Adaptable Software Reuse:

Binding Time Aware Modelling Language to Support Variations of Feature Binding Time in Software Product Line Engineering

by

Armaya'u Zango Umar

Supervisor:

Dr. Jaejoon Lee

Thesis submitted for the
degree of *Ph.D in Computer Science*

September 2019

Acknowledgement

All praises are to Allah whose divine destiny made it possible for me to earn a Ph.D.. Special thanks to my parents for nurturing me and supporting me with prayers and unconditional love through thick and thin. May Almighty reward them in abundance.

I most sincerely thank the National Information Technology Development Agency (NITDA) for providing me with three-year funding. NITDA staff have been very helpful and responsive to my incessant demands. Thank you NITDA and God bless the federal republic of Nigeria.

I am ever grateful to my supervisor, Dr. Jaejoon Lee, for being there for me throughout my Ph.D. journey. Jaejoon has been exceptionally patient, supportive and provided me with invaluable guides all along. I am also sincerely grateful to panel members of my Ph.D. appraisals - Prof. Pete Sawyer, Dr. Geral Kotanya, and Dr. Ioannis Chatzigeorgiou. Their regular and constructive feedbacks were instrumental to my Ph.D. success.

I also appreciate the support of my siblings for their prayers and words of encouragement. My special regards to my wife, Barira Hamisu, and my two daughters, Maryam and Asiya for their patience of my absence and my aloofness during the intense period of my research.

I will also like to appreciate my colleagues at Infolab. These include Mus'ab, Yusuf, Bruno, Ono, Assyl, Sunday, Juliana to mention but few. It has been a nice feeling being in their midst as 'co-travellers' and knowing fully there were always ready to help.

Abstract

Software product line engineering (SPLE) is a paradigm for developing a family of software products from the same reusable assets rather than developing individual products from scratch. In many SPLE approaches, a *feature* is often used as the key abstraction to distinguish between the members of the product family. Thus, the sets of products in the product line are said to have 'common' features and differ in 'variable' features. Consequently, reusable assets are developed with variation points where variant features may be bound for each of the diverse products.

Emerging deployment environments and market segments have been fuelling demands for adaptable reusable assets to support additional variations that may be required to increase the usage-context of the products of a product line. Similarly, *feature binding time* - when a feature is included in a product and made available for use - may vary between the products because of uncertain market conditions or diverse deployment environments. Hence, variations of feature binding time should also be supported to cover the wide-range of usage-contexts.

Through the execution of action research, this thesis has established the following: Language-based implementation techniques, that are specifically proposed to implement variations in the form of features, have better modularity but are not better than the existing classical technique in terms of modifiability and do not support variations in feature binding time. Similarly, through a systematic literature review, this thesis has established the following: The different engineering approaches that are proposed to support variations of feature binding time are limited in one of the following ways: a feature may have to be represented/implemented multiple time, each for a specific binding time; The support is only to execution context and therefore limited in scope; The support focuses on too fine-grained model elements or too low-level of abstraction at source-codes.

Given the limitations of the existing approaches, this thesis presents binding time aware modelling language that supports variations of feature binding time by design and improves the modifiability of reusable assets of a product line.

Declaration

I declare that the work in this thesis is my own work and has not been submitted either in whole or in part for the award of a higher degree elsewhere. Any sections of the thesis, which have been published, are clearly identified.

.....

Armaya'u Zango Umar

0.1 List of Acronyms

AOP	Aspect Oriented Programming
ATL	Atlas Transformation Language
DOP	Delta Oriented Programming
FOP	Feature Oriented Programming
JEE	Java Enterprise Edition
MDA	Model-Driven Architecture
MDD	Model-Driven Development
M2M	Model to Model
M2T	Model to text
MOF	Meta Object Facility
NATO	North Atlantic Treaty Organization
OCL	Object Constraint Language
OMG	Object Management Group
OOP	Object-Oriented Programming
OOSE	Object-Oriented Software Engineering
PIM	Platform Indipendent Model
PP	Pre-processing PSM Platform Specific Model
RV	Repetitive Viscosity
RBSE	Reuse Based Software Engineering
SPLE	Software Product Line Engineering
XML	eXtensible Markup Language
XMI	Interchangeable eXtensible Markup Language

Contents

List of Figures

Chapter 1

Overview

Software product line engineering (SPLE) is a paradigm for developing a family of software products from the same reusable assets rather than developing individual products from scratch. The driver of SPLE is pre-planned software reuse and within a specific problem area known as a domain.

In many SPLE approaches, a *feature* is often used as the key abstraction to distinguish between the members of the family. Thus, the sets of products in the product line are said to have 'common' features and differ in 'variable' features. Consequently, reusable assets are developed with variation points where variant features may be bound for each of the diverse products.

Variations in usage-context, a contextual setting in which the software product from a product line is deployed or supplied to, have been fuelling demands for adaptable reusable assets to support additional variations that may be required to increase the usage-context of the product line as a result of expanding markets.

Furthermore, *feature binding time* - when a feature is included in a product and made available for use - may vary between the products of a product line because of the different usage-contexts. Hence, variations of feature binding time should also be supported to cater for the diverse usage-contexts.

This chapter presents a general overview of the thesis. The chapter begins from the background theory of the thesis, which is reuse from the perspective of software engineering, and progressed to the focal theory, which is adaptable variations and feature binding time in software product line engineering. The chapter, also, highlights the research motivation; introduces the research challenges; outlines the research objectives; and enumerated the research questions. In addition, the chapter highlights the research contributions and presented an overview of the research methods that bound the research processes. Finally, the chapter presents a brief overview of the remaining chapters.

1.1 Software reuse

In the late 1960s, the committee on NATO Working Conference coined the term 'software crisis' to describe the lack of successes in developing reliable, large, and complex software[Nau68]. Software projects were either overrunning their budgeted costs, were being delivered behind schedule or were not adequately trustworthy. Appalling statistics, attributed to the United State General Accounting Office, were widely circulated as evidence of the 'software crises'. These statistics were later disputed on the premise that the prior studies only analysed failed projects and ignored successful ones [Gla94]. Nonetheless, conservative results estimated that a typical software project was exceeding its budget with an increase of 33-36% and was being completed with a delay of about 22% behind schedule[VG91].

In response to the 'software crisis', new 'software engineering' techniques and methods were developed. In this context, Reuse-Based Software Engineering (RBSE) was conceived as an engineering practice to reduce time-to-market, to reduce production costs, and to improve software quality. Intuitively, instead of developing software from scratch, each time, reusing existing artefacts results in speed gain. In addition, the reuse of existing artefacts reduces the efforts required to develop the new software and thus reduces the production costs. Similarly, the reuse of previously tested and trusted artefacts provides some quality assurance of the new system. Reusable artefacts were increasingly becoming available in different forms: a program library, an entire product or an abstract concept [Som11]. As hypothesized, RBSE recorded successes in reducing time-to-market and improving product quality [AE92, BBM96, FT96, PCH93].

1.2 Domain engineering as a scoped reuse

A pattern began to emerge on factors contributing to the success of software reuse. One of the factors is confining the reuse within a narrow and a special domain[Big98] A domain is a problem area and knowledge on how to develop software products in that area [CE00]. Example of a domain is a mobile telephone software domain and automotive software domain. Successful reuse was reported in a printer firmware domain [Big98] and in the network domain [Nei84]. Software reuse was also demonstrated to be effective in data structure component library [BST+94, CE00].

Consequently, both industry and academia begin to recognize that software reuse is more effective when it is deliberate rather than speculative; scoped rather than open-ended. This is due to the fact that software products in a specialized domain share substantial characteristics and differ in little distinctiveness. Thus, researchers increased focused on domain engineering as a systematic and planned software reuse paradigm[FPF+98,

KKL+98, Nei84]. Reuse within a specialized domain is in contrast to other established reuse based practices. For example, even though object-oriented software engineering (OOSE) targets software reuse and has been largely successful[LHKS92], OOSE approach to reuse is speculative. That is, the objective is to produce designs (both abstract and concrete) that may be reused 'as is' or modifiable to fit in an unforeseeable software system [BBM96, BD95].

1.3 Software Product Line Engineering as a form of systematic reuse

In the 1990s, Software Product Line (SPLE) researches emerged as a genre of systematic and planned software reuse[ABM00, BFK+99a, GKS+96, KKL+98]. SPLE is intended to address the two phases of reuse-driven software engineering systematically: i) engineering for reuse and ii) engineering with reuse. Engineering for reuse, known as *Domain Engineering (DE)* in SPLE, is the process for the analysis, design, and implementation of reusable assets. Engineering with reuse, known as *Application Engineering (AE)* in SPLE, is the process of product-specific analysis, design, and implementation using the reusable assets earlier developed.

Essentially, a domain is analysed, design and implemented in cognisance of common and variable characteristics of the products in the domain. The common and variable characteristics of the domain are often described as product *features* [GKS+96, KKL+98]. Subsequently, reusable assets are developed with pre-defined variation points. Pluggable options and variant assets are also developed to fit into the pre-defined variation points.

In order to derive individualised software product from the reusable assets, a valid combination of desired features are selected, a process known as *product configuration*. In most researches and industrial projects, a product configuration process triggers the selection and activation of specific variants assets for the specific product[ABM00, AK09].

1.4 Motivation

The adaptability of reusable assets is critical to the success of software reuse. Ideally, reusable assets should be adaptable with less effort. The following form the bases of this thesis findings on adaptability of reusable assets: a) an exploration of feature characteristics, in the source-codes, that have impacts on adding variations; b) investigation of the impact of language-based implementation techniques on the modifiability of source-codes to add variations.

Modifiable assets are needed to support additional variations that were not planned beforehand in order to increase the usage context of SPL as a result of expanding markets. For example, Kastner [KAB07] observed that, in Berkeley Database Engine (BDE) product line[Coo17], features such as *Statistics* and *Transactions* were implemented as mandatory. However, such features will have to be made optional to make BDE configurable to other usage-contexts such as smartcard products because the product cannot afford the footprint of the extraneous feature. Therefore, to increase the usage context of BDE, its source-codes must be modified to make *Statistics* and *Transactions* optional. Failure to inject additional variations may lead to the delivery of product with extraneous source-codes - which is not desirable for lean memory applications and may also cause a problem especially if the product is to be integrated within other software product[GAO95].

Furthermore, in SPLE where a *feature* is used as the key design abstraction, *feature binding* is a physical inclusion of the feature in a product configuration and making it available for use. Consequently, *Feature binding time* is when the feature is included in the product configuration and made available for use[ARR+16, CRE08, DFV03, VdH04, LK03, RSAS11]. i.e. when the variable assets are selected and activated for the specific product.

Feature binding time for some features may vary between product configurations because of uncertain market conditions or diverse deployment environments. For example, a binding decision of a *Power Saving* feature of a wireless protocol in a smart device may be made in advance if the deployment environment is known beforehand. While a battery-powered device, deployed in the field, certainly requires the *Power Saving* feature, a device in a residential environment with access to energy would not. However, if the deployment environment is not known until at deployment time, the binding decision about the same *Power Saving* feature would have to be delayed. In some cases, the deployment environment may change after initial deployment (e.g. a device may require additional security settings of a *Security* feature because of a change in the deployment environment). A wireless protocol supplier that covers the different deployment environments faces a challenge of variations of feature binding time. The supplier cannot simply deploy all the features in a device and leave it to the discretion of a final user to activate the ones needed because the device may be overwhelmed, revenues may be lost, and the device may also misbehave due to possible feature interaction[AABZ14, AKS+13, KK98]. Had it been the revenue loss is the only concern, the supplier can deploy all the features and provides license keys to activate the paid features.

In addition to managing variations of products of a product line, supporting variations of feature binding time is equally important. Without supporting variations of feature binding time, a product line assets have to go through an ad-hoc adaptation (often through *clone-and-own*)– which may affect time-to-market, product quality, and production costs [DRB+13, RK12]. – to support the change of binding times. Conventional approaches suggest narrowing and fixing the scope of a domain in order to have more commonality

and less variability or dividing the domain into multiple product lines[LKL02, PBvDL05]; the former constraints product line companies from expanding their products and the latter implies that the company has to maintain parallel assets for the different categories of customers which increase maintenance and production costs.

After circa three decades of SPLE researches, new implementation techniques have emerged to support product line variations. These implementation techniques have different mechanisms for decoupling of implementation modules in the source codes. However, not much is known about modifiability of source codes these techniques are used to implement and update reusable assets.

Similarly, much-needed attention was not adequately given to supporting variations of feature binding time. Most of the current approaches share one more of the following limitations: multiple representations/implementations of the same set of features, each for a specific binding time; are interventions targeted at fine-grained model/ program elements; are at low-level of abstraction; are limited in scope [ARR+16, CRE08, WJE+09].

1.5 About this thesis

This thesis focuses on the technological aspects of capturing and implementing variability in terms of products' features as well as the binding times of those features. Thus, certain processes such as domain scoping, organizational issues, and other managerial frameworks are out of the scope of this thesis.

The overall goal of the thesis is to make contributions that minimize the following challenges:

Ch1. The challenge of adapting software product line assets to accommodate emerging variations that were not planned beforehand. This challenge arises when a new usage context of software product line emerges and the product line company has reusable assets to enter the market but additional variations have to be injected in the existing assets.

Ch2. The challenge of managing variations of binding times to cater for different binding time requirements for the different categories of customers.

1.5.1 Research Objectives

In order to address the research challenges (Ch1 and Ch2) we set out the following objectives:

O1. To explore the properties of features, in the code-asset, that affect modifiability of injecting additional variations.

O2. To evaluate modifiability of language-based implementation techniques when injecting additional variations.

O3. To systematically investigate the current support for a flexible binding time.

O4. To propose and validate an improved approach to support variations of feature binding time and injection of additional variations.

Achieving the first objective (O1) is the first step to addressing the first research challenge (Ch1) and perhaps to triggering of subsequent researches to further understand the challenges and to propose alternatives solutions. Achieving the second objective (O2) has a two-fold benefit: a) to take a cue on the generally missing support, if any, for addressing the first (Ch1), and b) to expose the pros and cons of the relatively new language-based implementation techniques proposed or adapted to support flexible variations.

We set out the third objective (O3) for two purposes: a) to systematically consolidate the identification of research gap, and 2) to position our contributions in the context of the existing body of knowledge in a less partial manner. We set the fourth objective (O4) to simultaneously address Ch1 and Ch2 in a unified approach.

1.5.2 Research Questions

We re-structure the research objective into the following research questions:

RQ1. What are the characteristics of feature in the code-asset that affect the modifiability of injecting additional variations?

RQ2. How flexible are the new language-based implementation techniques on the injection of additional variations?

RQ3. What are the current approaches supporting variations of feature binding time?

RQ4. How can we simultaneously and uniformly improve the support for variations of feature binding time and injection of additional variations?

Answers to the above research questions will determine the extent we achieve the set out objectives.

1.5.3 Contributions

C1. We explored the properties of features at the implementation-level and contributed with the description of the properties that affect the flexibility of reusable assets. We experimented with prominent language-based implementation techniques that are proposed or adapted to support flexible variations; we evaluated their modifiability when injecting of additional variations and contribute with the exposition of their pros and cons. This contribution is the fulfilment of the first and second research objectives (O1 and O2).

C2. We conducted a systematic literature review (SLR) on current approaches proposed to support variations of feature binding time. We contribute to the field by shedding lights in terms of where the proposed interventions are desirable and where they may be limited. This contribution is the fulfilment of the third research objective (O3).

C3. We developed binding time aware modelling language, in a model-driven approach, to abstract-away the actual binding mechanisms used at the implementation level and, thus, raised the level of abstraction of binding time management. This contribution is a partial fulfilment of the fourth research objective (O4).

C4. We contribute with toolsets to encode binding time and check consistency at architecture model level. The toolsets have embedded capability to decide a model element, representing an executable artefact, to instantiate based on binding time decision. This contribution is also a partial fulfilment of the fourth research objective (O4).

1.6 Research methods

Adrion cited by Glass [Gla94] proposed four methods of software engineering researches. Two of the proposed methods are a) Empirical method and b) Engineering method. In the Empirical method, researchers do "*propose a model, develop statistical or other methods, apply to case studies, measure and analyse, validate the model, and repeat*". In the Engineering method, researchers do "*observe existing solutions, propose better solutions, build or develop measure and analyse, and repeat until no further improvements are possible*". Observe that each of the above methods consists of a series of research activities. To form a research lifecycle, the series of research activities are executed in a particular order and within a defined research boundary[Fli09].

