ENHANCED SOFTWARE PRODUCT LINE (EnSPL) FOR INDUSTRIAL TEST APPLICATIONS

NGO YU CHENG

UNIVERSITI SAINS MALAYSIA

2009

ENHANCED SOFTWARE PRODUCT LINE (EnSPL) FOR INDUSTRIAL TEST APPLICATIONS

by

NGO YU CHENG

Thesis submitted in fulfillment of the requirements for the Degree of Master of Science

July 2009

ACKNOWLEDGEMENTS

First of all I would like to express my utmost gratitude and appreciation to my supervisor, Dr. Vincent Khoo Kay Teong, for devoting his precious time to guide me to complete my thesis writing.

Next, I would like to convey my greatest gratitude to my dearest wife, Leh Cheu Peng, for her constant support and encouragements, without which I would not be able to complete this thesis.

I would also like to extend my sincere thanks to Tse Yi and Tin Tin for giving some constructive ideas during research and thesis writing.

My endless appreciation is also extended to Dr Azman for his encouragement during the most critical time, and in the hour of need and frustration.

Last but not the least, a special thanks is extended to Dr Shahida for her guidance during the thesis correction.

TABLE OF CONTENT

LIST OF FIGURE

LIST OF TABLE

LIST OF ABBREVIATIONS

CM	Configuration Management
DUT	Device Under Test
EnSPL	Enhanced Software Product Line
FODA	Feature-Oriented Domain Analysis
FORM	Feature-Oriented Reuse MehtodologyMethodology
MDIForm	Multiple Document Interface Form
MDA	Model Driven Architecture
MES	Manufacturing Execution System
MIL	Module Interconnection Language
PLE	Product Line Engineering
RDBMS	Relational Database Management System
SEI	Software Engineering Institute, Carnegie Mellon University
SCM	Software Configuration Management
SPL	Software Product Line
UML	Unified Modeling Language

BARISAN PRODUK PERISIAN DITAMBAHBAIK (EnSPL) UNTUK APLIKASI UJIAN PERINDUSTRIAN

ABSTRAK

Pendekatan-pendekatan barisan produk perisian masa kini menghasilkan produk perisian yang mempunyai ikatan kuat dengan aset-aset perisian. Oleh kerana aset-aset perisian mengharungi proses membina-balik, komposisi dan perubahan versi, dan perisian-perisian product juga mengalami proses penyesuaian dan pembetulan, teknik yang menekankan perubahan kod-kod perisian adalah tidak efektif untuk memisahkan aktiviti-activiti kejuruteraan domain dan kejuruteraan aplikasi. Pendekatan EnSPL dicadangkan dalam penyelidikan ini untuk menguruskan perubahan yang tidak dapat dielakkan dalam pengunaan dan perkembangan barisan produk perisian melalui dua-peringkat pengurusan penyusunan pembolehubah. EnSPL bersesuaian menjuruterakan perubahan dalam perkembangan perisian dan perubahan dalam produk-produk perisian melalui kejuruteraan satu-sistem, pengurusan menyusun perisian, skop produk, perkembangan barisan dasar perisian dan produk profail. Aplikasi pendekatan EnSPL telah terbukti dalam pembinaan aplikasi-aplikasi ujian perindustrian di salah sebuah syarikat antarabangsa di Malaysia. Peningkatan dalam pendekatan tersebut juga dibandingkan dengan pendekatan-pendekatan masa kini yang lain.

ENHANCED SOFTWARE PRODUCT LINE (EnSPL) FOR INDUSTRIAL TEST APPLICATIONS

ABSTRACT

Current software product lines (SPL) methodologies develop software products in the manner that is tightly coupled to the software assets. While software assets might go through a process of rebuilding, composition and version upgrading, and the same go to a specific software product's development that might require adaptation and correction, the code-based derivative techniques used in those methodologies are not effective to separate the two corresponding domain engineering and application engineering activities. An enhanced SPL (EnSPL) methodology was proposed in this research to manage the inevitable evolution of software product line development through the two-stage configurable variability management. The EnSPL appropriately engineered the software development variability and software product variability through the single-system engineering, software configuration management, product scoping, product baseline development and product profiling. The methodology applicability has been proven in the development of Industrial Test Applications in one of the multinational companies in Malaysia. The enhancement of the methodology was also evaluated in comparison to the current SPL methodologies.

CHAPTER 1

INTRODUCTION

1.1 Overview

A software product line (SPL) is a new paradigm in the software development process that emphasizes on the development of common architecture for a family of software products (Pohl *et al*. 2005). Conventionally, software product line (SPL) methodology emphasizes on a two-stage development process for a family of similar software products in which domain engineering activities develop common software assets and application engineering activities derive each specific software application. Since the first Software Product Line Conference (SPLC 2000) (Donohoe 2000), the SPL approach has gained serious attention from both the practitioners and researchers. The reason is rather simple. Obviously, the requirements for software applications are increasing and most of these software applications share commonality in software design with slight differences in software implementation. Thus, in the domain engineering stage of SPL, the commonalities of these applications are grouped together in the single effort of domain analysis and domain design. At the same time, variability is introduced to the design artifacts so that the common applications can be derived to form a specific software application in the application engineering stage.

