An Approach to Software Artefact Specification for Supporting Product Line Systems

Dr. Waraporn Jirapanthong

This research has been funded by
Dhurakij Pundit University
2008

รายงานผลการวิจัย

เรื่อง

การสร้างสิ่งประดิษฐ์เชิงซอฟต์แวร์สำหรับสนับสนุนการพัฒนากลุ่มซอฟต์แวร์แบบโปรดักต์ไลน์

โดย

ดร. วราพร จิระพันธุ์ทอง

รายงานการวิจัยนี้ได้รับทุนอุดหนุนจาก

มหาวิทยาลัยธุรกิจบัณฑิตย์

พ.ศ. 2551

Abstract

This research is aimed to resolve the difficulties of software specification, particularly software product line systems. One of the research's contributions is the meta model for product line systems. The model is used as the reference model for specifying of requirements and design artefacts. Moreover, we envisage the use of prototype tool as a general platform for creating the artefacts. Five tasks were created to demonstrate different situations of software product line development, involving different types if documents and different stakeholders. The experiments of artefact creation have been evaluated by considering two criteria: precision and recall measures; and satisfaction of users.

บทคัดย่อ

งานวิจัยนี้มีจุดมุ่งหมายที่จะแก้ไขปัญหาและความยุ่งยากในการพัฒนาซอฟต์แวร์ โดยเฉพาะการพัฒนาซอฟต์แวร์แบบโปรดักต์ไลน์ ในงานวิจัยนี้ผู้แต่งได้นำเสนอแม่แบบจำลองของการพัฒนาซอฟต์แวร์สำหรับระบบซอฟต์แวร์โปรดักต์ไลน์ ซึ่งประกอบด้วยสิ่งประดิษฐ์เชิงซอฟต์แวร์แบบต่างๆ โดยผู้แต่งกำหนดขอบเขตของงานวิจัยภายใต้กระบวนการพัฒนาสิ่งประดิษฐ์เชิงซอฟต์แวร์แบบความต้องการ ได้แก่ เอกสาร, ข้อความ, โน้ต หรืออื่นๆที่บันทึกความต้องการและรายละเอียดของซอฟต์แวร์ที่ต้องการและเอกสารการออกแบบระบบซอฟต์แวร์ นอกจากนี้ผู้แต่งได้สร้างแบบจำลองเครื่องมือเพื่อใช้ในการทดลองสร้างสิ่งประดิษฐ์เชิงซอฟต์แวร์ดังกล่าว ภายใต้รูปแบบการทดลองห้าแบบ การทดลองทั้งห้าแบบ ถูกสร้างเพื่อจำลองสถานการณ์การพัฒนาโปรดักต์ไลน์ที่แตกต่างกัน รวมถึงเกี่ยวพันกับการสร้างสิ่งประดิษฐ์เชิงซอฟต์แวร์ประเภทต่างๆกันและกลุ่มของผู้เกี่ยวข้องในการพัฒนาโปรดักต์ไลน์ที่แตกต่างกัน การทดลองถูกประเมินด้วยพิจารณาค่าความถูกต้อง (precision measure) และค่าความแม่นยำ (recall measure) รวมถึงค่าความพึงพอใจของผู้ทดลองใช้งาน

Acknowledgement

I am grateful to Dhurakijpundij University for the financial support for this research.

Declaration

Some of the material in this report has been previously published in the paper:

- W. Jirapanthong, "Techniques and Approaches for Developing Software Product Line", the 2007 International Conference on Software Engineering Research and Practice (SERP'07), Las Vegas, Nevada, USA, 2007

I grant powers of discretion to Dhurakijpandit University to allow this research to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Contents

List of Figures

List of Tables

Chapter I Introduction

In recent years we have been experiencing the proliferation of a large number of software systems that share a common set of features and have also their own distinct characteristics. Examples of such systems are found in the telecommunication domain in which products including personal digital assistants (PDAs), mobile phones, and pagers have many common characteristics. Other examples are found in the automotives, electronics, medical imaging, and elevator control domains. These systems are known in the literature as *product line systems* (Ardis and Weiss 1997, Bass et al. 2003, CAFE 2003, Clements and Northrop 2002, Clements and Northrop 2004, Staudenmayer and Perry 1996, Weiss and Lai 1999) and are characterized as being software systems that share a common set of features and are developed based on the reuse of core assets and addition of new functionalities.

According to the software product line development, the main activities are analysis, design, and implementation of similar and different aspects of the systems:

1) Analysis – this activity is aimed to explore and justify the requirements of product line systems which represent the common and variable aspects of the systems. In particular, the artefacts being generated from the analysis process of software product line systems are namely *reference requirements*.
2) Design – this activity is aimed to elaborate the requirements from the analysis process and to design the software systems for the product line. More specifically, the design presents the commonality and variability in design aspects. The artefacts being generated from the design process of software product line systems are namely *software product line architecture*.
3) Implementation – this activity is aimed to implement the requirements and design artefacts produced from previous processes as components and to assemble a software system which includes common and variable aspects of a

product line. In particular, the artefacts being generated from the implementation process are a set of *reusable software components* and software systems of product line.

In principle, the reference requirements, software product line architecture, and reusable software components are gradually generated during the process of software product line development. They are later reused for developing a software product member in an effective and efficient way. It is meant to support the risk reduction during the software development.

However, according to the majority of approaches being used in organisations, there are some issues found:

1) the software product line approaches proposed recently are not flexible, practical, and appropriate enough to the conventional approaches being used in the organisations;
2) different organisations have various behavioral cultures and traditions depending on the strategies and missions of the organisations. In particular, many organisations found difficulties to adopt the software product line approaches to fit into their strategies and missions.

Consequently, there are still errors and mistakes during the development of software systems that requires the reuse of software components. Also, invalid use of software product line approaches decreases the benefits of having the product line systems.

We advocate the fact that the software product line systems should be established in an organisation, since the systems support the reuse of software components which leads less error-prone and less time consuming. We expect to reduce the difficulties of development activities, in particular, analysis and design.

This research is aimed to develop the meta model of software specification in order to support the development of software product line systems. In particular, the main contributions of our work are:

Firstly, we have investigated which artefacts are playing the main roles in the process of product line system development and which artefacts are applied in organisations.

Secondly, we propose a model for specifying of software product line artefacts. The model is proposed by taking into consideration:

(a) the semantics of document types;

(b) the activities during the software development process; and

(c) the available techniques and tools being used in an organisations.

Next, we have justified the model for generating the artefacts in the domain of product line systems through five different scenarios. Each scenario presents the testing of generating the documents that occurs during the process of product line system development.

The remainder of this report is organized in five chapters as described below:

Chapter 2 presents a survey on product line, including the current methods and techniques for product line system development.

Chapter 3 presents the model that represents the different types of documents and describes the structure of each document type for specifying the commonality and variability for product line systems.

Chapter 4 presents a prototype tool to assist an end-user to apply the model for specification of software product line artefacts.

Chapter 5 contains a description of the experiments that we have developed to demonstrate the work and evaluates the experimental results of our work.

Chapter 6 discusses the conclusions and directions for future work.

Chapter II Literature Review

This chapter describes a literature of software product line including current problems, and existing approaches, techniques and tools in the domain of product line systems. The motivation and related terminologies are given in Section 2.1. In Section 2.2 presents the activities during software product line. Also, Section 2.3 illustrates existing methodologies proposed for software product line development.

2.1. Introduction to Product Line

Software reuse is the process of software development by using existing software artefacts (Department_of_Defense 1996). Over the last years, approaches and techniques for software reuse have been developed and extended. According to (Clements and Northrop 2002, ESAPS, Weiss and Lai 1999), software reuse at the largest level of granularity is supported by *product line.* This is to serve the reuse practice in an organization having a large number of products, which drives issues such as highly expensive, complex, and tedious tasks. The different exact definition of product line will be given in Section 2.1.1.

The idea of product line was motivated by the need to systematize a number of products more effectively and the fact that these products have a certain set of common and special functionalities. For example, a mobile-phone company has created a mobile-phone product line that contains a set of mobile-phones. Some lower-end mobile-phones have similar basic functionalities but different hardware capacities to offer competitive price. Mobile-phone network communications in some countries provide different standards of transmission and signaling and depend on regional diversity; thereby, a company provides different support for different regions.

2.1.1. Terminologies in Product Line

We describe below terminologies used in the domain of product line.

Product Line

Initially, Parnas (Parnas 1976) defined *program family* as a set of software programs constituted as a family whereby a program is developed by applying common properties of prior programs and adding extra properties to the program.

In (Bass et al. 2003, CAFE 2003, Clements and Northrop 2004, Staudenmayer and Perry 1996, Weiss and Lai 1999), *product family* is defined as a set of products sharing some common aspects and having some different aspects. The product family is aimed at gaining the market share under the same business domain and marketing factors. They also suggested *product members* that are products which are built-up by applying shared assets i.e. requirements, architecture, models, and source code in a product family. Eventually, *product line* and *business unit* are other terms found in the literature that have the same meaning as that of product family (Ardis and Weiss 1997, Bass et al. 2003, Clements and Northrop 2004).

According to (Bass et al. 2003, CAFE 2003, Clements and Northrop 2002, Staudenmayer and Perry 1996), product family takes into account both hardware and software systems. In (Clements and Northrop 2002), they suggested that a *software product line* is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and is developed from a common set of core assets in a prescribed way.

In this work, we focus on and call the software systems that are developed for product line as *product line systems*.

Features

The term *feature* has been initially used in (Kang et al. 1990). The authors defined a feature as a prominent and distinctive aspect or characteristic of a system that is visible to various stakeholders (e.g. end-users, domain experts, developers). In (Bosch 2000, Gibson et al. 1997, Griss 2000, Svahnberg et al. 2001), a feature is concerned with a logical behavior of a system that is specified as a requirement or set of requirements (i.e. functional and non-functional requirements). In (Bailin 1990), the author suggested a different definition whereby a feature refers to any distinctive or unusual aspect of a system that requires a decision for system engineering. In this work, we use the term feature as a user-visible aspect or as a characteristic of product line systems. A feature is related to other features and represented in a tree structure of And/Or nodes to express common and variable aspects within product line systems

Core Assets

Core assets (Clements and Northrop 2004) are those assets that form the basis for a product line. Core assets include requirements, architecture, and reusable software components, domain models, documentation and specifications, schedules, test cases, and work plans. In (Riebisch et al. 2002), the authors also suggested that core assets i.e. requirements, architectures, analysis models, design models, test cases, and source codes are reused between different product members in product line. A variant term of core asset is *platform* that is defined in the domain of Model-Driven Architecture (MDA).

Commonality Vs. Variability

According to (Bosch 1998, Clements and Northrop 2004, Weiss and Lai 1999), *commonality* is concerned with a set of similar functionalities or aspects between product members of product line and *variability* is defined as different functionalities or aspects between product members of product line.

2.2. Activities in the Process of Product Line System Development

According to the maturity level of an organization, the approaches for the development of product line can be categorised, namely *proactive*, *reactive*, and *extractive*. We describe below three types of approaches for the product line system development.

Proactive

The *proactive* approach (Krueger 2001) is an approach of the product line system development when an organization decides to analyse, design, and implement a line of products prior to the creation of individual product members. The product line is built-up and the core assets representing the commonality and variability are created. All product members are then created under the scope of the product line. The approach is viewed as a top-down developing strategy which requires the setting of broad goals and the goals are refined in later phases of the development.

Reactive

The *reactive* approach (Krueger 2001) is an approach of the product line system development when an organization enlarges the product line systems in an incremental way based on the demand for new product members or new requirements for existing products. The core assets need to be extended and evolved in such a way as to correspond to new requirements or new systems. This is caused by the fact that the customer requirements considerably influence the architecture and the design of products. On the other hand, a company that sticks strictly to the principles of made-to-order manufacturing will not allow an uncontrolled proliferation of variety due to the demands of individual customers. However, in reality, many companies have a production control concept based on customer requirements. So the problem occurs when the architecture and design of product line systems should be maintained. This level of development takes shorter time than the previous one since system developers only extend and adapt the available products.

Extractive

The *extractive* approach (Krueger 2001) is an approach of the product line system development when an organisation creates product line systems based on existing product members by identifying and using common and variable aspects of these products. The stakeholders i.e. domain experts and system developers analyse and define the product line by taking into consideration individual products' requirements. The approach is viewed as a bottom-up developing strategy that begins with existing artefacts e.g. requirements specification, design and source code, then creates the higher granularity level of each artefact as the core assets.

In the following section, we describe the activities occurred during the product line system development process. In addition to (Bosch 2000, Clements and Northrop 2002, Jazayeri et al. 2000, Thiel and Hein 2002), *software product line engineering* is a methodology for developing product line systems that focuses on activities of analysis, design, and implementation of product line as well as the use of the core assets inclusive common and variable artefacts potentially and effectively for product members.

Figure 2-1 illustrates the main activities of software product line engineering i.e. domain engineering and application engineering.

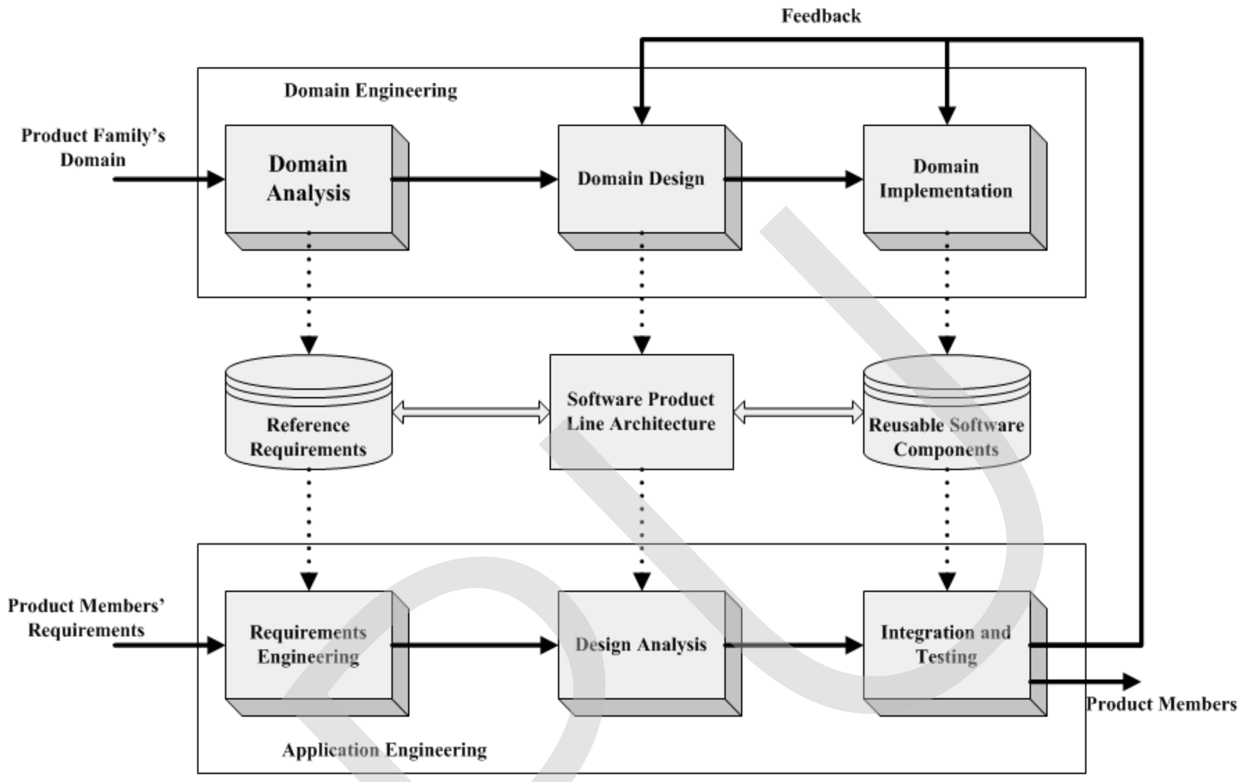


Figure 2- 1: Activities in software product line engineering adopted from (Clements and Northrop 2004)

2.2.1. Domain Engineering

Domain engineering is a systematic process for the creation of the core assets (Clements and Northrop 2004). There are three steps for domain engineering:

Domain Analysis

Domain Analysis is the process of identifying, collecting, organizing and representing the relevant information in a domain, based upon the study of existing systems and their developing histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain (Kang et al. 1990).

As shown in Figure 2-1, software artefacts that are produced during the activity of domain analysis are called *reference requirements*. The reference requirements define the products and their requirements in product line. The reference requirements contain commonality and variability of the product line. The following sub-activities occur during the domain analysis:

I. Scoping

According to (Arango and Prieto-Diaz 1991, Ardis and Weiss 1997), domain analysis for product line basically starts from *scoping*. Scoping is to identify the context of product members in product line e.g. functionalities and performances. The activity is concerned with domain knowledge obtained from domain experts and other sources such as books, user manuals, and design documents (Nuseibeh and Easterbrook 2000). The domain experts analyse and define the boundary of the product line and the standard terminologies in the product line. The product members are therefore defined.

II. Commonality and Variability

The activity of defining commonality and variability is to thoroughly discover and define commonality and variability in product line (Ardis and Weiss 1997, Weiss 1995). Many existing approaches are proposed to support the activity. Examples of such approaches are (Ardis and Weiss 1997, Bosch 2000, Clements and Northrop 2002, Svahnberg and Bosch 2000, Weiss 1995). The determination of whether a characteristic is a commonality or variability mostly depends on a strategic decision of organisations.

In particular, defining commonality is the determination of whether a requirement is served as the commonality of product line. Defining variability is the determination of whether a requirement is served as the variability of product line. Variability is represented as a set of *variation points*. Each variation point is a situation that product members can be specialized differently and dependent on a number of *variants*. Variants are possible variables for each variation point. A variation point is classified as: (i) *optional* – an aspect may exist in a product; (ii) *alternative* – an aspect can be specialized as one of the variants; and (iii) *optional alternative* – an aspect can be specialized as one of the variants or does not exist (Svahnberg et al. 2001). Variation points can appear at different phases of product line system development i.e. analysis, design, and implementation. At the state of domain analysis, a variability point is concerned with the highest abstraction level of an artefact.