In this thesis, we executed the research activities in two separate research lifecycles. The second and the third boxes of Fig.1.1, labelled as *Research life cycle 1* and *Research life cycle 2* respectively, highlight the research methods of this thesis. The first research life cycle is action research in which an open-source version of Oracle Berkeley Database

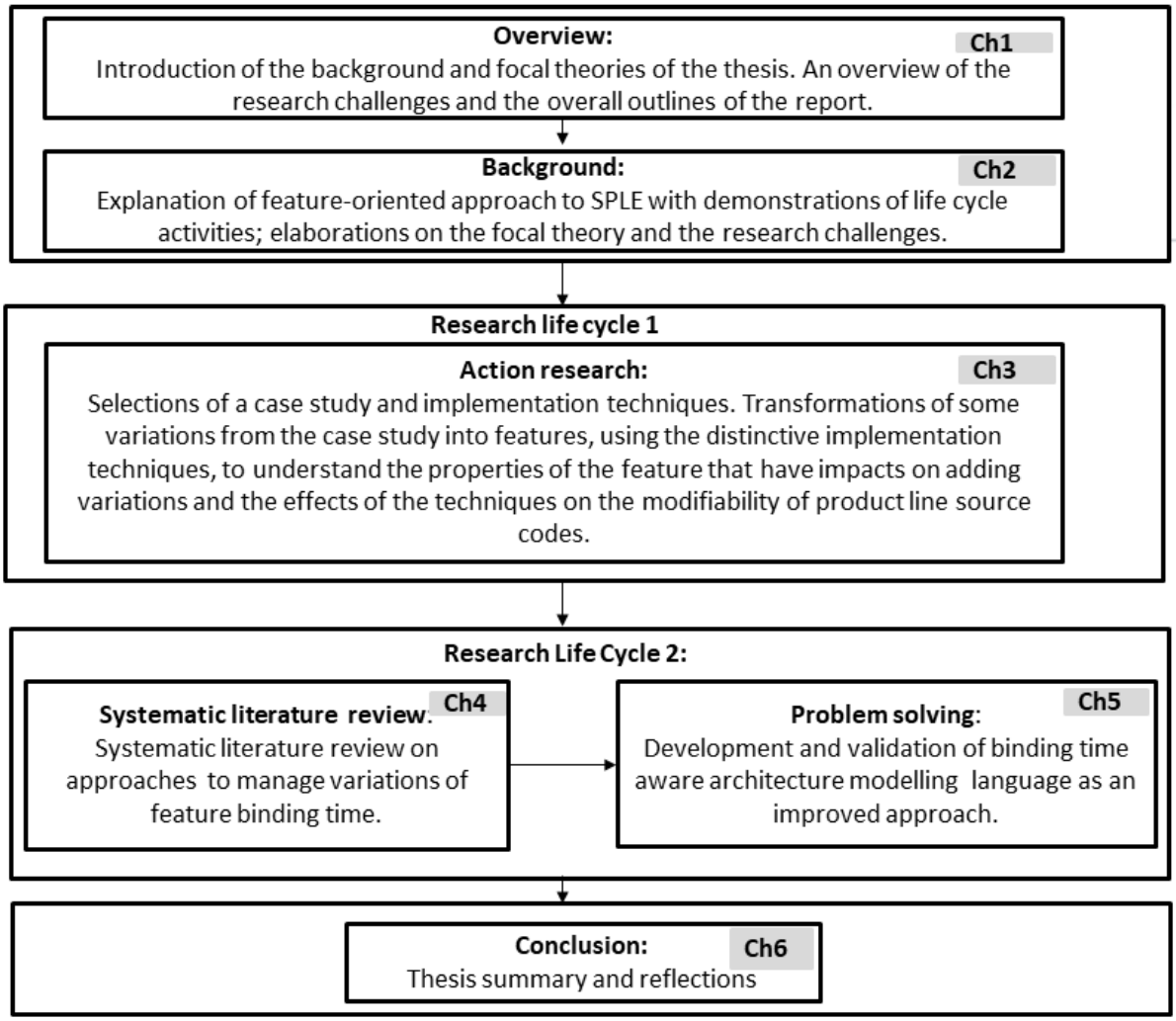


Figure 1.1: High-level outline of the thesis

Engine (BDE) was selected as a case study. Three, distinctive, modern language-based and one classical tool-based implementation techniques were also selected. We transformed some of the variations, from the case study, into features using the different techniques to understand the properties of a feature that have impacts on adding variations and effects of the techniques on the modifiability of product line source codes. The details of the lifecycle activities and the results are presented in *Chapter three* (indicated with the Ch3 in the figure).

The second research lifecycle is based on an engineering research method and comprises the two research activities: (i) A systematic review of the current approaches to managing variations of feature binding time the details of which are presented in *Chapter four* (Ch4)). (ii) Development and validation of binding time aware architecture modelling language as an improved approach. The details are presented in *Chapter five* (Ch5).

Also presented in Fig.1.1 are the outlines of the thesis's chapters. The topmost box contains the outline of the current chapter, *Chapter one*, and that of *Chapter two*. *Chapter two* elaborates on the concepts introduced in this chapter. Central to these concepts is the feature-oriented approach to SPLE. Thus, the chapter illustrates the feature-oriented approach to the lifecycle activities of both domain engineering and application engineering. The chapter also elaborates on the concepts of adaptable variations, feature binding time in software product line engineering, and the research challenges associated with both. *Chapter three* presents the details of the first research lifecycle. *Chapter four* presents the systematic literature review on approaches of supporting variations of binding time. *Chapter five* presents the detail design and validation of our approach to supporting variations of feature binding time. In *Chapter six* we present the summary of the thesis, revisit the contributions and discussed limitation and future research directions.

1.7 Chapter summary

This chapter provides a general overview of the thesis. The chapter highlights the background theory of the thesis, which is software reuse from the perspective of software engineering. The chapter, also, highlights the focal theory of the thesis, which is adaptable variations and feature binding time in software product line engineering. Further, the chapter highlights the research motivation, the research objectives, and the formulated research questions. Finally, the chapter outlines the research contributions and presents an overview of the research methods that bound the research processes.

Chapter 2

Background

This chapter explains the background concepts of the feature-oriented software product line. The chapter discusses the feature-oriented approach to capturing and implementing variations in SPLE by illustrating the lifecycle activities of domain engineering and application engineering with concrete examples. The chapter also explains the concept of feature binding time and its possible variations that may arise between different products. Lastly, we elaborate on the research challenges introduced earlier in *Chapter one*

2.1 Feature-oriented Software Product Line (SPLE)

Throughout the history of software engineering, X-oriented may be used to describe an engineering activity if X is the dominant technique, concept or an abstraction used throughout the activity.

In feature-oriented SPLE, a *feature* is used as the key design abstraction to differentiate between the products of the same family. The motivation behind feature-orientation is because a feature is recognized by different stakeholders. For example, customers recognized a feature as a service or capability of a product that satisfies their need. Requirement engineers can use a feature to describe functions that need to be developed. Developers can also use a feature to describe a unit of functionality that needs to be designed, developed, tested and maintained. Marketers can use a feature to promote a product to potential customers[LKL02].

As introduced in *Chapter one*, software product line approach to reuse is by means of two engineering phases: (i) domain engineering (DE) and, (ii) application engineering (AE). At the domain engineering phase, reusable assets are developed based on the analysis of common and variable characteristics of the products in the domain. At the application engineering phase, individual products are developed from the reusable assets.

In feature-oriented approach, the common and variable characteristics of the products are expressed in terms of features. Fig.2.1 summarizes the relationship between

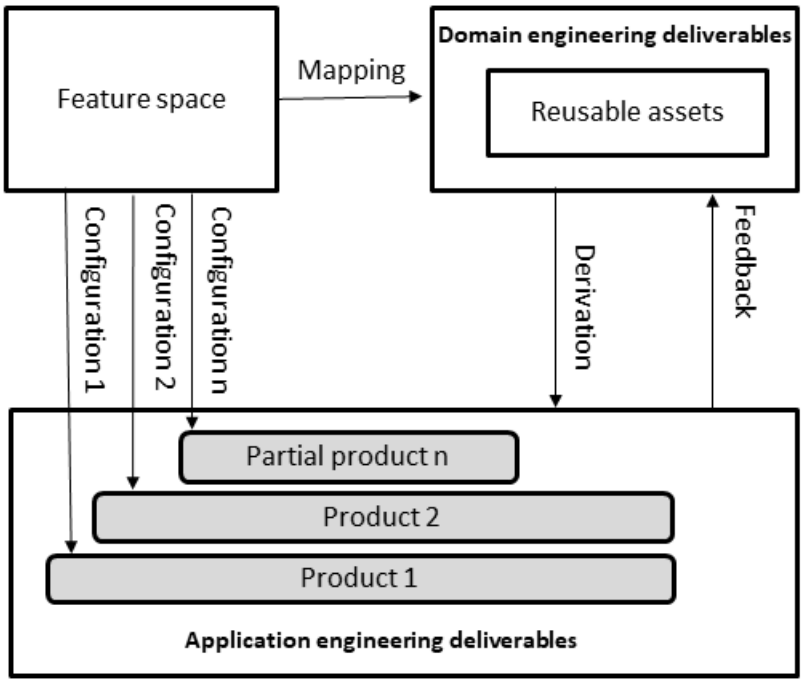


Figure 2.1: Relationship between feature space (top right) on one hand and the deliverables of both domain engineering (top left) and application engineering (bottom) on the other hand

feature space an the outputs of both domain engineering and the application engineering activities. As shown in Fig.2.1, features in the feature space are mapped to the reusable assets (deliverable of the domain engineering activities). Similarly, each product or partial product, as the case may be, is derived from the reusable asset based on feature configurations from the feature space (i.e. valid selection of features). Lastly, product-specific artefact may be fed-back into the reusable assets in the form of feedback.

In the next section, we discuss feature-oriented approach to domain engineering in more details. We also discuss feature-oriented approach to application engineering in *Feature-oriented application engineering* section.

2.2 Feature-oriented domain engineering

The term 'domain engineering' is attributed to neighbour's work of engineering reusable assets for a group of similar software products in a particular problem domain as oppose to developing individual products from scratch[Nei84]. Thus, domain engineering is also known as *family engineering* as the group of the products can be considered as members of the same family. In some approaches [BFK+99b], domain engineering is also referred to as *infrastructure construction phase* because the activities at the phase are

geared toward creating an infrastructure from which individual products in the domain can be created. *Infrastructure usage phase* (aka application engineering) is used to describe the phase in which the individual products are created.

Domain engineering in SPLE may have a narrower focus than in traditional domain engineering due to considerations of current market opportunities of the potential products. For example, smart elderly care product line, with focus on elderly people living in care homes, may be created from a smart living domain - the generic domain of interconnected smart devices. Similarly, a domain in SPLE often combines (part of) several horizontal technical domains into a single vertical application domain. For example, a banking information system product line is a vertical application domain that combines other technical horizontal domains such as database management systems (DBMS), network protocol, and user interface (UI).

The key activity in domain engineering, and which covers all the lifecycles, is advance planning for the commonality and variability (C & V) of the potential products in the chosen domain. Lifecycle activities at the domain engineering phase include: domain analysis, domain design, and domain implementation. Fig.2.2 highlights the key engineering activities in feature-oriented SPLE. The high-level activities which also constitute other sub-activities are domain analysis, domain design, and domain implementation.

Domain analysis activity is a systematic identification and organization of product features and the outputs from this activity are the analysis models. Domain design activity is the construction of the product line architecture that accommodates the diversities of the products in the domain and the outputs are various domain design models. Domain implementations are the conversion of the product line architecture into executable and the outputs are the implementation artefacts. The outputs are deposited into repository of reusable assets.

We present the details of the sub-activities of the three high-level activities and in the subsequent sections.

2.2.1 Feature-oriented domain analysis

Kang *et al* [KCH+90] introduced the idea of eliciting, designing, and implementing the common and variable characteristics of products in terms of "product feature". The product features may be identified from four viewpoints as follows [LKL02]:

- **Product capability**: A feature from the viewpoint of product capability describes a service provided by the product, a means to operate the product or how the information is presented to the product user. It may also be a representation of a certain quality attribute.

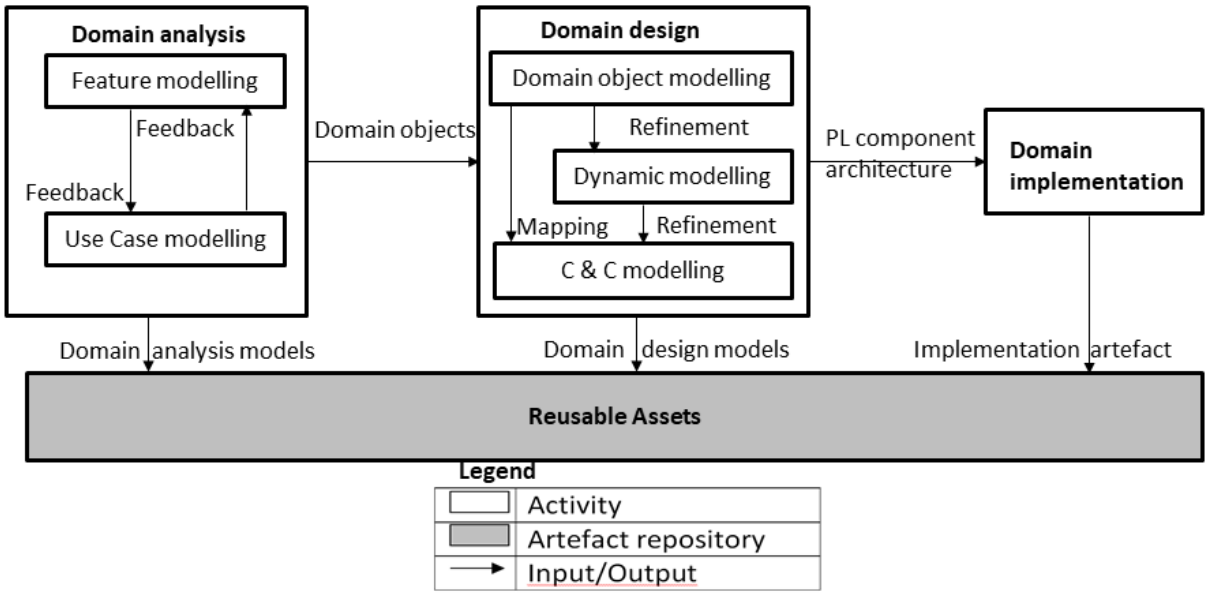


Figure 2.2: key engineering activities in feature-oriented SPLE

- **Product operating environment**: A Feature from this viewpoint represents an environmental constraint in which the product is deployed or used.
- **Domain technology**: A Feature from this viewpoint represents a way of implementing services or operations that are specific to the domain.
- **Implementation technique**: Features from the viewpoint of implementation technique are the available implementation decisions.

To analyse a domain, a good starting point is to use a set of exemplar products in the domain. In some domains, a group of products portfolio, such as basic and advance, can be identified. For example, a manual car without air-condition and sunroof may be consider a basic while the advance one will have full options (automatic, sunroof, air conditioned, cruise control, etc.).

Beginning from the small set of products, common and variable features should be identified. However, feature identification may not be completed by analysing small number of products in the domain. The followings sources should also be consulted:

- **Domain terminologies**: In a matured and standard domain, terminologies are already understood by the domain experts. Examples of domain terminologies are navigation in avionics domain, method of transfer in banking domain, cumulative grade point average (CGPA) in the information system of higher institutions of learning. In an immature domain, domain terminologies need to be clarified and standardized to have consistent meaning to all stakeholders.

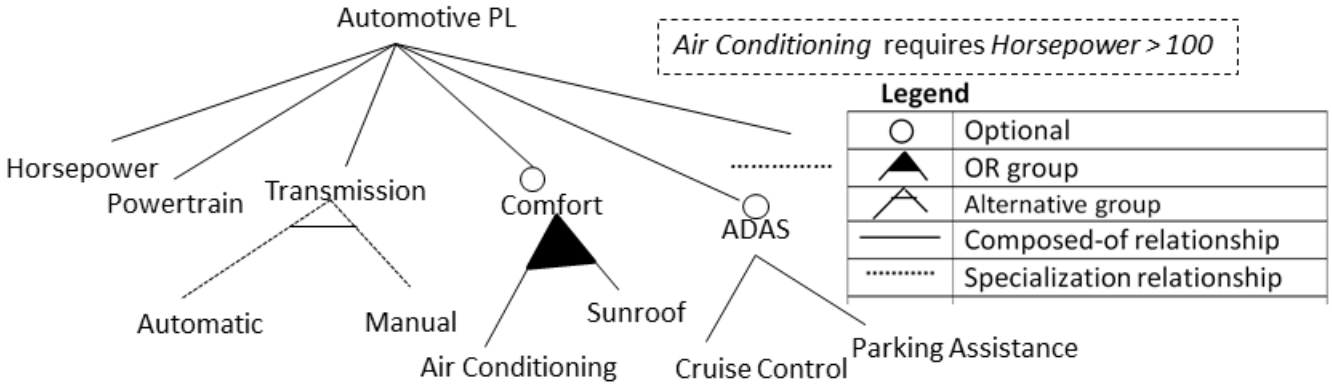


Figure 2.3: A partial feature model of Automative product line

- **Stakeholders**: Stakeholders such as domain expert, end users, product marketers, system designers are potential sources of information about products' features that need to be developed.
- **Existing documentation**: documentation such as user manual, design documents, and existing laws in the domain are also useful for feature identification.

Domain engineers should create a domain dictionary. The domain engineers should also define, precisely, each of the identified features, and the definitions should be added to the domain dictionary. Identified features should then be organized in a feature model. We discuss feature modelling in the next section.

2.2.1.1 Feature model

A Feature Model (FM) has been widely used to organize commonality and variability (C &V) information of a product line[MPH+07]. A feature model is a graphical tree structure in which product features of a product line are organized with their types and their relationships. Feature type is one of the following broad categories:

- **a mandatory type**: a feature of this type is a common feature that is manifested in all the products of a product line.
- **a variable type:** a feature of this type is either an optional, is part of an alternative group (only one feature in the group can be selected) or is part of inclusive OR group feature (more than one feature can be selected from the group). Unlike the alternative group, the OR group features are not mutually exclusive and can either contain all optional features or at least one of the features must be selected.

