The converted video files are generated separately.

Feature Name	Compatibility
Feature Type	Mandatory and Variation Point
Feature Path	Video-Converter → Compatibility
Feature Description	Shows in which operating systems the program runs.

Feature Name	Windows
Feature Type	Mandatory Alternative
Feature Path	Video-Converter → Compatibility → Windows
Feature Description	Indicates that this program version runs only on Windows operating systems.

Feature Name	OS X
Feature Type	Mandatory Alternative
Feature Path	Video-Converter → Compatibility → OS X
Feature Description	Indicates that this program version runs only on Mac operating systems.

Feature Name	Summary Report
Feature Type	Optional and atomic
Feature Path	Video-Converter → Summary Report
Feature Description	After finishing the conversion process, the program generates a file that provides users with detailed information such as the conversion duration time, Bitrates, frame size, etc.

5.1.2 Features References

We can add an element to FODA's basic elements for describing features — the *Feature Reference.* A feature reference graphically extends a particular feature in a feature diagram if that feature requires more explanations or has a graphical extension that might cause confusion in the feature diagram.

Thus, the reference feature can be modeled separately. A feature reference can also be used to simplify larger feature diagrams and divide them into segments where each segment represents hierarchal structure of features.

In Figure 5.1, notice that some features have the indicator **RF** at the end of their names. The features are **Choices** RF, **Multi-Files Input/output** RF, **Single-File Input/output** RF, and **Merge** RF, where **RF** is an acronym for the term *Reference Feature***.**

In addition to showing two different choices for selecting video files (*Single* and *Multiple* file selection), the *Choices* feature provides software developers with the available video formats that the video-converter program supports, including input video formats for the selection process and output video formats for the conversation process. Figure 5.2 shows the extension of the *Choices* feature.

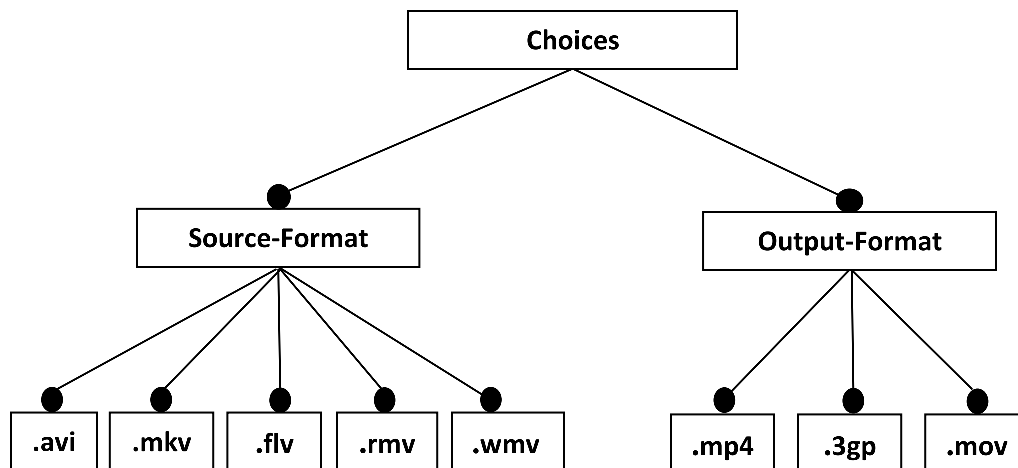

Figure 5.2: An extension to the *Choices* feature that represents available input/output video files formats

Since all sub-features of *Source-Format* and *Output-Format* are mandatory features, the video-converter program will support all of them (including both basic and Pro-versions). Source and output formats have been identified and included in the program's product line during domain analysis and requirements phases where market studies showed that the most preferable video formats are: **.mov** for Apple products including iPhones, **.3gp** for Windows products and Android-powered cell phones, and **.mp4** for Mac, Windows and Android devices.

Although we can include the *Choices* feature with its extension in our example, it would require two *Choices* features: one that represents selection choices and one that represents video formats. This leads to two redundant features, a condition that domain and requirements analysts try to avoid. Another possible representation would have a new mandatory feature called *Supported-Format* that shows available source and output video formats and directly link to the concept node.

Other referenced features are *Single-File Input/output* and *Multi-Files Input/output*, expressed in Figures 5.3 and 5.4, respectively.

The *Single-File Input/output* feature has an extension that provides the software developer with available formats that the program's basic version supports. As shown in Figure 5.3, *Single-File Input/output* supports all formats that the *Choices* feature provides in Figure 5.2. The difference between them is that the *Single-File Input/output* feature has alternative sub-features for both source and output formats, which tells the developer that a user can select and convert only one video-file at a time.

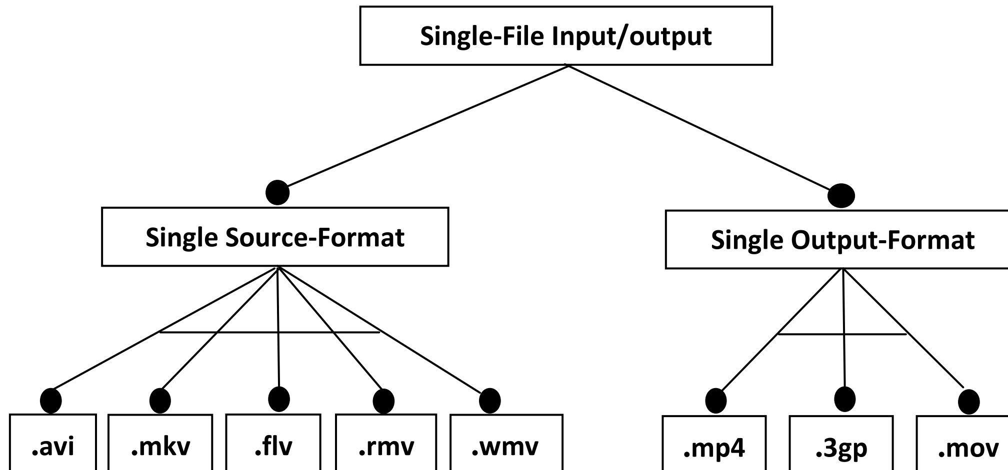

Figure 5.3: An extension to the *Single-File Input/output* feature which represents available input/output video file formats for a video-converter program that supports only one video conversion at a time (Basic version)

The extension of the *Single-File Input/output* feature also informs the software developer that all formats are included and supported by the program's basic version. Otherwise, the developer might think that only one input and output format is allowed since both *source* and *output* features have sub-features as alternatives. Although this functionality is not graphically represented in Figure 5.3, it is specified in the feature's description given in the previous section, since both feature diagrams and descriptions provide software developers with all the information they need.

Figure 5.4 shows the extension of the *Multi-Files Input/output* feature. The extension informs the software developer that a customer who purchases a Pro-version can select multiple video files at a time through the OR-feature set that indicates one-to-many (1…*) multiplicity (in this case the use can select up to five video files with formats that match the specified formats expressed in Figure 5.4).