III. Planning for Product Members and Features

According to (Arango and Prieto-Diaz 1991), one of the activities in domain analysis is to identify features of product members in product line. The features of a product line are planned for possible product members. In other words, the relevant requirements of product members are associated to the features of product line. The common and variable aspects of product line are accommodated and planned for product members.

Domain Design

> *Domain design* is the process of developing a design model from the products of domain analysis and the knowledge gained from the study of software requirements or design reuse and generic architectures (Garlan and Shaw 1993).

Software artefacts that are produced during the activity of domain design are called *software product line architecture* (see Figure 2-1). In (Bass et al. 2003, Jazayeri et al. 2000), *software architecture* forms the backbone of integrating software systems and consists of a set of decisions and interfaces which connect software components together. Software product line architecture differs from an architecture of single systems that it must represent the common design for all product members and variable design for specific product members (Linden et al. 2004). The following sub-activities occur during the domain design:

I. Software Product Line Architecture Definition

The activity of software product line architecture definition is to design the software architecture that describes commonality and variability of product members. The software product line architecture is composed of a set of architectural decisions, a set of reusable design artefacts, and a set of optional design artefacts.

The variability in software product line architecture is called *designed variability points* (Svahnberg et al. 2001). The software product line architecture can be elaborated into different levels of granularity. At higher levels, the software product line architecture does not entail shared artefacts between product members while at the low levels, the software product line architecture make a distinction between specific designs of product members.

II. Software Product Line Architecture Evaluation

The activity of software product line architecture evaluation is to evaluate the software architecture that describes commonality and variability of product members. The evaluation of software product line architecture is to assure that the architecture has the right properties and characteristics of product line.

For the evaluation of software product line architecture, the following must be considered: (i) the context for software product line architecture must be scoped and planned during domain analysis; (ii) the commonality of product line must be elaborated in several levels of the architecture; and (iii) the variability of product line must be identified and provided with a set of variants for each designed variability point in the software product line architecture.

However, Bosch (Bosch 2000) suggested that the maturity of software product line architecture can be viewed as three levels: (i) an *under-specified* architecture that defines common aspects but does not specify differences between product members; (ii) a *specified* architecture that defines both common and variable aspects for product members; however, does not define possible variables for variable aspects; and (iii) an *enforced* architecture that defines both common and variable aspects covering possible variables for all product members.

Many approaches and techniques are proposed to support domain design for product line. Relevant existing methodologies e.g. *model-based software engineering* (MBSE 1993), *organizational domain modeling (ODM)* (Simos 1995), *synthesis* (Campbell et al. 1990), *domain-specific software architecture (DSSA) program* (Tracz et al. 1993), *evolutionary domain life-cycle (EDLC)* (Gomaa et al. 1989) are applied for the development of software product line architecture. Some general-purpose techniques such as data flow diagrams, structured analysis and design techniques, entity relationship modeling (ERM), object models (e.g. UML (UML)), view point-oriented models (Finkelstein et al. 1990) can be also applied for the activity. Recently a number of methodologies such as (Atkinson et al. 2000, Batory et al. 2000, Bayer et al. 1999, Griss et al. 1998, Kang et al. 1998, QADA, Simos 1995, Weiss 1995, Weiss and Lai 1999) are proposed to particularly support the activity of domain design in the domain of product line.

Domain Implementation

> *Domain implementation* is the process of identifying reusable components based on the domain model and generic architecture (Clements and Northrop 2004).

Software artefacts that are produced during the activity of domain implementation are called *reusable software components* (see Figure 2-1). The activity is focused on the creation of reusable software components e.g. source codes and linking libraries that are later assembled for product members. In (Szyperski 1997), a reusable software component is a unit of composition with interfaces and independent context. The reusable software component is created and then integrated with other reusable software components for a particular product member. The set of reusable components are defined independently and provide the connectors for integration with other components to fit into a specific functionality. The components are viewed as black boxes whose data and implementation details are completely hidden and only interfaces are allowed. The development of components can be applied with relevant existing methods such as object-oriented methods e.g. (Bosch 2000, Szyperski 1997).

At the end of the domain engineering process, an organization is ready for developing product members. In the following section, we describe the activities for developing the product members in the software product line engineering.

2.2.2. Application Engineering

As shown in Figure 2-1, application engineering is another major activity of software product line engineering. According to (Northrop 2002), *application engineering* is a systematic process for the creation of a product member from the core assets created during the domain engineering. Domain engineering assures that the activities of analysis, design and implementation of product line are thoroughly performed for all product members, while application engineering assures the reuse of the core assets of the product line for the creation of product members.

The application engineering process for product line is comparably considered with the process for a single system (Clements and Northrop 2004). There are activities such as: (i) *requirements engineering*, which is a process that consists of requirements elicitation, analysis, specification, verification, and management (Fairley and Thayer 1997, Sommerville and Sawyer 1997, Sutcliffe and Maiden 1998); (ii) *design analysis*, which is a process that is concerned with how the system functionality is to be provided by the different components of the system (Sommerville 2000); and (iii) *integration and testing,* which is a process of taking reusable components then putting them together to build a complete system, and of testing if the system is working appropriately.

Requirements Engineering

The activity of requirements engineering focuses on identifying, colleting, organizing and representing requirements of a product member. The major difference between requirements engineering of an individual product and a product member is that stakeholders not only focus on the specific product but also on the scope of product line.

Technically, the requirements of product members are defined and scoped under the domain of the product line's requirements. A variability point of a requirement is bound with a variant for a particular product member during requirements engineering.

Design Analysis

Design analysis in application engineering must be consistent with the concept of design analysis in domain engineering. This activity is to analyse and design the architecture for a product member. Software product line architecture is refined and specialized for a particular product member. The software architecture of the product line is configured to fit for a product member based on the specific product's requirements. The configuration includes the addition and removal of designed variability points of the product line.

In (Bosch 2000), *architecture pruning* is an activity that the common aspects of software product line architecture is collected and the variable aspects for a specific product member are specified. The composition of common and variable aspects acquires the software architecture for a specific product member. Nonetheless, it is possible that a software product line architecture does not fulfill the complete design of a specific product. This needs an activity called *architecture extension* (Bosch 2000). The activity extends some aspects that are not included in the software product line architecture.

Integration and Testing

The usage of the core assets of product line and development of product members involve the following three steps: (i) discovering a set of reusable components for a specific product member; (ii) instantiating the variability points of the reusable components for a specific product member; and (iii) integrating and testing the reusable components for the product member.

2.3. Methodologies for the Development of Product Line Systems

In this section, we describe existing methodologies to support product line system development. In particular, object-oriented methodologies have been common and popular in the development of software systems. Many existing object-oriented methods are aimed at supporting the development of single software systems. Recently, some object-oriented methods have been extended and proposed for the development of product line systems. We describe below the methods and approaches for product line system development in the object-oriented paradigm.

COPA

Component-Oriented Platform Method (COPA) (America et al. 2000) is proposed for product families of software-intensive electronic products i.e. telecommunication, medical imaging, and consumer electronics. COPA defined *architectural* and *process frameworks*. The architectural framework consists of five views:

(i) *Customer view* – the view shows customer business models represented in customers language or textual language.

(ii) *Application view* – the view shows application models represented in UML diagrams

(iii) *Functional view* – the view shows functionalities and performances of systems represented in use cases

(iv) *Conceptual view* – the view presents *platform* and *product-specific* components created for product line and product member, respectively. In COPA, construction components are applied with some component-based techniques such as COTS, Microsoft's COM component model, Sun's JavaBeans, and OMG's CORBA.

(v) *Realization view* – the view illustrates specific techniques e.g. hardware infrastructure, hardware platform, operating systems. These are specified in a textual language.

The process framework consists of three main activities:

(i) *Product family engineering* – this activity is driven by policy and plans of an organisation. There are sub-activities during product family engineering such as domain modeling, requirements formulation, and commercial and technical design. These activities construct customer, application, and functional views. The architecture of product line is created during product family engineering. For example, COPA applied Koala for representing the product family architecture. According to Figure 2-1, this activity can be comparable with the domain analysis and domain design during domain engineering.

(ii) *Platform engineering* – this activity is concerned with technology and people management. Sub-activities can occur during platform engineering such as standard development, cooperating between stakeholders in product family engineering and product engineering to comprehend requirements of product line and product members, integrating and testing for product members, and maintenance of existing reusable components and platforms. This activity has sub-activities that are comparable with domain engineering including domain analysis, domain design, and domain implementation as shown in Figure 2-1.

(iii) *Product engineering* – this activity is concerned with the customer-oriented process. There are sub-activities during product engineering such as standard development, cooperating with customers to understand specific requirements, constructions of product members, and maintenance and support for product members. According to Figure 2-1, this activity can compare with application engineering including requirements engineering, design analysis, and integration and testing.

In the COPA method, the authors suggested the activities in software product line engineering and artefacts created during three activities. The artefacts are represented in UML diagrams, use cases, textual language, Koala language (Ommering et al. 2000), and component-based representation languages.

QADA

Quality-driven Development of Software Family Architectures QADA (QADA) is a quality-driven architecture-centric method for product line system development. The QADA method described the development of software product line architecture. The method includes five activities:

(i) *Requirements engineering* – this activity is aimed to capture and analyse requirements and context model. The requirements i.e. functional and non-functional requirements and context model i.e. hardware and software interfaces of a system, a set of constraints, rules, and standards are represented in textual language.

(ii) *Conceptual architecture design* – this activity is aimed to identify a conceptual architecture which is represented with three views namely, *structural view, behavior view*, and *deployment view*. The structural view is concerned with conceptual components and their relationships. The structural view is composed of three types of artefacts: (a) list of functional responsibilities represented in textual language; (b) table of non-functional requirements represented in text and table; and (c) decomposition model. The behavior view is concerned with dynamic actions and kinds of actions to which a system produces. The behavior is represented in a collaboration model. The deployment view is concerned with allocation of the conceptual components into hardware components. The behavior is composed of two types of artefacts: (a) table of deployment units represented in text and table; and (b) allocation model. Another type of artefact generated during conceptual architecture design is *design rationale* which represents design principles and rules.

(iii) *Conceptual architecture analysis* – this activity focuses on qualities, commonality, and variability of a system. Three types of artefacts are created namely: (a) *product line scope*, which represents a boundary of product line; (b) *taxonomy of requirements*, which describe syntactic architectural notations and are represented in domain models, relevant architectural views; architectural styles; environmental assumptions and constraints; and trade-off rationale; and (c) *knowledge base*, which allows the evaluation of collections of architectural styles and patterns in terms of both quality factors and concerns. The knowledge base in QADA contains materials, quality attributes, questions that describe the evaluation of artefacts.

(iv) *Concrete architecture design* – this activity focuses on providing a set of concrete software components and definition of interfaces between components. The activity is concerned with three views in the activity of concrete architecture design (structural view, behavior view, and deployment view). Firstly, the list of functional, non-functional requirements, and decomposition model from the conceptual architecture is designed and refined as *structural diagrams* that represent concrete components, interfaces and relationship. Secondly, the collaborative model from the conceptual architecture is defined and refined as *state diagrams* and *message sequence charts*. Thirdly, the table of deployment units and allocation model from the conceptual architecture are designed and refined as *deployment model*.

(v) *Concrete architecture analysis* – this activity is aimed to assess and evaluate the software product line architecture regarding expected changes. The analysis method consists of five sub-activities: (a) deriving of changes from the product line scope; (b) defining product-line architecture description; (c) defining scenario identification; (d) evaluating the effect of scenarios; and (e) identifying scenario interaction.

In the QADA method, the activities of domain engineering are defined. More specifically, the activity of requirements engineering is comparable with domain analysis in Figure 2-1, and the activities of conceptual architecture design, conceptual architecture analysis, concrete architecture design, and concrete architecture analysis are comparable with domain design in Figure 2-1. However, the QADA method does not cover an activity of application engineering in Figure 2-1. In addition, artefacts created during theses activities are represented by using textual language and UML diagrams.

KobrA

KobrA (Atkinson et al. 2000) is a component-based method for software product-line engineering that is developed by Fraunhofer IESE. In the KobrA method, the authors proposed a *Komponent* as a set of reusable components that satisfy a requirement or group of requirements. The Kobra method is divided in two main activities: (i) *framework engineering*, which defines a set of Komponents; and (ii) *application engineering*, which applies existing Komponents and constructs a product member.

Framework engineering consists of four activities, namely:

(i) *Context realization* – the aim of this activity is to define properties and scope of product line. The *business process models*, which describe the requirements and constrains of product line, and *decision models*, which describe common and variable requirements of product line, are created.

(ii) *Komponent specification* – the aim of the activity is to describe properties of a Komponent. The *structural model*, which is represented in UML class diagrams, *behavioural model*, which is represented in UML statechart diagrams, *functional model*, which is represented in Operation schemas, and *decision model*, which is represented in a textual language, are created.

(iii) *Komponent realisation* – the aim of the activity is to define the design of a Komponent. The *interaction model*, which is represented in UML collaboration diagrams, *structural model*, which is represented in UML class diagrams, *activity model*, which is represented in UML activity diagrams, and *decision model*, which is represented in a textual language, are created.

(iv) *Component reuse* – this activity focuses on applying existing components to develop new Komponent.

Application engineering consists of two activities:

(i) *Context realization instantiation* – the activity is aimed to identify relevant Komponents to be reused for a product member.

(ii) *Framework instantiation* – the activity is used to create a framework of a set of Komponents and relationships between those Komponents for a product member.

The Kobra method is defined to complete the activities in the development of product line systems. More specifically, the activities of context realization, Komponent specification, Komponent realization, and component reuse are comparable with the activities of domain analysis, domain design, and domain implementation as shown in Figure 2-1, respectively. Moreover, the activities of context realization instantiation, and framework instantiation cover the activities of application engineering including requirements engineering, design analysis, and integration and testing in Figure 2-1. Additionally, the method is systematic, scalable and practical for the development of product line systems. The artefacts created in the method are based on UML diagrams and textual language that are customised to fulfil the activities in the domain of product line systems.

PuLSE

Product Line Software Engineering (PuLSE) (Bayer et al. 1999) is a customizable software product line engineering approach. The PuLSE method consists of four main activities:

(i) *Initialisation* – the activity is aimed to analyse and evaluate a situation of an organisation.

(ii) *Infrastructure construction* – the aim of this activity is to define a scope and processes of product line. A scope model and definitions of product line are created.

(iii) *Infrastructure usage* – the aim of activity is to define and create product members.

(iv) *Evolution and management* – the aim of activity is to evolve the product line.

The PuLSE method consists of six technical components and three support components. The technical components are: (i) PuLSE-BC, which is used to support the analysis and evaluation of an organisation in the initialisation activity; (ii) PuLSE-Eco, which is used to support an economic analysis of product line; (iii) PuLSE-CDA, which is used to support a domain analysis of product line; (iv) PuLSE-DSSA, which is used to support a domain design of product line; (v) PuLSE-I, which is used to support the development of product member; and (vi) PuLSE-EM, which is used to support the evolution and management of product line.

The support components are: (i) *project entry points*, which are used to support analysis of an organisation' situation; (ii) *maturity scale*, which are used to support evaluation the adoption of product line; and (iii) *organization issues*, which are used to support maintenance of product line.

PuLSE defined the framework of components conducted by different activities. The activity of initialization is comparable with domain analysis in Figure 2-1. The activity of infrastructure construction has sub-activities in common with domain analysis, domain design, and domain implementation. Moreover, the activity of infrastructure usage is comparable with application engineering including requirements engineering, design analysis, and integration and testing as shown in Figure 2-1. In addition, software product line architecture and other artefacts in product line are represented as a set of prescribed components.

FAST

Family-oriented Abstraction, Specification and Translation (FAST) (Weiss 1995) is a software product line method that initially described two main activities in software product line engineering. The activities, which resemble the main activities depicted in Figure 2-1, are:

(i) *Domain engineering*, which defines product line and the core assets of the product line; and

(ii) *Application engineering*, which develops product members by using the core assets of the product line.

FAST describes a domain specific language AML (Application Modeling Language) for specifying the requirements of product line. The requirements of product line represented in the language are then specialized for product members. However, the definition and specification of requirements are restricted.

RSEB

Reuse-Driven Software Engineering Business (RSEB) (Jacobson et al. 1997) is proposed to focus on achievement of business goals and improvement of business performance. In (Jacobson et al. 1997), they proposed to apply use cases to describe reference requirements of product line and UML diagrams to describe the software product line architecture. They also defined activities in the development of product line systems:

(i) *Requirements engineering*, where variability is specified as use cases;

(ii) *Architectural family engineering*, where the software product line architecture is created in UML diagrams;

(iii) *Component system engineering*, where reusable components are developed; and

(iv) *Application system engineering*, in which product members are developed.

The activities defined in RSEB are comparable with ones shown in Figure 2-1. More specifically, the activity of requirements engineering in RSEB is concerned about domain analysis and requirement engineering defined in (Clements and Northrop 2004). The activities of architectural family engineering, and component system engineering have likewise sub-activities in domain design and domain implementation, respectively. Moreover, the activity of application engineering in RSEB covers the activities of design analysis, and integration and testing as shown in Figure 2-1.

SPLIT

Software Product-Line Integrated Technology (SPLIT) (Coriat et al. 2000) is a systemic approach for the development of product line systems. SPLIT suggested a life-cycle of the development process which consists of two activities. The activities, which resemble the main activities depicted in Figure 2-1, are:

(i) *Domain engineering*, which reference requirements, software product line architecture, and reusable components are created; and

(ii) *Application engineering*, which product members are developed.

There are four approaches applied in SPLIT:

(i) The approach called *SPLIT/Cloud* is applied to develop the reference requirements of product line systems. In this activity there are artefacts created: business process, capability, functional area, force, functional requirement, and non-functional requirement. In SPLIT, they described two situations of requirements engineering: the first one is the development based on existing products; and the second one is the development from scratch. The product line system development based on existing products consists of activities: (i) define reference requirements i.e. functional and non-functional; (ii) identify and organize the requirements of each product member; (iii) define artefacts that represent high-level views of functional requirements of each product member; (iv) define artefacts that represent high-level views of non-functional requirements of each product member; (v) map high-level views of functional and non-functional requirements to the reference requirements

The product line system development from scratch consists of activities: (i) define the domain of product line; (ii) scope the domain; (iii) identify the requirements of the product line; (iv) determine COTS used in the product line domain by applying with COTS model; (v) define reference requirements i.e. functional and non-functional; (vi) define a business process; (vii) define capabilities related to each business process; and (viii) define forces related to each non-functional aspect.