Fig.2.3 depicts a hypothetical feature model of an automotive product line adapted from Kang *et al* [KCH+90]. In the feature model, *Powertrain* and *Transmission* features are mandatory because they must be available in every car. *Comfort* and *Advanced Driver Assistance (ADAS)* are optional features because they do not have to be fit in every vehicle. *Automatic* and *Manual* are alternative features since the two features cannot co-exist in the same car. *Air Conditioning* and *Sunroof* are OR group features because a car can have zero or more of the features.

Relationships between features can be hierarchical or horizontal (aka cross-tree constraints). The major hierarchical relationship is a composed-of relationship. The feature on top of the composed-of hierarchy, the parent feature, represents an abstraction over the constituent features (children-features) beneath the hierarchy (e.g. *Transmission* in Fig.2.3). Features in a composed-of relationship may also exhibit a generalization/specialization association. In this case, the main feature is the generic feature that represents a suppression of detail differences of the children's features. Each child feature is a specialization of the generalized feature. For example, in Fig.2.3, *Manual* and *Automatic* are specialization of *Transmission* feature.

In contrast to the hierarchical relationship, horizontal relationships are usually presented separately from the feature model as a series of dependency rules. These are also called composition rules because the rules are specifications of how features should be composed. For example, in Fig.2.3, *Air Conditioning requires Horsepower >100* represents a composition rule. In other words, a dependency constraint from *Air conditioning* to *Horsepower*.

A selection of a valid combination of features is known as as *product configuration*. For example, the following is a valid configuration of features from Fig.2.3: *Horsepower 150, Powertrain, Automatic, Air conditioning*.

Features organized in the feature model may be validated with a use case model. Beginning from the feature modelling and then the use case modelling is recommended when the domain is familiar and significant expertise exists. In an unfamiliar domain, it might be more effective to start with use case modelling before feature modelling.

We discuss use case modelling in the next section.

2.2.1.2 Use case model

A use case model describes a software product from the user's perspective. In feature-oriented domain analysis, a use case model can be used to support the identification of variations in the features of a product line. Thus, variant model elements such as actors, systems, and use cases should be identified by exploring user-level semantics of the

collective products. Occasionally, the product line has to be explored at a sub-function level to clarify variation.

To validate the identified features using the use case model, the domain engineers should do the following checks:

- Check if each use case can be mapped to a feature. If not, there may be a feature that is missing from the feature model.
- Check if all the features are addressed by the use cases. If not, there may be use cases that are missing.
- Check if variations of use cases and variations of features are consistent.
- Check if the use cases are consistent with the semantics of the features defined in the dictionary.

.

In the next section, we demonstrate an example of domain analysis with the feature and use case models.

2.2.2 Domain analysis example

To present a concrete example of the domain analysis, we introduce Education Software Product Line (EduPL), an enterprise product line for universities initiated by a partner university from a developing country. In that country, in which EduPL was conceived, a common regulatory body dictates most of the critical operations of the universities. Hence, the product line approach is expected to be beneficial.

We begin with an example of feature modelling of EduPL in the next section.

2.2.2.1 Feature modelling example

Fig.2.4 shows a partial feature model of EduPL. As shown in the figure, we identified the following features as some of the services of the products: *User Account, User Notification, Admission, Registration, and Result Computation*. The *User Account* feature provides services for maintaining an account with the institution's portal when the product is deployed. Since validating the user while creating an account is available only for some products, *User Validation* is an optional child feature of *User Account*.

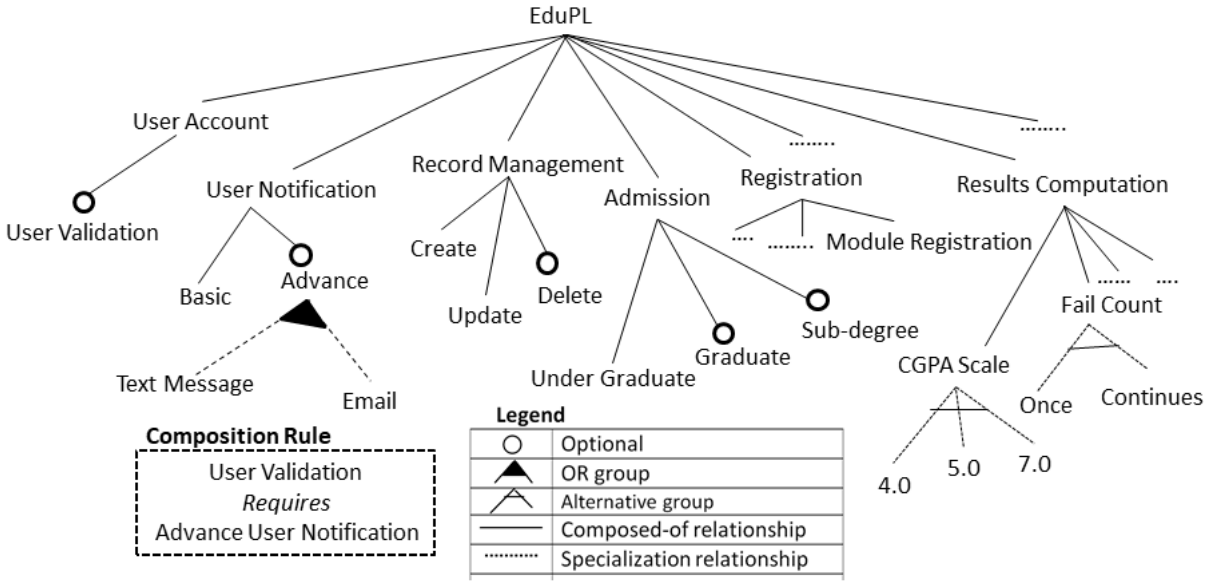


Figure 2.4: A partial feature model of Education Software Product Line (EduPL), an enterprise product line for universities

User Notification is a service feature and is decomposed into *Basic* and *Advance*. *Basic* provides display services of status messages on a user's screen. *Advance* provides services for sending messages to a user via email or telephone or both. Hence, *Text Message* and *Email* are modelled as OR group features. *Email* and *SMS* are two methods of implementing user notification - hence they were identified from implementation technology perspective.

Admission feature provides services for application into various programmes of an institution and it is decomposed into three children: (1) *Under Graduate* - provides services for admission into under graduate programmes; (2) *Graduate* - provides services for admission into post-graduate programmes; *Sub-degree* - provides services for admission into sub-degree programmes.

Registration feature provides services for enrolling accepted applicants and thereby changing their status from applicants to students. One of the children features of *Registration* feature is *Module Registration* which provides services for students to select modules and appropriate staff to validate the selections. The registered modules would later be retrieved and assigned marks, after several assessments and semester examination.

Result Computation feature provides service for evaluating students' academic performance based on the marks they score from the registered modules. One of the performance indicators is cumulative grade point average (CGPA) which is used to determine students current standing using aggregate score of all examined modules do date. *CGPA Scale* and *Fail Count* are some of the children feature of *Result Computation* and they

represent features from domain methods.

Different institutions adopt varying scale for computing CGPA; hence *CGPA Scale* represents an alternative feature group. *Fail Count* represents how a repeatedly failed module would affect student's CGPA. With the *Once* child feature of *Fail Count*, the weight of a failed module is added to the denominator part of computing the average only once, no matter how many time a student failed the module, hence less impact on student CGPA. With the *Continuous* count, weight for the repeatedly failed module would continue to be added to the denominator part of computing the average until the failed module is passed in the subsequent attempt, hence negatively affects CGPA of students who repeatedly failed modules.

Record Management is a feature identified from the perspective of products operation and represents one of the critical operations in EduPL. The feature is decomposed into *Create, Update and Delete* for record creation, record updates and records removal operations respectively. *Delete* is an optional child feature of *Record Management* because some institutions have a policy to not remove a stored record. However, that policy may change with an increasing concern about users' lack of control on their private data. For instance, European General Data Protection Regulation (GDPR) [Uni18] mandated that data controllers must remove a private data of their subject within a specified duration upon request.

It may be counter-intuitive as to why *Record management* is a root feature. One may ask why not as a child feature of *User Account, Application, and Registration* features since each of the mentioned features requires records to be managed. However, it is important to note that features in a feature model should not be organized as functional dependencies or call hierarchies; features should be organized so that commonalities and variabilities can be recognized easily. The functional dependencies should be analysed later during domain design. Thus, feature that is related to many other features should also be moved to the root.

As with a single product development, sometimes it is useful to decompose the overall product line into subsystems for further analysis and design. The next paragraph presents the high-level subsystems of EduPL.

EduPL Subsystem Overview

Fig.2.5 presents the package diagram of EduPL subsystems. Note, however, without domain knowledge it may not be possible to break the product line into subsystems without further exploration. In Fig.2.5, *Account Management* subsystem encapsulates *User Account, User Notification, and Record Management* aspect of the *User Account* feature. The *Admission Management* subsystem encapsulates *Admission* feature and its related record operations. *Admission Management* is dependent upon *Account Management* because having an account is a pre-requisite to starting an application for admission. Similarly, *Enrolment Management* depends on *Account Management* and *Admission Management*.

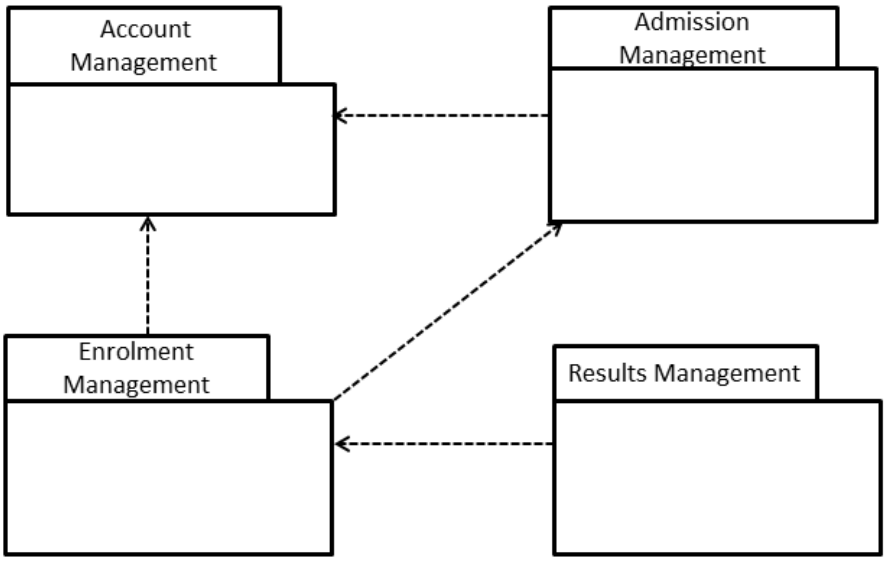


Figure 2.5: subsystem overview of EduPL(an enterprise product line for universities)

Lastly, *Results Management* depends on *Enrolment Management*.

in the subsequent illustrations, we focus on the User *Account Management* subsystem for brevity. In the next section, we present an exploration of *Account Management* with use case to validate the identified features from the feature model of Fig.2.4.

2.2.2.2 Use case modelling example

Fig. 2.6. depicts the use case model of the *Account Management* subsystem. Gomaa *et al* [Gom05] illustrate a variant model element of a use case model with UML stereotype. However, to avoid cluttering the model, we represent a variant model element with dotted outlines.

In the model of Fig.2.6, the use case *Create Account* represents an interaction between a new user and the system (the software product) to create an account and the use case is mandatory. While creating an account, the system can optionally validate the new user. Thus, *Validate Account* use case optionally extend *Create Account* use case. Similarly, *Validate Account* use case uses *Send Notification* use case to complete its role. *Send Notification* is an optional use case and represents sending notification via text message or email. *Display Status* is a mandatory use case and it represents basic user notification. The *Update Account* is a mandatory use case and corresponds to the update operation of an account. Lastly, *Delete Account* is an optional use case that corresponds to the optional delete operation of an account.

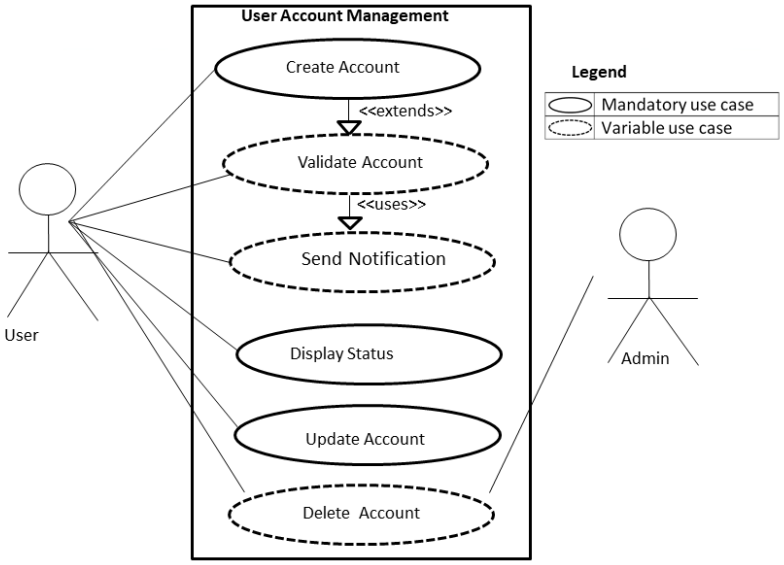


Figure 2.6: A use case model for *Account Management* subsystem of EduPL

Table 2.1: Feature to use case mapping

Feature	**Use Case**
Create	Create Account
Validate User	Validate Account
Advance Notification	Send Notification
Update	Update Account
Basic Notification	Display Status
Delete	Display Status

Table 2.1 presents the mapping from features to use cases. In the table, *Create, Update, and Delete* features are mapped to *Create Account, Update Account, and Delete Account* use cases respectively. *User Validation* feature is mapped to *Validate Account* use case, *Advance Notification* feature is mapped to *Send Notification* use case. Lastly, *Basic Notification* is mapped to *Display Status* use case.

When variations are fully captured and (partially) validated with the use case, we transition into domain design.

2.2.3 Domain design

Domain design is the activity of crafting the domain architecture, or more specifically, a product line architecture (PLA). Architecture is a "*principle design decision about a software product*" [TMD10]. A PLA, therefore, is an embodiment of "principal design decisions" about all potential products of a product line. As with the architecture of a single software product [Ran98, Kru95], PLA may have several representations and each of the representations should accommodate diversities of the products.

In feature-oriented domain design, PLA model can be obtained through gradual refinements of domain objects that should be identified from the products' features. In the refinement process, we construct other models to understand both the static and dynamic structure of the domain. We also consider the dominant architectural style of the intending products. In this thesis, we refer to the component and architecture model as the terminal PLA while the various models leading the terminal PLA as intermediary design models. In the subsequent paragraphs, we discuss the refinement process and the required models. We begin with a model that is formed by the objects in the domain.

A domain object model (DOM) represents the domain structure in terms of objects and their relationships. To design DOM, we identify candidate objects from the feature model. Not every feature can be mapped to a domain object though. Some features manifest as operations on domain objects or parameters on a generic object and therefore are not candidate objects themselves.

After identifying candidate domain objects, we then construct the domain object model by adding the semantic relationship between the objects. A relationship between features can also be directly translated into the relationship between objects. The domain object model should then be refined by exploring detail interaction between the domain objects.

Interactions between the domain objects must be explored with one or more dynamic models. The level of detail exploration varies from domain to domain. It may be sufficient to explore the flow of execution of operations to identify key components and

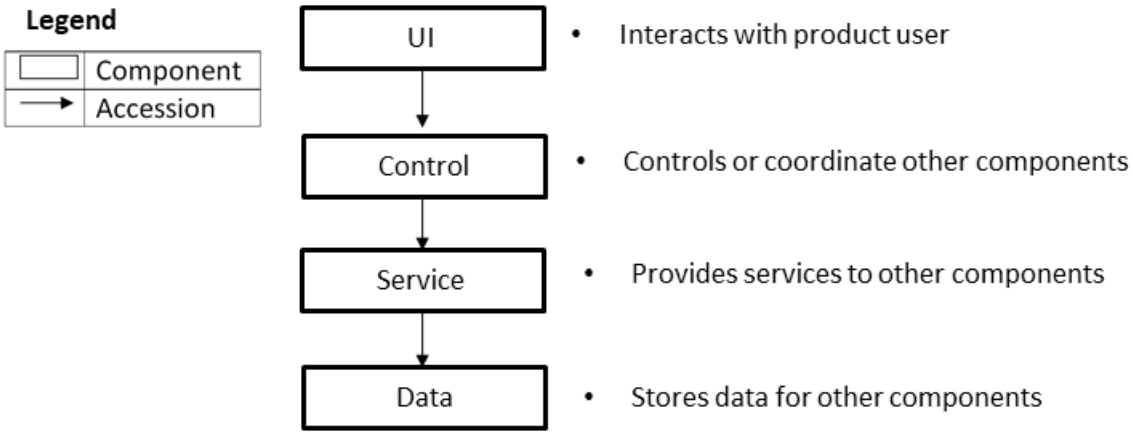


Figure 2.7: roles for layered components in N-tier architecture pattern

their connections. In that case, the fine-grained details are further explored by developers. In the cases of fluid domains, such as predominantly state-based, details interaction could be explored in great detail. So, also, in domains that are highly sensitive to business rules such as insurance.