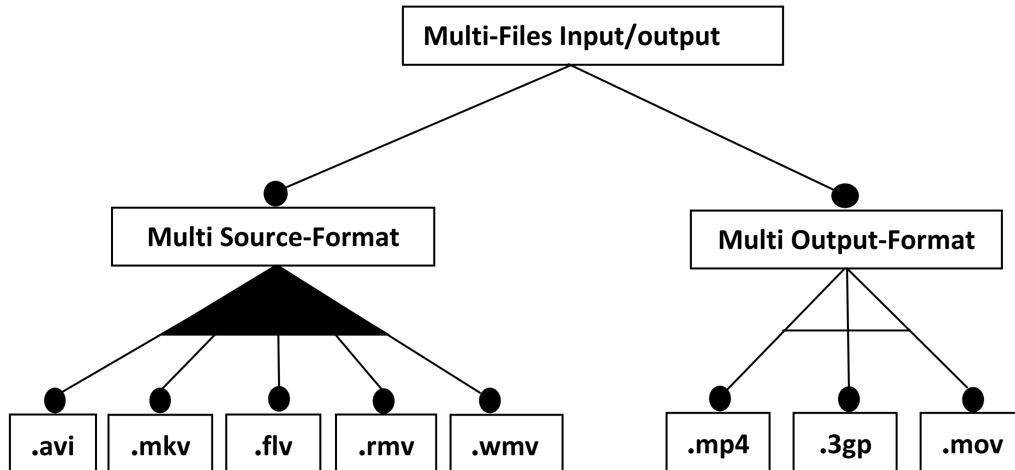

Figure 5.4: An extension to the *Multi-File Input/output* feature that represents available input/out video file formats for a video-converter program that supports multiple video conversions at a time (Pro-version)

The extension of the *Multi-Files Input/output* feature also informs the developer that the program allows selecting multiple videos with multiple formats and converts these videos to a singular format. As shown in Figure 5.4, the *Multi Output-Format* sub-feature has three sub-features as alternatives which indicate that only one format is allowed for converted videos. A user can select (1…5) formats in the *source* OR-feature and select exactly one format from the three available formats expressed in the *output* alternative feature. The description of the *Multi-Files Input/output* feature provided in the previous section indicates that all formats are included and supported by the program's Pro-version.

The third referenced feature is the OR-feature *Merge*, a feature that represents the functionality of merging converted video files into one single file. The extension of the *Merge* feature is represented in Figure 5.5.

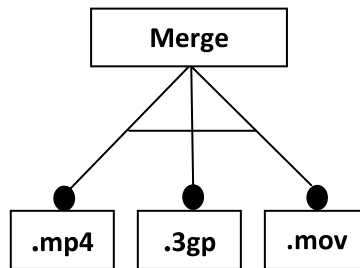

Figure 5.5: An extension to the *Merge* feature shows available formats that Pro-versions of the video-converter program support.

Merge has a set of three alternative sub-features which indicates possible formats for the generated video file that merges all converted video files. The extension of the *Merge* feature informs the software developer that a user should have the choice between .mp4, .3gp, or .mov as the format of the generated video file that merges all converted videos.

5.1.3 Features Constraints and Dependencies

Feature constraints, dependencies, and relationships between features are information that cannot be graphically expressed using feature diagrams. A complete feature model consists of these aspects in addition to feature diagrams and descriptions. As stated in Section 4.4.2, FODA's feature models consist of feature diagrams, feature descriptions, composition rules, and rationales for features. Feature interdependencies and constraints are captured and expressed by using composition rules.

In the literature, there are several constraints and dependencies that specify relationships between features in feature diagrams. The most widely used constraints are *requires* and *excludes* relations that are composition rules defined by the FODA method [KCH+90]. We illustrate these rules by applying them to the video-converter example.

- *Requires* composition rule: If feature X requires feature Y, the functionality of feature X is dependent on the functionality of feature Y. For example, the *Merge* feature requires the *Choices* → *Multi-Files* feature, since *Merge* is a feature that exists only if the optional feature *Multi-Files* is selected through the program's Pro-version.

- *Excludes* composition rule: If feature X excludes feature Y, then it is not possible to have both features X and Y in the same product. Since there is no such case applicable in the video-converter product line, we can assume, for example, that the *Summary Report* feature is not supported by the *OS X* feature and summary report files are only generated using the *Windows* operating systems. Based on this assumption, the relation would be: *Summary Report* excludes *OS X*, which indicates that these features cannot be included in the same program generated from the product line.

Requires and excludes rules can be applied to some features as explanations or constraints specific for those features without enrolling other features. For example, consider the *Multi-Files* feature in the video-converter feature diagram. The feature's functionality is to select multiple video files as inputs to convert them through the program's Pro-versions. We can add a constraint for this feature that informs the software developer that the maximum number of video files selected to convert is ten videos at a time.

Another example is the *Multi-Conv.* feature, which provides the functionality of converting multiple video files and has a set of OR-features: *Merge* and *Separate*. A constraint can be added to *Multi-Conv*. that specifies recommended system requirements required to perform it (e.g., RAM: 512MB or higher). Constraints and dependencies can be mentioned in feature descriptions or specified explicitly through a textual note next to the feature's graphs in the feature diagram. Figure 5.6 shows all constraints that have been discussed in this section added to the video-converter feature diagram.

In the literature, there are other types of dependencies that specify relationships between features in feature diagrams. Table 5.1 shows the most commonly used constraints in feature modeling research (e.g., [ZHT08] and [DK02]) in addition to FODA's composition rules.

Constraint	***Meaning***
Same	Indicates that feature X is the same as feature Y. For example if we mention the *Summary Report* feature in another feature decomposition in the same diagram, we can represent it as SR and link between the two feature (e.g., ◄-- *same*-►) which indicates "*Summary Report*" *same* "SR"
Extends	Indicates that feature X extends Feature Y if X adds to the functionality of Y. For example, the *Language* feature extends the *User Interface* feature.
Includes	Indicates that feature X has feature Y inside of it. For example, the *Multi-Conv.* feature includes the *Merge* feature.