(ii) The approach called *Daisy* is applied for developing software product line architecture. In Daisy, a software system product line architecture (SSPLA) description is based on three architectural views: (a) business view; (b) subsystem view; and (c) technology view. The business view represents subject area and analysis pattern. The subsystem view represents subsystem, architectural pattern, process, architectural guidelines, architectural constraints and information. The technology view represents component model, computing infrastructure and deployment. The views are represented in UML diagrams.

(iii) The approach *Ladder* is applied for developing reusable components. In Ladder, they suggested the transformations, composition, splitting up, abstraction, refinement, development branch for reusable components development as well as COTS adaptation.

(iv) The approach *Wheels* is applied for supporting sub-processes during domain engineering and application engineering in SPLIT.

The SPLIT method is applied in ESAPS (ESAPS) and CAFÉ (CAFE 2003) projects. The method itself is composed of other methods to support each activity in software product line engineering. Otherwise, artefacts produced by using these methods are represented in i.e. UML diagrams, use cases, component-based representation languages.

Additionally, the concept of feature-orientation is not completely new in software engineering and there have been efforts to apply the concept of features to express aspects of a software system. Examples of feature-oriented methods are FODA (Kang et al. 1990), FORM (Kang et al. 1998), and FeatuRSEB (Griss et al. 1998), which are increasingly important to software product line engineering due to several reasons:

(i) The fact is when developing the product line, stakeholders communicate with each other in terms of product features. It becomes an effective media of communication between customers and system developers.
(ii) Due to a large size and diversion of requirements for product line systems, specifying and representing the requirements becomes primary tasks in domain analysis as these activities are supported by the feature-oriented methodologies.
(iii) Features can be used as the basis for analyzing and representing commonalities and variabilities of product members under the same product line. Additionally, the feature-oriented methodologies offer a way to classify various requirements.

FODA

Feature-Oriented Domain Analysis (FODA) (Kang et al. 1990) is proposed to support the activity of domain analysis. In FODA, the activities are described and cover the activity of domain analysis depicted in Figure 2-1. Three activities are:
(i) *Domain analysis*, which focuses on scoping of product line and identifying product members;
(ii) *Feature analysis*, which develops a list of common and variable aspects of product line; and
(iii) *Feature modelling*, which models the common and variable aspects as *a feature model.*

FODA is an initial method that defines a feature model for representing common and variable aspects of product line. Identification of features requires domain knowledge obtained from the domain experts and other sources such as books, user manuals, design documents, etc. In FODA, the authors described that domain experts and system analysts can use standard terminologies to communicate with each other in mature and stable domains. Therefore, analyzing the domain terminology is an effective and efficient way to identify the features of a given domain. However, in prior to feature identification, standard terminologies and domain scope should be done since they are not available in immature or emergent domains. Feature models are used as a mechanism to facilitate different perceptions of domain concepts and scope which cause confusion between stakeholders.

The authors defined three types of features: (i) *mandatory* features, which represent the commonality of product line; (ii) *alternative* features, which are specialized for product members; and (iii) *optional* features, which may or may not exist in product members. The feature model consists of elements such as: (a) a *tree-structured diagram* which represents characteristics of product line; (b) a *definition* for each feature; and (c) *composition rules* which are defined rationally between features. There are two types of rules: (i) one feature *requires* another feature: and (ii) one feature is *included* in another feature.

FORM

Feature-Oriented Reuse Method (FORM) is an extension of FODA that provides the activities of domain analysis and the development of core assets. Three activities are concerned:

(i) *Feature modelling* – that is a process for defining features of product line systems. The authors proposed to apply the extension of the feature model from FODA for representing features. They proposed the classification of with respect features to their purpose as: (a) a set of *capability* features that express the characteristics of distinct services, operations, functions, or performances, (b) a set of *operating environment* features that represent attributes of the environment in which an application is used and operated, (c) a set of *domain technology* features that represent the domain of realization (e.g., navigation methods in the aviation domain), and (d) a set *of implementation technique* features that represent implementation details at lower and more technical levels e.g. abstract data types and sorting algorithms. Kang et al. pointed out that a domain technology feature is more specific to a given domain and may not be usable in other domains while an implementation technique feature is more generic and may be used in other domains.

(ii) *Architecture modelling* – that is a process for defining software product line architectures. Artefacts created during this process are viewed a hierarchy and consists *subsystem model, process model* and *module model*. These models are represented the commonality and variability of the product line.

(iii) *Component engineering* – that is a process for defining reusable components. In (Lee et al. 2000), the authors described the technique used in the activity of component engineering in the FORM method. The authors described principles for the creation of reusable components by mapping features created during the activity of feature modeling. The principles are (a) capability features can be modeled as an object or group of objects that provide a similar set of operations. The object or group of objects is specified with a parameter for a particular product member; (b) operating environment features can be modeled as an object or group of objects that provide a set of operations for different requirements of product members; (c) domain technology features are modeled to be specific for the domain of product line; and (d) implementation technique features should be used to implement domain-specific objects. For example, a communication method feature (e.g. synchronized or asynchronized communication) depends on the implementation languages or platforms. However, the mapping of the feature model and product member is not described.

In the FORM method, the activities of domain engineering are defined. More specifically, the activities of feature modeling, architecture modeling, and component engineering have likewise sub-activities in domain analysis, domain design, and domain implementation defined in (Clements and Northrop 2004). However, the FORM method does not cover an activity of application engineering.

FeatuRSEB

Featuring RSEB (FeatuRSEB) (Griss et al. 1998) is a combination of RSEB method (Campbell et al. 1990) and FODA (Kang et al. 1990). The FeatuRSEB method includes the activities defined in RSEB which are requirements engineering, architectural family engineering, component system engineering, and application engineering. The method adapted using a feature model by adding UML-based relationships i.e. dependency and refinement. The feature model is used to represent common and variable RSEB models. In other words, the feature model is used to represent an association between RSEB models in product line.

2.4. Summary

This chapter has provided background information for product line systems. It has presented the terminologies, existing problems, current approaches and current techniques in the domain of product line. In the next chapter, we present a model for software product line specification.

Chapter III Software Artefact Specification for Product Line Systems

This chapter describes an approach to software artefact specification for supporting product line systems. The approach includes the types of documents represented software artefacts created during the phase of domain analysis and domain design (according to Figure 2-1) are defined in Section 3.2 and Section 3.3 respectively. The summary of the approach is also given in Section 3.4. Section 3.5 summarises of the chapter.

3.1. Introduction

According to the literature and survey of techniques which are applied by organizations in Thailand, we present the approach to software artefact specification for product line systems that is suitable to software development. Our work concentrates on documents generated during the phases of domain analysis and domain design.

Particularly, the approach includes two main essentials: (i). the types of documents represented software artefacts created during the phase of domain analysis; and (ii) the types of documents represented software artefacts created during the phase of domain design.

Additionally, we believe that a feature-based object-oriented engineering approach is required when developing product line systems. A feature-based approach is important to support domain analysis and domain design, enhance communication between customers and developers in terms of product features, and assist with the development of software product line architecture. On the other hand, an object-oriented approach is necessary to assist with the development of the various product members. As the following section, we elaborate the idea of applying featured-based objected-oriented engineering approach. We describe each type of software artefacts.

3.2. Requirements Artefacts

The reference requirements created during the domain analysis phase is represented by feature model (Kang et al. 1998) and use case (Cockburn 1997). In the following, we described the details.

3.2.1. Use Case

Use case is a textual specification language that captures a contract between the stakeholders of a system about its behavior (Cockburn 1997). Examples of the approaches proposed to apply use cases in the activities of product line system development are (America et al. 2000, Griss et al. 1998, Jacobson et al. 1997). In our work, we represent the functional requirements of product line as use-cases by adapted the template proposed in (Fantechi et al. 2004). The authors proposed to express the requirements of product line systems by extending the use case definition given by Cockburn (Cockburn 2000). In particular, the variability is expressed in use cases by using special tags. The tags indicate the variable requirements of product line that need to be specialized for a product member. They proposed three types of tags:

(i) *alternative tag,* which represents variable requirements with a predefined set of requirement variants;

(ii) *parametric tag*, which represents variable requirements that requires the instantiation of specific parameters for a product member, and

(iii) *optional tag*, which represents variable requirements which may or may not be instantiated for a product member.

In our template, a use case is composed of:

(1) **Use_Case** – the element consists of three attributes, which are information of the use case:

(a) *Use_Case_ID* – this attribute is identified as a use case;

(b) *System* – this attribute specifies which domain of product line is; and

(c) *Product_Member* – this attribute specifies for which product member the use case is specified.

(2) **Existential** – this element is used to represent the existential of a use case. It consists of an attribute *Commonality_Variability* – this attribute can be (i) mandatory, which indicates a use case must be satisfied by product members; (ii) *alternative*, which indicates a use case must be satisfied and altered for particular product members; and (iii) *optional*, which indicates a use case may or may not be satisfied by a product member.

Moreover, in the case that the attribute *Commonality_Variability* is specified as "alternative", the element **Existential** can consist of sub-element **Variant_Point**. The element *Variant_Point* specifies a particular point of the use case's variability. The element *Variant_Point* can consist of a sub-element either **Variant** or **Parameter**. The element *Variant* specifies a set of alternatives for the particular variant point, as the element *Parameter* specifies the domain of the *Variant_Point*. Note that in the case that the attribute *Commonality_Variability* is either mandatory or optional, the element *Variant_Point* may not exist.

(3) **Title** – the element *Title* is the title of use case.

(4) **Description** – the element *Description* is specified for a brief textual description.

(5) **Level** – the element describes the level of functionality that it describes within a system.

(6) **Preconditions** – the element describes the conditions that must be satisfied before its execution.

(7) **Postconditions** – the elements describes the conditions that must be satisfied after its execution.

(8) **Primary_actors** – the element specifies primary users of the use case.

(9) **Secondary_actors** – the element specifies secondary users of the use case.

(10) **Flow_of_events** – the element specifies a list of the events that trigger the use case and the specification of the normal events that occur within it. The element

Flow_of_events consists of the sub-element **Event**, which specifies a particular event being preceded in the use case.

(11) **Exceptional events** – the element describes the events that do not always occur when the use case is executed.

(12) **Superordinate use case** – the element specifies a use case for which the use case is elaborated.

(13) **Subordinate use cases** – the element specifies a use case to which the use case is specified.

Figure 3-1 illustrates an example of a use case *Sending a Message* from a mobile phone for product member PM1 of the mobile phone case study. The use case is identified with *UseCaseID* ("UC1"), *System* ("MobilePhone"), and *Product_Member* ("PM1"). The use case contains *Existential* element which its attribute *Commonality_Variability* is specified as "Alternative". The sub-element *Varaint_Point* ("v1") is declared along with the sub-element *Variant* which includes a set of possible values for *v1*. It also contains elements i.e. *Title, Description, Level, Preconditions, Postconditions, Primary_actor, Secondary_actors, Flow_of_events, Exceptional_events, Superordinate_use_case,* and *Subordinate_use_case* that describe the context of the use case

Use_Case *UseCaseID*="**UC1**"
System="MobilePhone"
Product_Member="**PM1**"
Existential *Commonality_Variability* = "Alternative"
Variant_Point v1
Variants v1 {keying-in a phone number of a receiver, selecting a phone number from a list of contacts}
Title *Sending a Message*
Description The phone is able to send a text message. The user can specify an address of a receiver by selecting from a list of contacts.
Level User Goal
Preconditions The user has already selected function of sending a text message from the main menu.
Postconditions The phone has sent the message.
Primary_actor The user
Secondary_actors -
Flow_of_events
Event 1 The system shows an editor for writing a message.
Event 2 The user inputs a phone number by [v1].
Event 3 The system displays the phone number to which the message is being sent.
Event 4 The user enters the message and confirms sending the message.
Event 5 The system sends the message and displays an acknowledge on the screen.
Exceptional_events -
Superordinate_use_case –
Subordinate_use_case –

Figure 3- 1: Use case *Sending a Message*

3.2.2. Feature Model

A *feature model* is a software artefact that describes the abstraction of domain knowledge obtained from domain experts such as system users, analysts, and system developers, as well as other sources such as books, user manuals, design documents, and source programs. This technique was initially proposed in FODA to assist the activity of domain analysis. Many approaches apply and extend the definition of a feature model to support the development of product line systems. As the following, we summarise different aspects of the feature modeling technique being applied in those approaches.

Types of Features in a Feature Model can be (i) *mandatory features* (Bosch 1998, Clements and Northrop 2002, Griss et al. 1998, Kang et al. 1990, Kang et al. 1998, PuLSE, Weiss 1995) are compulsory for product members in a family; (ii) *optional features* (Bosch 1998, Clements and Northrop 2002, Griss et al. 1998, Kang et al. 1990, Kang et al. 1998, PuLSE, Svahnberg et al. 2001, Weiss 1995) may exist in a specific product member or not; and (iii) *alternative features* (Bosch 1998, Clements and Northrop 2002, Kang et al. 1990, Kang et al. 1998, PuLSE, Weiss 1995) or *variant features* (Griss et al. 1998), are a set of possible features that can be selected for a specific product member. Moreover, (Svahnberg et al. 2001) define a feature type *external features* that is a feature unavailable in a system but needs to be satisfied by an external system.

Notations of Features in a Feature Model can be different. As shown in Figure 3-2, a feature may be depicted as a round or a rectangle with its name inside. Many approaches applied the feature notation defined in (Kang et al. 1990). However, some approaches applied a UML class diagram for expressing features, for example (Griss et al. 1998). Moreover, different types of a feature i.e. mandatory, optional, and alternative are represented in different notations.

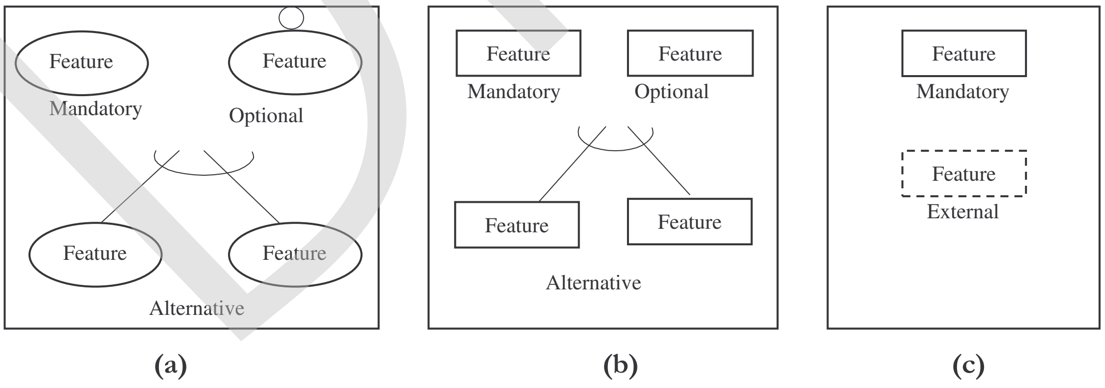


Figure 3-2: different notations for different types of a feature: (a) (Kang et al. 1990); (b) (Griss et al. 1998, Kang et al. 1998); and (c) (Svahnberg et al. 2001)

Ideally, features are atomic units that can be put together in a product without difficulty. However, features are generally not independent and several types of relations can exist between them. According to (Gibson et al. 1997), feature interaction is defined as a characteristic of a system whose complete behavior does not satisfy the separate specifications of all its features. The types of relationships express the rules of feature interaction. These relationships are considered when features are selected for product members. They represent which features must be selected together and which features must not. Table 3-1 shows different types of relationships between features.

Table 3- 1 presents the classification of relationships between features:

Relationship type	Description
depends-on (Griss et al. 1998)	Indicating that a feature relies on an existence of another feature
mutually exclusive (Griss et al. 1998)	Indicating that two features must exist at the same time
conflicting (Griss et al. 1998)	Illustrating that related features have conflicting requirements.
composed-of (Kang et al. 1998), *composition* (Svahnberg et al. 2001)	Indicating that a feature is composed of other features
generalization/specialization (Kang et al. 1998), *OR specialization* (Svahnberg et al. 2001)	Indicating that a child feature is specialized from a parent feature
implemented-by (Kang et al. 1998)	Indicating that a feature is implemented by another feature
XOR specialization (Svahnberg et al. 2001)	Indicating that children features are mutually exclusive

Our approach, we extend the feature model proposed in FORM (Kang et al. 1998) which is based on the feature model proposed by (Kang et al. 1990). More specifically, the authors enhanced the feature model with a textual specification for each feature. Our feature model describes the requirements artefacts of a product line system and illustrates the features available in the line. Figure 3-3 presents an example of a graphical hierarchy of feature for a mobile phone product line, while Figure 3-4 presents an example of a textual specification for feature *Text Messages*.

As shown in Figure 3-3, a feature is represented by a name and can be (i) *mandatory*, when it must exist in the applications in the domain; (ii) *optional*, when it is not necessary to be presented in the applications in the domain; or (iii) *alternative*, when it can be selected for an application from a set of features that are related to the same parent feature in the hierarchy.

The features can be classified into four groups namely (i) *application capabilities*, signifying features that represent functional aspects of the applications (e.g. calling, connectivity, personal preference, and tool features); (ii) *operating environments*, signifying features that represent attributes of the environment in which product members are used and operated (e.g. network, input and output methods, and operating system features); (iii) *domain technologies*, signifying features that represent specific implementation and technological aspects of the applications in the domain (e.g. WAP and XHTML[1] browser types; specific Java application support like mobile media and wireless messaging application programming interface; SMTP, POP3, and IMAP4[2] network protocol features); and (iv) *implementation techniques*, signifying features that represent more general implementation and technological aspects of the applications, but not necessary specific for the domain (e.g. PGP and DES encryption methods; AMR, MIDI, and MP3 sound formats; and 3GPP and MPEG[3] video format features).

Feature can also be related by different types of relationships. Examples of these relationships are (i) *composed_of*, (ii) *generalisation/specialization*, and (iii) *implemented_by* relationship types.

[1] WAP: Wireless Application Protocol; XHTML: Extensible HyperText Markup Language.

[2] SMTP: Simple Mail Transfer protocol; POP3: Post Office Protocol; IMAP4: Internet Message Access Protocol.