Part of domain design is consideration of architectural pattern and style of the intending software products. An architectural pattern is a generic solution to a reoccurring architectural problem that can be adapted to suit many contexts. For example, multi-tier architecture is a pattern to solve the problem of logical separation between data, computation, and control. The architectural style is a class of architecture that is found repeatedly in practice and often dictates the topological arrangements of architectural components. However, the two (architectural style and architectural patterns) are not strictly different[TMD10]. Generally, patterns and styles facilitate some quality attributes such as maintainability and ease of evolution. An architectural pattern or style may have a significant influence on how domain objects interact and how they are refined into components and connectors. Therefore, we should decide on the architectural pattern or style we wish to enforce. For example, when Adopting an N-tier architecture pattern, to separate data, computation, and control, we may assign roles to layered components as indicated in Fig.2.7.

In Fig.2.7, *UI* is a component that interacts with user to convey system status and forward user command/request to *Control* component. *Control* is a component that coordinates or control steps of operations or other components. *Service* is a component that provides services to other components by performing some computation, processing or transformation. The domain objects should then be refined into components and connectors based on the dynamic exploration and the adopted architectural pattern.

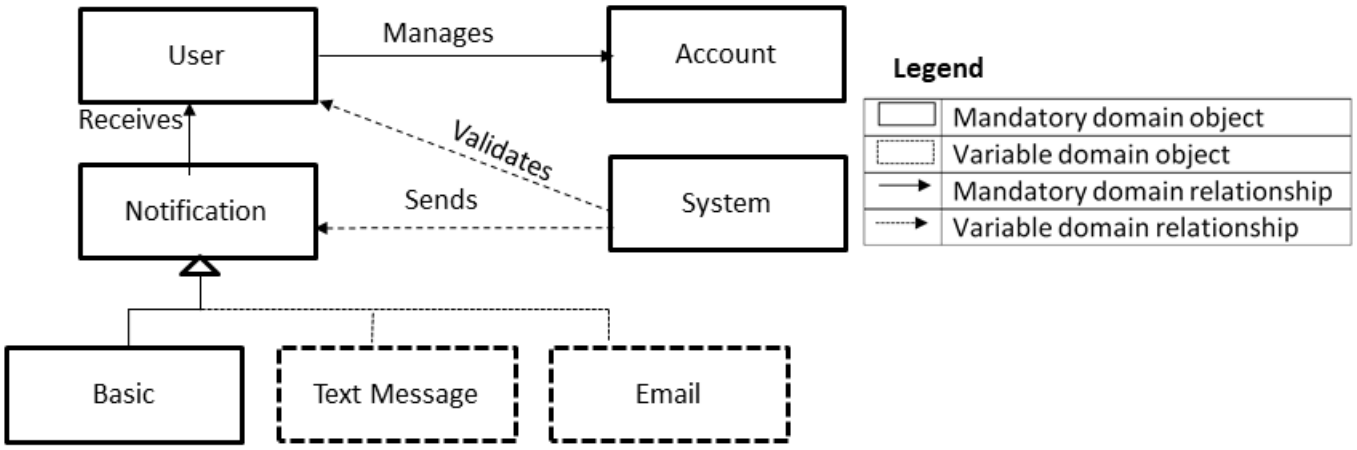


Figure 2.8: a domain object model (DOM) for *Account Management* subsystem

In the next section, we illustrate an example of PLA model that is derived from the gradual refinements domain objects.

2.2.4 Domain design example

This section illustrates an example of domain object model and its gradual refinements to component and connector architecture model. Along the line, we briefly illustrate the role of architecture style and domain exploration with a dynamic model.

Domain object model

Fig.2.8 depicts the domain object model of User *Account Management* subsystem. The figure shows the following domain objects as identified from the product features: *User, Account, Notification, System, Basic (Notification), Text Message (Notification),* and *Email (Notification).* In the same Fig.2.8, the following are the relationships between objects: A *User* 'manages' *Account*; *System* 'validates' *User* and also 'sends' *Notification* to *User; User* 'receives' *Notification.* The 'manages' relationship is a combination of 'creates', 'updates' and 'deletes' relationships that are between the *User* and the *Account* objects. In addition, the Generalization/ Specialization relationship between *Notification* and its children features in the feature model of Fig.2.4 are translated into Generalization/ Specialization between *Notification* object and its sub-types.

Next, we explore the flow of operations between the domain objects using an activity diagram.

Activity diagram

Activity diagram is one of the dynamic models that can be used to explore concise domain

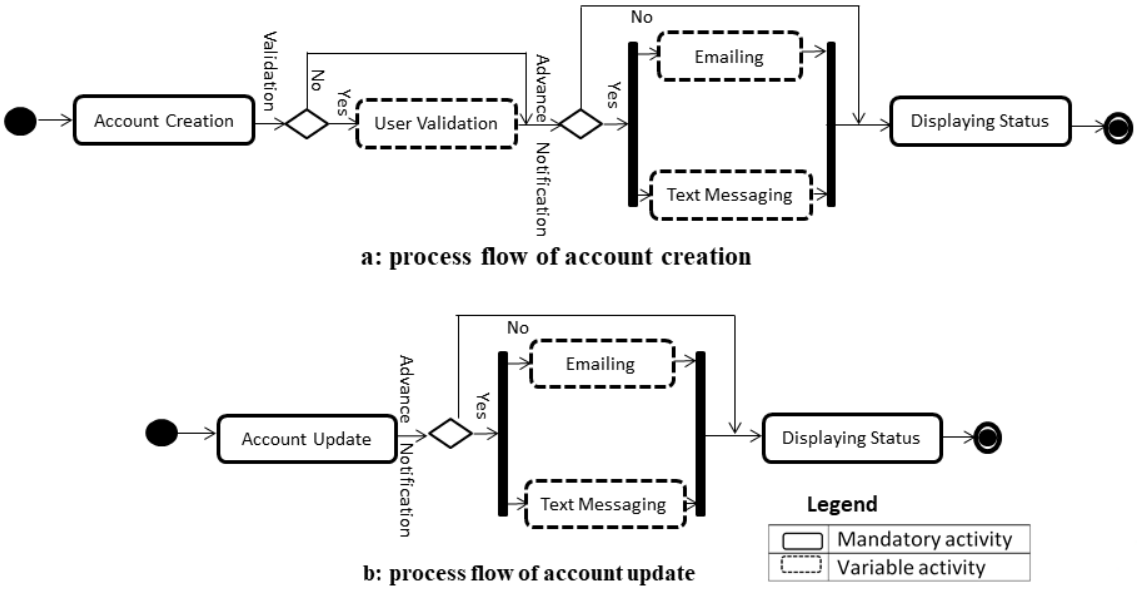


Figure 2.9: Activity diagram showing (a) process flow for account creation and (b) process flow for account update

operations by showing the flow of executions from one activity to another. In domain engineering, it suffices to explore the domain only at a high-level and not the detail messages transfer between activities. One of the goals is to identify key operations that must be realized by the intending components and connectors. Another goal is to identify variations of activities between products of a product line.

Fig.2.9 illustrates the key operations in *Account Management* subsystem with a UML activity diagram. In the figure, unlike in UML, we use dotted outlines to depict a variable activity and the decision branch to depict a variation point. Fig.2.9a shows the flow of activities in the process of account creation while that of Fig.2.9b shows the flow of activities in the process of account update. The flow of activities in the process of account deletion (not shown in the figure) follows a similar pattern with that of Fig.2.9b.

Subsequently, we refined the domain objects into components and connectors based on the dynamic flow of the activity diagram of Fig.2.9 and the N-tier architecture pattern of Fig.2.7.

Components and connectors PLA model

Fig 2.10 depicts component and connector model of the *Account Management* subsystem. We use UML notations of component and connector but with slight modifications. We used dotted outlines to represent the variants architectural elements rather than the usual stereotype. However, we retained the usual ball to denote providing connection (provided interface) and the socket to denote requiring connection (require interface).

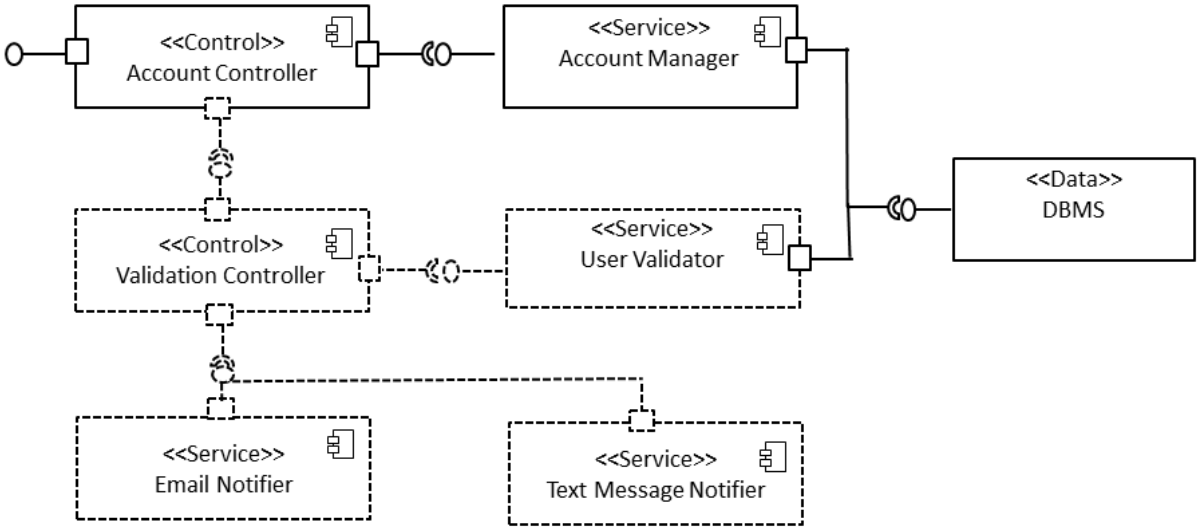


Figure 2.10: components and connectors PLA model that is derived from the gradual refinements of DOM. In the figure, dotted outlines represent variable architectural elements

Objects that are playing a system role in the domain object model should be analysed further. If the role is global, the object should be refined into the overall product control component. For example, a component that coordinates the overall operations of other components in a mobile telephone domain may be refined into *Phone Controller* component. Similarly, the component coordinating the overall *Account Management* subsystem may be refined into *Account Controller* component. However, if the role is local, it should be refined into a control component that is controlling the local activities. For example, in the DOM of Fig.2.7, *System* plays the local role of coordinating user validation, hence we refined it into *Validation Controller* component to coordinate *User Validator*, *Email*, and *Text Message* service components.

We refined the domain objects, based on the N-tier architecture and the activity diagram, into two control components, one data component and four service components. The control components are*Account Controller* and *Validation Controller*. The data component is the *DBMS*. finally the service components are *Account Manager*, *User Validator*, *Email*, and *Text Message* (Fig.2.10).

From the top-left to top-right of Fig.2.10, the *Account Controller* Control component receives a request to manage an account and forward the same to its required connection to *Account Manager* service component. In addition, *Account Controller* optionally requires connection to *Validation Controller*. The top-right component, *Account Manager* service component, manages the account related records in the persistent storage. Hence, it requires a connection to the DBMS component.

From the middle of Fig.2.10, *Validation Controller* is connected to the conditional

required end of the *Account Controller*. That is, *Validation Controller* receives request to validate user, from *Account Controller*, and forward the same to its required connection to *User Validator* service component. In addition, the component also required connection to components providing notification services(i.e *Email* and *Text Message* service components). The *User Validator* component generates token to be sent to the user to be validated and record the same token in the persistent storage. Hence, it also requires connection to *DBMS*.

Before implementing the domain, further exploration is still needed for the nature of communications between components. Also, internal components implementation should be explored by the developers. Preferably, each implementation details of a component should remain local to it and not cross over to other components.

When domain design is completed. The next step is domain implementation. We discuss popular approaches to domain implementation in the next section.

2.2.5 Domain Implementation

Domain implementation is the activity of converting the domain design into executables. The domain may be implemented beforehand as part of the domain engineering, or maybe deferred until a product, from the product line, is requested. In the latter case, the domain is implemented in concurrence with application engineering.

In this section, we discuss the implementation as part of domain engineering. We also discuss the domain implementation in concurrence with the application engineering in *Feature-oriented application engineering* section.

At the implementation level, a feature manifests as one of the following:

a. one or more major units of implementation (e.g. modules, components, classes) exclusive for the realization of the feature;

b. many fragmented minor units of implementation (e.g. functions, attributes, clauses) scattered in other major units;

c. a combination of (a) and (b).

Kästner *et al* [KAK08], in an exploratory study, refer to the manifestations of feature in the implementation artefacts as granularity of variations. In summary, a feature may be traced to many parts of the implementation artefacts and the parts may be coarse-grained or fine-grained implementation units[KAK08]. Conversely, an implementation artefact can (in parts) implement more than one feature.

There are different, non-mutually exclusive, techniques to implement the domain design. Below are four of the most popular techniques:

Annotative techniques

In the annotative approach, source-codes fragments for variable features are annotated to be conditionally compiled. That is, source-codes of variable features are only compiled if the feature is included in the product configuration. Consequently, the technique is also called conditional compilation. A simple implementation of a conditional compilation is to have an additional facility, in the development environment, that pre-processes the source-codes before compilation and comments-out the part of which the corresponding variable features are not in the product configuration. The approach is also called pre-processing because the annotated source-codes fragments are pre-processed before compilation.

Listing 2.1 shows an example of annotations (pre-processor directives in this case) that are used to mark features. In the source-codes snippet, the //#ifdef ....//#endif in line 7-9 annotated code fragment of the *User Validation* feature. If the feature is not included in a given product configuration, the annotated source-codes will not be compiled. Likewise //#ifdef ....//#endif in line 11-15 annotated source-codes fragments of *Delete* feature. The portion of the annotated source-codes would be commented out if the *Delete* feature is not in the product configuration.

Annotative approaches have been heavily criticized for source-codes obfuscation, difficult to understand and maintain the source-codes especially when the #ifdefs are heavily nested[MP02, ZBP+13, LAL+10]. The nesting of #ifdefs is necessary when a parent feature is optional and also some of its child features are optional. It is also difficult to reason about individual features because the annotations may span several places in several physical files. Thus, the annotative technique lacks traceability between a feature and its source-codes implementation. To remedy some of its shortcomings, Kästner *et al*[KA09] proposed an approach to simulate feature traceability in the pre-processing technique, through assigning of unique background colour to source codes of the same feature [KA09]. In the approach, a developer can also view or query program source codes of a particular feature of interest and thereby improving modularity.

Annotative approaches can be used to implement a feature of any level of granularity because the technique is insensitive to the syntax of the target programming language. Pre-processing of annotations has been popular with the developers [LAL+10, HZS+16] and has been in existence even before the emergence of SPLE researches.

Listing 2.1: Examples of pre-processor directives used to implement variations

```
1 ..............
2 public class AccountController {
3 ........................
```

```
4 public String createAccount (....){
5 //forward request to Account Manager Service
6 // component to actually create the account
7 #ifdef User Validation
8 //forward request to Validation Controller component
9 #endif
10 }
11 #ifdef Delete
12 public String DeleteAccount (....){
12 //forward request to Account Manager Service
13 // component to actually remove the account
14 }
15 #endif
16 ............
17 }
```

Compositional techniques
In the compositional technique, a domain is implemented as combinable implementation units. Approaches in this class of technique can be language-based or tool-based. In the language-based category, the technique depends on programming language support. Various strategies, such as interface-implementation separation, are used to decouple implementation units. Example of language-based composition approaches includes component/service composition, plugin composition, and independent module linking. In the tool-based category, the technique does not depend on programming language support but rather a tooling facility that is external to the programming language used. Examples of a tool based composition approaches are physical file inclusion to the build path and version control system.

A common limitation of the compositional technique is an inability to separate fine-grained source-codes of a feature. However, In the language-based approaches, configuration parameters can be used to tailor coarse-grained components and therefore taking care of fine-grained variations.

Activation/deactivation techniques
In the activation/deactivation approach, variable features are simply turned off or on depending on the product configuration. In this approach, source-codes for all the variant features that may be used have to be compiled and shipped with the product. If a feature is deactivated its various execution paths in the source-codes are sidestepped. That is, feature execution is controlled with conditional statements such as *if.. else* and *switch*. Feature activation conditions may be read from an external source and stored as global variables and the conditional statements test the values stored in the global variable. Depending on the property of variation point, the global variable may be set once-and-

for-all or it may be changeable. A license key, for activating product features, can be regarded as a sophisticated implementation of activation/deactivation technique.

Listing 2.2: Examples of global variables used used to implement variations

```
1 ..............
2 public class AccountController {
3 static boolean USER_VALIDATION = false;
4 static boolean DELETE= false;
5 ........................
6 public String createAccount (....){
7 //forward request to Account Manager Service
8 // component to actually create the account
9 if (USER_VALIDATION){
10 //forward request to ValidationController component
11 }
12 }
13 if (DELETE)
14 public String DeleteAccount (....){
15 //forward request to Account Manager Service
16 // component to actually remove the account
17 }
18 }
19 ..............
20 }
```

Listing 2.2 depicts an example of the activation/deactivation technique in the source-codes snippet. In the snippet, the feature activation is controlled using a combination of global variables and conditional statements. In line 3-4 of the listing, *USER_VALIDATION* and *DELETE* are Boolean global variables that take the value of true or false. The global variables are used in line 9 and 13 to activate part of *User Validation* and *Delete* features respectively. Subsequently, source-codes segments of the corresponding feature are only executed when the corresponding global variable is set to true (i.e. the feature is activated).