Table 5.1: Other types of feature dependencies used in feature diagrams

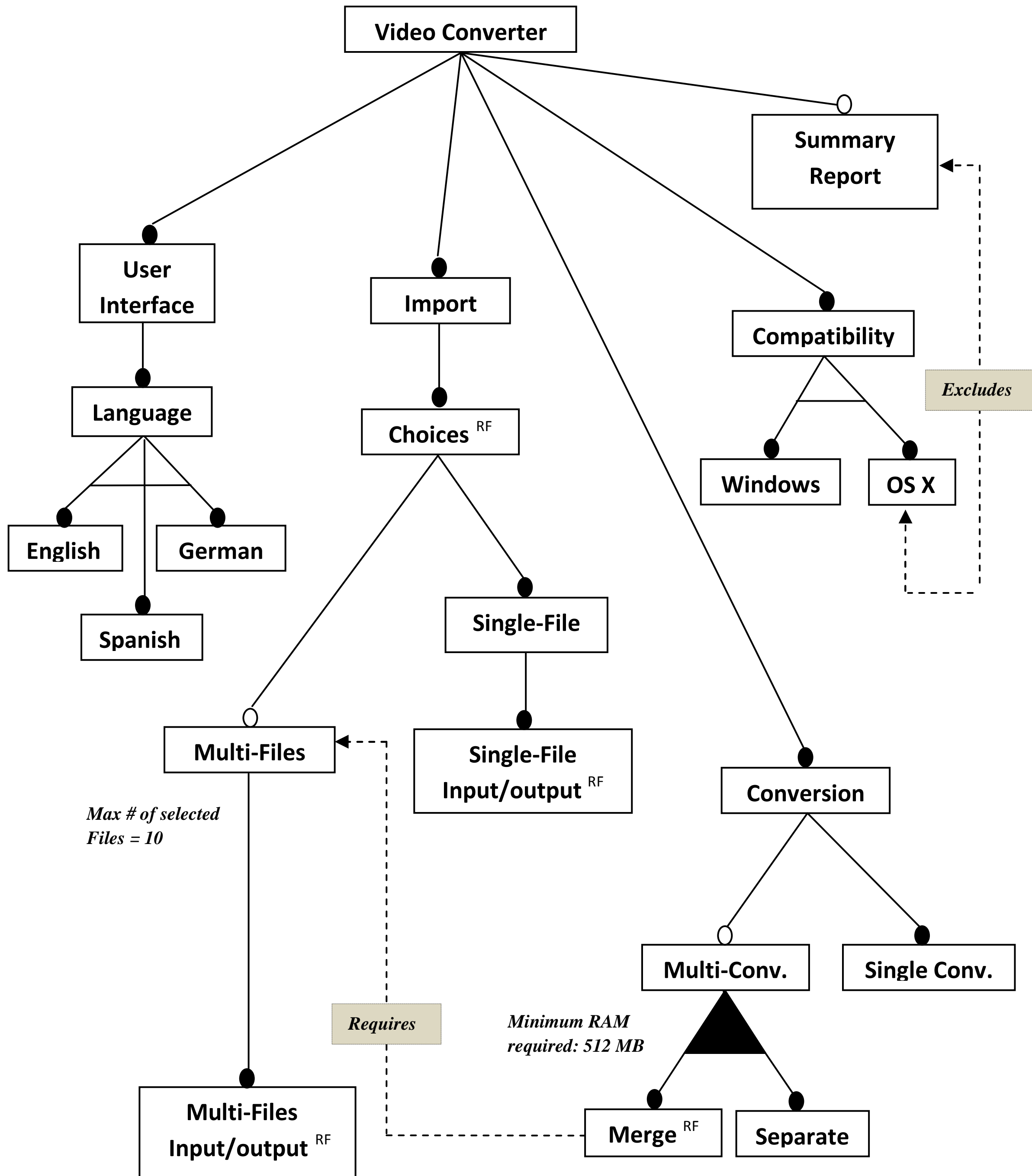

Figure 5.6: A feature diagram of a video-converter product line with constraints

5.1.4 Possible Configurations of the Video-Converter Product Line

Besides capturing and modeling common and variable requirements of software product lines, feature models provide stakeholders with the total number of possible configurations for each product line. Since a feature model represents a complete product line, it represents the product line's features in a concise and abstract way that makes it much easier to know how many products can be instantiated from a product line from other modeling techniques such as object-oriented models (e.g., UML class diagrams).

In the video-converter product line example, the number of possible products generated from the diagram would be as follows (without considering requires/excludes rules):

- *Video-Converter* → *User Interface* → *Language*. **Three** possibilities:
  - *Video-Converter* → *User Interface* → *Language* → *English*
  - *Video-Converter* → *User Interface* → *Language* → *Spanish*
  - *Video-Converter* → *User Interface* → *Language* → *German*
- *Video-Converter* → *Import* → *Choices*. **Two** possibilities:
  - *Import* → *Choices* → *Single-File*
  - *Import* → *Choices* → *Single-File* and *Multi-Files*
- *Video-Converter* → *Conversion*. **Four** possibilities:
  - *Conversion* → *Single-Conv.*
  - *Conversion* → *Single-Conv.* and *Multi-Conv.* with *Merge*
  - *Conversion* → *Single-Conv.* and *Multi-Conv.* with *Separate*
  - *Conversion* → *Single-Conv.* and *Multi-Conv.* with both *Merge* and *Separate*.

- *Video-Converter* → *Compatibility*. **<u>Two</u>** possibilities:
  - *Video-Converter* → *Compatibility* → *Windows*
  - *Video-Converter* → *Compatibility* → *OS X*
- *Video-Converter* → *Summary Report*. **<u>Two</u>** possibilities:
  - *Video-Converter* → *Summary Report*
  - *Video-Converter* → Without *Summary Report*

As a result, the number of possible system configurations of the video-converter product line would be: $3 \times 2 \times 4 \times 2 \times 2 = \mathbf{96}$ possible video-converter instances, where each instance represents a configuration that can serve as a specification of a software program.

By adopting a software product line as a reuse strategy for developing reusable core assets and using these assets to build concrete applications, the company that develops the video-converter program would have the ability to produce 96 different video-converter products to satisfy customer orders by capturing, analyzing, and modeling the product line's common and variable requirements using feature models.

Using the feature diagram that represents the video-converter product line, two possible video-converter products out of 96 would be:

- A *Video-Converter* program that has a *User Interface* (commonality) that supports the *English Language* (variability) with the *Single-File Selection* (commonality) and *Single-File Conversion* (commonality) features; *Compatible* with *OS X* (variability) operating systems and does not include the *Summary Report* (variability) feature.

- A *Video-Converter* program that has a *User Interface* (commonality) that supports the *German Language* (variability), supports *Single-File Selection* (commonality), *Single-File Conversion* (commonality), *Multi-Files Selections* (variability), and *Multi-Files Conversions* (variability) with *Merge* (variability) and *Separate* (variability) features added to multiple conversions; *Compatible* with *Windows* (variability) operating systems and includes the *Summary Report* (variability) feature.

If we consider requires/excludes rules, then the number of possible video-converter instances would be reduced. The *Summary Report* feature is not supported by computers that work on *OS X* operating systems. The *Summary Report* has two possibilities multiplied by other features' possibilities. If we exclude all *OS X* computers from generating summary reports, then the number of possible instances will be reduced to a half. The total number of possible instances after applying the exclusion rule is: 96/2 = 48 different products.

The "Requires" rule does not affect the number of possible instances because it is obvious that the users who buy the basic version will not have the choice of selecting multiple files. Thus, they will not have the *Multi-Conv*. feature and its two child features *Merge* and *Separate*.

5.2 Feature Modeling versus Object-Oriented Modeling

As stated in Section 3.5.1, one of the shortcomings of Object-Oriented Analysis and Design (OOAD) methods in the context of software reuse is their insufficient support for modeling variabilities. In comparison to software product line techniques that utilize feature models as a part of the domain analysis process for representing related software systems, OOAD methods lack an abstract and concise model for representing variabilities.

We explain these aspects by providing a feature diagram example that represents a simple software product line for a company that manufactures cellular phone products. We then transform the feature diagram into a UML class diagram, a static structure diagram that is widely used in the modeling of object-oriented systems during analysis and design phases, and demonstrate why software product lines strategy is considered a better approach than OOAD in modeling common and variable features across related software systems.

5.2.1 A Software Product Line for Cellular Phones Using Feature Diagram

Figure 5.7 shows a small feature diagram in the domain of a cellular phone industry that illustrates the structural view of a software product line created to provide software for cellular phones. The feature diagram starts with *Cellular Phone* as a concept feature that-represents the cellular phone's software product line. *Cellular Phone* has three mandatory features that support software for *Microphone*, *Input-Device*, and *Network Mode* and one optional feature, *Bluetooth.*

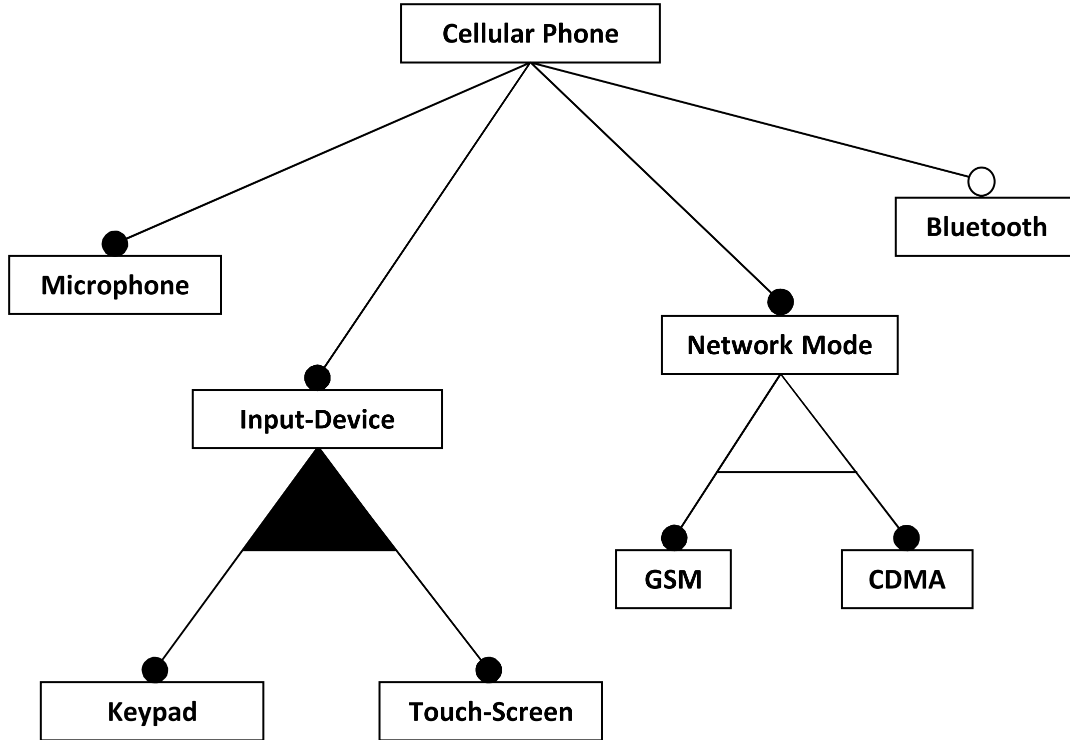

Figure 5.7: A feature diagram of a simple software product line for cellular phones