[3] AMR: Adaptive Multi-Rate; MIDI: Musical Instrument Digital Interface; MP3: MPEG Audio Layer III; 3GPP: 3rd Generation Partnership Project; and MPEG: Moving Picture Experts Group.

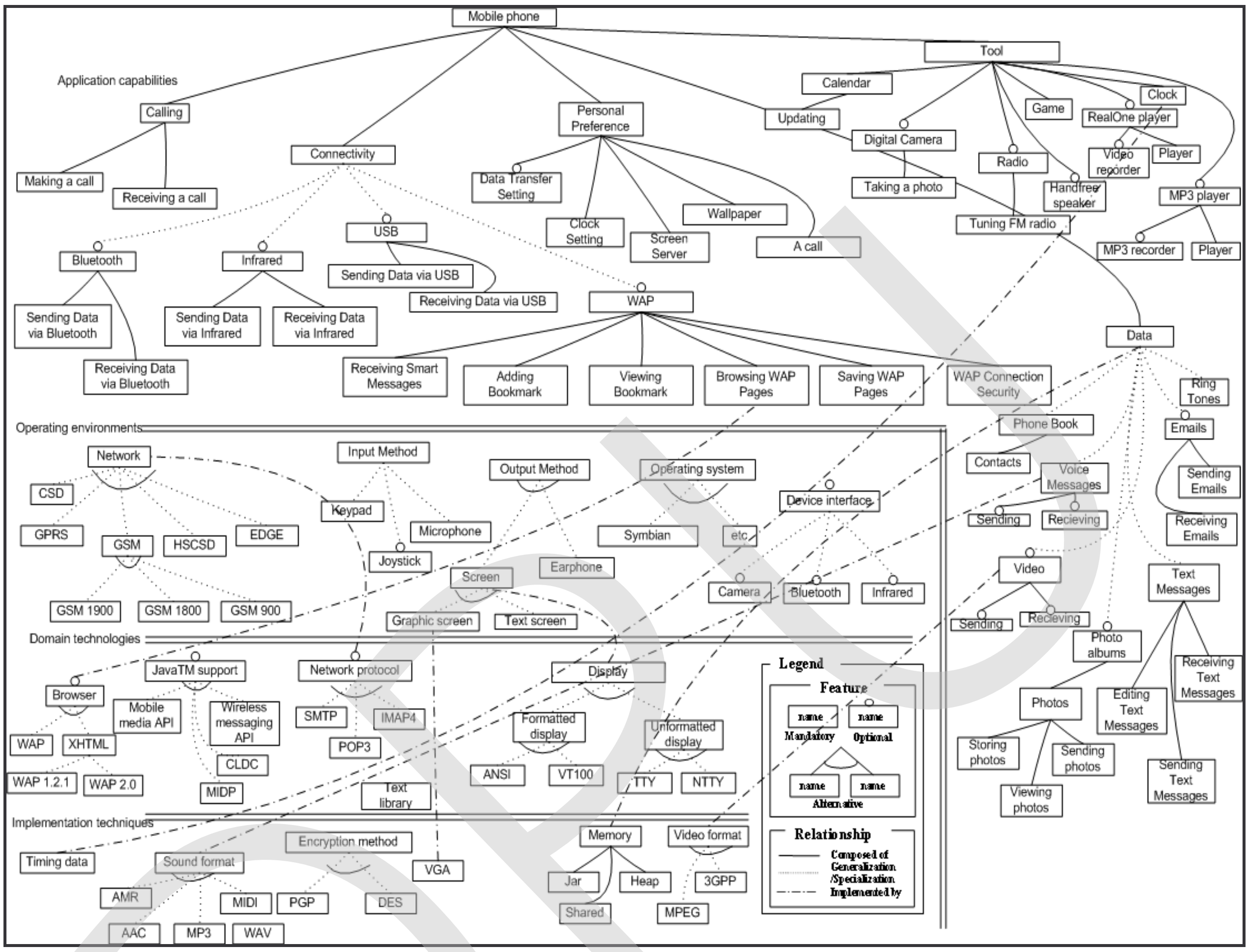


Figure 3- 3: The feature model of the mobile phone

As shown in Figure 3-4, the textual specification represents (i) a *name*, (ii) a *description*, (iii) *issues and decisions* representing trade-offs, rationale, or justifications for including the feature in an application, (iv) a *type* such as application capabilities, operating environments, domain technologies, and implementation technologies, (v) *commonality* indicating if a feature is mandatory, optional, and alternative, (vi) relationship with other features such as composed-of, implemented-by, generalisation/specialization, (vii) *composition rule* representing mutual dependency and mutual exclusion relationships to indicate consistency and completeness of a feature, if any, and (viii) *allocated-to-subsystem* indicating the name of a subsystem that contains the feature, if any

Feature-name:	Text Messages
Description:	The phone can edit, send, and receive a short text message
Issues and decision:	Text message over mobile phone is a way of communication
Type:	Application capability
Commonality:	Mandatory
Composed-of:	Sending Text Messages, Receiving Text Messages, Editing Text Messages
Composition-rule:	-
Allocated-to-subsystem:	Messaging

Figure 3- 4: Features in textual specification language.

As described earlier, the requirement artefacts are specified in use case and feature model. Since the use case represents the requirements of a product member, the feature model represents the requirements of a product line. Both types of artefacts are elaborated into other types of software artefacts in subsequent activities of product line engineering i.e. design and implementation. The following section, we present the software artefacts being applied with the process of software product line design.

3.3. Design Artefacts

According to the literature, some approaches such as (Clauss 2001, Gomaa 2004, Keepence and Mannion 1999) are proposed to adapt UML diagrams for modeling software product line systems. In our approach, we adopt UML class diagram, statechart diagram, and sequence diagram to present the software product line architecture. In the following, we described the details.

3.3.1. Class Diagram

In our approach, we extend the class diagram presented in (Clauss 2001) by adding some elements. The diagram consists of elements as described following:
(1) **Class Diagram** – the element consists of three attributes, which are information of the class diagram:

(a) *Class_Diagram_ID* – this attribute is identified as a class diagram;

(b) *System* – this attribute specifies which domain of product line is; and

(c) *Product_Member* – this attribute specifies for which product member the class diagram is specified.

(2) **Existential** – this element is used to represent the existential of a class diagram. It consists of an attribute *Commonality_Variability* – this attribute can be (i) *mandatory*, which indicates a class diagram must be satisfied by product members; (ii) *alternative*, which indicates a class diagram must be satisfied and altered for particular product members; and (iii) *optional*, which indicates a class diagram may or may not be satisfied by a product member.

(3) **Class** – the element **Class** specifies a system component that is composed of *attributes*, which describe properties of a particular class, and *methods*, which specify operations of the particular class. According to (Clauss 2001), a class can be one of three types for expressing variability in product line: (i) *variationPoint*, which represents a variation point of product line; (ii) *variant*, which represents an alternative of a particular variation point; and (iii) *optional*, which represents an optional class.

(4) **Relationship** – the element **Relationship** describes an association between classes. To capture and represent variability of a product member, classes can be associated by applying one of two relationship types: (i) *generalization/specialization*, which associates between classes typed of variationPoint and variant; and (ii) *association with cardinality 0...1*, which associates between any class and a class typed of optional.

As shown in Figure 3-5, we illustrate an extract of a class diagram for a product member. An example of representing variability is that a class *DisplayScreen* is typed of *variationPoint* as classes *GraphicColourScreen* and *TextScreen* are typed of *variant*. The relationship between the class *DisplayScreen* and classes *GraphicColourScreen* and *TextScreen* is *generalization/specialization*.

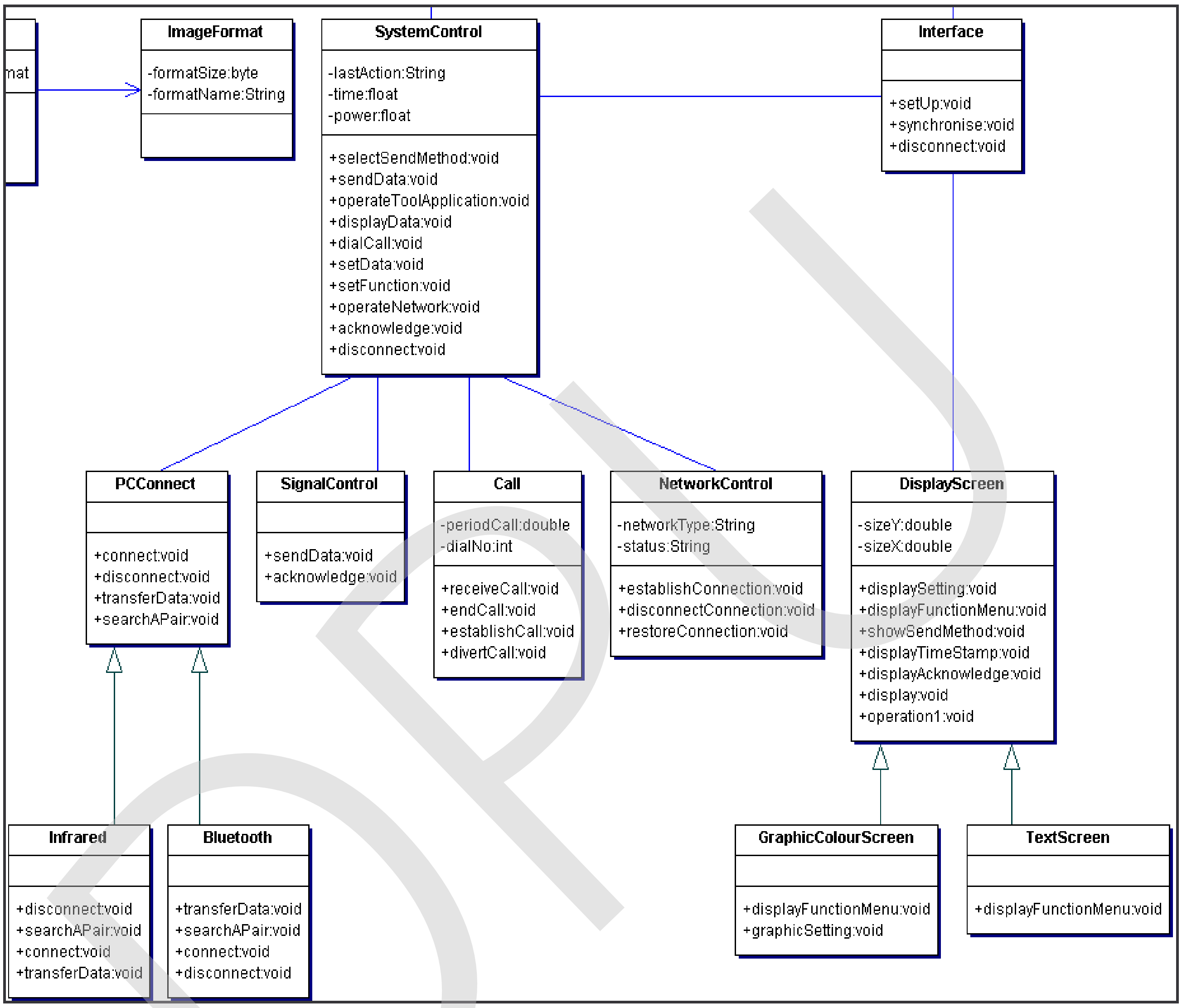


Figure 3- 5: An extract of a class diagram

3.3.2. State Chart Diagram

In our approach, we propose the extension of UML state chart diagram by adding some elements. The diagram consists of elements as described following:

(1) **State Chart Diagram** – the element consists of three attributes, which are information of the state chart diagram:

(a) *State_Chart_Diagram_ID* – this attribute is identified as a state chart diagram;

(b) *System* – this attribute specifies which domain of product line is; and

(c) *Product_Member* – this attribute specifies for which product member the state chart diagram is specified.

(2) **Existential** – this element is used to represent the existential of a state chart diagram. It consists of an attribute *Commonality_Variability* – this attribute can be (i) *mandatory*, which indicates a state chart diagram must be satisfied by product members; (ii) *alternative*, which indicates a state chart diagram must be satisfied and altered for particular product members; and (iii) *optional*, which indicates a state chart diagram may or may not be satisfied by a product member.

(3) **State** – the element **State** specifies the system's particular status. The states of the diagram can be representing some aspects of the variability. We define three types of a state for expressing variability in a product line: (i) *variationPoint*, which represents a state that initiates a variation point of product line; (ii) *variant*, which represents an alternative states of a particular variation point; and (iii) *optional*, which represents an optional state.

(4) **Transition** – the element **Transition** describes a driving method to transform a state to another state. To capture and represent variability of a product member, a transition can be specified as one of three transition types: (i) *variantTransition*, which describes one of possible driving methods to transform a state to another state; (ii) *parameterTransition*, which describes a transition requiring a parameter to drive the method; and (iii) *optionalTransition*, which describes a possible driving method to transform a state to another state.

As shown in Figure 3-6, we illustrate an extract of a state chart diagram. An example of representing variability is that states *Idle* and *SavingPhoto* are typed of *variant* state which can be transformed from the state *DisplayingArea* which is considered as *variationPoint*. The transitions from the state *DisplayingArea* to the state *Idle* and end state are *alternative transitions*.

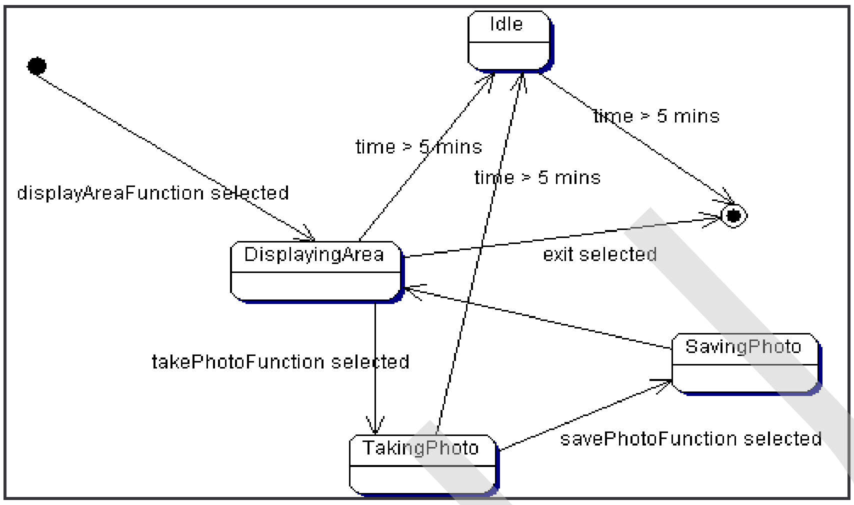


Figure 3- 6: An extract of a state chart diagram

3.3.3. Sequence Diagram

We also propose the extension of UML sequence diagram by adding some elements. The diagram consists of elements as described following:

(1) **Sequence Diagram** – the element consists of three attributes, which are information of the class diagram:

(a) *Sequence_Diagram_ID* – this attribute is identified as a sequence diagram;

(b) *System* – this attribute specifies which domain of product line is; and

(c) *Product_Member* – this attribute specifies for which product member the sequence diagram is specified.

(2) **Existential** – this element is used to represent the existential of a sequence diagram. It consists of an attribute *Commonality_Variability* – this attribute can be (i) *mandatory*, which indicates a sequence diagram must be satisfied by product members; (ii) *alternative*, which indicates a sequence diagram must be satisfied and altered for particular product members; and (iii) *optional*, which indicates a sequence diagram may or may not be satisfied by a product member.

(3) **Sequence** – the element **Sequence** specifies an interaction between an object and actor, or between objects. We propose three types of sequences for expressing variability in a sequence diagram: (i) *variationPoint*, which represents a sequence that initiate a variation point of later sequences; (ii) *variant*, which represents an alternative sequence of a particular variation point; and (iii) *optional*, which represents an optional sequence.

(4) **Message** – the element **Message** basically represent a called operation from an object interacting to another object. A message can be representing the variability of a product member. Specifically, we propose three types of messages for expressing the variability: (i) *variantMessage*, which is one of possible messages being sent from a *varaintPointSequence* to another sequence; (ii) *parameterMessage*, which is a message requiring a parameter to drive the method and (iii) *optionalMessage*, which is an optional message that may or may not be sent on a sequence.

As shown in Figure 3-7, we illustrate an extract of a sequence diagram. An example of representing variability is that the sequence *2.1.1* is specified as a variant point *v1* and then the sequences *2.1.1.1, 2.1.1.2,* and *2.1.1.1.1* are variants of the variant point *v1*.