The activation/deactivation technique can be used to implement a feature at all levels of granularity at different phases of the product lifecycle. However, the technique is associated with an over-bloated product because of the variable features must be delivered even if they would never be used. Similar to annotative technique, It is also difficult to reason about individual features because several conditional statements may have to be used to activate/deactivate a single feature. In practice, this technique is often used in combination with other techniques such as the compositional approach.

Model Driven Development (MDD)
Model Driven Development (MDD) is a superset of Model Driven Architecture (MDA) - an Object Management Group (OMG) initiative for raising the level of abstraction of software development [Fra03]. One of the benefits of MDA is gaining more value from a model in the sense that implementation artefacts such as executable can be generated from the model. Similarly, proven practices such as design patterns, architectural patterns, and architectural styles can be codified and enforced in the models. Thus, a model is not only a design artefacts but also part of implementation artefactss.

Recently, MDD is gaining popularity for the domain implementation and the technique is orthogonal to the many other techniques including the compositional, the annotative, and the activation/deactivational approaches. MDD can be used to model combinable components as in compositional technique [VV11]. Model elements can also be filtered in consistent with a product configuration similar to annotative technique[VV11, CA05, WG18, BW14]. Global variables can also be modelled on model-driven components. Configuration files may then be generated from the model elements and those values are fed into conditional statements to activate/deactivate desired features.

Other techniques such as generative [CE00], frame technology[MP02], and language-based [Bat05a, SBB+10] initiatives can also be used for the domain implementation.

We have so far discussed the lifecycle activities of domain engineering phase - the phase to create reusable assets for a family of products. In the next section, we discuss the lifecycle activities of feature-oriented application engineering phase - the phase to create individual products from the reusable assets produced at the domain engineering phase.

2.3 Feature-oriented application engineering

Application engineering is a phase in which the specific products are derived from the reusable assets that were developed earlier at the domain engineering phase. Ideally, application engineering should be less stressful. Some enthusiastic of automated production of software advocate for eliminating the logical separation between the domain engineering and the application engineering by reducing the later to simple selections of features and then 'push-button' to create the product [Kru06]. In reality, however, the required efforts in application engineering can range from a thorough analysis and complete implementation of tailored abstract models to the push-button activity.

In the through analysis and complete implementation scenario, an application engineer receives high-level requirements of the products; matches the requirements into features; derived, from the reusable assets, abstract models (both analysis and design) that are specific to the product. The terminal architecture model would then have to be

implemented entirely. This is the case when the domain is implemented in concurrence with the application engineering.

Simple feature selections and then 'push-button' to create a product can be achieved in the case of a specialized and narrow technical domain such as database and network protocols. It is also possible in a portfolio of products, such as a microwave oven product line where the features to be selected are limited to those pre-planned (i.e. the product line company is no willing to offer product-specific extensions). Usually, the domain has been implemented with annotative, MDD, or generative techniques. To derive a product in that scenario, an application engineer selects the desired features and then push the button to, automatically, generate/assemble implementation artefactss in consistent with the feature selections. In that case, analysis and design models that are specific to the product may also be created through the push-button activity.

Somewhat in the middle of the two extremes is to reuse the analysis and implementation of a similar product already created, rather than to develop the terminal abstract model from scratch. Similarly, product-specific extensions may have to be developed.

Recall, also, various techniques can be used as a stand-alone or in combination to implement the domain. Regardless of the magnitude of the efforts required and the techniques used, the following activities can be identified as part of application engineering lifecycles: product requirement elicitation and product instantiation. We explain the two activities in the next two sections.

2.3.1 Product requirement elicitation

Product requirement elicitation begins when a customer made initial contact and specify his requirement. The customer's requirements should be matched to the pre-planned features. The application engineer should explain, to the customer, each of the variable features and its implication to the overall product. The application engineer should also guide the customer to resolve the variations by selecting/deselecting optional features and choosing between the alternatives. If all the requirements match the pre-planned features then no further analysis is required.

If, however, the customer requirements or part of the requirements differ from the pre-planned features, the application engineer should still match the extraneous requirements with respect to the pre-planned features. Therefore, the requirements should be compared to closely related features and the difference may be modelled either as an extension or alternative to an existing feature. In that case, a new node has to be created on the feature model; both the analysis and the design models have to be updated to accommodate the new feature.

The extraneous requirements may be modelled as a product-specific extension if there is no existing feature to be compared with. In that case, the domain models should not be updated but the feature may still be suggested to similar customers. In some cases, the customer may negotiate with the product line company to hide the specific extension as it may be the customer's secret advantage over their competitors [HKM+13].

To complete the product requirement elicitation, the product analysis models such as the use case model should be derived from the use case model produced from the domain analysis activity. The customer specific-extensions, if any, have to be reflected in the use case model of the product. Deriving the use case for the specific product can be done manually or with the assistance of a tool. In the manual approach, the domain use case model would be documented but without the model elements that correspond to the deselected features. In other words, the model elements of the use case model are filtered in consistent with the feature selection. In the tool approach, the use case model could be filtered automatically in consistent with the feature selections.

When requirement elicitation for the specific product is completed, the next activity is product instantiation.

2.3.2 Product instantiation

To instantiate a specific product, specific artefactss realizing all of the product's features have to be included in the specific configuration. At this stage, all the design models, including that of product-specific extensions, must have been completed. Consequently, a product can be instantiated through one or more of the following activities:

- Manual tailoring and implementation of an abstract terminal model;
- Automatic derivation of source-codes from the reusable assets;
- Automatic generation of source-codes from the design models.

In the manual tailoring and implementation of the abstract terminal model, intermediate models such as the domain object model and the activity diagram are filtered in consistent with the feature selections and then documented for future references. Similarly, the terminal architecture model is filtered in consistent with the feature selections. The filtered terminal model will then have to be fully implemented if no product has been instantiated already (i.e. the product is the first to be instantiated from the product line). However, in the case of instantiating the subsequent products, prior source-codes should be reused.

Automatic derivation of source-codes, from the usable assets, is one of the button-push approaches to product instantiation. This is the case when the domain is fully

implemented (source-codes exist in its entirety) and in which the feature selections are connected to the source-codes. Similarly, product-specific extensions, if any, must also have been implemented. The design models, both the intermediate and the terminal models (if they exist) are filtered in consistent with the feature selection and then documented for future references.

In the automatic generation of implementation artefacts from the architecture models, a product is instantiated through the generation of source-codes from the tailored terminal model. This is the case when the domain is implemented with MDD or generative technique. Intermediate models such as the domain object model and the activity diagram can either be abstract or concrete and should also be tailored in consistent with the feature selection and then documented for future references. Unlike the automatic derivation from the reusable assets, in this approach, source-codes do not exist in its entirety but can be generated, either fully or partially, from the design model.

It may be possible to select architecture configuration as a part of the product instantiation if the quality attributes were not already captured as features. For example, in EduPL, we may have a quality attributed related feature called *Users Load* with children features as $<$*Five Thousand* and $>$ *Five Thousand* as alternative and the choice between the alternatives determines the connection mechanism to be used between architectural components and whether or not the application and the persistent storage would be distributed on separate servers. In the absence of quality attributes related features from the selectable features, separate selections of architecture configurations are required.

In the next section, we illustrate an example of feature-oriented application engineering.

2.3.3 Application engineering example

In this section, we illustrate an example of feature-oriented application engineering using the *Account Management* subsystem of EduPL. We begin with an example of requirements elicitation of a specific product in the next section.

2.3.3.1 Product requirements elicitation example

Assuming management information (MIS) Officer, from a potential customer university, has established a contact with the EduPL company for a software product and explained that one of their high-level requirements is to manage accounts of their portal users. The application engineer should match the requirements of managing accounts of the portal users to the features of *Account Management* subsystem and explain, to the MIS officer, the variable features and their implications. The application engineer should then

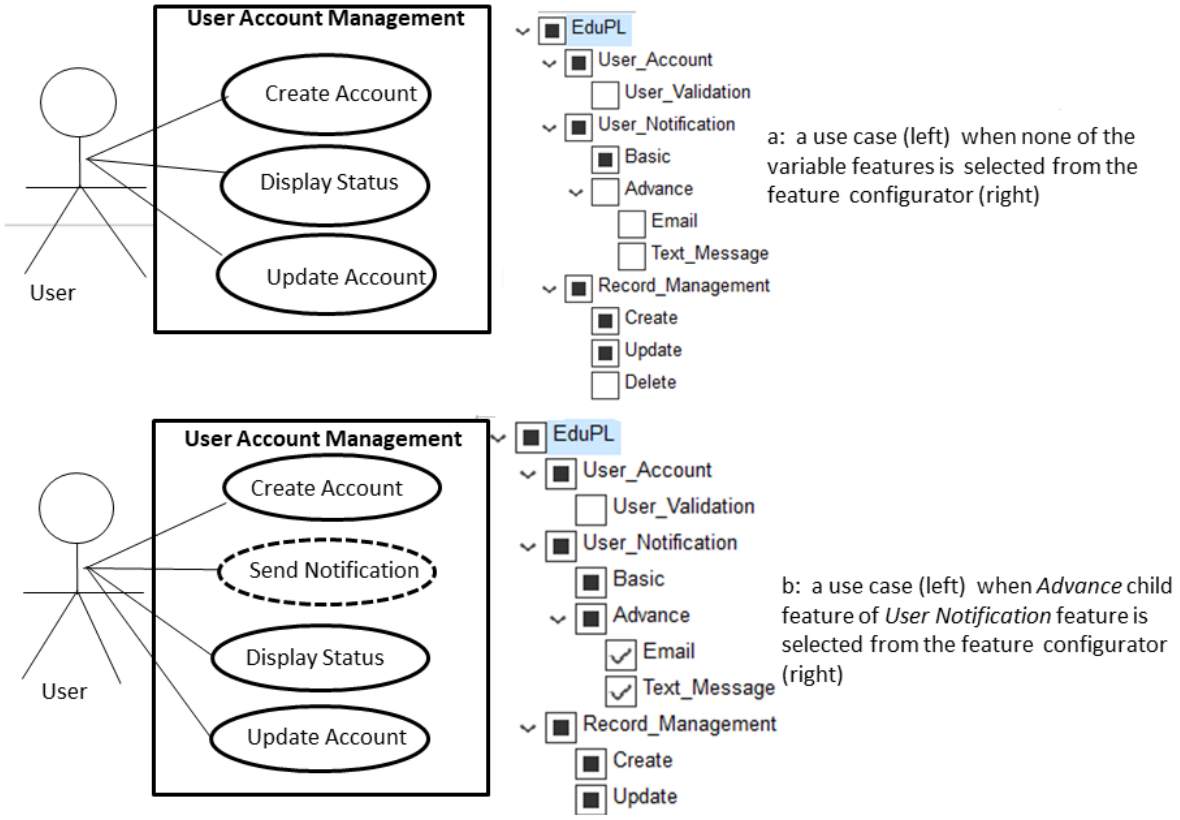


Figure 2.11: use case models (left) in consistent with the feature selections from the configuration interface (right)

help the MIS officer to resolve the variations by selecting/deselecting from the variable features. Variations resolution decisions are usually presented in a configuration interface. Example of a configuration interface is feature configurator depicted at the right hand side of Fig.2.11a and Fig.2.11b.

After resolving the variations, the application engineer will then has to filter the use case model, and any other analysis model, in consistent with the selected features. The left hand side of Fig.2.11a and Fig.11b shows examples of two instances of use case models. In each of the figures, the variable model elements of the use cases have been filtered in consistent with the features selected from feature configurator depicted at the right hand side of the figure. In the Fig.2.11a, the customer selected none of the variable features. i.e. the customer is happy with the basic features of *Account Management* subsystem. On the other hand, in Fig.2.11b, the customer selected *Email* and *Text Message* as variable features. Recall that *Email* and *Text Message* are OR children feature of *Advance*, which is also a child feature of *User Notification*. Hence, the *Send Notification* use case in the use case model corresponds to the *Advance* feature in the feature model (refer to Table 2.1).

The example so far assume that customer requirements match the pre-planned features of the *Account Management* subsystem. In reality, that is not always the case. Consider, for example, a case of highly exclusive university where a user has to pay a certain fee to create an account. Such university would require a *Payment* feature as their specific extension. The reusable assets of EduPL will have to be updated with the extension if many other exclusive universities are also likely to require the *Payment* feature.

When the requirements of the specific product are captured and documented, the next activity is product instantiation. We illustrate an example of product instantiation in the next section.

2.3.3.2 Product instantiation example

Assuming the application engineer wants to instantiate the products based on the filtered use cases of Fig.11a and Fig.11b. The application engineer should filter the components and connectors architecture model to be consistent with the use cases and by extension to be consistent with the feature selections. The model will then have to be implemented.

Fig.2.12a and Fig.2.12b depict components and connectors model corresponding to the use cases of Fig.11a and Fig.11b respectively. In this example, the implementation of the first product depicted in Fig.2.12a should be reused in the implementation of the second product represented by the model in Fig.2.12b.

We have so far discussed the background concepts of feature-oriented software product line engineering. The focus of the discussion has been on the feature-oriented approach to capturing and implementing variations. Moving forward, we shall elaborate on the key focus of this thesis. In the next section, we begin with the adaptability of reusable assets to accommodate emerging variable features.

2.4 Research focus

Recall, in *Chapter one*, we introduced the adaptability of reusable assets and variations in feature binding time as the focus of this thesis. In this section, we elaborate on both beginning from the adaptability of reusable assets.

2.4.1 Adaptable reusable assets

Usage-context is any contextual setting in which software product is deployed or supplied to[LK10, HT08]. For example, the Berkeley Database Engine (BDE)[Coo17], introduced in *Chapter one*, is deployable into many usage-contexts: due to its small footprint, it suits embedded devices such as smartphones and smart cards; it appeals to messaging servers and directory servers applications because of its high concurrency; its fast front-end cache can as well make it desirable for e-commerce applications. Similarly, some products of a product line are supplied to many other product lines. For example, vehicle infotainment products are often supplied to different car product lines in an automotive domain. Application domains such as smart home, as discussed in *Chapter two*, also have wide-range of usage-contexts and multiple chains of product lines[HT08, HMT09].

Variations in usage-contexts, for products of a product line, have been fuelling demands for adaptable reusable assets. The reusable assets should be flexible to address variations that may emerge with additional usage-contexts. This is the case when some mandatory features that were hardcoded to manifest in all the products have to become optional in other products[HMT09]. More specifically, product line assets, especially source-codes, have to be retrofitted to inject the emerging variations.

Injecting additional variations to existing reusable assets is not the only way to address the emerging variations though. The main assets may be cloned and extended with the specific variation requirements [SB99]. The clone can be created by either physically copying the original assets to a new location, or by using configuration management branches. However, cloning introduces additional maintenance overhead because some tasks, such as testing and bug fixing, need to be performed on each of the duplicated copies[DRB+13].

In software engineering literature, changes to inject additional variations may be classified under adaptive maintenance, which is crucial to software development life-cycle. Adaptive maintenance, of single product development, roughly consumes about 60 percent of total maintenance costs[Gla01]. The maintenance cost of a product line is intuitively higher than that of single product development because of the likelihood to modify the assets is relatively higher in the product line development. In addition, the assets modifications in the product line development propagate changes to many parts of the assets due to dependencies between features[CCG+16].

In addition to the adaptability of reusable assets to accommodate wide-ranging and emerging variable features, binding time of the features and their possible variations that may arise between products is another focal point of this thesis and we elaborate on that in the next section.

2.4.2 Feature binding

Feature binding is a physical inclusion of the feature in the product configuration and making it available for use. Our view of feature binding is consistent with the view articulated in the current literature[LK03, Lee, VdH04]. More specifically, and in this thesis, feature binding means the binding of a feature to a product in contrast to the binding of a feature to the product execution context. This section explains the two forms of feature bindings.:

Feature binding to a product
Feature binding to a product is about a physical inclusion of both the variable structure and the variable behaviour in the product configuration. For example, assuming the following are access control features of a smart home product line: *Keypad, Finger Print, and Remote Control*. From the variation type point of view, these features belong to OR group and each of the three is a specialization of the *Access Control* feature. Hence, all the three features can be included and made available to use in a specific smart home. However, even though all the three features can be bound to the smart home product (i.e. included and made available for use), only one of the three features would be executed at a time because they represent different forms of the same thing: access control.

At any given time, a feature in operation is said to be bound to the execution context of the product and that is our next explanation.

Feature binding to the execution context of a product
Feature binding to execution context is about the product exhibiting the specific variable behaviour at a given time. For example, the access control handler of the smart home product, in the smart home example, has to be given a reference to one of the three features at a time and that is the variable feature in the execution context at that moment. In the implementation, there are two modes of binding a feature to the execution context of a product[CE00]:

i Static feature binding mode: the feature in the current execution context can be determined by looking at the static structure of the source code. For example, if a reference to the *Keypad* feature is hard-wired in all the places where access control is required, the *Keypad* feature is said to be statically bound.

ii Dynamic feature binding mode: The feature in execution context is not known until at invocation time because one of several variants features may be invoked at execution time[BLL+14, TSCS16]. That is, the feature in the current execution context cannot be determined by looking at the static structure of the source codes. For example, if more than one features are included for the access control, any of the included features may be used (bound to execution context) to access the smart home.