The *Microphone* feature is a mandatory and atomic feature that indicates that the cellular phone software must provide a functionality to support the phone's microphone in all phones' products. This is because each cellular phone requires a software support for a microphone.

The *Input-Device* feature is a mandatory feature that has a direct set of OR-sub-features: *Keypad* and *Touch-Screen. Input-Device* indicates that each software product instantiated from the cellular phone product line must provide functionality to phones that support a physical *Keypad*, a *Touch-screen*, or both.

The *Network Mode* feature is a mandatory feature as well as a dimension since it has a set of alternative sub-features, *GSM* and *CDMA*, which are the two major radio systems used in cell phones.

GSM is an acronym for Global System for Mobile Communications; a 2G technology that functions based on a combination of Time Division Multiple Access (TDMA) and Frequency Division Multiple Access (FDMA). *CDMA* is an acronym for Code Division Multiple Access, a spread spectrum technology that transmits multiple digital signals simultaneously over the same carrier frequency. GSM phones require SIM cards to store user account information, while user account information is stored using internal memories in CDMA phones. AT&T and T-Mobile are examples of carriers that use GSM technology, while Verizon Wireless and Sprint are examples of carriers that use CDMA.

Since GSM and CDMA are completely different radio systems, a cellular phone can only support one of the systems, and thus we use an alternative relation between them. The last feature is the optional feature *Bluetooth*, which indicates that the software may or may not include support for the Bluetooth service. This is necessary since not all phone products support the *Bluetooth* feature.

The number of possible software products (system configurations) generated from the phone software product line would be:

- *Cellular Phone* → *Microphone*. **One** possibility
- *Cellular Phone* → *Input-Device*. **Three** possibilities:
  - *Cellular Phone* → *Input-Device* → *Keypad*
  - *Cellular Phone* → *Input-Device* → *Touch-Screen*

- *Cellular Phone* → *Input-Device* → *Keypad and Touch-Screen*

- *Cellular Phone* → *Network Mode*. **Two** possibilities:
  - *Cellular Phone* → *Network Mode* → *GSM*
  - *Cellular Phone* → *Network Mode* → *CDMA*
- *Cellular Phone* → *Bluetooth*. **Two** possibilities
  - *Cellular Phone* → *Bluetooth*
  - *Cellular Phone* → Without *Bluetooth*

As a result, the number of possible configurations of the software product line for the cellular phone product would be: $1 \times 3 \times 2 \times 2 = \mathbf{12}$ possible software instances (products).

5.2.2 Transforming a Cellular Phone Feature Diagram into a UML Class Diagram

Inspired by [DK02], Figure 5.8 shows one possible implementation of the cellular phone example using a UML class diagram where each feature of the cellular phone software product line represents a class.

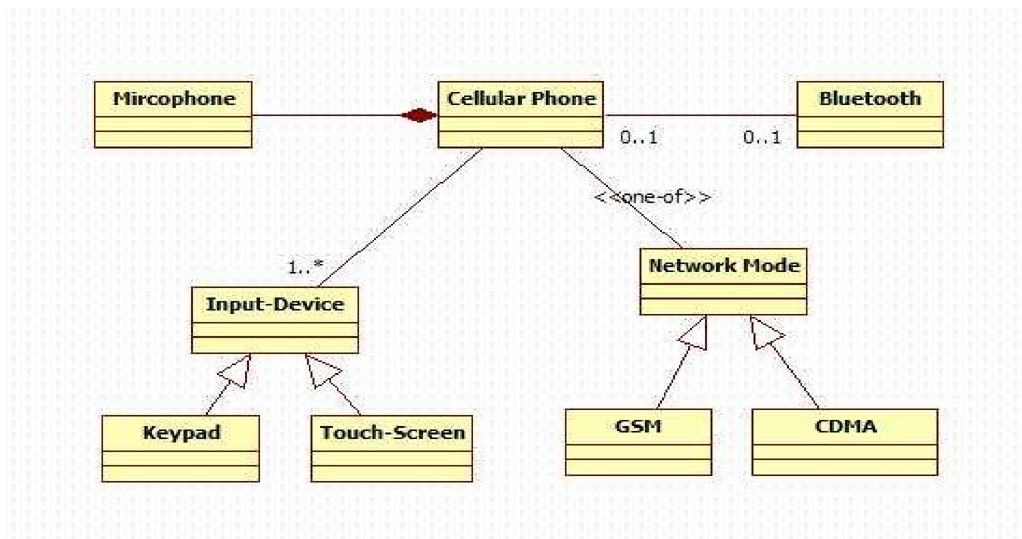

Figure 5.8: UML class diagram that shows one possible implementation of the cellular phone software

In Figure 5.8, the mandatory feature *Microphone* is represented in the UML class diagram through a class that is connected to *Cellular Phone* using a composition (part-of) relationship. Composition is a dependency between the container class (in this case the *Cellular Phone* is the container) and the instance (*Microphone*). *Cellular Phone* is considered the owner of *Microphone*, which is created or destroyed when *Cellular Phone* is created or destroyed.
The more-of choice that the extension point *Input-Drive* provides is represented through subclasses *Keypad* and *Touch-Screen*, both of which are connected to the super-class *Input-Drive* using generalization.

Generalization (also called inheritance) indicates a parent-child relationship where the subclass is the specialized form of the super-class, and the super-class is the generalized class of the subclass. The super-class *Input-Drive* is connected to *Cellular Phone* using association with a multiplicity one-to-many (1…*) as *Input-Drive* provides three choices.

The one-of choice that the variation point *Network-Mode* provides is represented through subclasses *GSM* and *CDMA*, both of which are connected to the super-class *Network Mode* using generalization. The super-class *Network Mode* is connected to *Cellular Phone* using association with a stereotype <<one-of>> [DK02] that indicates only one allowed choice. Notice that we could use a multiplicity instead of the stereotype, which would be *1* in this case.

The *Bluetooth* class is connected to *Cellular Phone* using association with a multiplicity of 0 to 1 (0...1), which explains that *Bluetooth* is an optional class, and therefore, an optional dependency between *Bluetooth* and the *Cellular Phone* classes is required.

As we can see from the phone class diagram, variable characteristics of the phone software system can be represented using several variability mechanisms. Also, the same variable feature might be implemented in different mechanisms. For example, Input-Drive's subclasses *Keypad* and *Touch-Screen* are implemented using a single inheritance mechanism where a subclass can inherit from only one super-class. The association multiplicity (1…*) indicates that three choices can be made: *Keypad*, *Touch-Screen*, or both.

Another way to model a feature with direct OR-sub-features is by replacing single inheritance with a multiple inheritance mechanism [Cza98], where a class can inherit features from more than one super-class.