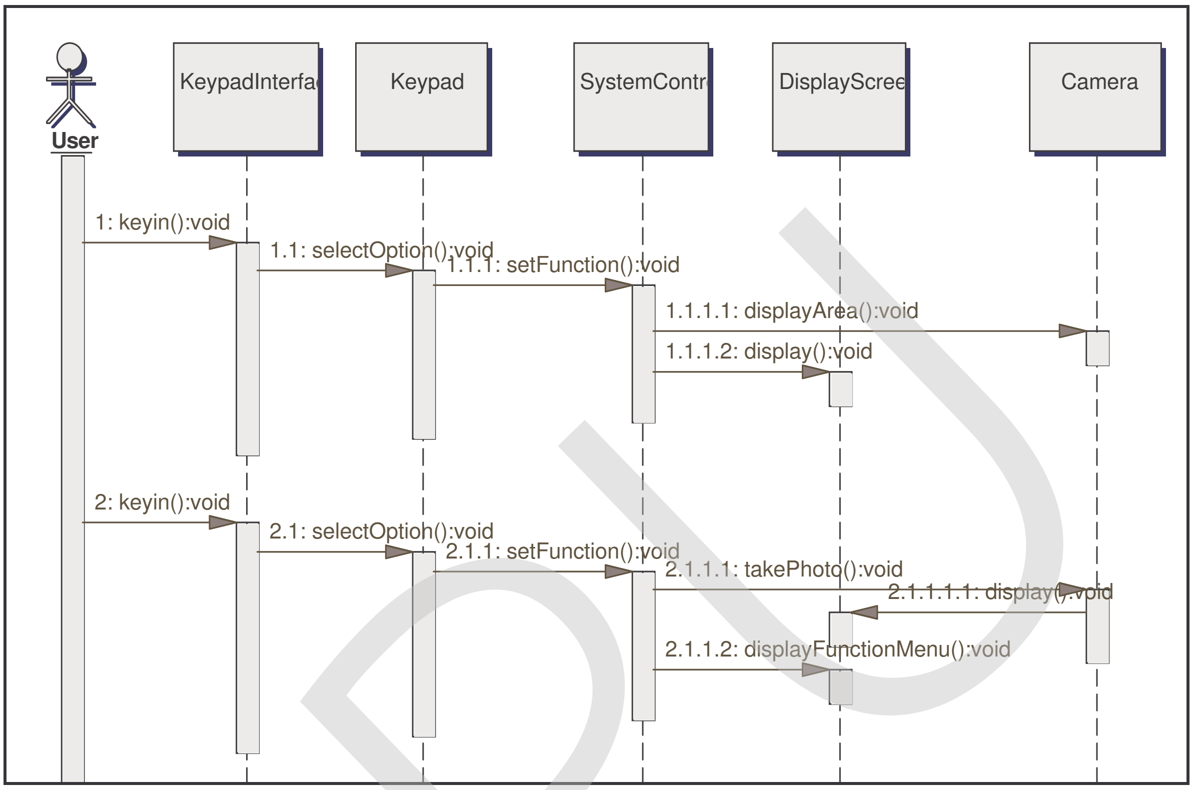

User
KeypadInterfa
Keypad
SystemContr
DisplayScree
Camera
1: keyin():void
1.1: selectOption():void
1.1.1: setFunction():void
1.1.1.1: displayArea():void
1.1.1.2: display():void
2: keyin():void
2.1: selectOption():void
2.1.1: setFunction():void
2.1.1.1: takePhoto():void
2.1.1.1.1: display():void
2.1.1.2: displayFunctionMenu():void

Figure 3- 7: An extract of a sequence diagram

3.4. Summary

This chapter described the meta model for specification of a product line system. It has presented the types of artefacts proposed in our approach for specifying the software artefacts created during product line analysis and design. It also described and gave examples of each types of software artefacts. In the next chapter, we describe a prototype tool which is implemented in order to assist the use of our model.

Chapter IV Prototype Tool

This chapter presents the prototype tool that we have implemented to demonstrate our work. It aims to illustrate how the tool can facilitate the activity by generating software artefacts, particularly use case and feature model. Section 4.1 describes the overview and functionalities of the tool. Section 4.2 presents the interfaces of the tool. Section 4.3 summarises the chapter.

4.1. Overview

In order to evaluate and demonstrate our approach, we have implemented a prototype tool. We envisage the use of our tool as a general platform for creating the software artefacts for product line's analysis and design. The tool has been implemented in Java.

Figure 4-1 illustrates the architecture of our tool. The tool is composed of four components, namely:

(a) *Interface* – This component provides the user interfaces for a user to specify the type of documents to be created.

(b) *Generator* – This component generates a document according to the specific type.

(c) *Display* – This component presents a document created by the tool.

(d) *Converter* – This component transforms a document to be in XML format.

According to Figure 4-1, the *Interface* component is responsible for communication with a user for specifying the type of document being created. The component *Generat*or is composed of two sub-components: *implemented* and *embedded*. Consider in Case (a), a user requests to create a document typed of use case or feature model, the *implemented generator* component is invoked. Consider in Case (b), a user requests to create a document typed of class diagram, state chart diagram, or sequence diagram, the *embedded generator* which is embedded with some existing tool is invoked to provide the creation of the document.

The *Display* component is used to display a document created by the tool. Additionally, the *Converter* component can be used to transform the document created by the tool into XML format.

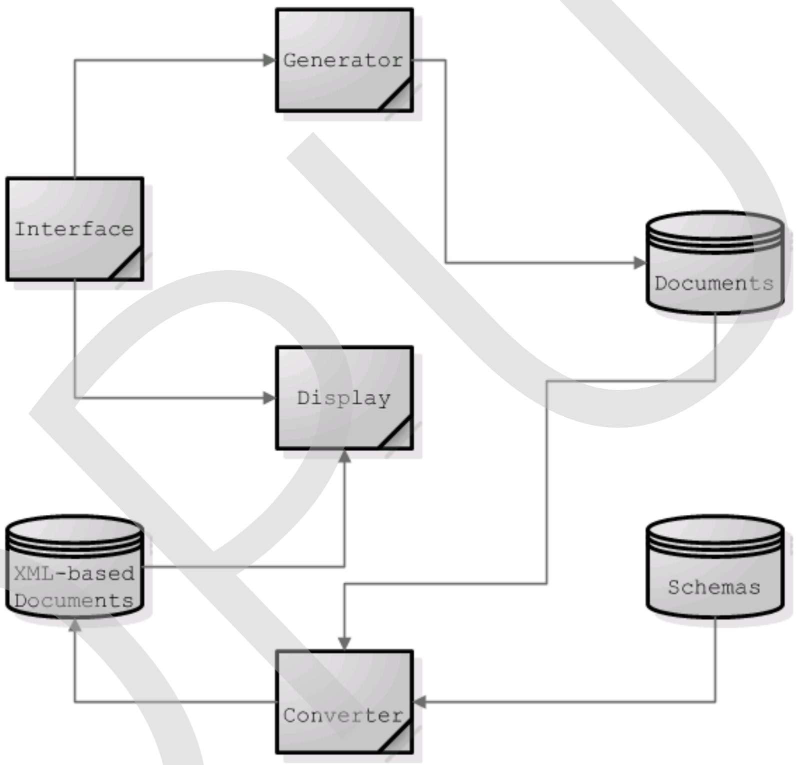


Figure 4- 1: The Architecture of our Tool

The various components of the prototype tools support various functionalities. These functionalities include:

(i) *Document Creation*, which is concerned with the generation of different typed documents based on the specification of a user;

(ii) *Document Visualisation,* which is concerned with the representation of the document generated in (i).

In the following, we explain these functionalities in more details.

I. Document Creation

In order to support this functionality, the tool provides sophisticated user interfaces in which a user can select a type of documents to be created. More specifically, this functionality allows the user to create a document according to the proposed model. For any of cases (a) or (b) above, the user can select to create a document for the whole product line or a particular product member. In the other word, the tool can generate a document in different levels of granularity, namely: (a) at the level of specific product members; and (b) at the level of product line. The generation of documents is executed by the components in the tool i.e. *implemented generator* and *embedded generator*. Particularly, we apply some existing UML tools i.e. Borland Together [ ] to create a class, sequence, and state chart diagrams for the embedded generator component.

In Section 4.2, we show user interfaces for this functionality and an example of using the interfaces in which the type of documents a user has selected to be created.

II. Document Visualisation

The generated documents are recorded and represented in XML documents. The XML-based documents represent the structure of a document including proposed elements as described in Chapter 3. This functionality provides a user to access a concrete element of each document in term of XML element.

4.2. Interface

This section illustrates the user interfaces of the prototype tool and describes how a user can execute the various activities supported by the tool. We illustrate the use of the tool by giving examples based on the mobile-phone systems.

4.2.1. Selecting Specific Type of a Document

As shown in Figure 4-2, this interface supports the functionality of *Document Generation*. It allows a user to specify the types of documents to be created and consists of four main parts:

(a) a panel that is composed of two sub-panels, namely *requirement*, and *design*. Each sub-panel contains different icons representing the various types of documents of each development phase. The *requirement* sub-panel contains *use case* and *feature model* icons representing documents produced during the analysis phase. The *design* sub-panel contains *class diagram*, *statechart diagram* and *sequence diagram* icons representing documents produced during the design phase. Table 4-1 shows all the icons representing the various documents.

(b) a panel that shows the selected type of document to be created.

(c) a panel with two buttons "OK" and "Cancel", which either presents the next interface or abort the command of document creation, respectively. In the case (a) a user selects to create either use case or feature model, the tool calls the next interface; or case (b) a user selects to create class, sequence, or state chart diagram, the tool invokes embedded tool, specifically, Borland Together.

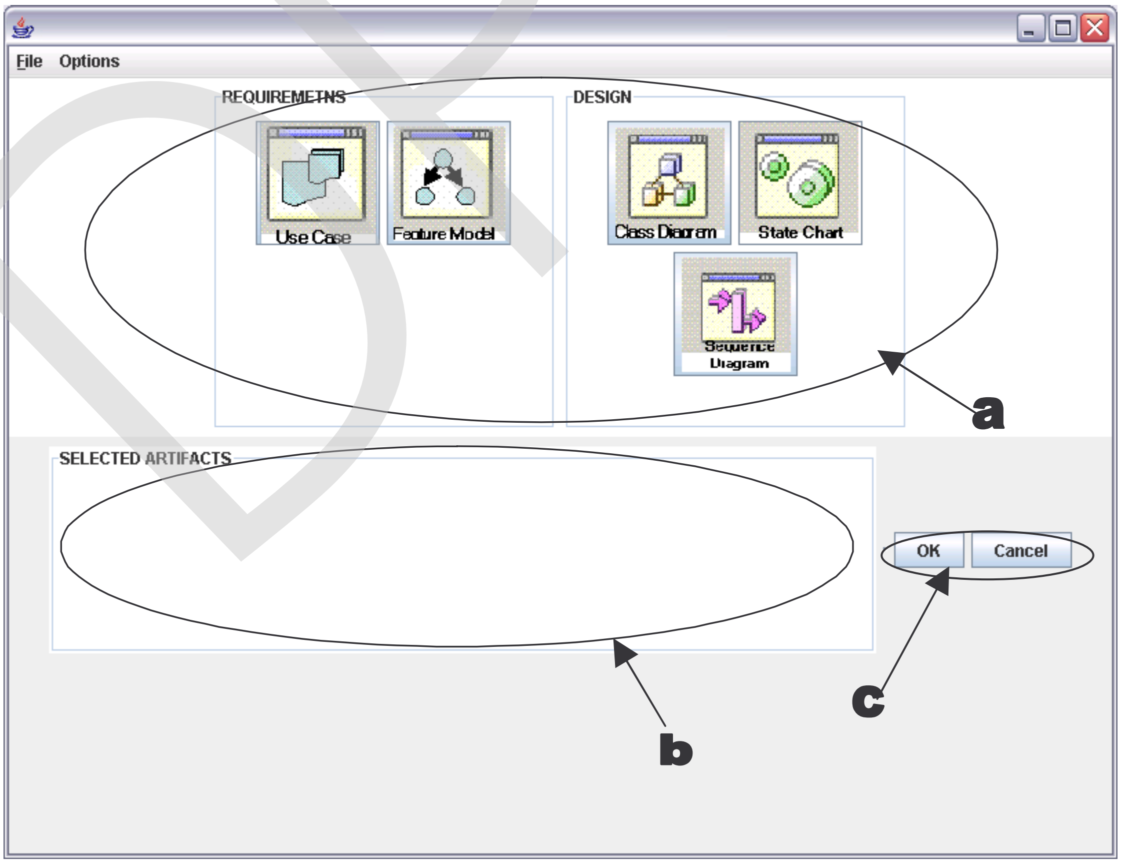


Figure 4- 2: An interface for specifying the type of software artefact to be created

Table 4- 1: Icons in panel (a)

Sub-panel	Icon	Software Artefact Type
Requirements	Use Case	Use case
Requirements	Feature Model	Feature model
Design	Class Diagram	Class diagram
Design	Sequence Diagram	Sequence diagram
Design	State Chart	Statechart diagram

Figure 4-3 illustrates how to work with the interface.

- Firstly, the user selects a type of document by clicking on its respective icon in panel *a*.
- Secondly, the system displays the selected type of document in panel *b*.
- Next, the user either selects the "OK" button to precede an action of document creation or the "Cancel" button to abort the action.

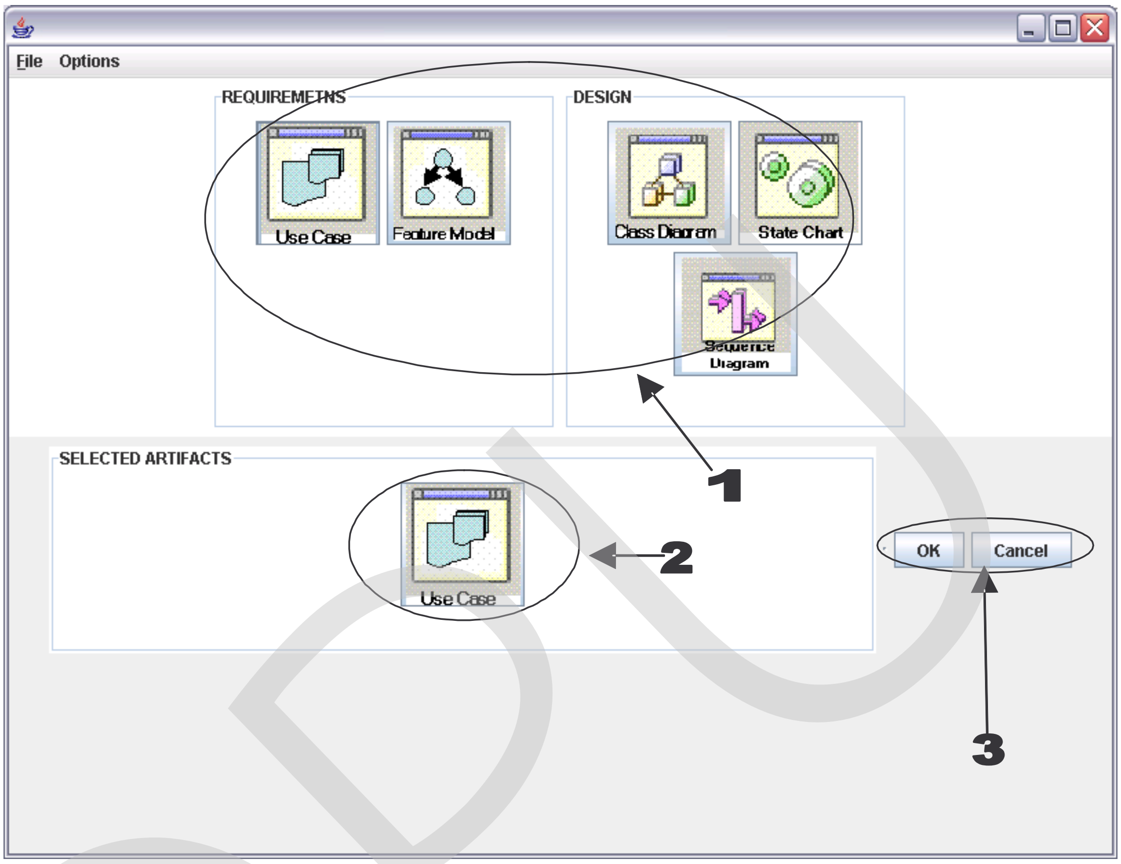


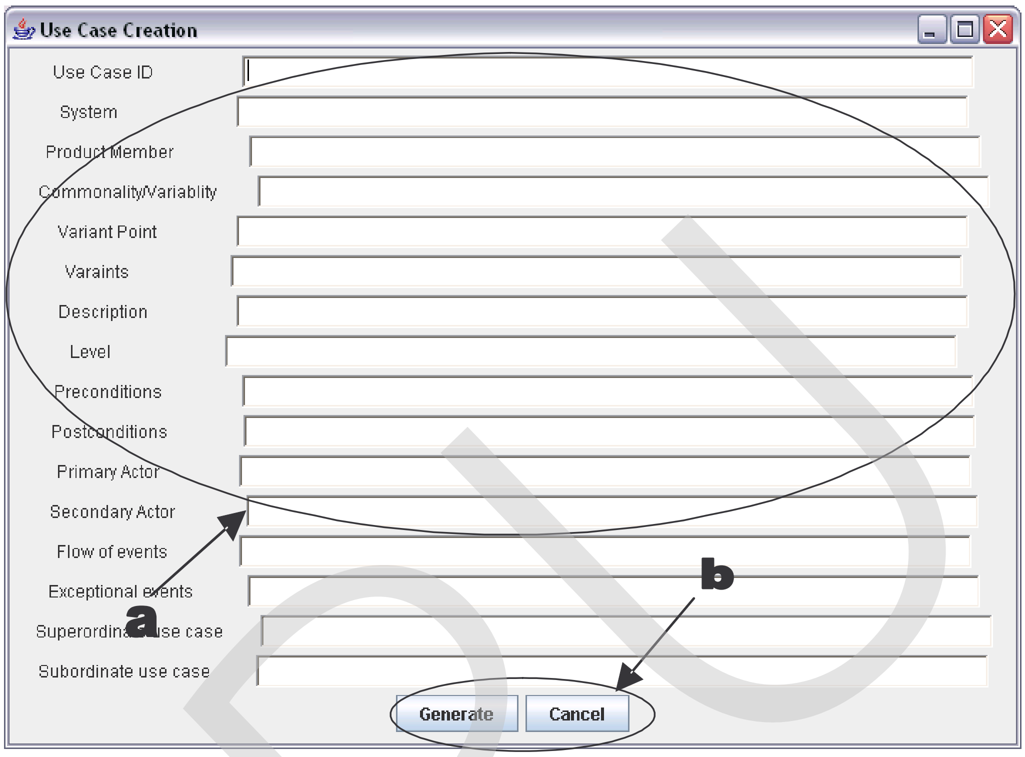
Figure 4- 3: Example interface demonstrating specifying of types of document artifacts

4.2.2. Specifying a Use Case

As shown in Figure 4-4, this interface also supports the functionality *Document Generation*. This interface consists of:

(a) a panel that shows a list of elements to be consisted in a document. The elements following from the type of document selected from the previous interface (Figure 4-3). The example in Figure 4-4 shows the elements for a use case.

(b) a panel with two buttons, namely "Generate", and "Cancel", which are related to actions for generating a use case, and aborting the creation, respectively.