This chapter will discuss the research motivation, problem statement, research objectives, research approach and thesis outline.

1.2 Motivation

The inherent complexity of managing and integrating the two isolated activities of SPL methodology has prompted researchers to look for an improved methodology (Atkinson *et al*. 2000, McGregor 2005 and Krueger 2007). It has been noted that the recent proposed SPL methodologies by McGregor (2005) and by Krueger (2007) have proposed automatic generation of a software product over the manual derivation technique.

McGregor (2005) reported that the generation technology, claimed as the more feasible approach, could be used to transform some form of specification into the executable software product, but the expressiveness of specification language, which might complicate the transformation infrastructure still required further investigation for simplification. On the other hand, Krueger (2007) proposed that product configurator was used to generate the software product automatically from the product feature profile and core assets. Even though the conceptual model of the methodology has been described (Krueger 2007), the practical implementation technique has still yet to be practically demonstrated.

1.3 Problem Statement

The advancement of SPL in enhancing the efficiency of software development is widely recognized and appreciated by organizations and research communities (McGregor *et al*. 2002, Sugumaran *et al*. 2006). Many SPL methodologies have been proposed in the literature, some of them use the two-stage engineering (domain engineering and application engineering) approach (Atkinson *et al*. 2000, van Ommering *et al*. 2000),

while others use a single-stage or single-system engineering (domain engineering only) approach (Krueger 2002, Krueger 2006a, Krueger 2006b). However, all these methodologies have issues arise from the management of increasing variability, tightly coupled relationship of a specific software product and software assets, and least effectiveness in software evolution and maintainability. The single stage approach such as 3-Tiered Methodology (Krueger 2007) has reduced variability through consolidation of domain engineering and application engineering. Nevertheless, it has only been presented conceptually without much detailing in system design and implementation. Although there are several success stories in the software product line development (Product Line Hall of Fame 2008, Linden *et al*. 2007), they are not being linked to any of the methodological steps. Therefore, the question remains if current two-stage and single stage methodology can be further enhanced? Or if the single-stage methodology as proposed by Krueger (2007) is effective to improve the two-stage methodology, can it be described in more concise methodological steps?

1.4 Research Objectives

This research was conducted to achieve the following objectives:

a. To enhance the current software product line (SPL) methodology by appropriately re-engineering the software development variability and product variability through the two stages of configuration management.
b. To describe the SPL development with more concise methodological steps through the feasibility study in the domain of industrial test applications.

1.5 Research Approach

The research work has started with studies of the fundamental technologies of SPL such as the management of commonality and variability in SPL and its derivation techniques, and factors that have brought software product line development into the limelight as new software development paradigm of software engineering. It has also been recognized that SPL is software-family-based development, reuse-driven, economically driven, and architecture-based, which is very much different from the traditional software development. The SPL success factors are mainly contributed by the architectures, domain-specific software management, configuration management and business models (Mili *et al*. 2002).

The literature study for understanding the approach or chronological order used in various methodologies to manage SPL gave an insight of the specific SPL methodological design. For example, KorbrA (Atkinson *et al*. 2000, Atkinson and Muthig 2006) denotes itself as a component-based incremental product line development and methodology for modeling architectures, and the 3-Tiered Methodology (Krueger 2007, Krueger 2006a, Krueger 2006b) focuses on the automated application engineering to consolidate the engineering management of software assets and specific software products creation. The EnSPL has drawn ideas from many of these SPL methodologies but builds its own way towards more effective software product line management, especially in the variability management aspect. Below is the illustration of the variability management in EnSPL as in comparison to other methodologies. KorbrA and 3-Tiered

Methodology are selected for comparison because each represents its kind in the advancement of SPL methodologies or SPL generation.