Distinguishing between the two forms of bindings (binding to product and binding to execution context) is important to avoid limiting the scope of feature binding to only the binding to a product execution context. In most cases, a discussion/technique with a limited scope of binding a feature to execution context focuses only on the existence of alternative execution path or absence of it (e.g. if..else for the dynamic feature binding mode and absence of if..else for the static feature binding mode). Limiting the scope of feature binding to execution context may result in one or more of the following consequences:

- Limiting post-deployment variations to only those affecting behaviour (i.e. runtime variations) of the product and thereby losing sight of features that may need to be bound after product deployment but their binding to execution context is static such as in software upgrade.
- Taking engineering decisions that complicate upgrading a product with prior-deselected features. For example, by closing down a variation point using a static reference of a fixed feature where binding alternative variants may also be possible in the future.
- Impede effective communication between separate groups of researchers or practitioners which affects sharing of knowledge, sharing of best practices, making comparisons between alternative approaches, and advancements of the state-of-the-art.

2.4.3 Feature binding time

Feature binding time is a phase in the product lifecycle in which the feature is included and made available for use. In this thesis, we propose to abstract away the actual binding techniques used. The logic being, for the product line customer, the technique used should not matter but rather when the feature is bound. Thus we consider the following binding time as an abstraction over various underlying binding techniques:

Pre-deployment time:
A feature is bound to a product before the product is shipped to the customer site. Binding techniques that may be used at this phase include pre-processing and product builds. For example, when building a special Linux distribution for a firewall router, all features that are irrelevant to routing and security should be stripped off at this phase. Similarly, features that are needed to optimize routing and security should be bound at this phase.

Deployment time:
A feature is bound at installation time at the customer site before the product is put to use. Binding techniques that may be used at this phase include activation/deactivation of features, plugin registration. Configuration interfaces that may be used at this phase include installation wizards and configuration files. For example, binding of a *Language* feature to an operating system product may be decided at this phase.

Post-deployment time:
A feature is bound when the product is already in operation at the customer site. For example, a new device driver of an operating system may be downloaded and bound when a new device is plugged. Field-programmability in embedded domain, where a customer downloads and activates a new feature is a form of feature binding at post-deployment time.

Some features may be required to support multiple binding times for different customers and this requires variations in feature binding times. For example, when a product line should support other product lines as customers [HMT09](i.e., a product line of product lines). We discuss this in the next section.

2.4.4 Variation of feature binding time

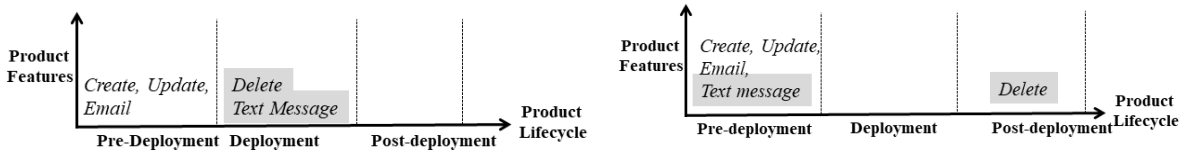
In this section, we elaborate on the variation of feature binding time introduced earlier in *chapter one*.

Traditionally, decisions about the feature binding time are fixed during the analysis and design of product line assets, because binding time is viewed as a property of a variation point [CB13]. However, when a product line targets various categories of customers, there may be a demand for variable binding time for some of the features. In the example of a smart home, a decision to bound some features may be made when ordering the smart home for some customers (if the deployment environment is known). For other customers, the decision may not be made until at a point of installation. In addition, a previously deployed smart home may be upgraded with additional features. In this case, the supplier of smart home products is faced with variations of feature binding time.

Another example is EduPL introduced in *Feature-oriented domain analysis* section. EduPL provides software products to various market segments including universities and Enterprise Resource Planning (ERP) vendors. In this example, consider the following two partial configurations for two different categories of customers:

(i) A configuration for a customer from a market segment of ERP vendors (Fig.2.12a) who requests the binding of *Email* at pre-deployment time, but requests the deferments of binding decision on *Delete* and *Text Message* to the deployment phase to allow their salesman to negotiate with their respective customer over the price of a product.

(ii) The second configuration is for a customer university; the customer requests the binding of the *Text Message* at the pre-deployment phase (Fig.2.12b), whereas the *Delete* feature was not included because their current policy prohibits the deletion of stored records. However, they ask the *Delete* feature to be purchasable at a later



(a) Feature binding time decision for an ERP vendor

(b) Feature binding time decision for a customer university an ERP vendor and a university

Figure 2.13: Variation of feature binding time between

time, because they expect to update the policy to allow the deletion of the stored record in the near future.

The example of the enterprise software product line also illustrates a variation of feature binding time between the two configurations (i) and (ii): That is, the binding of *Delete* is at deployment phase in the configuration (i), but at the post-deployment phase in the configuration (ii); the binding of *Text Message* is at the deployment phase in the configuration (i), but at pre-deployment phase in the configuration (ii).

So far we elaborated on feature binding time and its possible variations between products of a product line. In the next section, we briefly discuss why variations in feature binding time need to be managed.

2.4.5 Justifications for managing variations of feature binding time

There are several overlapping justifications for the demands of variations of feature binding time. These are related to agility in responding to customer demands, costs and efforts saving to the product line company, and non-functional requirements of the products. This section briefly explains some of these justifications.

Customer request to upgrade already deployed product with prior-deselected features

The ability to deliver diverse products means that prior-deselected features may be requested later. This may be due to an increase in customer operating capacity or improve financial status. Customers may also change their mind and request the forgone alternative feature. This is because the alternative features may be as a result of alternative domain rules and the customers of the product line can be at liberty to choose/change between the alternative rules. For example, in EduPL the children of the *Fail Count* feature represent alternative domain rules of computing CGPA. In this example, one of the institutions was using the *Continuous* alternative feature, later the University senate voted in favour of *Once* alternative feature and requested for the changeover. The request

that ought to be served with a simple change of binding the alternative feature ended up triggering the whole application engineering lifecycle activities.

A product line company should be able to serve the request to replace the alternative or to offer a prior-deselected feature within acceptable costs, efforts, and within a reasonable amount of time. managing variations of feature binding time by making the feature binding flexible would make the product upgrade less time consuming and maintain product quality because ad-hoc changes to the product are minimised. Minimising delay in the product upgrade also benefits customers since they can put the requested feature into operation without unnecessary delay.

Diverse usage-contexts

A usage-context - a contextual setting in which software product is deployed or supplied to- is one of the key determinants of feature binding time. As mentioned earlier, Alternative domain rules often manifest as alternative features and not knowing which of the alternatives domain rules a customer uses means that binding the relevant features would have to be delayed. This case is aggravated if orders come from different suppliers in the middle of a software supply chain as in the case of EduPL and many consumer electronics [HKM+13]. It is also the case if the products in the product line have diverse deployment environments.

Loss of revenue, security, resource consumption

Loss of revenues, security, and resource consumption are also significant drivers for the flexible binding requirement. Ideally, a product line company should be able to deliver what customer has paid for and no more so that the production effort is proportional to the revenue accrued. Recall, from the domain implementation techniques, with the activation oriented technique the whole features may be included in the product and license keys can be provided to activate the paid features. In that approach, customers only get values in what they paid for. However, that does not address resource consumption and security concerns.

The ability to deliver both low and high-end products from the same reusable assets means that the source-codes for the selected features have to be totally removed from the low-end products. Simply deactivating deselected features and making them unavailable while their corresponding source-codes remain in the product is unacceptable because the product memory would be overwhelmed. Where resource consumption and loss of revenue are not concerns, security can be a concern since the extraneous features may increase the vulnerability for attacks. Thus, managing variations of binding time by making feature binding flexible would address the revenue, security, and resource consumption concerns.

In the next section, we further elaborate on the specific challenges the thesis aim to address.

2.5 Specific research challenges

This section revisits the specific challenges associated with both the adaptability of reusable assets and the variations of feature binding time. Although these challenges may be relevant to many application domains, we focus on embedded database and information system domains.

2.5.1 Research challenge on adaptable reusable assets

Over the years, SPLE researchers proposed new language-based implementation techniques to support flexible variations. Existing languages for advanced separation of concerns were also adapted for product line development. The alternative techniques represent different mechanisms for decoupling and synthesis of implementation units in the code-asset.

The research challenge is to unravel the adaptability of reusable assets when the modern language based implementation techniques are used to implement and update reusable assets. Existing studies are limited in one or more of the following ways:

- No exploration of feature property in the source-codes that affects the modifiability.
- Limited coverage of the modern-language based implementation techniques and thereby living-out the key techniques that were designed mainly to implement variations.
- No evaluation of modifiability with respect to adding variations

2.5.2 Research challenge on variations of feature binding time

The challenge to support variations of feature binding time stems from the fact that domain implementation techniques and their associated feature binding mechanisms either deliver a fixed target binding time with no option to defer or to advance the binding time or deliver flexible binding time but with certain limitations. For example, some of the annotative approaches such as a pre-processing technique [HZS+16] can be used to implement features at all levels of granularity, but features can only be bound before compilation. That is, at pre-deployment time. On the other hand, some compositional approaches such as plugin architecture [CEM04] can be used to bind features at different binding time (through the plugin registration at the different phases) but cannot be used to implement fine-grained variations [KAK08]. Similarly, although MDD is a flexible

implementation technique, current researches [BW09, SE08] are limited to supporting variations between static and dynamic modes.

Other researches to circumvent the limitations of feature binding techniques tend to follow one of the following ways:

- Employs different sets of techniques for the different binding times[VdH04]
- Flexible composition of fine-grained model slices [WJE+09]
- Implementing each feature with a specific binding time as a separate concern[ARR+16, CRE08].

Most of these approaches do not consider explicit variation points at the architecture level. In addition, they are either too fine-grain or too low-level. Where the approaches consider variation points at the architecture level, they are limited to the scope of binding a feature to execution context [BW09, SE08].

2.6 Chapter summary

We discussed the background concepts that are related to the feature-oriented approach to capturing and implementing variations in SPLE. We also elaborated on the focus and the research challenges of this thesis: Adaptability of reusable assets to accommodate emerging variations and the associated research challenge; feature binding time and its possible variations that may arise between different products and the associated research challenge.

In the next chapter, *Chapter three*, we discuss the detail processes and outcomes of our action research to investigate the flexibility of language-based implementation techniques that are proposed to support flexible variations.

Chapter 3

Language-based approaches to flexible variations: Action research

This chapter presents the details of the action research that was executed to investigate the adaptability of reusable assets when adding variations. Part of the investigation is an exploration of the properties of features, in the code-asset, that affect modifiability when injecting additional variations. In the exploration, we observed that an intersection between program elements of different features negatively affects the modifiability of code-assets when injecting additional variations.

In addition to the exploration, we selected and compared the following three language-based implementation techniques: (i) Feature-Oriented Programming (FOP), (ii) Aspect-Oriented Programming (AOP), and (iii) Delta-Oriented Programming (DOP). We compared the techniques relative to Pre-Processing (PP) of annotations (a classical variability implementation technique) using modifiability metrics. Furthermore, we discussed the techniques with respect to supports for modularity and multiple binding time.

The results showed that DOP in its current implementation is the steepest in modifiability because it lacks the necessary semantics to support flexibility. AOP is the second steepest because of the high number of language constructs that have to be used to untangle a feature in the code-assets. FOP has moderate modifiability and the best form of modularity. Overall, none of the language-based implementation techniques is better than pre-processing in terms of modifiability. Similarly, language-based implementation techniques specifically proposed to implement features in SPLE (Feature-Oriented Programming and Delta-Oriented Programming) have better modularity but do not support variations in feature binding time.

The action research fulfills our first and second research objectives (O1 and O2). Hence, the findings provide answers to our first and second research questions (RQ1 and RQ2).

Before we go into the details of the action research, in the next section, we clarify the meaning of the key terms that appear repeatedly in the chapter.

3.0.1 Definition of terms

Definition 1 (Code-asset, program element)
Code-asset (CA) is a pair $\langle E, R\rangle$ where E represents the set of program elements of CA, and R is the set of relationships between the program elements. A program element e can be an attribute, an operation or declaration. Let Att be the set of attributes of CA, Op be the set of operations of CA and Dec be the set of declarations of CA. The set E of program elements of CA is defined as $E = Att \cup Op \cup Dec$

Definition 2 (Feature module)
A feature module f1 consists of a set of program elements, Ef1, such that $Ef1 \subset E$.

Definition 3 (Exclusivity)
Program element ef1 is said to be exclusive to a feature f1 if it satisfies a disjoint union relationship with other features in the code-asset CA. i.e.$ef1 \cap ef2 \cap ef3 \cap ef3 \cap \cdots efn = \varnothing$

Definition 4 (Intersection) *An intersection between programs elements of different features ef1 and ef2 is defined as ef1#ef2, which is a modification and or integration of ef1 and eF2 so that they work correctly together* [BHK11].[1]

Next section presents the settings of the action research.

3.1 Study settings

This section explains the action research settings which include the criteria for assessing the flexibility of reusable assets and the selected case study. The section also demonstrates the mechanisms of the selected implementation techniques when used to decouple features in the code-asset using a small example of injecting additional variation to the case study.

3.1.1 Evaluation criteria: flexibility

In this work, we used modifiability and modularity as constituents of flexibility as suggested by Sturm *et al* [SDS10]. In addition, we propose to include the support for multiple binding times as one of the constituents of flexibility because it is being

[1]Batory and Kim [BHK11] refer to this as feature *interaction* in the original definition. We use the term intersection to distinguish it from the interaction as a problem of unexpected side effects when two or more features are combined to work together [AABZ14]

recognized as an important aspect of SPLE [DFV03, VdH04, CRE08, RSAS11]. The three constituents of flexibility (modifiability, modularity, and support for multiple binding time) are potential indicators to maintainability, time-to-market and production costs. We further elaborate on these constituents of flexibility in the following:

3.1.1.1 Modifiability

Modifiability is the ease of accommodating changes. In this work, we adopted repetitive viscosity (RV) measure[RKS00] to compare the modifiability of the implementation techniques. RV is regarded as immediate efforts required to make the changes with the technique in observation. The lower the value of viscosity obtained from using a technique to modify program elements, the better the technique.

We instantiate the repetitive viscosity with three sub-metrics: #C: number of fragments created - this is the case when some fragments should be newly added to inject the variation (e.g. new delta in DOP or new aspect in AOP). #M: number of program elements moved - these elements are removed from their original location and inserted at a different place (e.g., an attribute declaration is moved to the new fragment created). #V: number of mechanisms added to implement variations - these elements solely belong to a variation mechanism (e.g., #ifdef for the pre-processing technique).

Where it is not possible or it does not make sense to derive the RV interms of #C, #M, and #V, we qualify the RV of each of the techniques as *low*, *moderate* or *high*. For example it does not make sense to quantify RV of intersection within a method an a constructor and in which the values can be reasonably generalizable due to variations in parameters/signatures, line of codes and scattering of intersections.

3.1.2 Feature modularity and support for multiple binding time

In general, modularity is a desired property of an implementation technique because it minimises complexity and enables separation of concerns[BOVH17]. In this work, we consider modularity from two different perspectives: horizontal and vertical modularity.

The horizontal modularity is used to evaluate whether a technique provides a mechanism to modularize implementation artefacts at the same level of abstraction with a feature (i.e., decomposing a product line assets into feature modules providing traceability from an abstract feature in the analysis to feature module in the implementation). Similarly, the vertical modularity means a sub-hierarchical organization of program elements within the feature module.

In addition to modularity, an implementation technique is said to support multiple

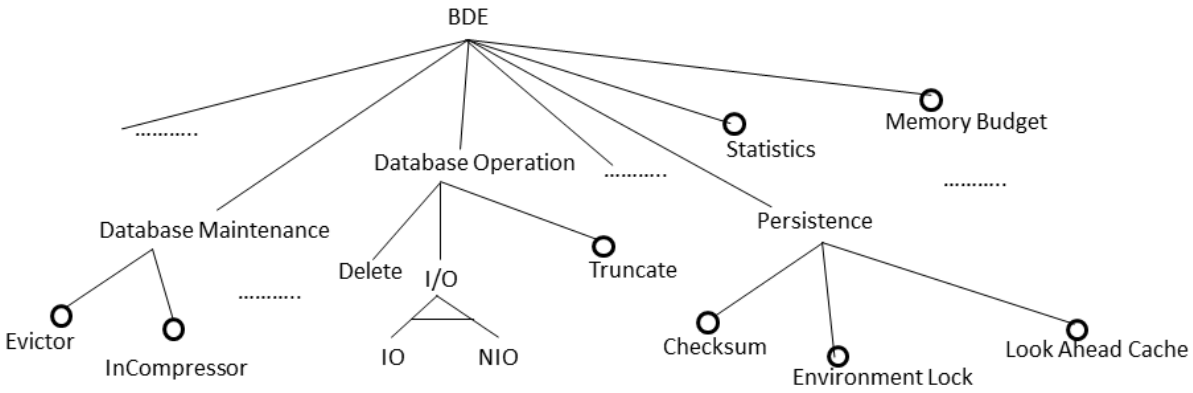


Figure 3.1: Selected BDE variations transformed into features.

binding times if it can be used to bind a feature at more than one phase of the product's lifecycle.