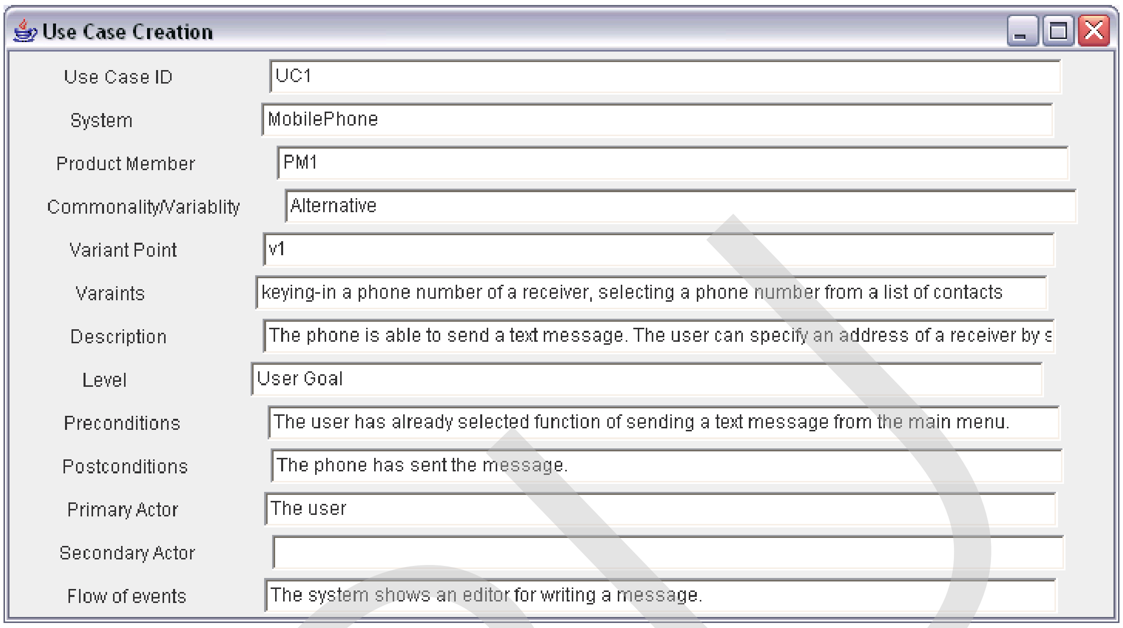


Use Case Creation

Use Case ID	
System	
Product Member	
Commonality/Variablity	
Variant Point	
Varaints	
Description	
Level	
Preconditions	
Postconditions	
Primary Actor	
Secondary Actor	
Flow of events	
Exceptional events	
Superordinate use case	
Subordinate use case	

Figure 4-4: An interface for creating a use case

The example shown in Figure 4-5 follows from the specification in the example in Figure 4-3. This shows the case in which the user has selected to create a use case.



Use Case ID	UC1
System	MobilePhone
Product Member	PM1
Commonality/Variablity	Alternative
Variant Point	v1
Varaints	keying-in a phone number of a receiver, selecting a phone number from a list of contacts
Description	The phone is able to send a text message. The user can specify an address of a receiver by s
Level	User Goal
Preconditions	The user has already selected function of sending a text message from the main menu.
Postconditions	The phone has sent the message.
Primary Actor	The user
Secondary Actor	
Flow of events	The system shows an editor for writing a message.

Figure 4- 5: Example of specifying a use case

4.2.3. Specifying a Feature Model

As shown in Figure 4-6, the interface also supports the functionality Document Creation. This interface consists of:

(a) a panel that shows a list of elements to be contained in each feature;

(b) a panel that shows the tree-structure of a feature model in which contains specified features;

(c) a panel with two buttons, namely, “Generate” and “Cancel” which are related to actions for generating a feature model, and aborting the creation, respectively.

The example is shown in Figure 4-7.

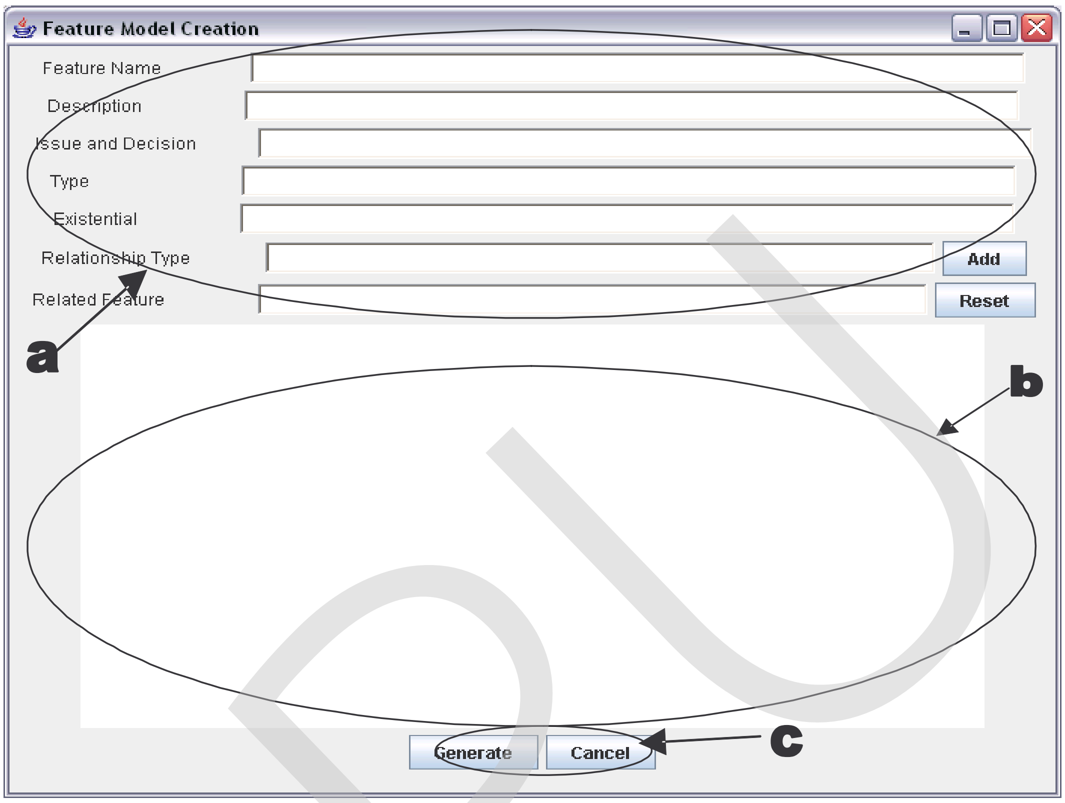
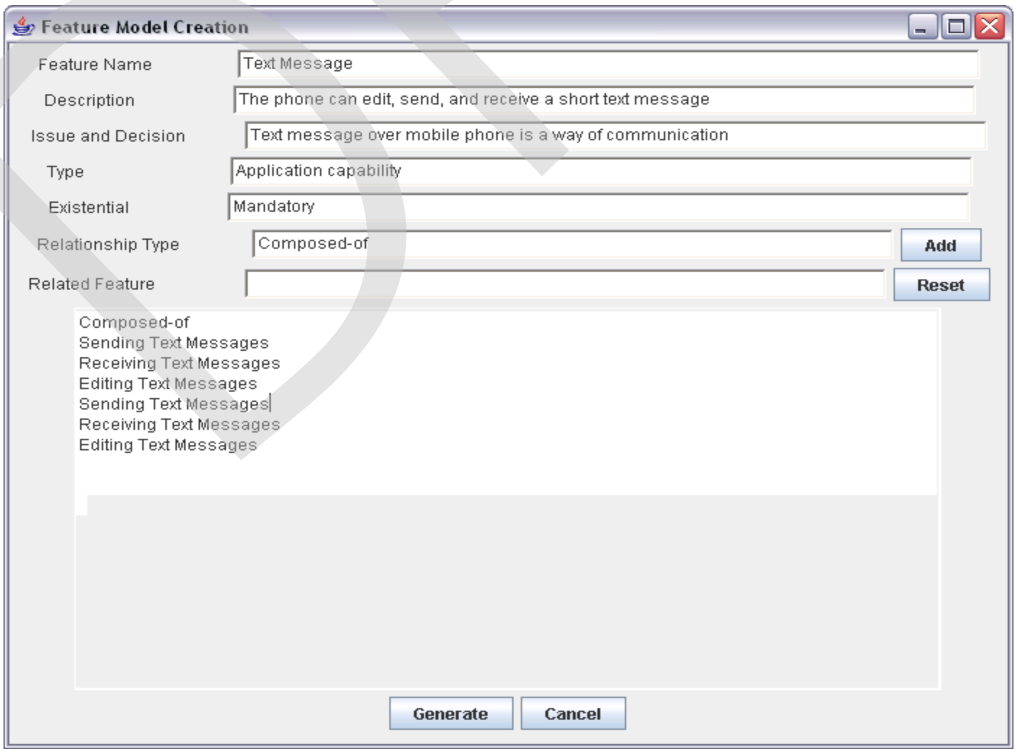


Figure 4- 6: An interface for specifying a feature model



Feature Name	Text Message
Description	The phone can edit, send, and receive a short text message
Issue and Decision	Text message over mobile phone is a way of communication
Type	Application capability
Existential	Mandatory
Relationship Type	Composed-of
Related Feature	

```
Composed-of
Sending Text Messages
Receiving Text Messages
Editing Text Messages
Sending Text Messages
Receiving Text Messages
Editing Text Messages
```

Figure 4- 7: Example of specifying a feature model

4.2.4. Converting a document into XML

To support the functionality *Document Visualisation*, the tool allows a user to transfer a document into the Extensible Markup Language (XML) format. Our approach supports the XML technology since there are several reasons:

(a) XML has become the de facto language to support data interchange among heterogeneous tools and applications; and

(b) the existence of large number of applications that use XML to represent information internally or as a standard export format (e.g. Unisys XML exporter for Rational Rose (IBM), Borland Together (Borland), ArgoUML (ArgoUML).

Figure 4-8 presents an interface to transform a document into XML. Initially, the documents of our concerned are translated into XML by using a *Converter* component. In the case of the class, sequence, and state chart diagrams, the XMI format is generated by using commercial XMI exporter (e.g. Unisys XMI exporter for Rational).

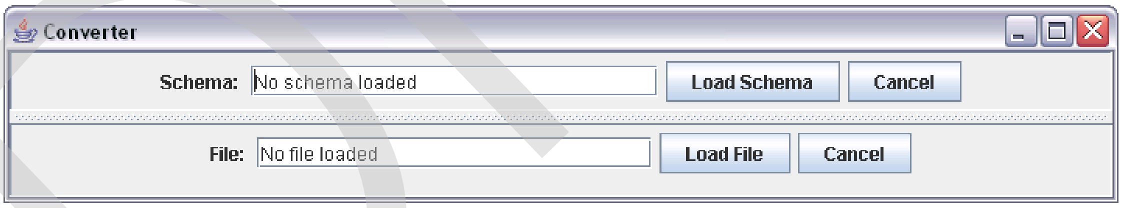


Figure 4- 8: An interface to transfer a document into XML

4.3. Summary

This chapter has presented the prototype tool including its functionalities and user interfaces. The chapter has illustrated the use of the tool to partly support the specification of software product line artefacts.

Chapter V Evaluation and Analysis

In this chapter, we evaluate and analyze our work. Section 5.1 describes an overview of our evaluation, the different scenarios used to evaluate our work, and an outline of how the evaluation was conducted. Section 5.2 presents the results of the evaluation and analyze these results. Section 5.3 summarises the chapter.

5.1. Evaluation Objectives and Methods

The objective of the evaluation is to:

evaluate whether the model helps an organization in making software product line artefacts more precise and consistent.

According to the above evaluation objective, this testing was inducted by concerned the following factors:

5.1.1. Selection of Participants

The testing scenarios used in our evaluation were based on two main factors. **The first factor** was concerned with the different ways in which organizations can develop product line systems. As proposed in (Krueger 2001), organisations can develop product line systems in three different ways:

(a) when an organisation decides to analyze, design, and implement a line of products prior to the creation of individual product members (*proactive approach*);
(b) when an organisation enlarges the product line systems in an incremental way based on the demand for new product members or new requirements for existing products (*reactive approach*); and

(c) when an organisation creates a product line based on existing product members by identifying and using common and variable aspects of these products (*extractive approach*).

According to the organizations, we found that these approaches are not mutually exclusive and can be used in combination. For instance, it is possible to have product line systems initially created in an extractive way to be incrementally enlarged over time by using a reactive approach.

Particularly, we randomly selected up to 50 organisations of different business areas such as software production, financial, trading, logistics, airlines, insurance and so on by analysis the infrastructure of the organizations. We applied CMM standard for justifying the organisations to be participated in our study. Specifically, the criteria which we applied to justify the organisations for our testing are: (i) maturity, (ii) size, and (iii) number of software products.

The second factor was concerned with the participants involved in the product line system development process. Participants are stakeholders who are involved in this process ranging from market researchers, to product managers, requirement engineers, product-line engineers, software analysts, and software developers. These stakeholders contribute in different ways to the product line system development process, have distinct perspectives of the system, and have distinct interests in different aspects of the product line systems. For example, a market researcher may be interested in the requirements and features of a new product member to be developed, while a software developer may be interested in the design and implementation aspects of this new product member. Therefore, the stakeholders would be interested in different types of documents that may assist them in their various tasks during system development. Note that these participants have experienced in the legacy system of their organizations but not necessarily have the knowledge of software product line systems.

5.1.2. Test Cases

In order to take into consideration the various ways of developing product line systems, the heterogeneity of stakeholders, and document types. The five scenarios used in our testing include:

(a) the creation of a new product member from existing product line;
(b) the creation of product line from already existing products in their organisations;
(c) changes to a product member in a product line;
(d) changes to the core assets of a product line; and
(e) impact of changes to the core assets of a product line to a product member.

For each of these scenarios we have identified the stakeholders involved in the process and the types of documents according to the meta model that are related to the scenarios. We asked our participants to perform some of above tasks twice: (i) by applying the prototype tool prepared by the author (see in Chapter 5); and (ii) by manually performing. Manual practice may subject to applying any existing software of the organizations. The results of each task are software artefacts which are developed according to the meta model. The types and number of software artefacts are various in each task. The author has prepared the software artefacts for some tasks as required. Each task is described below:

(a) Task 1: The creation of a new product member from existing product line

This situation occurs when an organisation wants to enlarge its system and creates a new product member. In this case, a set of requirements and design are being generated by considering the existing documents e.g. feature model, class diagrams. As shown in Figure 5-1, the stakeholders involved in this case are:

(a) market researchers (or persons who act as market researchers) that are responsible to identify the feasibility of producing a new product and the features that this new product should include from a commercial point-of-view;

(b) requirements engineers and product managers (or persons who act as requirements engineers and product managers) that specify the requirements of the new product;

(c) product line engineers, product managers, and software analysts (or persons who act as product line engineers, product managers, and software analysts) that identify which aspects in the core assets of a product line are related to the new product;

(d) software analysts and software developers (or persons who act as software analysts and software developers) that analyse existing product members and identify the commonality and differences between existing product members and the new product; and

(e) software developers (or persons who act as software developers) that design the new product by reusing parts of existing product members and specifying new aspects of the product being developed.

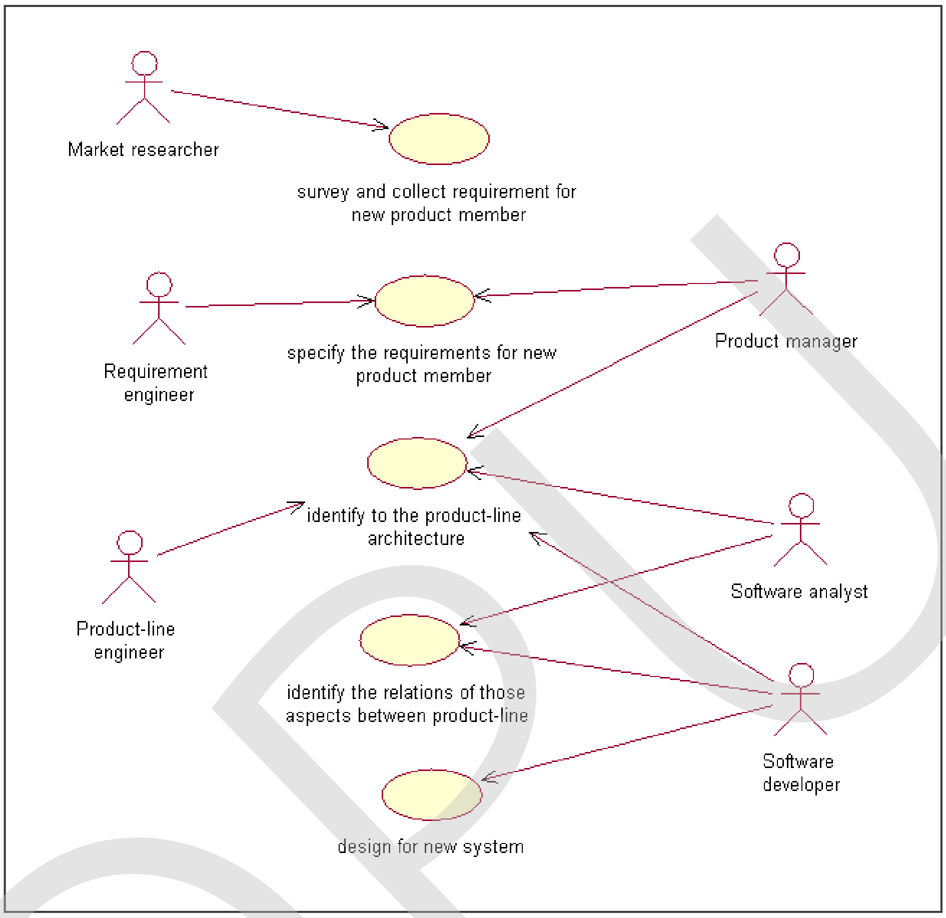

Market researcher
survey and collect requirement for new product member
Requirement engineer
specify the requirements for new product member
Product manager
identify to the product-line architecture
Software analyst
Product-line engineer
identify the relations of those aspects between product-line
Software developer
design for new system

Figure 5- 1: Scenario for Task 1

For this scenario, it is necessary to compare various documents of a product line, documents of existing product members and new product member.

(b) Task 2: The creation of product line from already existing products

As shown in Figure 5-2, the stakeholders involved in this case are:

(a) product managers (or persons who act as product managers) that identify which aspects of the product members should be part of the product line;

(b) product line engineers, software analysts, and software developers (or persons who act as product line engineers, software analysts, and software developers) that design the documents at the product line level; and

(c) software analysts and software developers (or persons who act as software analysts and software developers) that develop the documents at the product line level.