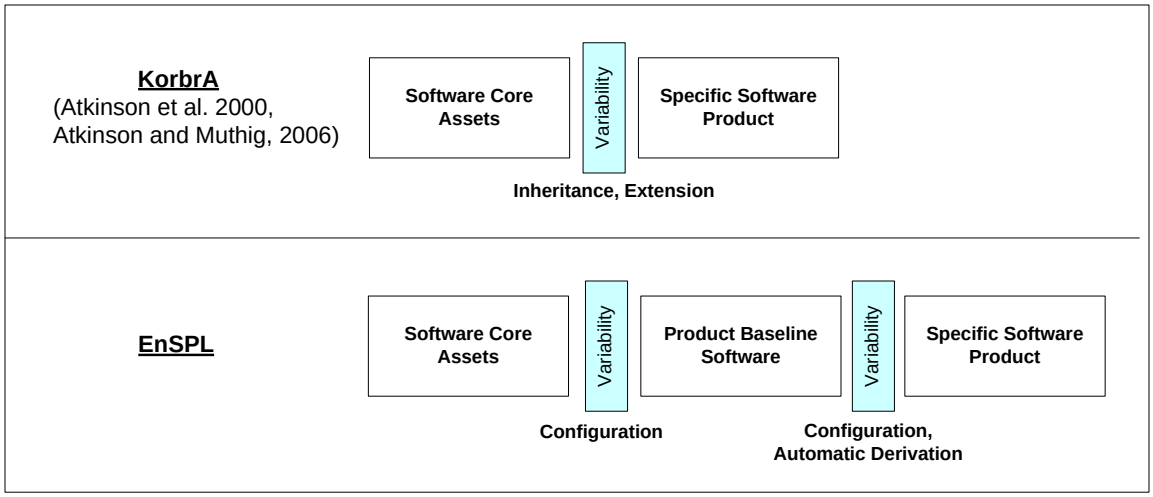

Figure 1-1 Proposed EnSPL and its comparison with other methodologies

As it shall be discussed later in Chapter 3 and Chapter 4, EnSPL separated variability management into two stages of configurable management. The first configuration stage of EnSPL focuses on the *programming in the large* (version, rebuilding, composition), while the second configuration stage emphasizes on the *programming in the many* (multiple software product development, process supports, concurrency).

The essence of the EnSPL variability management approach is that both software assets development and specific software products developments are loosely coupled from each others through the intermediary of product baseline software development. The impacts are tremendous because the product baseline software has reduced the combinatory complexity among the composition of software assets themselves, and among the

software assets and the specific software products. Besides, the approach also facilitates the generative programming (Czarnecki and Eisenecker 1999, Jarzabek and Knauber 1999) for the automatic derivation of a software product.

The design of the EnSPL was based on the literature review of software product line, and the practical experience of actual industrial test applications development in one of the multinational companies in Malaysia. The proposed EnSPL is evaluated qualitatively through the comparison with two other typical SPL methodologies in literature namely KorbrA and 3-Tiered Methodology. A prototype is developed as proof of concept to the proposed EnSPL, and finally the summary of enhancement in the proposed EnSPL is elaborated with comparison to the KorbrA and the 3-Tiered Methodology in terms of reusability, configurability, adaptability and software maintainability.

1.6 Thesis Outline

This research is first conducted through the literature study of a software product line in Chapter 2. The development of software product line requires the paradigm shift from the development of a traditional individual software product to the development of common product line architecture and variability management for a family of software products. The software product line activities are discussed as consisting of domain engineering activities, enabling technology of reusable software assets development and application engineering activities. Several SPL methodologies in the literature have been reviewed for the understanding of the various approaches, and compared from the view point of the product line engineering approach and the product derivation technique.

In Chapter 3, the EnSPL methodology is presented on software product line development from the domain analysis to specific software product development through various stages of feature modeling, product scoping, product baseline software development, and product profiling. The methodology has also been described by several architectural view models such as logical view, process view, development view and physical view. The variability management concept and approach taken in EnSPL is also elaborated in this chapter. The chapter ends with more concise elaboration of the development view on the software design and realization aspect.

This is followed by Chapter 4, which discusses the contribution of EnSPL in comparison to other SPL methodologies such as KorbrA and 3-Tiered Methodology. Chapter 4 ends with the demonstration of seven concise methodological steps of EnSPL in the domain of industrial test systems. Chapter 5 concludes the outcome of this research.

CHAPTER 2

LITERATURE REVIEW

2.1 Software Product Lines

A software development process has evolved along the path of single-system development. However, due to the increasing demand for software systems and their customization, the conventional single system approach is no longer cost effective and also rather time consuming to meet with each system requirement. Building a common platform to accommodate for the common development practice and design becomes a necessity. Systems are built by using the common platform and adding with the specific code for customization. The process is analogous to building a manufacturing production line that assembles parts to build a complete product. The production line is designed to accommodate most of the common parts with some variation to fix in the customization parts that produce customized products. The idea has been introduced into the software engineering mainstream as a new software development paradigm known as a software product line (SPL) engineering.

Northrop (2007) gives the definition of SPL as

> *"A software product line is a set of software-intensive systems that share a common, managed feature set satisfying a particular market segment's specific need or mission and that are developed from a common set of core assets in a prescribed way"*

The interest in software product lines emerged from the field of software reuse when developers realized that they could obtain much greater reuse benefits by reusing software architectures instead of reusing individual software components. The software industry is increasingly recognizing the strategic importance of software product lines (Clements and Northrop 2001).