3.1.3 The case study: Oracle Berkeley Database Engine (BDE)

Oracle Berkeley Database Engine (BDE), Java Edition, is an embedded storage engine designed to support integrations of functionalities and storage requirements of an application in a single binary installation. It is specific for applications targeting Java Virtual Machine (JVM) and where no separate installation of a database server is required. Hence, the BDE runs in the same memory address space with the integrated application and thereby eliminating the overhead of process switching. As highlighted in the *Introduction section*, BDE products can be embedded in a wide range of applications.

BDE has many features that are extraneous to the requirements of some applications in the domain. For example, features for concurrent access and atomic transaction are not required in an application with single access and simple data requirements. Similarly, by default, BDE gathers statistics of almost every operation of the database such as tree traversal and memory usage. The implementation of the statistics collection adds a significant footprint that may become a burden to some applications that do not require the collection of statistics. Therefore, those features should be variable and the code-asset should reflect their variability.

We selected the BDE as a case study because its legacy code-asset requires additional variations to derive customized products for the different applications and has been used in previous researchers related to flexible variations and binding times[KAB07, ARR+16].

Fig.3.1 depicts the partial feature model of BDE in which 11 features there were

Table 3.1: List of 11 variations selected for transformation into features

Feature name	explanation
Checksum	Feature for calculating checksum for every database entry to be written to the log file and using the same for validation while reading back the entry.
Delete	Feature implementing delete operation of the database.
Environment Lock	Feature responsible for locking the database to preserve its integrity on concurrent access.
Evictor	Feature responsible for maintaining memory consumption within certain threshold specified in the database configuration.
Incompressor	Removes delete entries and empty nodes from the tree
IO	A variant implementation of database input and output operation.
Look ahead cache	Feature for maintaining cache on log files that are lineup for cleaning.
Memory budget	Feature for monitoring overall memory usage of the database.
NIO	A variant implementation of database input and output operation.
Truncate	Feature for deleting the database and creating new one.
Statistics	Feature responsible for gathering statistics about several operations of the database.

not variables will have to be made variable to increase its usage-context. All the features were identified from previous studies [KAB07, ARR+16] and vary in size of line of codes. We limited our selection to 11 features because the retrofitting of variations into features is tediously repetitive. Table 3.1 presents a brief explanation of the selected features.

3.1.4 Case study exploration

To study how features are implemented in the code-asset, we followed an iterative process depicted in Fig.3.2. We started with consulting BDE documentations [Coo17, Tec] to have reasonable domain knowledge. We then checked the features already identified from the previous studies[ARR+16, KAB07]. We then searched through the code-asset to trace the identified features. With the support of a tool, we marked program elements of the traced features with annotations. In the process of marking the BDE code-asset, we made the following observation: the code-asset of BDE is a union of feature modules and the program elements in the feature modules intersect with each other. Fig.3.3 depicts this observation in a Venn diagram.

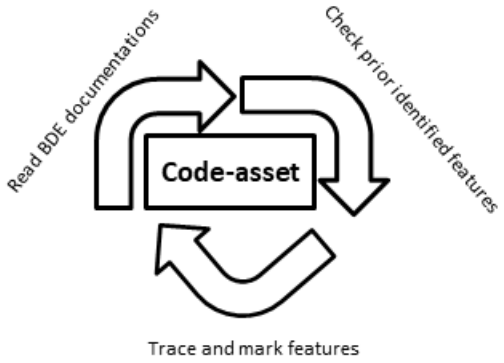


Figure 3.2: Iterative process for study of features in code assets

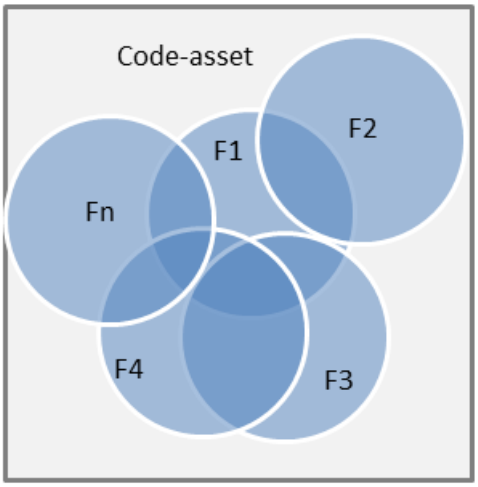


Figure 3.3: Union an intersection of feature modules in code-asset

In Fig.3.3, each of the feature modules, represented by F1...Fn, internally contains program elements (attributes, declarations, and operations) that are only for the containing feature as well as program elements that are shared with other features. We refer to the program elements that are not shared with any other feature as *exclusive* elements to the given feature, and the shared part as an *intersection* between features. We define *exclusivity* and *intersection* (semi-formally) in definition 3 and definition 4 respectively at *Definition of terms* section.

Consequently, injection of additional variations requires decoupling of unions between features in the code-asset: The program elements of the feature to be made variant (both the exclusive and the intersection parts) have to be traced in the code-asset; Both the exclusive and the intersection parts have to be separated; The separated parts should be added to a product configuration only if the variant feature is selected.

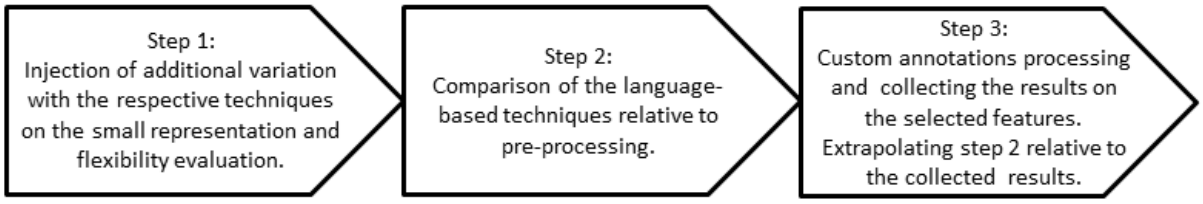


Figure 3.4: Research execution steps

We noticed this observation to be true in other implementation techniques. The following are examples: using various design patterns in OOP[GHJ+95, SSS17], one should be able to encapsulate features to some extent. For example, *Keypad*, *Finger print*, *Remote control* as OR group sub-features of *Access Control* feature in a smart home can be implemented using *Strategy pattern*. In that case, a developer implements each feature as a separate strategy for accessing the smart home, and each of the strategies only contains program elements exclusive to one of the features. At the point of invocation, however, a request to access the smart home has to be resolved to one of the concrete strategies, and that is the point of intersection.

Similarly, when using class inheritance, as a form of polymorphism, to implement an optional feature as a subclass of one of the mandatory classes, the sub-class is the exclusive program element to the optional feature. In that case, intersection points are places where references are made to the subclass.

Thus, thinking in terms of exclusiveness and intersection may be beneficial in the evaluation of other implementation techniques. Intuitively, an implementation technique that limits the number of intersections may be better for a product line that is not stable and with the tendency of the emergence of new usage-contexts. Of particular note, *Factory pattern* encapsulates a specific form of intersection- a point to create one of the variant objects.

Following the observation about the exclusiveness and the intersection, we adopted a *reductionism* strategy and focus on a small representation of both the exclusivity and the intersection. Reductionism is a scientific approach to understanding complex phenomena by focusing on a thorough investigation of part of the phenomena in order to understand the whole[ESSD08]. While the reductionism approach is not normally reported explicitly in software engineering researches, it has been applied to several software engineering researches implicitly [BOVH17, RKS00, MLWR01, DSW14].

Figure 3.4 depicts how the rest of the study was executed. As shown in Fig.3.4, after the exploration of features in the code-asset, the next step (step 1 in Fig.3.4) of the study is the injection of additional variation to the small representation with each of the selected techniques. In the same step 1, we discussed the flexibility of each of the techniques using modifiability metrics. We discuss the details activities of step 1 in the

next section (*Study execution*).

In step 2, we compared the three techniques relative to pre-processing of annotations. Our hypothesis is that: *'new' language-based implementation techniques proposed to support flexible variations are also better in modifiability when injecting additional variations than the classical implementation technique.*

In step 3, we defined custom annotations based on various categories of exclusiveness as well as the intersections. We marked program elements of the selected features, from BDE code-asset, with the custom annotation and implemented a custom annotation pre-processor. We collect statistics about the eleven features at compile-time and extrapolated the results obtained from the small representation (from step 2). We discuss the activities of step 2 through step 3 in the *Custom annotations* section.

3.2 Study execution

From SPLE literature, we selected four distinctive variability implementation techniques for the study: (i) Pre-processing (PP) of annotations (ii) Aspect-Oriented Programming (AOP) (iii) Feature-Oriented Programming (FOP) and (iv) Delta-Oriented Programming (DOP). PP is not a language-based implementation technique but is widely used in practice[HZS+16, LAL+10] and therefore we used it as a benchmark. AOP augment Object-Oriented Programming (OOP) with mechanisms to separate crosscutting concerns. The last two (FOP and DOP) are specifically designed to implement feature modules in SPLE. These techniques differ in how they can be used to decouple and synthesize implementation modules. They also differ in how they deal with the binding time of features.

The main focus of this chapter is the comparison of language-based implementation techniques when we have to refactor variations of a legacy code-asset into features. We, therefore, illustrate each of the techniques with a simple case of untangling part of the *Statistics* features of BDE to change the type of *Statistics* from mandatory to optional

Note that, the legacy implementation of BDE has no embedded variation mechanism. The following sections merely illustrate how the respective techniques might be used to inject additional variations to the existing code-asset (i.e. to transform some variations into features) assuming BDE has been developed with the respective techniques.

Fig.3.5 illustrates the code-asset in which the *Statistics* is still mandatory. In the figure, there are four Java classes: *Cleaner, LongStat, FileSelector, and DatabaseImpl.* Within the classes, the program elements of *Statistics* are shaded in grey.

In the *Cleaner* class at the top-left of Fig.3.5, there are two attributes, *stats* an

Figure 3.5: Part of BDE code-asset before injecting variation. *Statistics* source-codes to be decoupled are shaded grey

nCleanerRuns (line 4-5) that are exclusive to *Statistics*. In addition, there is an intersection within the class constrcutor (line 9). The entire *LongStat* class, at the middle-left of Fig.3.5, is exclusive to *Statistics*. In the *FileSelector* class, at the bottom-left of Fig.3.5, the method *loadStat()* (line 7-9) is also exclusive to *Statistics*.

In the *DatabaseImpl* class at the right hand side of the Fig.3.5, the attribute, *stats* (line 7) and the method, *getEmptyStat()*(line 9-11), and the static inner class *StatsAccumulator*(line 35-36) are exclusive to *Statistics* . In addition, intersections can be observed within the *walkDatabaseTree* method (line 25-26, line 31-32, and even within its parameters on line 20). The method has thirty-seven (37) lines of statements that are not part of the intersection (not shown in Fig.3.5).

The *DatabaseImpl* class is an interesting one because it has program elements that are exclusive to *Statistics* as well intersections. In the subsequent discussions about the implementation techniques, we use the *DatabaseImpl* class to illustrate the modifiability metrics when decoupling exclusive elements and intersection within a method. We also use the *Cleaner* class to illustrate the decoupling of an intersection within a constructor.

In each of the techniques, we discuss the case of decoupling the program elements exclusive to *Statistics* using the *DatabaseImpl* class and we present the discussion as (*Modifiability Case 1: decoupling of program elements exclusive to a feature*). We discuss the intersection of program elements using the *walkDatabaseTree* method and we present the discussion as *Modifiability Case 2: decoupling intersection within a method*. Lastly, we discuss the case of decoupling an intersection within a class constructor using the *Cleaner* class and present the discussion as *Modifiability Case 3: decoupling intersection within a constructor*

We begin with the pre-processing of annotations in the next section.

3.2.1 Pre-processing with Antenna tool

In *Chapter two*, we illustrated how pre-processing of annotations (pre-processing in short) can be used to implement a domain. To be consistent with the presentation of other techniques to be discussed in this section, we repeat the pre-processing here with the example of changing the feature type of *Statistics* from mandatory to optional.

Figure 3.6: Injection of variation to the part of BDE code-asset with pre-processing

Fig.3.6, illustrates the use of pre-processing to inject variation in the code-assets depicted in Fig.3.5 and make *Statistics* optional. In the figure, program elements of *Statistics* are annotated with //#ifdef *Statistics*.... //#endif. For the annotations, we used Antenna, a non-native pre-processing facility introduced in Java and integrated with a SPL implementation tool-suite[TKB+14].

Of particular note in Fig.3.6 is that the annotations were used to mark program elements of different levels of granularity[KAK08]. For example, in the *Cleaner* class, the top box at the left-hand side in the figure, the two attributes that are exclusive to *Statistics* are annotated (line 4-7). These attributes are some of the fine-grained variations. In addition, object instantiation which represents an intersection within the constructor

Table 3.2: Repetitive Viscosity of (Virtual)decoupling of program elements that are exclusive to the *Statistics* feature in the *DatabaseImpl* class with PP

Technique	**#C**	**#M**	**#V**	Total
PP	0	0	3	3

Table 3.3: Repetitive Viscosity of (Virtual)decoupling intersections between the base-program and *Statistics* within the *walkDatabaseTree* method and within the constructor of the *Cleaner* class with the PP technique.

Decoupling scenario	**RV**	**Remarks**
intersections within a method	low	simple annotations are required.
intersections within a constructor	low	simple annotations are required

of the *Cleaner* class (line 10-13) is also annotated. In contrast, the *LongStat* class, at the middle box in the same left-hand side, is annotated in its entirety. It means that the whole *LongStat* class is only included in a product configuration when the *Statistics* feature is selected (The class is exclusive to *Statistics* and at coarse-grained level). Similarly, in *FileSelector* class at the bottom box in the left hand side of the Fig.3.6, the method that is exclusive to *Statistics*, *loadStat()*, is entirely annotated (line 6-10).

So far we illustrate how the code-assets can be annotated to introduce variations, next we demonstrate the modifiability metrics when the pre-processing is used to inject variations. We also briefly discuss modularity and the support for variations in feature binding time when the technique is used to implement variations.

Modifiability Case 1: decoupling program elements exclusive to feature with PP
To decouple program elements that are exclusive to *Statistics* in the *DatabaseImpl* class with PP, no #C or #M is required because PP is not sensitive to the semantic of the underlying programming language. However, #V is required in the form of annotations. In this example, three annotations are required (see lines 7-9, 11-15,and 47-50 in the *DatabaseImpl*class of Fig.3.6). Thus, Table 3.2 represents the modifability of program elements that are exclusive to *Statistics* and within the *DatabaseImpl* class.

Modifiability Case 2: decoupling intersection within a method with PP
To decouple the intersections within the *walkDatabaseTree* method with PP, simple annotations are required to virtually decouple the intersections. The annotations required in this case are three in total (see Fig.3.6). One of the annotations is used to surround the parameter, *statsAcc*, of the type *TreewalketStatsAccumulator* (line 26-28 in fig 3.6). Other annotations are in lines 34-35 and lines 42-43. Thus we rate the PP low in this regard as depicted in the first row of Table 3.3.

Modifiability Case 3: decoupling intersection within a constructor with PP
To decouple the intersection within the constructor of the *Cleaner* class with PP, single annotation suffices since the 35 *Statistics* related attributes (not shown in Fig.3.6) are adjacent to each other as one block of code. Hence, we rate the RV of PP low (see the second row of Table 3.3).

Modularity and support for variations in feature binding time in PP
As a classical tool-based technique, PP has no support for modularity. As such, variation mechanisms are scattered in the code-assets with no support for feature traceability [KA09].

With respect to support of flexible binding time, with PP, a feature has to be bound before compilation at pre-deployment time and not at any other binding time. Hence, pre-processing has no support for multiple binding time.