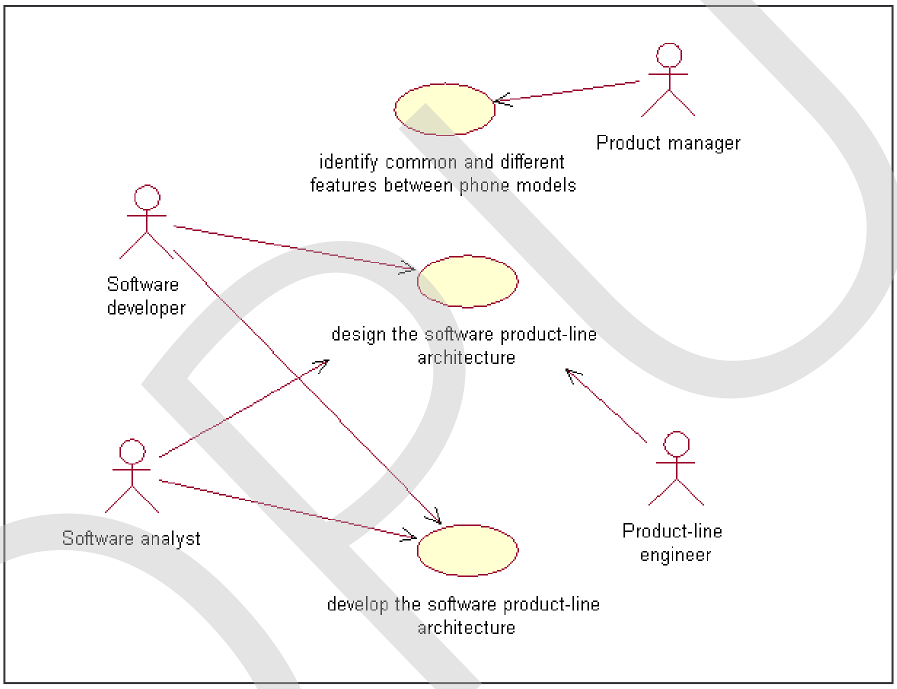

Product manager
identify common and different features between phone models
Software developer
design the software product-line architecture
Software analyst
Product-line engineer
develop the software product-line architecture

Figure 5- 2: Scenario for Task 2

For this scenario, it is necessary to compare various documents of existing product members.

(c) Task 3: Changes to a product member in a product line

As shown in Figure 5-3, the stakeholders involved in this case are:

(a) software analysts (or persons who act as software analysts) that specify changes to be done in a design part of a product member; and

(b) software analysts and software developers (or persons who act as software analysts and software developers) that identify the effects of these changes in the other related design software artefacts.

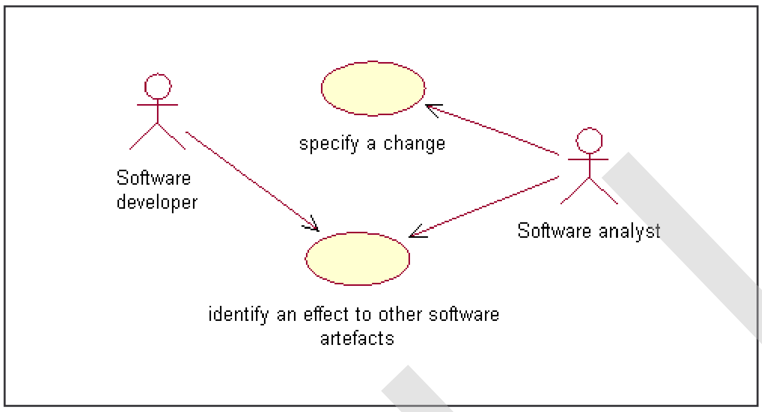


Figure 5- 3: Scenario for Task 3

(d) Task 4: Changes to the core assets of a product line

As shown in Figure 5-4, the stakeholders involved in this case are:

(a) market researchers (or persons who act as market researchers) that identify new features of the system; and

(b) product-line engineers (or persons who act as product-line engineers) that identify which aspects in the core assets of the product family are related to the new features and the effect of these new features to the other documents at the product line level.

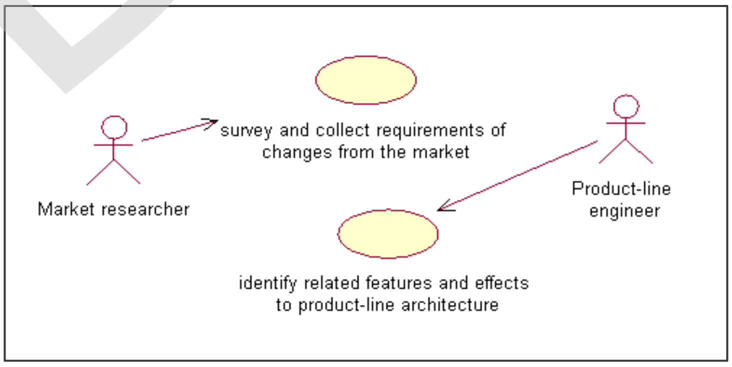

survey and collect requirements of changes from the market
Market researcher
Product-line engineer
identify related features and effects to product-line architecture

Figure 5- 4: Scenario for Task 4

(e) Task 5: Impact of changes to the core assets of a product line and product members

As shown in Figure 5-5, the stakeholders involved in this case are:

(a) product line engineers (or persons who act as product line engineers) that identify the changes to be done at the subsystem; and

(b) software analysts and developers (or persons who act as software analysts and developers) that identify the effect of these changes at the product member design documents.

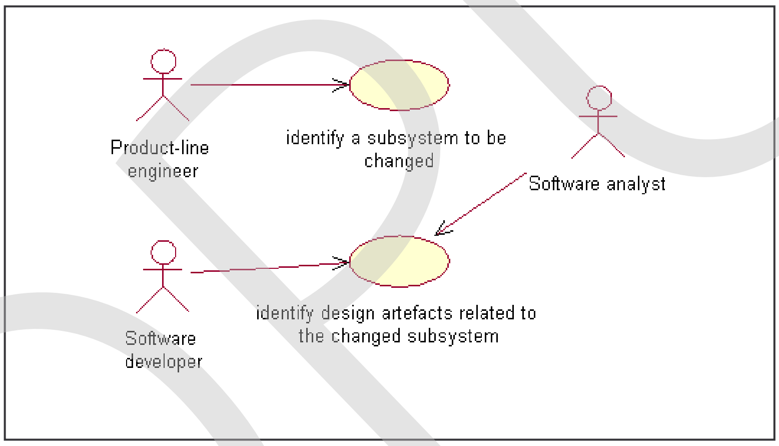

Product-line engineer
identify a subsystem to be changed
Software analyst
Software developer
identify design artefacts related to the changed subsystem

Figure 5- 5: Scenario for Task 5

5.1.3. Measurement of Test

In this evaluation, we have conducted sets of testing related to five different scenarios of product line system development. The tests are justified by concerning two aspects. **Firstly**, we have used the following standard definition of *recall* and *precision* given in (Faloutsos and Oard. 1995). The authors described that precision measure represents the soundness of documents to be retrieved due to an inquiry and recall measure represents the proportion of the relevant documents. We then adopt the measurement techniques to capture the commonality and variability of a product line system. Particularly, we compare the creation of software product line artefacts according to the meta model (a) by applying the prototype tool and (b) by manually performing. As the following, the precision and recall are calculated by:

$$\text{Precision} = |ST \cap UT| \;/\; |ST|$$
$$\text{Recall} = |ST \cap UT| \;/\; |UT|$$

where

- ST is the set of artefacts which are available in a system;
- UT is the set of artefacts which are specified by participants; and
- $|X|$ denotes the cardinality of the set X, in which represents the artefacts are specified validly.

Note that an artefact which is considered in a test is a fine-grained element of document. **Secondly**, we measured the time to complete a task when users were proceeding each one of test cases with normal procedure and available tools, and the time to complete the same task with the proposed model and prototype tool. We also asked the participants to fill in our questionnaire containing the questions with a five-point scale to measure aspect of use. The score of each aspect of use—easy to decide to next step, easy to understand the requirements and design, easy to literate and locate information and overall satisfied with analysis and design based on a five-point scale that score 1 = Strongly Disagree, 2 = Disagree, 3= Neutral, 4 = Agree and 5 Strongly Agree.

5.2. Evaluation Results and Analysis

In this section, we present the results of our evaluation for objective described in Section 5.1 and analyze these results.

In the tests, we have organized groups of participants which properties are fitted into our criteria (see Section 5.2.1). Moreover, the tests are subject to the development of software domain with which the participants are familiar. Every test, we have prepared the participants the software requirements in text, as some tests, some software artefacts are provided by the author. This is due the different objectives of the tests as described in Section 5.1.2. Additionally, since it is an agreement between the author and participants. We do not explicit the source and profile of participating organisations in this report due to the confidential issues. Each group is assigned to perform some tasks (as describe in Section 5.2.2). Table 5-1 shows the participation of each group for each task.

Table 5- 1: Participation of each group for each task

	Task 1	Task 2	Task 3	Task 4	Task 5
Group 1	✓		✓	✓	
Group 2	✓	✓	✓	✓	✓
Group 3		✓			
Group 4		✓			
Group 5	✓	✓	✓		✓

Table 5- 2: Summary of requirements and design artefacts created or changed in each task by all groups

	Task 1 (from 3 tests)	**Task 2** (from 4 tests)	**Task 3** (from 3 tests)	**Task 4** (from 2 tests)	**Task 5** (from 2 tests)
No. of *expected* requirements artefacts to be created or changed	243	1078	113	40	44
No. of *actual* requirements artefacts created or changed	227	971	100	35	40
No. of *expected* design artefacts to be created or changed	347	1417	111	67	40
No. of *actual* design artefacts to be created or changed	270	1313	100	56	40
Total no. of *expected* artefacts to be created or changed	590	2495	224	107	84
Total no. of *actual* artefacts that are created or changed	497	2284	200	91	80

We manually counted the number of software artefacts created in the tests. As shown in Table 5-2, the number of requirements artefacts are created by applying the prototype tool is 243 as the number of requirements artefacts are manually created is 227. These numbers are accumulated from three tests (performed by three groups as shown in Table 5-1). The figures in the table show the difference of the numbers of software artefacts that are created in the same task and having the same software requirements. Moreover, Tables 5-3 to 5-7 show a summary of the number of requirements and design artefacts created or changed in each task by different group.

Table 5-3: Summary of requirements and design artefacts created in Task 1

	UT	ST
No. of requirements artefacts identified by Group 1	166	172
No. of design artefacts identified by Group 1	154	192
Total no. of artefacts identified by Group 1	320	364
No. of requirements artefacts identified by Group 2	36	43
No. of design artefacts identified by Group 2	28	61
Total no. of artefacts identified by Group 2	64	104
No. of requirements artefacts identified by Group 5	25	28
No. of design artefacts identified by Group 5	88	94
Total no. of artefacts identified by Group 5	113	122

Table 5-4: Summary of requirements and design artefacts created in Task 2

	UT	ST
No. of requirements artefacts identified by Group 2	368	437
No. of design artefacts identified by Group 2	554	615
Total no. of artefacts identified by Group 2	922	1052
No. of requirements artefacts identified by Group 3	266	272
No. of design artefacts identified by Group 3	354	371
Total no. of artefacts identified by Group 3	620	643
No. of requirements artefacts identified by Group 4	102	97
No. of design artefacts identified by Group 4	87	99
Total no. of artefacts identified by Group 4	189	196
No. of requirements artefacts identified by Group 5	235	272
No. of design artefacts identified by Group 5	318	332
Total no. of artefacts identified by Group 5	553	604

Table 5-5: Summary of requirements and design artefacts changed in Task 3

	UT	ST
No. of requirements artefacts identified by Group 1	16	17
No. of design artefacts identified by Group 1	15	12
Total no. of artefacts identified by Group 1	31	29
No. of requirements artefacts identified by Group 2	22	23
No. of design artefacts identified by Group 2	12	14
Total no. of artefacts identified by Group 2	34	37
No. of requirements artefacts identified by Group 5	62	73
No. of design artefacts identified by Group 5	73	85
Total no. of artefacts identified by Group 5	135	158

Table 5-6: Summary of requirements and design artefacts changed in Task 4

	UT	ST
No. of requirements artefacts identified by Group 1	27	32
No. of design artefacts identified by Group 1	34	40
Total no. of artefacts identified by Group 1	61	72
No. of requirements artefacts identified by Group 2	8	8
No. of design artefacts identified by Group 2	22	27
Total no. of artefacts identified by Group 2	30	35

Table 5-7: Summary of requirements and design artefacts changed in Task 5

	UT	ST
No. of requirements artefacts identified by Group 2	23	27
No. of design artefacts identified by Group 2	28	25
Total no. of artefacts identified by Group 2	51	52
No. of requirements artefacts identified by Group 5	17	17
No. of design artefacts identified by Group 5	12	15
Total no. of artefacts identified by Group 5	29	32

Additionally, Table 5-8 shows, for each case, a summary of the number of artefacts created or changed in the tests. In the table, ST is the set of artefacts expected to be created or changed; and UT is the set of artefacts created or changed.

Table 5- 8: Summary of artefacts involved in the tests

	Test 1	Test 2	Test 3	Test 4	Test 5
$\mathbf{UT}_{\text{group 1}}$	320		31	61	
$\mathbf{ST}_{\text{group 1}}$	364		29	72	
$\lvert \mathbf{ST}_{\text{group 1}} \cap \mathbf{UT}_{\text{group 1}} \rvert$	309		26	58	
$\mathbf{UT}_{\text{group 2}}$	64	922	34	30	51
$\mathbf{ST}_{\text{group 2}}$	104	1052	37	35	52
$\lvert \mathbf{ST}_{\text{group 2}} \cap \mathbf{UT}_{\text{group 2}} \rvert$	61	887	32	28	43
$\mathbf{UT}_{\text{group 3}}$		620			
$\mathbf{ST}_{\text{group 3}}$		643			
$\lvert \mathbf{ST}_{\text{group 3}} \cap \mathbf{UT}_{\text{group 3}} \rvert$		607			
$\mathbf{UT}_{\text{group 4}}$		189			
$\mathbf{ST}_{\text{group 4}}$		196			
$\lvert \mathbf{ST}_{\text{group 4}} \cap \mathbf{UT}_{\text{group 4}} \rvert$		179			
$\mathbf{UT}_{\text{group 5}}$	113	553	135		29
$\mathbf{ST}_{\text{group 5}}$	122	604	158		32
$\lvert \mathbf{ST}_{\text{group 5}} \cap \mathbf{UT}_{\text{group 5}} \rvert$	110	548	129		26

Table 5-9 shows the results of our testing for each test in terms of recall and precision rates. The results shown in Table 5-9 provide positive evidence about our approach to apply the meta model to specify software product line artefacts at a high level of recall and precision.

Table 5- 9: Precision and Recall Rates (%)

	Test 1	Test 2	Test 3	Test 4	Test 5	Average Precision/ Recall of each group
Precision of group 1	0.85		0.90	0.80		**0.85**
Precision of group 2	0.59	0.84	0.86	0.80	0.83	**0.78**
Precision of group 3		0.94				**0.94**
Precision of group 4		0.91				**0.91**
Precision of group 5	0.90	0.91	0.82		0.81	**0.86**
Average Precision of all tests	0.78	0.90	0.86	0.80	0.82	**0.83**
Recall of group 1	0.97		0.84	0.95		**0.92**
Recall of group 2	0.95	0.96	0.94	0.93	0.84	**0.92**
Recall of group 3		0.98				**0.98**
Recall of group 4		0.95				**0.95**
Recall of group 5	0.97	0.99	0.96		0.90	**0.96**
Average Recall of all tests	0.96	0.97	0.91	0.94	0.87	**0.93**

We applied the histograms to compare the precision and recall in the testing. Figure 5-6 shows that the precision figures in all the cases and the recall figures in all the tests are not so significant. On average, the performance of our approach in terms of precision and recall measurements in tests seems to be consistent.

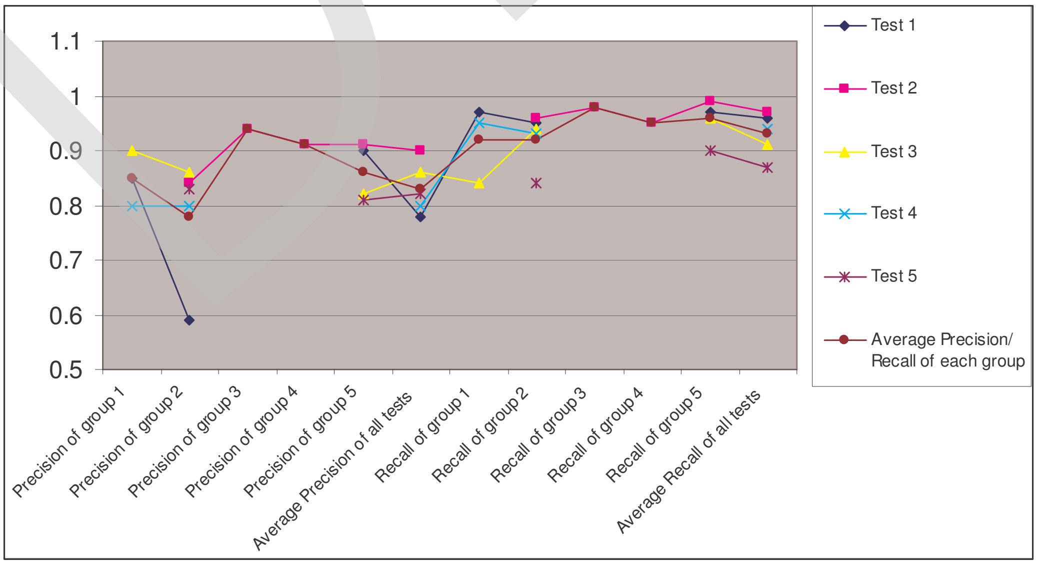


Figure 5- 6: Precision and recall figures of each group as well as the average precision and recall of all tests

Additionally, the time spent during the generation of the software product line artefacts in the tests varies depending on the size of the artefacts and the number of requirements and design artefacts. For example, in Task 2, the processing time in the tests with groups 2 and 5 took much longer than groups 3 and 4 which are significantly smaller. Moreover, the experience and expertise of stakeholders who involve the specification process also contributes to an increase of the processing time. Table 5-10 shows the summary of approximate time spent in each test. As shown in Table 5-10, some cells has no data since those groups did not participate in the tasks. For example, group 1 took part in Tasks 1, 3, and 4 but not in Tasks 2 and 5.