Unlike other software engineering practices, SPL proposes a more systematic approach and has an integrated development life cycle where application engineering and domain engineering processes and their interaction are guided by a set of products in a specific domain. The Software Engineering Institute (SEI) of Carnegie Mellon University views SPL as consisting of three essential activities: core asset development, product development and management as shown in Figure 2-1. (Northrop 2002, Northrop 2007).

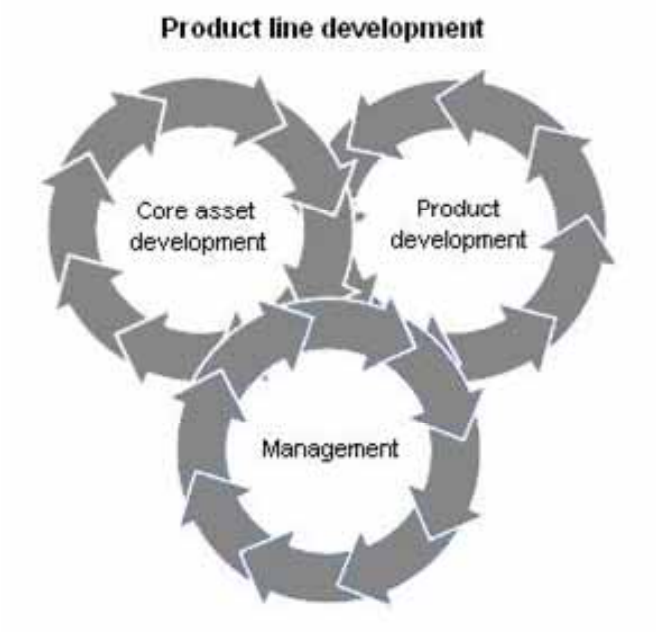

Figure 2-1 Essential product line activities (Northrop 2002, Northrop, 2007)

Each rotating circle represents one of the essential activities. All three are linked together and in perpetual motion, showing that all three are essential, are inextricably linked, can occur in any order, and are highly iterative. Core assets might be evolved from activities such as mining from existing products for generic assets, revision of existing core assets or new development. There is a strong feedback loop between the core assets and the products. Core assets are refreshed as organizations develop new products. Management will ensure that both core assets development and product development are managed carefully at all levels.

The product line engineering starts with development of core assets, which often included but are not limited to the architecture, reusable software components, domain models, requirements statements, documentation and specifications (Sugumaran *et al*. 2006, Northrop 2002). The architecture is the key asset among the collection of core assets. Each system in software product line is developed by taking applicable components from a pool of common asset base and tailoring them through preplanned variation mechanisms and predefined assembly guide or plan. The process of core assets development and product development is an incremental and iterative activity. It must be guided by operational management to ensure that the development engage in a standard process and follow the same rules with proper documentation.

The SPL analyzes requirements of all candidate applications in the domain by commonality and variability. The commonality denotes the common requirements shared

by all applications while the variability denotes the specific requirements of those applications (Coplien *et al*. 1998). Each individual software product is constructed by utilizing the common artifacts known as core assets and adding in with the specific parts. The SPL enhances the efficiency of the activity of software development by reducing the repeating software design and development as undergone by conventional software development since most of the software developed within the same domain share more commonality than variability. For example, applications in automotive industry (Thiel and Hein 2002a, 2002b) exhibit almost the same characteristic to assemble parts and do test and measurement. Therefore, applications are not necessary to be developed from scratch.

2.1.1 Managing Commonality

Production lines in manufacturing plants have long been designed in such a way to use common parts for as many products as possible. Many products are designed to use common parts as many as possible to the significant economies of production and maintenance. Although software product lines are different, they share the same objectives as the manufacturing production lines.

Managing commonalities in SPL is not simply a sharing of the common codes of various applications, but it also includes design reuse. It is the architectural blocks that are in great value to the organizations adopting the concept of SPL. Commonality in the architectural design greatly reduces the amount of time required to develop an application. Software engineers are able to focus on the problems on hand and develop codes to solve

them rather than being forced to attend the design issues as well. This greatly enhances the efficiency of their solution skill while maintaining the same standard of design that facilitates the maintenance in the long run of a product life cycle.

Companies recognize that building systems from common assets can yield amazing improvements in productivity, time to market, product quality, and customer satisfaction. In any circumstances, they strive to manage the system from various perspectives ranging from the conventional practice that emphasize the commonalities of low level components such as operating systems, programming languages, service components and code reuse, to the more appealing SPL commonality approach such as reference architecture, application framework, and design reuse. Managing commonalities in large scale development such as the SPL require a careful consideration from many aspects. Sugumaran *et al*. (2006) outlines the three important aspects which are an organizational aspect (centralized or distributed asset development), a technical aspect (core asset development, product development, and runtime dynamism) and process aspect (proactive, reactive, and extractive models). The organizational aspect and technical aspect are in correspondence to the three essential activities described by Northrop (2007) (refer to Figure 2-1).