As shown in Figure 5.9, we can add another subclass called *KeypadTouch-Screen*–that explicitly indicates the combination of *Keypad* and *Touch-Screen* in a phone product. *KeypadTouch-Screen* is represented using a multiple inheritance mechanism as it inherits features and characteristics from two parents, the *Keypad* and *Touch-Screen* classes.

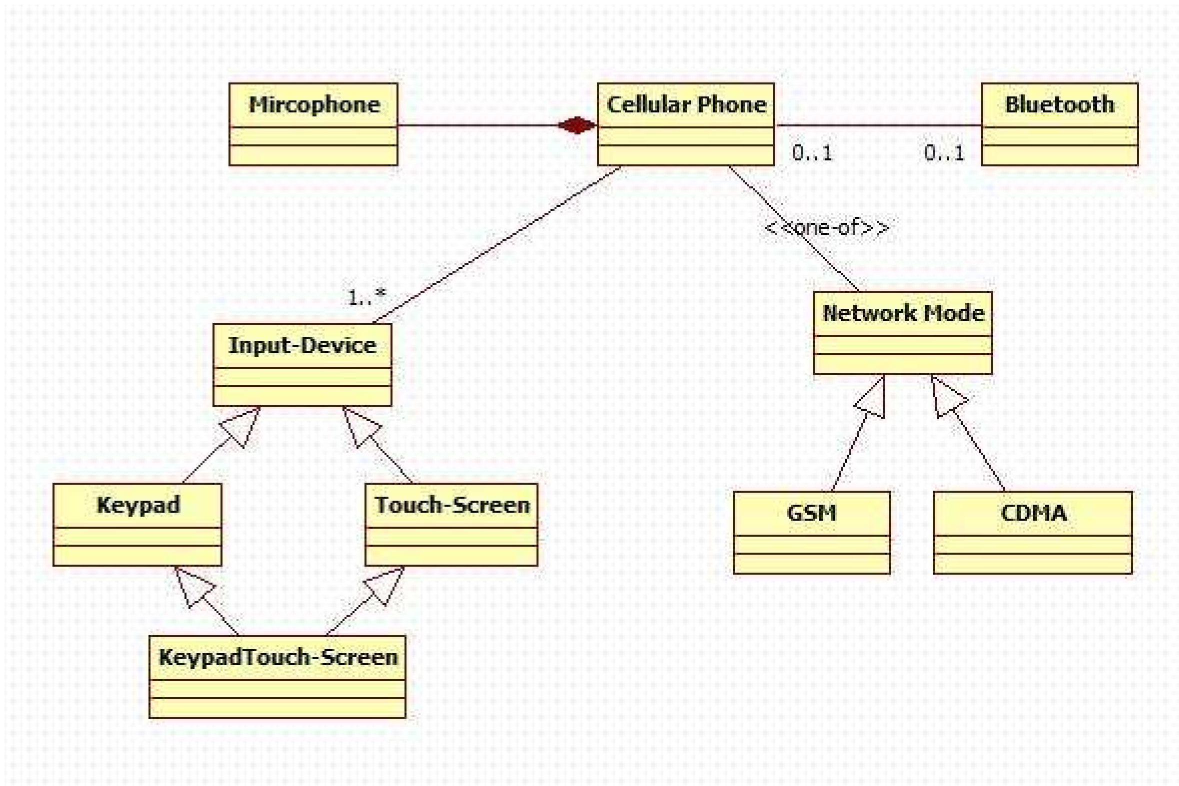

Figure 5.9: UML class diagram that shows another possible implementation of the cellular phone software using a multiple inheritance mechanism

Notice that the *KeypadTouch-Screen* subclass can be connected directly to the *Input-Device* class through a single inheritance mechanism as shown in Figure 5.10.

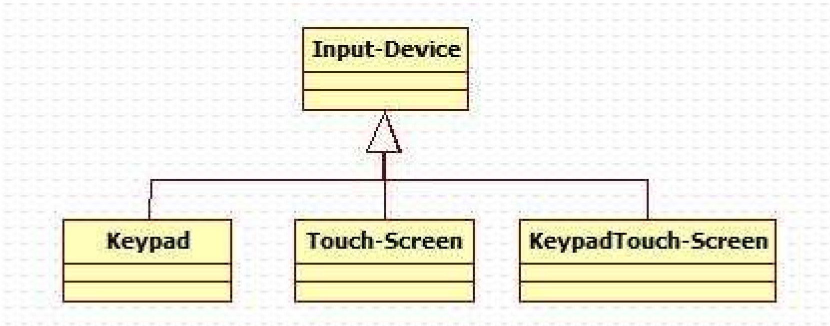

Figure 5.10: The extension point *Input-Device* with OR-features represented using a single inheritance mechanism

The single inheritance mechanism is preferred for use in UML modeling over multiple inheritance mechanism since the later has more complex relationships when representing features that have more than 2 OR-sub-features that result in deep and complex hierarchies.

Another alternative to a multiple inheritance mechanism is parameterized inheritance: by passing the super-class as a parameter to its subclasses. Multiple and parameterized inheritance in addition to other variability mechanisms used in object-oriented models such as static and dynamic parameterizations are discussed in [CE00] and [CE99-B].

In summary, variability mechanisms such as composition, inheritance, association, and parameterization used in OOAD modeling techniques such as UML class diagrams can support modeling variabilities for software systems, where the same variable feature might be implemented using different variability mechanisms. In contrast, the domain engineering process utilizes feature models as a part of the domain analysis phase to represent common and variable features across related software systems in a domain.

In comparison to OOAD variability mechanisms, feature models are more abstract, concise, and able to represent the concept and its corresponding features at the highest level of abstraction [CE99-A].

Another weakness of OOAD methods is when representing variation points (e.g., using UML class diagrams). When doing so, a decision should be made from the start of drawing the diagram to select what implementation mechanisms are used to represent a particular variation point (e.g., inheritance, aggregation, and classification). On the other hand, feature models offer an appealing technique to model variabilities without being restricted to any particular variability mechanism [CE99-B].

As stated in the previous chapter, there are some methods proposed in the software reuse community that attempt to combine domain engineering with OOAD for the purpose of providing OOAD methods with techniques to support software reuse through focusing on classes of systems rather than single systems to develop reusable software parts. Such attempts include *Object Oriented Role Analysis and Modeling* (OOram) [RWL96] and *Sherlock* [SVV+00], where both of them are domain analysis and engineering approaches that use object-oriented modeling techniques such as use-case and class diagrams for modeling common and variable features for a family of systems in a certain domain area.

Hassan Gomaa [Gom04] presents interesting work on designing software product lines using UML. Attempts to model software variabilities using UML are illustrated in [Cla01-A] and [Cla01-B]. Other studies that investigate ways of integrating feature models into UML models are discussed in [VS06], [PDH+11], and [GFD98].

5.3 Summary

This chapter provided practical examples on software product lines that summarized the ideas presented in the previous chapters on how to organize and store past experience in building systems in a particular domain in the form of reusable assets through the domain engineering method. The first example illustrated how domain engineering is utilized in software product lines through feature and domain analysis: the first phase of domain engineering that attempts to define, capture, and model common and variable requirements of the concepts in a particular domain area using feature models.

The first example presented a complete feature model for a simple video-converter software product line. Through that example, we discussed how a feature model is used to represent common and variable requirements for a family of systems in an abstract way. It also showed the possible configuration space of all software products of a software product line in terms of features. In the second example, we presented a feature diagram of a software product line for cellular phones and converted it into a UML class diagram to demonstrate the advantages that software product lines and feature modeling provide over OOAD and object-oriented modeling techniques in the context of software reuse and modeling variability in software systems.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The main objective of this thesis was to show how the feature modeling process contributes in constructing software product lines and software families.