Table 5- 10: Summary of approximate time spent in each test (hours)

	Task 1	Task 2	Task 3	Task 4	Task 5
Group 1	22	-	6	1	-
Group 2	12.5	37	2	0.5	4
Group 3	-	4	-	-	-
Group 4	-	6.5	-	-	-
Group 5	3	16	0.5	-	3.5

Moreover, after completely all tasks, the subject was observed regarding attitudes toward various aspects of software specification without and with the meta model. The results are summarized in Figure 5-7. Figure 5-7 depicts how participants evaluated the applying of the meta model for specifying software product line artefacts i.e. requirements and design artefacts through our questionnaire. As seen in the figure, the participants agreed average of 3 scores ease of deciding next step in specification with the conventional software engineering methodologies, while they agreed 4.3 scores with our approach. Similarly, ease of understanding the rational of systems, the participants agreed on average of 2.3 scores with the conventional software engineering methodologies and 4.5 scores with our approach. Ease of locating the information, the participants agreed 3.1 scores and 4.4 scores for the conventional documents and our proposed documents respectively. On the average, participants feel more satisfied with specification of software product line systems when applying our meta model than conventional methods and techniques.

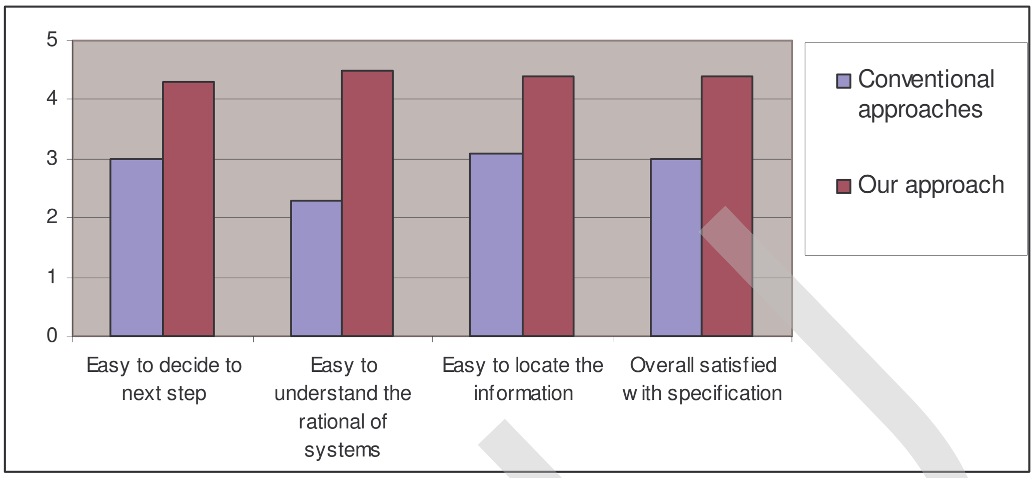


Figure 5- 7: Comparison of Qualitative User Evaluation on conventional software engineering approaches and our approach for specification of software product line systems

5.3. Summary

This chapter has illustrated the tests and their results. We have observed and evaluated the results of software product line specification by applying with the precision and recall measurements. In addition to, the explanations of results have been given. The evaluation and analysis leads to the conclusion of this research that will be presented in next chapter.

Chapter VI Conclusions and Future Work

We provide in this chapter the conclusions, findings, and future work of this research. Section 6.1 presents the overall conclusions. The findings of this research and the future work are described in Section 6.2 and Section 6.3, respectively. The final remarks are listed in Section 6.4.

6.1. Overall Conclusions

This research has contributed to specifying the software product line artefacts in a systematic way. We summarize below the contributions in this research.

A meta model for product line systems – In this research, we proposed a meta model for product line systems in Chapter 3. The concepts and motivation are derived from the background in Chapters 2 and from the survey of the organisations. The model is composed of document types. Firstly, it includes a set of documents created during the analysis process of product line systems. Two types of documents are concerned, namely: (a) *feature model* used to represent reference requirements of product line systems; and (b) *use case* used to represent requirements of product members. Seconding, it includes a set of documents created during the design process of product line systems. Three types of documents are concerned, namely; (a) *class diagram,* (b) *sequence diagram*, and (b) *statechart diagram* used to represent design models of software product line.

The demonstration and evaluation of the approach – The prototype tool in Chapter 4 is implemented in Java to facilitate the demonstration and evaluation of the approach. The main functionalities of the tool are namely: (a) *Document Generator*, specifying documents for software product line systems; and (b) *Document Presenter*, recording and representing the documents created by Document Generator.

Additionally, five cases are created to demonstrate different situations of software product line development, involving (a) different types of documents; and (b) different stakeholders. The experiments of document creation have been evaluated by considering two criteria: (i) easy and (ii) specifying documents. For the latter criteria, the *precise* and *recall* measurements are used.

6.2. The Findings

We summarize below the findings in this research.

6.2.1. Problems of the Establishment and Maintenance of Product Line Systems in Organisations

Many approaches have been proposed to support the development of product line systems. However, there are many associated problems which we describe in this section.

I. The Difficulty to Get Support from Organisations

Due to timing constraints, an organisation usually considers available methodologies rather than establishing product line. Additionally, an organisation has defined and used the current development process for a certain period of time. The organization prefers adopting familiar and practical techniques to support the development process rather than unfamiliar techniques.

II. The Uncontrolled Growth of Variety

Ideally, the establishment of product line needs to have a stable and clear vision of domain; however, it needs to be flexible enough to evolve new requirements. Practically, an organization is uncertain about requirements of product members and develops extra options to anticipate all possible requirements.

III. The Difficulty in Communication

Product line system development is a collaborative process where people from various disciplines need to communicate each other. In other words, communication is required to facilitate and improve the software system development. For example, Meyer (Meyer 1998) suggested that the interaction between stakeholders e.g. between the development team and manufacturing team should be concerned. In addition to (Finkelstein and Guertin 1998), the authors proposed that good communication provides the right requirements at the right time and the right place. Precise requirements must be known in order to facilitate actual implementation.

However, it is not easy to support communication between various groups of stakeholders in an organisation. Successful communication between stakeholders depends on various factors such as: (i) sufficient resources e.g. staff or tool to facilitate the communication; (ii) differences in organisational cultures; (iii) distinct organisational structures; and (iv) stakeholders' attitudes and aspirations. Unsuccessful communication in an organization leads to misunderstanding and lacks of some concepts during the development of software systems.

IV. The Difficulty of Defining Commonality and Variability

Defining commonality and variability of product line is to thoroughly discover the product line descriptions including all common and possible variable aspects. However, there are two issues which cause the difficulty of the practice. These issues are:

Different Perspectives

It is difficult to share views between different products and represent opinions between different tools. For example, sales engineers can offer a new combination of requirements, which seem perfectly reasonable from a customer viewpoint, but appear to be unproved in the technology domain. This difficulty to describe different perspectives of an artefact causes the difficulty of defining commonality and variability.

Lack of Knowledge

Defining commonality and variability of product line needs stakeholders who have enough experience, knowledge, responsibility and authority. However, it is not easy to find stakeholders who are qualified and also available to take this task in charge.

V. The Difficulty of Documenting Management

Data in product line systems rapidly grow as the number of product members in product line increases. Bosch (Bosch 1998) described that stakeholders need to interpret documents and discover relevant documents; therefore, it is important to specify the documents clearly and validly. However, there is a large number and heterogeneity of artefacts and relationships between those artefacts in the domain of product line systems. It is difficult to document the semantics between documents. The difficulty of documenting management leads the following issues: (i) *missing semantics* – documents miss to express the semantics of the context; (ii) *failure of interpreting the semantics* – stakeholders fail to interpret the semantics of documents; (iii) *missing of relevant documents* – stakeholders miss discovering all related documents of interest to them; and (iv) *failure of searching documents* – it is difficult to locate the documents efficiently and promptly.

VI. The Confliction and Dependency between Artefacts in Product Line Systems

Ideally, a feature is an atomic unit and a set of features can be put together to fit with a product member's requirements. However, features are not actually independent. Adding or removing a feature to or from product line has an impact on other features. Additionally, a feature is also related to other types of artefacts in a product line. Therefore, adding or removing an artefact has also an impact on other different artefacts. It leads a difficulty to development and maintenance of product line systems.

VII. The Difficulty to Specialise Variability

Variability can be specialized in different phases i.e. design, implementation, compile, linking, or run-time. However, there are some difficulties in specialization for variability such as: (i) *feature interaction* – specialization of a feature can lead other features in a product line to have unexpected results; and (ii) *separation of concern* – some variability are separated into different artefacts; however, this can lead to the difficulty of specialization.

VI.1 VIII. Issues of Evolution of Product Line Systems

There are some situations that require the evolution of product line systems such as: (i) there is a change on existing product line; and (ii) the core assets of product line have missed some functionalities. These situations occur when the maturity level of product line systems in an organization has grown. The organisation requires a software process which implements new requirements and maintains the consistency of existing systems. However, the issues of evolution are found and defined in (Bosch 2000).

6.2.2. Precision and Recall Measurement

This research has shown that some degree of systematic process in creating software artefacts which can be partly facilitated by the prototype tool. The creation of documents captures the semantics that are represented through the structure of each document type. As shown in this research, the results of creation are measured by using *precision* and *recall* rates. The average precision measured as 85.3% and average recall measured as 83.3%. The results shown in the research are giving positive to the approach.

6.2.3. Benefits

The research has demonstrated the possible situations of the use of meta model during the development of product line systems. We describe below the benefits of use:

I. Reuse

The research has found that the degree of reusing core assets of product line systems affects the cost of the development of the systems. The cost of the product line system development depends on the proportion of reuse of the core assets for the development of product members. However, the poor reuse would have caused higher cost to the product family system development. The specification of software product line artefacts influences the development by reducing the cost i.e. effort and time.

II. Understanding

The research has shown that different stakeholders, who have different experiences in the product line system development process, have different perspectives regarding to software artefacts. Several artefacts are used to represent stakeholders' requirements and design. Coarse-grained software artefacts such as common and variable aspects in feature models and fine-grained software artefacts such as ones in use cases are illustrated though the schema of each software artefact type and can facilitate the understanding of the generated documents.

6.3. Future Work

A number of possible directions for further investigations have been identified. We provide in this section future work of the research, what needs to be done to improve the approach and to increase the benefits of the approach:

- **Tool for Document Generation and Visualisation:** As shown in this research, a large number of various artefacts can be generated for a product line system. It is therefore believed that the approach could benefit by providing tool fully support for the specification of documents. In addition, sophisticated techniques for visualization could support the use of documents more efficiently.
- **Domain Implementation:** The research has focused on two main activities of product line system development i.e. analysis and design. The approach could be expanded to cover the activity of implementation in order to complete the whole life-cycle of the development of product line systems.

6.4. Final Remarks

This research has presented the approach for software product line specification. The research in this research has been contributed to:

- provide the background of product line systems (Chapter 2);
- present the meta model (Chapter 3);

- illustrate the prototype tool (Chapter 4);
- demonstrate and evaluate the approach (Chapter 5).

Bibliography

America, P., H. Obbink., J. Muller, and R. Van Ommering. 2000. COPA: A Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products. *Tutorial in: The First Conference on Software Product Line Engineering (SPLC1)*, Denver, Colorado.

Arango, G., and R. Prieto-Diaz. 1991. Domain Analysis Concepts and Reseach Directions. *Domain Analysis and Software Systems Modelings*: 9-31.

Ardis, M. A., and D. M. Weiss. 1997. Defining Families: The Commonality Analysis. Pages 649-650. *the 19th International Conference on Software Engineering*. ACM Press New York, NY, USA, Boston, Massachusetts, United States.

Atkinson, C., J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wust, and J. Zettel. 2002. *Component-based Product Line Engineering with UML*. Addison-Wesley.

Bass, L., P. Clements, and R. Kazman. 2003. *Software Architecture in Practice*. Addison-Wesley Professional.

Bayer, J., O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. 1999. PuLSE: A methodology to develop software product lines. Pages 122-131. *the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99)*, Los Angeles, CA, USA.

Borland. Borland Together Control Center 6.2.

Bosch, J. 1998. Product-Line Architectures in Industry: A Case Study. Pages 544 - 554. *the 21st International Conference on Software Engineering*. IEEE Computer Society Press, Los Angeles, California, United States.

—. 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. Addison Wesley.

—. 2001. Software Product Lines: Organizational Alternatives. *the 23rd International Conference on Software Engineering*.

CAFE. 2003. from http://www.esi.es/en/projects/cafe/cafe.html.

Campbell, G. H., Jr., S. R. Faulk, and D. M. Weiss. 1990. Introduction To Synthesis, INTRO_SYNTHESIS_PROCESS-90019-N. *Software Productivity Consortium*, Herndon, VA, USA.

Clauss, M. 2001. Modeling variability with UML. *GCSE 2001 - Young Researchers Workshop*.

Clements, P., and L. Northrop. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA.

—. 2004. A Framework for Software Product Lines Practice. http://www.sei.cmu.edu/productlines/framework.html

Cockburn, A. 1997. Structuring Use-Cases With Goals. *Journal of Object-Oriented Programming* Sep/Oct: 35-40.

—. 2000. *Writing Effective Use Cases*. Addison-Wesley, Boston

Coriat, M., J. Jourdan, and F. Boisbourdin. 2000. The SPLIT Method. Pages 147-166. *the First Software Product Lines Conference (SPLC1)*, Denver, Colorado, USA.

Fantechi, A., S. Gnesi, G. Lami, and E. Nesti. 2004. A Methodology for the Derivation and Verification of Use Casees for Product Lines. Pages 255-264. *the 3rd International Conference, SPLC 2004*. Springer Verlag, Boston, MA, USA.

Gomaa, H. 2004. *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures*. Addison Wesley Professional.

Griss, M. L. 2000. Implementating Product-Line Features with Component Reuse. *the 6th International Conference on Software Reuse*. Springer-Verlag, Austria.

Griss, M. L., J. Favaro, and M. d. Alessandro. 1998. Integrating feature modeling with the RSEB. Pages 76-85 in P. Devanbu and J. Poulin, eds. *the 5th International Conference on Software Reuse*. IEEE Computer Society Press.

Halmans, G., and K. Pohl. 2003. Communicating the Variability of a Software-Product Family to Customers. *Journal of Software and Systems Modeling*: Springer.

Jacobson, I. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional.

Jacobson, I., M. Griss, and P. Jonsson. 1997. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley Professional.

Jazayeri, M., A. Ran, and F. V. D. Linden. 2000. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley Pub (Sd).

Kang, K., S. Cohen, J. Hess, W. Novak, and A. Peterson. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Kang, K. C., S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. 1998. FORM: a feature-oriented reuse method with domain-specific architectures. *Annals of Software Engineering* 5: 143-168.

Keepence, B., and M. Mannion. 1999. Using Patterns to Model Variability in Product Families. *IEEE Software* 16: 102-108.

Krueger, C. W. 2001. Software Mass Customization. http://www.biglever.com/papers/BigLeverMassCustomization.pdf.

Lawrence-Pfleeger, S., and S. Bohner. 1990. A Framework for Software Maintenance Metrics. *IEEE Conference on Software Maintenance*.

Lee, K., K. C. Kang, W. Chae, and B.W. Choi. 2000. Feature-based Approach to Object-Oriented Engineering of Applications for Reuse. *Software-Practice and Experience* 30: 1025-1046.

Linden, F. v. d., J. Bosch, E. Kamsties, K. K¨ans¨al¨a, and H. Obbink. 2004. Software Product Family Evaluation. Pages 110-129. *the Third International Software Product Line Conference, SPLC 2004*. Springer Boston, MA, USA.

Northrop, L. M. 2002. SEI's Software Product Line Tenets. *IEEE Software* 19: 32-40.

Ommering, R. v., F. v. d. Linden, and J. Kramer. 2000. The Koala component model for consumer electronics software. *IEEE Computer* 33: 78-85.

Parnas, D. 1976. The Design and Development of Program Families. *IEEE Transactions on software engineering* SE-2.

QADA. from http://www.vtt.fi/ele/research/soh/projects/families/qada.htm.

Redondo, R. P. D., M. L. Nores, J. J. P. Aris, A. F. Vilas, J. G. Duque, A. G. Solla, B. B. Martinez, and M. R. Cabrer. 2004. Supporting Software Variability by Reusing Generic Incomplete Models at the Requirements Specification Stage. Pages 1-10. *8th International Conference, ICSR 2004*, Madrid, Spain.

Schmid, K., and M. Schank. 2000. PuLSE-BEAT -- A Decision Support Tool for Scoping Product Lines. Pages 65-75. *the International Workshop on Software Architectures for Product Families*. Springer-Verlag

Svahnberg, M., J. Gurp, and J. Bosch. 2001. On the Notion of Variability in Software Product Lines. Pages 45-55. *the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001).*

Szyperski, C. 1997. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional

Thiel, S., and A. Hein. 2002. Systematic Integration of Variability into Product Line Architecture Design. Pages 130 - 153 *the Second International Conference on Software Product Lines (SPLC2)*. Springer-Verlag.

Tracz, W., L. Coglianese, and P. Young. 1993. A domain-specific software architecture engineering process outline. *SIGSOFT Software Engineering Notes* 18: 40-49.

UML. from http://www.uml.org.

Weiss, D. 1995. Software Synthesis: The FAST Process. *the International Conference on Computing in High Energy Physics (CHEP)*, Rio de Janeiro, Brazil.

—. 1998. Commonality Analysis: A Systematic Process for Defining Families. *Second International Workshop on Development and Evolution of Software Architectures for Product Families*.

Weiss, D., and C. T. R. Lai. 1999. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison Wesley, Reading, MA.

Biography

Name	Dr. Waraporn Jirapanthong
Education Background	• PhD. in Computer Science, Software Engineering Group, City University, London, UK. • MSc. in Computer Science (Best Science Student with the Highest GPA Award from Professor Taeb Nilanithi Foundation, Thailand, 2001), Faculty of Science, Mahidol University, Thailand. • BSc. in Computer Science (First Class Honours), Faculty of Science, Thammasat University, Thailand.
Employment	Lecturer, Faculty of Information Technology, Dhurakij Pundit University