The core asset development activities are the software commonality development, which is initiated from the context analysis of product family and the exploration for reuse. The engineering process starts by identifying the product functional features and also the quality attributes of the system. Product functional features consist of feature lists, feature

description, and services. Quality attributes are the desirable non-functional features such as usability and scalability. Early identification of the required features will help to incorporate such a feature into the core asset development during the design stage; otherwise it can still be incorporated into the final system through the refinement process but this is less efficient due to the increased overheads.

One of the examples in managing the commonalities in software engineering practice is the implementation of common interface. A set of standardized interfaces is abstracted away from the actual implementation detail which is encapsulated into a separate object module. Here, the commonality is the interface, and the variability is the implementation code. Nevertheless, the most noted commonality in SPL is the reference architecture that it defines. During the domain analysis of the proposed EnSPL, features identified in the domain are captured in the hierarchical feature diagram. Commonality in features has been implemented as feature-based software modules, which implement the same interface. In this regard, the commonality in requirement in terms of features is implemented as common and individual software assets, which can be incorporated into the main stream development through the standard interface. However, the implementation of each software asset varies according to the feature that they characterize.

2.1.2 Managing Variability

Variability is the inherent part of the product line design. It provides options to describe different products. For simple variability, a property file or wizard is sufficient to express

the variability that is needed to describe a product. For greater freedom to describe a product, a tabular or tree-based configuration is being used. A Software Configuration Management technique (White 2000, Pavel *et al*. 2004) is also being used to manage variability in product line. The technique provides better guidance because variability is defined from a set of available options. In a more advanced approach, domain specific language (DSL) may be used to structure and manage the variability elements through powerful graphical or textual languages (Mernik *et al*. 2005). The DSL approach can express variability more efficiently where, for example, the type A can be connected to type B and later can be linked to type C in a creative way.

Variability defines how each application differs from one another within the same domain in terms of services and functionality. These variabilities are incorporated into the common design framework of SPL to form a complete application (Krueger 2002, Pohl et al. 2005). The merging of variabilities and commonalities forms what is called the product line's application.

In SPL, variabilities are sometimes used interchangeably with “features”. The use of features is motivated by the fact that end users and engineers often speak of product characteristics in terms of “features the product has and/or delivers”. They communicate requirements or functions in terms of features and, to them, features are distinctively identifiable functional abstractions that must be implemented, tested, delivered, and maintained (Kang *et al*. 1990, Lee *et al*. 2000).

The usability of software product line increases with the support of more features. Features are normally incorporated to the common asset framework through variation points or hotspots (Schmid, 1997). With the increase in the number of features supported, the variation points are increasing as well. This may possess greater challenges in managing variations throughout the product line life cycle because of the increasing complexity in implementation. Variability, however, is important in the design of product line because it facilitates the incorporation of new software products in a product family, thus adding value to the product line.

Pohl (2005) differentiates variability of a software product line from the viewpoint of the end users. The external variability denotes the variability of domain artifacts that are visible and configurable by the end users, who participate in the product development and decide on the unbound variability of product line artifacts. The internal variability, on the other hand, denotes the variability of domain artifacts that are hidden from the end users. The definition and resolution of internal variability are only within the responsibility of product line provider. In the proposed EnSPL development, internal variability is used to define the structural aspect of product line development through aggregation or generalization for the development of product baseline software while external variability is used to define the behavioral aspect of specific software product development by configuration management.

2.1.3 Reusable Architecture

The objective of SPL is to facilitate the development of similar software products in the same domain through construction of common software architecture and derivation of it into a customized application. Software architecture is often regarded as the key factor for product line success because it embodies the earliest design decision and provides a framework within which reusable components can be developed and integrated (Kaisler 2005). Good and manageable architecture is always necessary to produce high quality products that belong to the family. Baldwin *et al.* (2006) emphasized that generally the "manageable" software architectures (such as Windows and Office) give rise to product lines and product families, while "unmanageable" architectures (such as Apache and Linux) give rise to modular clusters and open source communities.

The difference between software architecture in general and product-line architecture is that most software architectures are application architectures that focus on a single, monolithic application. The SPL emphasizes on developing a product-line domain (reference) architecture that is used by all products in the family. Such architecture will go through the maturing process until it is robust, flexible, and highly customizable to facilitate the instantiation of the core architecture for multiple products. However, it is often harder to develop the reference architecture in comparison to the development of single application architecture. This is due to the need to incorporate variation point plug-in to the design of reference architecture (Mili et al. 2002).

Developing the reference architecture for SPL is always a difficult task because it means developing an overall application framework that can be instantiated into various customized applications (Mili et al. 2002, Woods and Rozanski 2005). Issues that could arise from this activity are that: 1) Measuring the conformance of the instantiated architecture to the reference architecture, or more specifically, how to determine if products are verified with respect to the product line architecture. 2) Synchronizing, modifying, and maintenance of the product line architecture so that it can evolve with new product requirements in terms of technological change, fixing existing problems, or adding new functionalities.