In the literature, there are a limited number of examples provided by domain analysts to explain how feature models can be adopted to perform feature analysis studies for creating software product lines. Although these examples effectively explain the basic rules of using feature models in the domain analysis phase, most of the work is done from an industrial perspective (e.g., a vehicle product line).

In this thesis we explained the idea of feature modeling by presenting a case study that discusses the process from a software perspective. In our video-converter example, we performed a complete feature analysis study for creating a line of related video-converters; these share common components while allowing variabilities among them to produce different products. Feature models are the most successful techniques for modeling these variabilties among similar systems in a domain.

The second case study illustrated the difference between feature diagrams and object-oriented modeling notations that use UML diagrams and demonstrated that feature diagrams are more concise and express commonalities and variabilities at the highest level of abstractions.

6.2 Future Work

Constructing software product lines is a challenging task. This thesis presented the first step of this task which concentrates on the domain analysis phase. Domain analysis is the first phase of the domain engineering process that attempts to both analyze existing software systems to manage their common and variable parts and develop new reusable components for a specific domain. After performing a feature analysis of the concepts in a target domain, the types of software components needed are identified. Developing these reusable components requires designing a product line architecture that specifies the rules on how these components will be connected in addition to the relationships, interactions, and dependencies among them. Since this thesis explored the first phase of the domain engineering process, our future research can focus on the next two phases: domain design and implementation with more concentration on investigating architectural design patterns for creating software product lines.

BIBLIOGRAPHY

[Ant93] G. H. Anthes. Software reuse plans bring paybacks. Computerworld. Vol. 27, No. 49, pp.73-76. 1993

[AR07] Elena Alana and Ana Isabel Rodriguez. Domain Engineering Methodologies Survey. GMV Innovating Solutions. GMV-CORDET-WP202-RP-001, Issue 1, pp. 3-38. 2007

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In: Henk Obbink and Klaus Pohl (editors). The 9th international Software Product Line Conference (SPLC 2005), volume 3714 of Lecture Notes in Computer Science. pp. 7–20. 2005

[BEG12] Ebrahim Bagheri, Faezeh Ensan and Dragan Gasevic. Decision Support for the Software Product Line Domain Engineering Lifecycle. Automated Software Engineering. Vol. 19, No.3, pp. 335-377. September 2012

[BFK+99] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. PuLSE: A Methodology to Develop Software Product Lines. Proceedings of the 1999 symposium on Software reusability (SSR '99), pp. 122-131. 1999

[BHS+04] Yves Bontemps , Patrick Heymans , Pierre-Yves Schobbens and Jean-Christophe Trigaux. Semantics of FODA feature diagrams In Tomi Männistö and Jan Bosch, editors, Proceedings of Workshop on Software Variability Management for Product Derivation - Towards Tool Support, Boston. August, 2004

[Big94] Ted J. Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. Technical Report: MSR-TR-94-19. In third International Conference on Software Reuse, pp.102-109, Rio de Janeiro, Brazil. November 1994

[Big97] Ted J. Biggerstaff. A Perspective of Generative Reuse. Technical Report MSR-TR-97-26, Microsoft Corporation. December, 1997

[BK91] Rajiv D. Banker and Robert .J. Kauffman. Reuse and Productivity in Integrated Computer-Aided Software Engineering: An Empirical Study. MIS Quarterly - Special issue on the strategic use of information systems. Vol. 15, No.3, pp. 375-401. September 1991

[BNG+12] Ebrahim Bagheri, Tommaso Di Noia, Dragan Gasevic and Azzurra Ragone: Formalizing interactive staged feature model configuration. Journal of Software: Evolution and Process. Vol. 24, No. 4, pp. 375-400. 2012

[Bro87] Frederick P. Brooks Jr. No Silver Bullet - Essence and Accidents of Software Engineering. Computer, Vol. 20, No. 4, pp. 10-19. April 1987

[BS09] G.N.K. Suresh Babu and DR.S.K.Srivatsa. Analysis and Measures of Software Reusability. International Journal of Reviews in Computing (IJRIC.). E-ISSN: 2076-3336. 2009

[BSR04] Don Batory, Jacob Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering. Vol. 30, No.6, pp. 355–371. 2004

[BW96] A.W. Brown and K.C. Wallnau. Engineering of Componnet-Based Systems. *Component-Based Software engineering*, IEEE Computer Society Press. 1996

[CE99-A] Krzysztof Czarnecki and Ulrich W. Eisenecker. Components and Generative Programming. Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7). ACM SIGSOFT Software Engineering Notes. Vol. 24, No. 6, pp. 2-19. November 1999

[CE99-B] Krzysztof Czarnecki and Ulrich W. Eisenecker. Synthesizing Objects. Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99). PP. 18-42. 1999

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000

[CHE05] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. Software Process Improvement and Practice. Vol.10, No.1, pp.7-29. 2005

[CHW98] James Coplien, Daniel Hoffman and David Weiss. Commonality and Variability in Software Engineering. IEEE Software. Vol.15, No.6, pp. 37-45. November 1998

[CK05] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-Based Featur Modeling and Constraints: A Progress Report. International Workshop on Software Factories at Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05). 17, October 2005

[Cla01-A] Matthias Clauß. Generic Modeling using UML extensions for Variability. Workshop on Domain Specific Visual Languages at OOPSLA. 2001

[Cla01-B] Matthias Clauß. Modeling variability with UML. In GCSE 2001Young Researchers Workshop. 2001

[Cle10] Paul Clements. Product Lines 2.0. Presented at the Product Lines Approaches in Software Engineering (PLEASE) Workshop. Cape Town, South Africa. May 2010

[CN01] P. Clements and L. M. Northrop, Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, Massachusetts. August, 2001

[CN07] P. Clements and L. M. Northrop. A Framework for Software Product Line Practice, Version 4.2. Software Engineering Institute (SEI), Carnegie-Mellon University. 2007

[CSP+92] Sholom G. Cohen, Jay L. Stanley, Jr, **A.** Spencer Peterson and Robert W. Krut, Jr. Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain. Technical Report: CMU/SEI-91-TR-28 ESD-TR-91-28, Software Engineering Institute (SEI), Carnegie Mellon University. June 1992

[Cyb96] Jacob L. Cybulski. Introduction to Software Reuse. Technical Report TR 96/4. The University of Melbourne, Department of Information Systems, Melbourne. 1996

[Cza98] Krzysztof Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. PhD dissertation. Technische Department of Computer Science and Automation, Technical University of Ilmenau. October 1998

[CZZ+05] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. An Approach to Constructing Feature Models Based on Requirements Clustering. Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE ’05), pp. 31–40. June, 2005

[Dag00] James C. Dager. Cummin’s. Experience in Developing a Software Product Line Architecture for Real-time Embedded Diesel Engine Controls. In Patrick Donohoe, editor, Software Product Lines: Experience and Research Directions. Proceedings of the First Conference on Software Product Lines (SPLC1), pp. 23-46 Kluwer Academic Publishers. 2000

[DF87] R. Prieto-Diaz and P. Freeman. Classifying Software for Reusability. IEEE Software. Vol. 4, No. 1, pp. 6-16, January 1987

[DFF95] Ruen Prieto-Díaz, Bill Frakes and Christopher Fox. DARE: A Domain Analysis and Reuse Environment. Sponsored by Defense Advanced Research Projects Agency. Defense Small Business Innovation Research Program. ARPA Order No. 6685. Issued by U. S. Army Missile Command Under Contract # DAAH01-93-C-R302. Reuse Inc. Phase 2 Final Report. July 199

[DFG92] Ruen Prieto-Díaz, Bill Frakes and B.K Gogia. DARE: A Domain Analysis and Reuse Environment. Sponsored by Defense Advanced Research Projects Agency. Defense Small Business Innovation Research Program. DARPA Order No. 5916. Issued by U. S. Army Missile Command Under Contract # DAAH01-92-C-R040. Reuse Inc. Phase I Final Report. August 1992