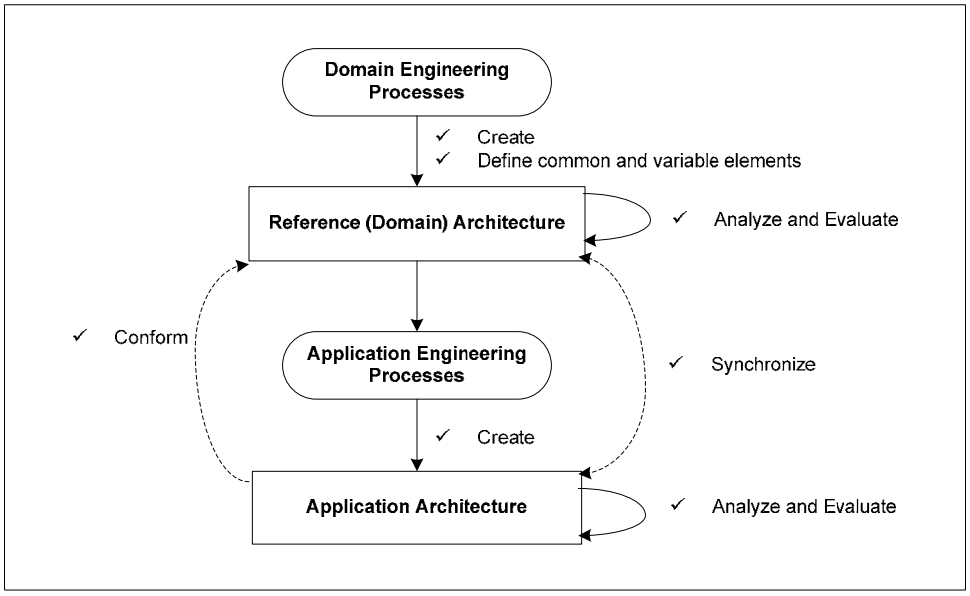

Figure 2-2 Product-line and product-specific architectures (Mili *et al*. 2002)

Figure 2-2 illustrates the activities concerning software architectures in a SPL context: creation, synchronization, conformance, analysis and evaluation. The outcome of the domain engineering process is reference architecture while the outcome of the application engineering process is application architecture. In a software product line, the software

architecture is used by all products in the family. It is known as domain architecture or reference architecture. Such architectures are generally robust, flexible, and highly customizable to facilitate the instantiation of the core architecture in multiple products (Pohl 2005). Normally, reference architecture is developed as framework such as KorbrA (Atkinson et al. 2000), and application architecture is instantiated from the domain architecture when specific application requirements are known. EnSPL proposes development of application framework where multiple feature-based software modules can be plugged in by configuration to form the product baseline software.

Besides, software architecture is also defined in terms of identified system components, how they are connected, and the nature of the connections (protocols for communication, synchronization and data access) (Kruchten 1995). The software architecture development has diversified further from a code reuse to design reuse, by emphasizing on the patterns of relationships among the elements of the architecture. A number of general-purposed architectures frequently occur, such as pipes/filters, client/server, three-tier, and layered (Buschmann et al. 1996).

In contrast to the class library approach, the structure of interconnections between components is reused, rather than just the components themselves. Software architecture reuse is based on the definition of ‘architectural styles’, which are a description of families of architectural designs that share a set of common assumptions (Hong 2005, Bushmann et al. 1996, Mary 1995). Architectural styles improve communication between designers, who can refer to a shared terminology for reusing design solutions. However,

depending on the domain requirements, normally different architectural style is selected. For example, for real-time systems such as industrial process control, an architectural style called event-based approach for the independent component systems (Bushmann et al. 1996) is used.

2.2 Software Product Line Activities

A software product line covers processes for building and managing of typically separated two key areas: domain engineering and application engineering as illustrated in Figure 2-3. Domain engineering covers activities on analyzing and identifying commonality and variability of domain applications. It is started with the domain analysis which produces domain models that are normally served as reference to the subsequent software core asset development such as architecture and software components (Figure 2-3). On the other hand, the application engineering is activities on specific software product development. It instantiates the domain models and specializes on them to cater for specific application requirements.

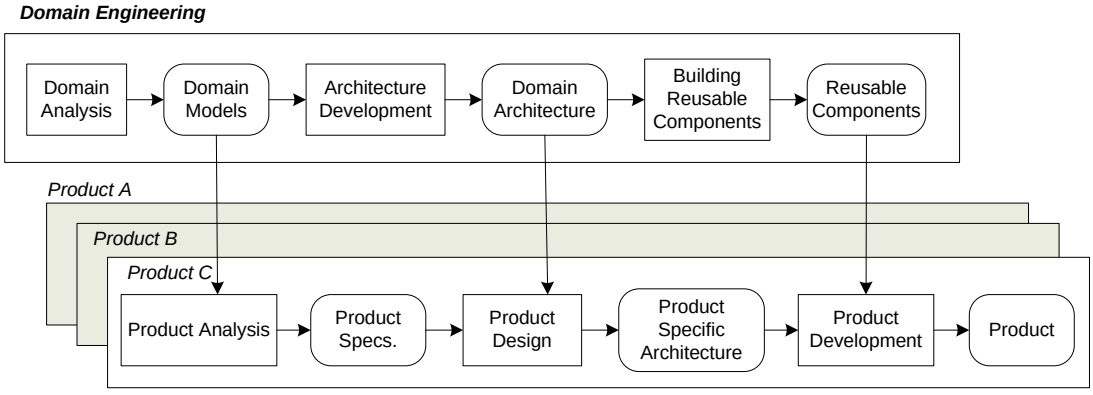

Figure 2-3 The SPL lifecycle (Mili *et al*. 2002)

2.2.1 Domain Engineering

In domain engineering, the variability and commonalities of the product line's reusable core assets such as requirements, architectural elements, or solution components are captured in the domain models. Domain analysis is performed to analyze features of the software family that are common, optional, or alternatives. Domain analysis technique such as Feature-Oriented Domain Analysis (FODA) (Kang *et al*. 1990) had been used extensively (Kang *et al*. 1998, Kang *et al*. 2002, Krueger 2007) to model domain capabilities in terms of features.

2.2.1(a) Feature-Oriented Domain Analysis (FODA).

FODA method is one of the domain analysis techniques emphasizing on those features that a user commonly expects in application in a domain (Kang et al. 1990). This method identifies feature as a prominent or distinctive user-visible aspect, quality, or characteristic of a software system. Therefore, features are said to be used to parameterize domain products from the user's perspective. However, features are still far abstracted from implementation and realization of effective measure on feature componentization and feature dependencies are still an active research area (Lee and Kang 2004, Cho et al. 2008)

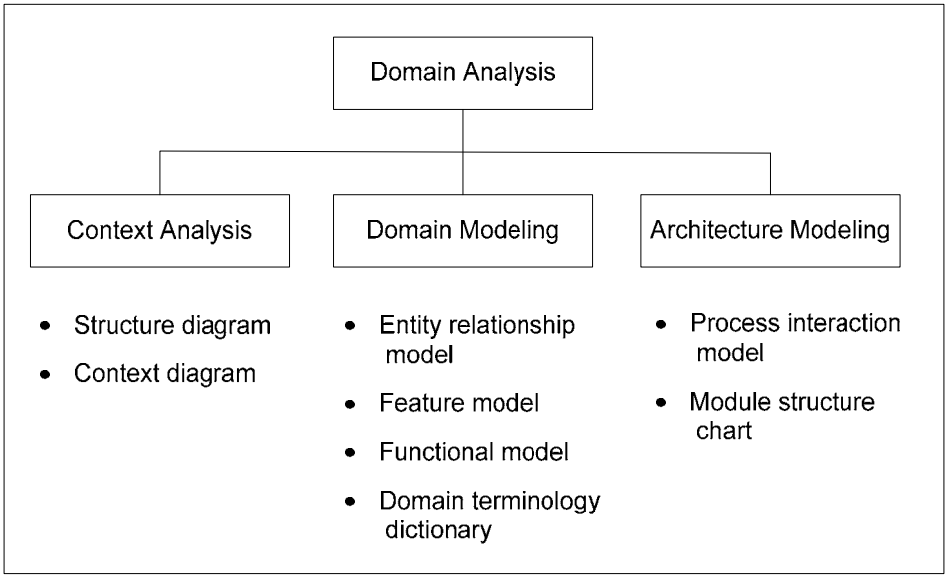

Figure 2-4 Phases of FODA process (Kang *el al*. 1990)

Figure 2-4 shows the FODA analysis process that focuses on the three sets of representation of the domain: context analysis, domain modeling and architecture modeling. Context analysis determines the boundary of a domain to analyze, with the objective to identify the relationship between applications in the domain. Domain modeling activities will use the information sources and other products of context analysis to support the creation of a domain model. In architecture modeling, the structure of implementations of software in the domain is established by using information from a domain model. The representations of domain models in feature model provide a mapping to the architectural models for constructing applications (Kang et al. 1990).

The next few sections will focus on the *Feature Model* as the single important domain model from FODA to the proposed EnSPL methodology of this thesis. Feature model

contributes to the variability management of EnSPL methodology as the conceptual level variability management which will be mapped to the class model at the implementation level, as discussed in section 4.3.3. Other domain model such as entity relationship model is captured as feature diagram as shown in Figure 2-5. The architecture modeling activities such as process interaction model and module structure chart are discussed as 4+1 View Model of Software Architecture (Krutchten 1995) in section 3.3.