[DK02] Arie van Deursen and Paul Klint. Domain--Specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology. Vol.10, No. 1, pp. 1-17. October 2002

[DKV00] Arie van Deursen, Paul Klint and Joost Visser. Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices. Vol. 35, No. 6, pp. 26-36. June 2000

[ESJ11] Peter Ebraert, Quinten David Soetens, and Dirk Janssens. Change-based FODA diagrams: bridging the gap between feature-oriented design and implementation. Proceedings of the 2011 ACM Symposium on Applied Computing (SAC'11), pp.1345-1352. March 2011

[FBD+91] William B. Frakes, Ted J. Biggerstaff, Ruben Prieto-Diaz, Kazuo Matsumura and Wilhelm Schaefer. Software reuse: Is it delivering? International Conference on Software Engineering - ICSE , pp. 52-59. 1991

[FDF98] William Frakes, Ruen Prieto-Díaz and Christopher Fox. DARE: A Domain Analysis and Reuse Environment. Partly sponsored by ARPA/US army Missle Command under SBIR contract number DAAH01-93-C-R302. Annals of Software Engineering. Vol. 5, pp.125-141. 1998

[Gac95] Cristina Gacek. Exploiting Domain Architectures in Software Reuse. Technical Report: USC/CSE-95-TR-501. In Proceedings of the ACM-SIGSOFT Symposium on Software Reusability (SSR'95), ACM Press, Seattle, WA. PP. 229-232. April 1995

[GE06] Daniel D. Galorath and Michael W. Evans. Software Sizing, Estimation, and Risk Management: When Performance Is Measured Performance Improves. Auerbach Publishing, Philadelphia, PA. 2006

[GFD98] Martin L. Griss, John Favaro and Massimo d'Alessandro. Integrating Feature Modeling with the RSEB. Proceedings of the 5th International Conference on Software Reuse (ICSR '98), pp. 76-85. 1998

[GHJ+04] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley. 2004

[GK05] Dharmalingam Ganesan and Jens Knodel. Identifying Domain-Specific Reusable Components from Existing OO Systems to Support Product Line Migration. International Workshop on Reengineering Towards Product Lines. 2005

[Gom04] Hassan Gomaa. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison Wesley. 2004

[GZP94] John D. Gannon, Marvin V. Zelkowitz and James M. Purtilo. Software Specification: A Comparison of Formal Methods. Greenwood Publishing Group Inc., Westport, CT. 1994

[HaR94] Frederick Hayes-Roth . Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program. Technical Reoprt. Teknowledge Federal Systems.Version 1.01. 1994

[Har02-A] Maarit Harsu. A survey of domain engineering. Technical report 31, Institute of Software Systems, Tampere University of Technology. December, 2002

[Har02-B] Maarit Harsu. FAST product-line architecture process. Technical report 29. Institute of Software Systems, Tampere University of Technology. January 2002

[Hen96] Scott Henninger. Accelerating the Successful Reuse of Problem Solving Knowledge through the Domain Lifecycle. Proceedings of the International Conference on Software Reuse, pp. 124 – 133. Orlando, FL. April 1996

[HPL+95] Barbara Hayes-Roth, Karl Pfleger, Philippe Lalanda, Philippe Morignot, and Marko Balabanovic. A Domain-Specific Software Architecture for Adaptive Intelligent Systems. IEEE Transactions on Software Engineering. Vol. 21, No. 4, pp. 288-301. April 1995

[JGP12] B.Jalender, A.Govardhan, P.Premchand. Designing code level reusable software components. International Journal of Software Engineering and Applications (IJSEA). Vol.3, No.1, pp. 219 – 229. January 2012

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Nowak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report: CMU/SEI-90-TR-21, Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, Pennsylvania. November 1990

[KKL+98] Kyo C. Kang , Sajoong Kim , Jaejoon Lee , Kijoo Kim , Gerard Jounghyun Kim and Euiseob Shin. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering. Vol.5, pp. 143-168. 1998

[KLM08] Gerald Kotonya, Simon Lock and John Mariani. Scrapheap Software Development: Lessons from an Experiment on Opportunistic Reuse. "Software, IEEE". Vol. 28, No. 2, pp. 68-74. 2011

[Krc95] Philippe B. Kruchten. The 4 + 1 view model of architecture. IEEE Software. Vol. 12, No. 6, pp. 42–50. November 1995

[Kru92] KRUEGER, C.W. Software reuse. ACM Computing Surveys (CSUR). Vol.24, No. 2, pp. 131 – 183. June 1992

[Kru06] Charles W. Krueger. Introduction to the Emerging Practice of Software Product Line Development. In Methods & Tools, http://www.methodsandtools.com. Fall 2006

[KS91] Yongbeom Kim and Edward A. Stohr. Software Reuse: Issues and research directions. Center for Research on Information Systems. Information Systems Department, Leonard N. Stern School of Business, New York University. Center for Digital Economy Research Stem School of Business. Working Paper IS-9 1-15. June 1991

[KZ96] Robert Krut and Nathan Zalman. Domain Analysis Workshop Report for the Automated Prompt and Response System Domain. Software Engineering Institure (SEI) at Carnegie-Mellon University. Special Report CMU/SEI-96-SR-001. May 1996

[LG84] Robert G. Lanergan and Charles A. Grasso. Software Engineering with Reusable Designs and Code. IEEE Transactions on Software Engineering. Vol. 10, No. 5, pp. 498-501. September 1984

[**Lin02]** Frank van der Linden. Software Product Families in Europe: The Esaps and Café Projects. IEEE Software. Vol. 19, No. 4, pp. 41-49. July 2002

[LLC04] László Lengyel, Tihamér Levendovszky and Hassan Charaf. Constraint Handling in Feature Models. The 5th International Symposium of Hungarian Researchers on Computational Intelligence. Budapest, Hungary, pp. 279-290.November 2004

[Mat04] Mari Matinlassi. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. Proceedings of the 26th International Conference on Software Engineering (ICSE'04), pp. 127-136. 2004

[Mat84] Yoshihiro Matsumoto. Some Experience in Promoting Reusable Software: Presentation in Higher Abstract Levels. IEEE. Transactions on Software Engineerin. Vol. 10, No.5, pp. 502–512. September 1984

[McC89] Carma McClure. CASE is Software Automation. Englewood Cliffs, New Jersey, Prentice Hall. 1989

[McI68] Douglas McIlroy. Mass Produced Software Components. Software Engineering: Report on a Conference Sponsored by the NATO Science Committee. Garmisch, Germany, Oct. 7-11, 1968. Brussels, Belgium: Scientific Affairs Division, NATO. pp. 138-150. 1968

[MHS05] Marjan Mernik, Jan Heering and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. ACM Computing Surveys (CSUR). Vol. 37, No. 4, pp. 316–344. December 2005

[ML97] M.H. Meyer and A.P.Lehnerd . The Power of Product Platforms: Building Value and Cost Leadership. The Free Press, New York, 1997

[MMM98] A. Mili, R. Mili and R.T. Mittermeir. A Survey of Software Reuse Libraries. Annals of Software Engineering. Vol.5, pp. 349–414. 1998

[MMY+99] A Mili, H. Mili, S. Yacoub, and E. Addy. An Introduction to Software Reuse.Technical Report, West Virginia University. August 1999

[MS90] Neel Madhav and Sriram Sankar. Application of Formal Specification to Software Maintenance. Technical Report No. CSL-TR-90-436. Program Analysis and Verification Group (PAVG) Report No. 48). August 1990

[MYA+99] Ali Mili, Sherif Yacoub, Edward Addy and Hafedh Mili. Toward an Engineering Discipline of Software Reuse. IEEE Software. Vol. 16, No. 5, pp. 22-31. September/October, 1999

[NAG13] The NAG Library. http://www.nag.co.uk/numeric/numerical_libraries.asp (22, Apr. 2013)