2.2.1(b) Feature Model

A feature model captures domain features and their relationship. It is usually illustrated in the form of a feature diagram as shown in Figure 2-5.

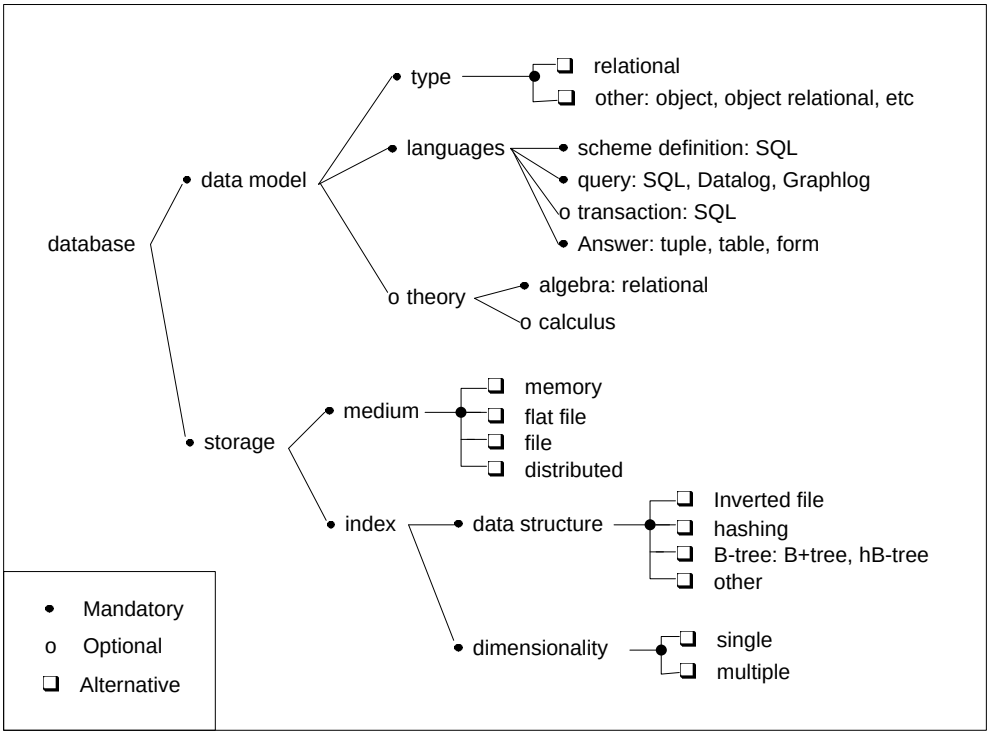

Figure 2-5 Feature diagram: mandatory, optional, alternative (Butler *et al*. 2002)

For a product line or framework, a feature model classifies each feature as mandatory, optional or alternative in order to express the commonality and variability across applications in the product line. Common features among different products are modeled as mandatory features, while different features among them may be optional or alternative (Kang *et al*. 1990). Optional features represent selectable features for products of a given product line and alternative features indicate that no more than one feature can be selected for a product.

There are several views of structure for a feature model. The basic concept is a finite set F of the features. The usual relationship of interest is the sub-feature relationship that defines a hierarchy of features. There is also a dependency relationship between features, and the actual dependency may be specified using a constraint. The classification of feature model can be considered from the view point of *variability* (Butler *et al*. 2002) and *kind-of* (Kang *et al*. 1998). The *variability* viewpoint map F to the set of {mandatory, optional, alternative}, and the *kind-of* viewpoint map F to the set of {capability, technology, environment, implementation}. The set of {mandatory, optional, alternative} helps to segregate the aspect of commonality and variability in the SPL analysis, while the set of {capability, technology, environment, implementation} give a broader view of the domain feature space that must be ultimately mapped to the domain design models.

2.2.1(c) Issues related to feature model

Large software systems are increasingly expressed in terms of the features they implement. Consequently, there is a need to express the commonality and variability

between products of a product family in terms of features (Kang *et al*. 2002, Lee and Muthig 2006). Unfortunately, technology supported (SPL methodology, domain modeling, feature modeling, etc.) for the early aspect of a feature is currently limited to the requirements level. There is a need to extend this support to the design as well as implementation so that domain models such as a feature model can be directly mapped to the implementation level.

Features are used in requirements engineering to define optional or incremental units of change. They have many-to-many relationships to the individual requirements. Thus, tracing features to the implementation of feature-based software components is complex. In ideal cases, a particular feature implementation is localized to a single module; but in many cases, features will cross-cut multiple components (Cho et al. 2008).

2.2.2 Reusable Software Assets – Enabling Technology

Software product line development is about the systematic reusability of software assets. These software assets can be created by a variety of techniques found in literature such as architectural patterns, design patterns, framework development and component-based development depends on the domain-specific design and implementation. Therefore, understanding and selecting the right reusable techniques and relating them to the domain-specific problems are crucial for the success of software product line development.