[Nei80] James M. Neighbors. Software Construction Using Components. PhD dissertation. Department of Information and Computer Science, University of California, Irvine. 1980

[Nei84] James M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. IEEE Transactions on Software Engineering. Vol. SE-10, No. 5, pp. 564 – 574. September 1984

[Nei89] James Neighbors. Draco: A Method for Engineering Reusable Software Systems. In: Ted J. Biggerstaff and Alan J. Perlis (eds.). Software Reusability: Concepts and Models. ACM Press Frontier Series. Vol.1, pp. 295-319. Addison-Wesley. 1989

[NR68] P. Naur and B. Randell, eds., Software Engineering. Report on a conference sponsored by the NATO Science Committee. Garmisch, Germany. Scientific Affairs Division NATO, Brussels, Belgium. 7-11 October 1968

[Nor02] Linda M. Northrop. SEI's software product line tenets. IEEE Software. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. Vol. 19, No. 4, pp. 32-40. July/August, 2002

[Nor08] Linda M. Northrop. Software Product Lines Essentials. Software Engineering Institute (SEI), Carnegie Mellon University. Pittsburgh, PA 15213-2612. 2008

[OxD12] "Domain". Oxford Online Dictionaries. Copyright © 2012 Oxford University Press. The University of Oxford, England. http://oxforddictionaries.com/definition/english/domain. (4, August, 2012)

[OxD12] "Feature". Oxford Online Dictionaries. Copyright © 2012 Oxford University Press. The University of Oxford, England. http://oxforddictionaries.com/definition/english/feature. (8 Nov. 2012)

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM. Vol.12, No.15, pp. 1053-1058. December 1972

[Par76] David L. Parnas. On the design and development of program families. IEEE Transactions on Software Engineering. Vol. SE-2, No. 1, pp. 1-9. March, 1976.

[PBL05] Klaus Pohl, Günter Böckle and Frank J. van der Linden. Software Product Line Engineering: Foundations, Principles and Techniques. Springer, 2005

[PC91] A. Spencer Peterson and Sholom G. Cohen. A Context Analysis of the Movement Control Domain for the Army Tactical Command and Control System (ATCCS). Software Engineering Institure (SEI). Special Report CMU/SEI-91-SR-003. April 1991

[PDH+11] Thibaut Possompès, Christophe Dony, Marianne Huchard, Hervé Rey, Chouki Tibermacine, and Xavier Vasques. Design of a UML profile for feature diagrams and its tooling implementation. Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011). 2011

[Pin93] B. Joseph Pine. Mass Customization: The New Frontier in Business Competition. Harvard School Business Press, Boston, Massachusetts, 1993

[Pre05] Roger S. Pressman. Software Engineering: A practitioner's Approach. Sixth Edition. McGraw-Hill Education. 2005

[Pri90] Ruben Prieto-Diaz. Domain Analysis: An Introduction. ACM SIGSoft Software Engineering Notes. Vol.15, No. 2, pp. 47-54. April 1990

[PY93] Jeffrey S. Poulin and Kathryn P. Yglesias. Experiences with a Faceted Classification Scheme in a Large Reusable Software Library (RSL). International Business Machines Corporation. Presented at the Seventeenth Annual International Computer Software and Applications Conference (COMPSAC'93), Phoenix, AZ, pp. 90-99. November, 1993

[RBS+02] Matthias Riebisch , Kai Böllert , Detlef Streitferdt and Ilka Philippow. Extending Feature Diagrams With UML Multiplicities. Proceedings of the 6th Conference on Integrated Design & Process Technology, IDPT-2002. June, 2002

[RDV12] Steven Raemaekers, Arie van Deursen, and Joost Visser, An Analysis of Dependence on Third-party Libraries in Open Source and Proprietary Systems. In proceedings of the Sixth International Workshop on Software Quality and Maintainability (SQM 2012). 2012

[RWL96] Trygve Reenskaug, P. Wold and O.A. Lehne. Working with objects. The OOram Software Engineering Method. Prentice Hall. 1996

[Sam97] Johannes Sametinger. Software Engineering with Reusable Components. Springer, Berlin .March 3, 1997

[SCK+96] Mark Simos, Dick Creps, Carol Klinger, Larry Levine, and Dean Allemang. Organization Domain Modeling (ODM) Guidebook, Version 2.0. Informal Technical Report for Software Technology for Adaptable, Reliable Systems (STARS). STARS-VC-A025/001/00. 14 June, 1996

[SDL08] System Development Life Cycle. Texas Project Delivery Framework, Version 1.1. Texas Department of Information Resources. May 30, 2008

[Sim95] Mark Simos. Where the Rubber Meets the Road: Applying Organization Domain Modeling (ODM) on the STARS Army/Unisys Demonstration Project. Organon Motives. 1995.

[SL03a] IMSL Fortran Library User's Guide STAT/LIBRARY Volume 1 of 2. Statistical Functions in Fortran. Visual Numerics, Inc 2003

[SL03b] IMSL Fortran Library User's Guide STAT/LIBRARY Volume 2 of 2. Statistical Functions in Fortran. Visual Numerics, Inc 2003

[Som10] Ian Sommerville. Software Engineering, 9th edition. San Francisco: Addison-Wesley Publishing Company. 2010

[SVV+00] Giancarlo Succi , Andrea Valerio , Tullio Vernazza , Massimo Fenaroli and Paolo Predonzani. Framework extraction with domain Analysis. ACM Computing Surveys, Vol. 32, No. 1. March 2000

[TC92] W. Tracz and L. Coglianese. DSSA Engineering Process Guidelines. Technical Report. ADAGE-IBM-9202. December 1992

[Tra95] Will Tracz. DSSA (Domain-Specific Software Architecture): Pedagogical Example. ACM SIGSOFT Software Engineering Notes. Vol.20, No.3, pp. 49-62. July 1995

[VH01] Wim van Vuuren and Johannes I.M Halman. Platform-Driven Development of Product Families: Linking Theory with Practice. Eindhoven Centre for Innovation Studies, The Netherlands. Working Paper 01.06. September 2001

[VS06] Valentino Vranic and Jan Snirc. Integrating Feature Modeling into UML. Conference Proceedings NODe 2006, GSEM 2006. Vol. 88. 2006

[Wen89] P. Wegner. Capital-intensive software technology. In: Ted J. Biggerstaff and Alan J. Perlis. Software Reusability: Concepts and Models. Vol.1, pp. 43-97. ACM Addison Wesley. 1989

[WL99] David M. Weiss and Chi Tau Robert Lai. Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley, Reading, Massachusetts. 1999

[ZHT08] Lamia Abo Zaid, Geert-Jan Houben, Olga De Troyer, and Frederic Kleinermann. An OWL-Based Approach for Integration in Collaborative Feature Modeling. In the 4th Workshop on Semantic Web Enabled Software Engineering (SWESE), workshop at the International Semantic Web Conference (ISWC), pp. 93-100. October 2008

[Znt00] Formal specification-Z notation-syntax, type and semantics. Consensus working draft2.6. Developed by members of the Z Starndars Panel. BSI Panel IST/5/-/19/2 (Z Notation). ISO Panel JTCI /SC22/WG19 (Rapporteur Group for Z). Project number JTC1.22.45. Project editor: Ian Toyn. August 24, 2000

VITA

HAZIM SHATNAWI

EDUCATION

B.A, Computer Science and Information Systems, Jordan University of Science and Technology, JORDAN, August 2006

WORK EXPERIENCE

Graduate Teaching Assistant, February 2013 – Present
Computer Science Department, University of Mississippi

Graduate Teaching Assistant and Instructor, Sept. 2009 – August 2012
Modern Languages – Arabic Department, University of Mississippi

Graduate Research Assistant, August 2008 – May 2009
The National Center for Physical Acoustics, University of Mississippi