3.2.2 Feature-Oriented Programming (FOP) with Jak language

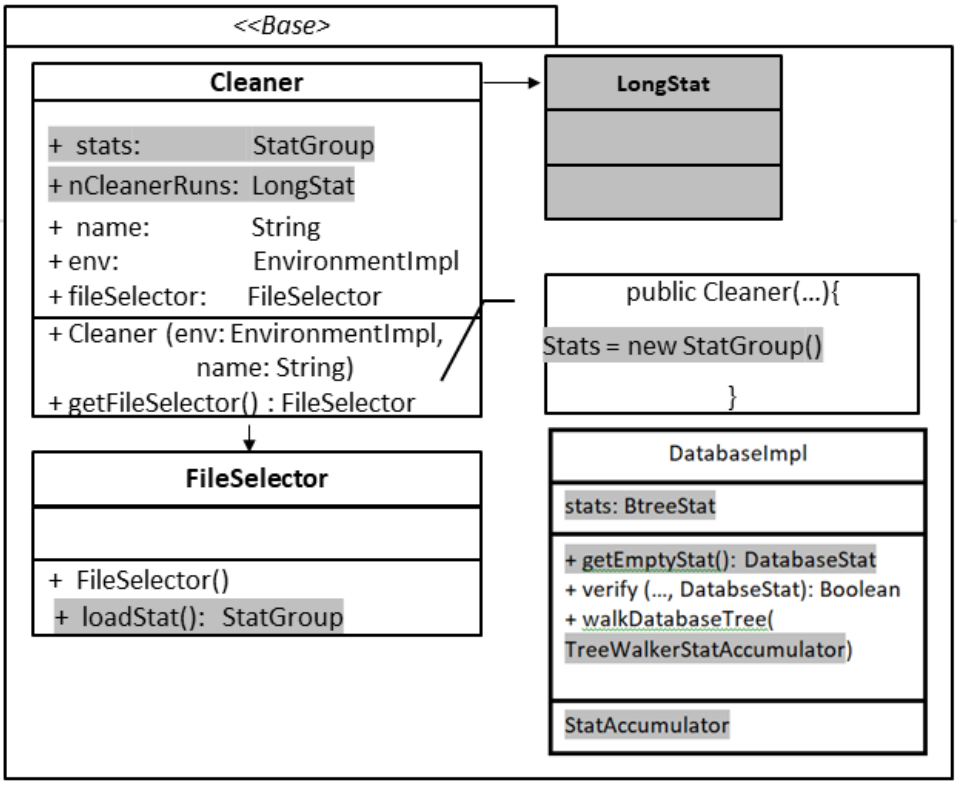


Figure 3.7: UML package representing part of BDE code-asset before injecting variation with FOP

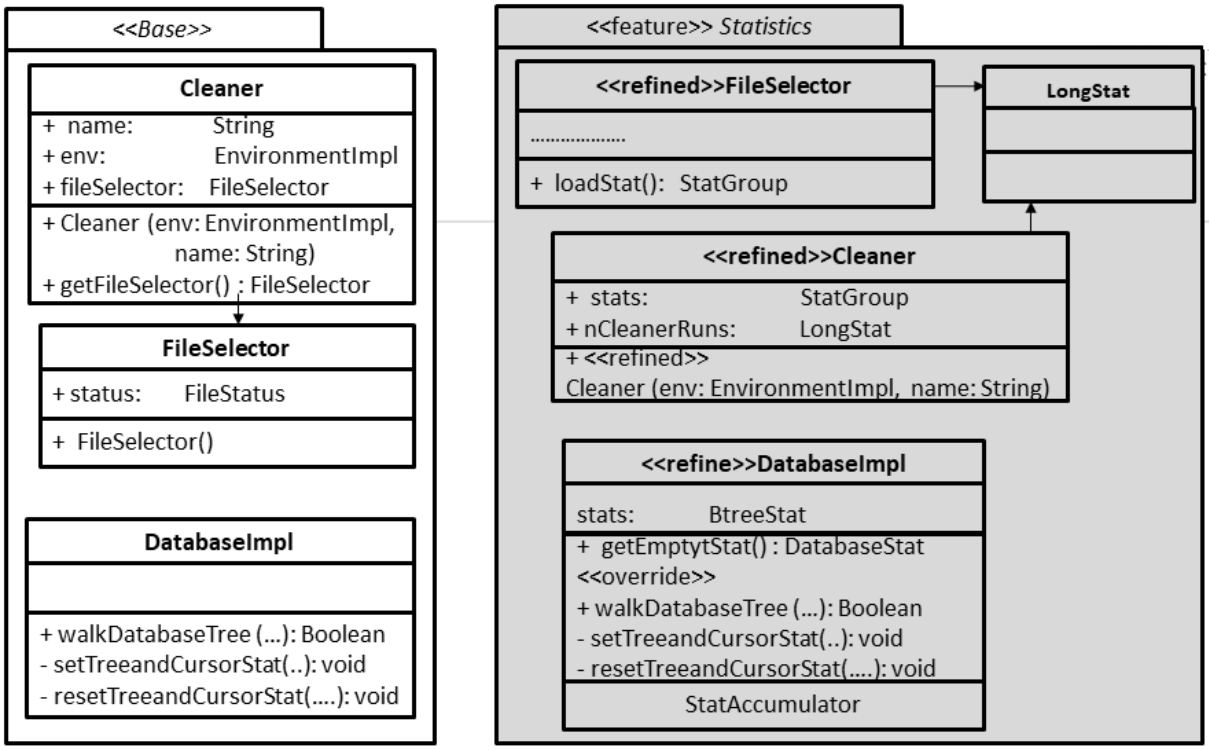


Figure 3.8: Injection of variation with to part of BDE code-asset with FOP

Proponents of Feature-Oriented Programming (FOP) are of the view that a feature is an increment in software functionality. As such, a feature module can be implemented as a *refinements* to a *base-program* (a basic implementation of a product line)[SB02, Bat05b, Pre97]. The refinements can be pure addition of new program elements or modification of existing ones. The concept of refining the base-program is in line with SPLE approaches in which mandatory features are implemented as monolithic program and variable features are implemented as extensions to the mandatory features of the base-program. Terms such as *kernel* [Gom05], *core-module* [SBB+10] are also used to describe the representation of all mandatory features, with which variable features may extend.

Fig.3.7 depicts a UML package containing program elements, representing the base-program, as in Fig.3.5, and in which the *Statistics* feature is still mandatory. Thus, the package contains the same classes as that of Fig.3.5. In Fig.3.7, the program elements of *Statistics* are shaded in grey. The callout shape, placed beneath the *LongStat* class, illustrates an object instantiation (*stats = new StatGroup*) within the constructor of the *Cleaner* class.

Fig.3.8 illustrates conceptual injection of variation with FOP. In the figure, the base-program and the *Statistics* feature occupy separate packages (illustrated with the UML package diagram and $\langle\langle feature \rangle\rangle$ stereotype). On one hand, all the program elements related to *Statistics* feature are removed from the package of *Base* (see the left-hand

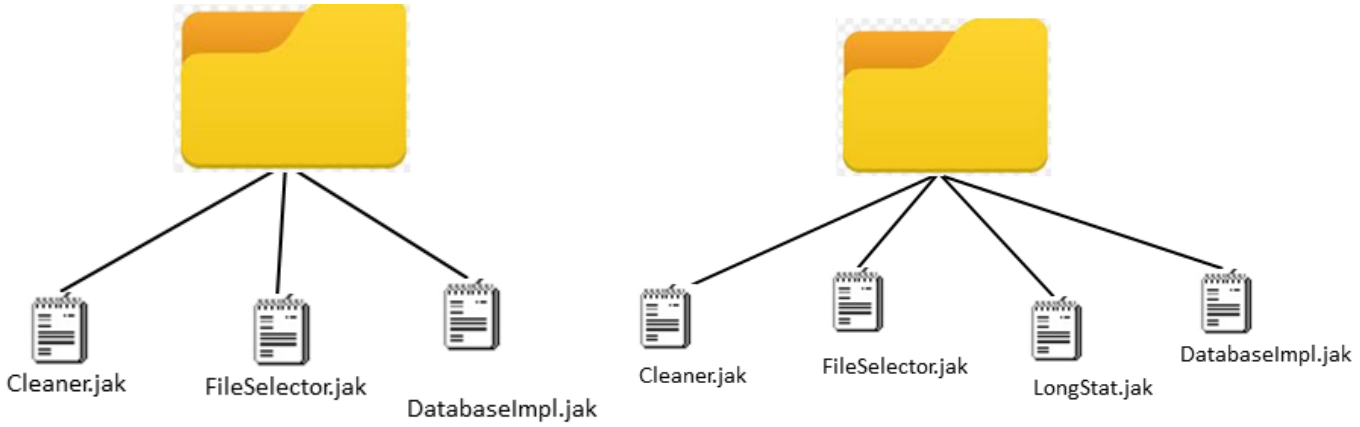
side of Fig.3.8). On the other hand, *Statistics*, as a variable feature, modifies program elements of the *Base* using a series of refinements (see the right-hand side of Fig.3.8). The refinements are illustrated with $\langle\langle refined \rangle\rangle$ stereotype and include the addition of the following program elements that were removed from the base-program: the method, *loadStats()*, in the *FileSelector* class (the top class in the *Statistics* package); the attributes (*stats* and *nCleanerRuns*) in the *Cleaner* class; the static inner class *StatAccumulator* in the *DatabaseImpl* class. The refinements also include constructor modification of the same *Cleaner* class;

Other forms of the refinements is overriding a method which is of two forms: (i) overriding an existing method to introduce the source-codes removed from the original method and (ii) introducing empty method in the base-program and overriding it in the decoupled feature - this is the case when a hook has to be introduced in the base-program and in which the variant feature extends. For example, in Fig.38, *Statistics* related source-codes are removed from the *walkDatabase* method of the *DatabaseImpl* class of the *Base* (i.e base-program). Thus, the method is overridden in the *DatabaseImpl* class of *Statistics* to re-introduced the removed source-codes that are relevant only to *Statistics*. This is the case of override (i) above. In contrast, *setTreeandCursorStat(...)* and *resetTreeandCursorStat(...)* methods in the *DatabaseImpl* of *Base* were introduced mainly as hooks to be extendend by their counterparts in the same *DatabaseImpl* of *Statistics*. This is the case of ovverride (ii) above.

In summary, *Statistics* refines *Base* when the feature is selected but everything in the *Statistics* package is not included in the product configuration if the corresponding feature is not selected.

Jak is a Java language extension designed to reify FOP in the implementation. With the Jak language, each feature module is confined to a separate directory structure known as *containment hierarchy*. For example, Fig.3.9a and Fig.3.9b depict the containment hierarchies for *Base* and the *Statistics* respectively. The duplicated classes, *Cleaner*, *FileSelector*, and *DatabaseImpl*, in the *Statistics* containment hierarchy contain only the modifications of the original classes of the *Base*.

FOP is one of the prominent techniques proposed to implement flexible variation in SPLE[SB02, Pre97, Bat05b]. In this study, we used FeatureIDE[TKB+14], an eclipse plugin that integrated the Jak language compiler and the tools for the synthesis and the translation from Jak to Java. From the domain implementation technique perspective, FOP approach is compositional. Thus, the composition of program elements of the selected features is gradual refinements of the base-program and the composed program elements are then transformed into target programming language (Java in our example). Next, we discuss the three cases of modifiability.



(a) Containment hierarchy of *Base* in which program elements of *Statistics* are removed from the 3 classes

(b) Containment hierarchy of *Statistics* in which *LongStat* class is created and other 3 classes refined to form program elements of *Statistics*

Figure 3.9: Containment hierarchies for *Base* and *Statistics* in Jak language

Table 3.4: Repetitive Viscosity of decoupling of program elements that are exclusive to the *Statistics* feature in the *DatabaseImpl* class with FOP.

Technique	#C	#M	#V	Total
FOP	1	3	1	5

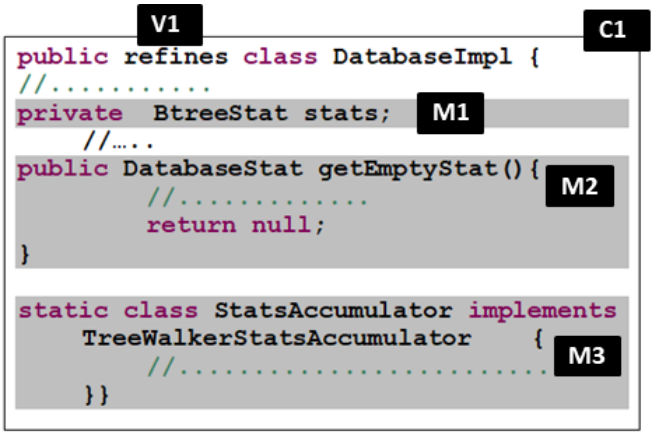


Figure 3.10: Decoupling of program elements that are exclusive to *Statistics* in the *DatabaseImpl* class with FOP

Modifiability Case 1: Decoupling program elements exclusive to feature with FOP

To decouple program elements that are exclusive to *Statistics*, in the *DatabaseImpl* class, with FOP, the class has to be re-created in the *Statistics*' containment hierarchy, which

Table 3.5: Repetitive Viscosity of decoupling intersections between the base-program and *Statistics* within the *walkDatabaseTree* method and within the constructor of the *Cleaner* class with the FOP technique.

Decoupling scenario	**RV**	**Remarks**
intersections within a method	moderate	requires the creation of extension points.
intersections within a constructor	moderate -	block move operations and simple refinements

makes #C to be one (1). One (1) #V is required, to refine the *DatabaseImpl* class in *Statistics'* containment hierarchy. The three program elements that are exclusive to *Statistics* have to be removed from the base-program and moved to the refined class (see Fig.3.10): this makes #M to be three (3). Thus, the total RV of decoupling program elements that are exclusive to *Statistics* in *DatabaseImpl* class is five (5) (see Table 3.4).

Modifiability Case 2:decoupling intersection within a method with FOP
To decouple intersection within a method with FOP and when the intersections are neither at the beginning nor at the end of the method, but in the middle as in the *walkDatabaseTree* method of the *DatabaseImpl* class (see Fig.3.11a), the points of intersections have to be extracted into empty methods, known as *hooks* or *hotspots*[KAB07]. Fig.3.11a shows two hooks created (C1 and C2) and two placeholders for the hooks are also created (C3 and C4).

The C5 in Fig.3.11b denotes the re-creation of the *walkDatabaseTree* in the new containment hierarchy. In addition, the lines of statements that are for *Statistics* are moved from the base-program to the newly created *walkDatabaseTree*(see M1-M3 in Fig.3.11b). The mechanisms used are the overriding of the *walkDatabaseTree* method, using the construct *Super()* (V1 in Fig.3.11b) and overriding of the two hook methods (V2 and V3 in Fig.3.11b). Thus we rate the RV of FOP for decoupling intersection within a method as moderate (see the first row of Table 3.5).

Modifiability Case 3: decoupling intersection within a constructor with FOP
To decouple intersection within the constructor of the *Cleaner* class with FOP, the *Cleaner* class has to be re-created, in the *Statistics'* containment hierarchy (Fig.3.12), which makes the #C to be one (1). The program elements that are related to *Statistics* have to be removed from the legacy base-program and moved to the new class as one block operation (M1 in Fig.3.12). Only two (2) #Vs are required, in form of modification, to the *Cleaner* class itself (V1 in Fig.3.12a) and its constructor(V2 in Fig.3.12). Thus, we rate RV of FOP moderate -. The minus (-) means that the moderation is closer to low due to the few number of move and refine operations (see the second row of Table 3.5).

(a) re-organized *walkDatabaseTree* method in which program elements of *Statistics* are removed in preparation for refinement with FOP

(b) the *walkDatabaseTree* method in which the removed program elements of *Statistics* are re-introduced with FOP refinement

Figure 3.11: decoupling intersection within a method in FOP

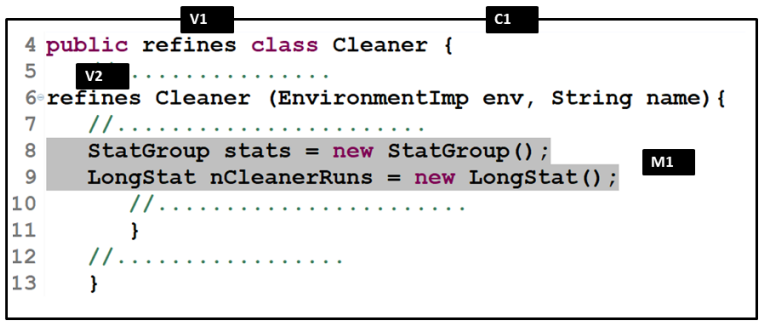


Figure 3.12: decoupling intersection within a constructor with FOP

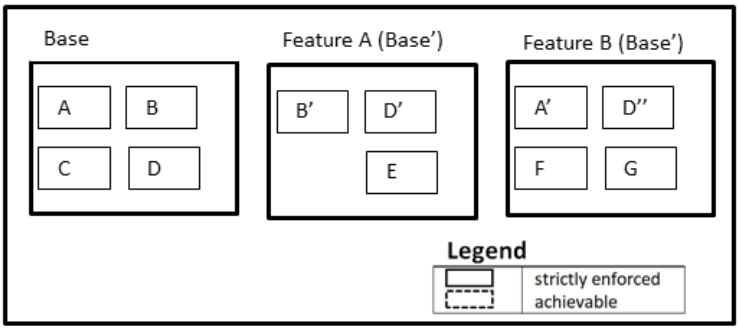


Figure 3.13: Modularity with FOP

Modularity and support for variations in feature binding time in FOP

FOP support both the horizontal and the vertical modularity as conceptually depicted in Fig.3.13. In the figure, program elements of each feature are confined to a dedicated containment hierarchy. As such, the horizontal modularity is strictly enforced(see Fig.3.13). In addition, the typed program elements, donated with letters in Fig.3.13, are also strictly bounded in separate files. Thus, vertical modularity is also strictly enforced. For example, in Fig.3.13, program element D' is a refinement of program element D of the base-program; it is only for *feature A* and is traced to a separate file under the containment hierarchy of *feature A*. Likewise, a separate refinement of program element D, denoted as D", is only for *feature B* and is traced to a separate file under the containment hierarchy of *feature B* [Bat05b].

With respect to support for flexible binding time, FOP supports the two modes of feature binding to executing context: (i) static binding mode and (ii) dynamic binding mode. With static binding mode, program elements and their refinements are statically composed. With dynamic mode, the *mixin* inheritance style is used to compose program elements with their refinements. The two different binding modes are supported in Jak with two separate tools: *jampack* for the static binding mode and *mixin* for the dynamic binding mode. However, from the perspective of feature binding to a product, the variability has to be resolved before compilation at pre-deployment time[Bat05b]. That is, FOP does not support feature binding after compilation at either deployment or post-deployment time.

3.2.3 Aspect Oriented Programming (AOP) with AspectJ

Aspect Oriented Programming (AOP) is a programming paradigm for modularization of crosscutting concerns. A concern is a specific requirement of a software product. In SPLE, variable features can be implemented as concerns that are separable from the base-program [CLK08, FCS+08, KAB07, ARR+16]. Thus, the set of mandatory features

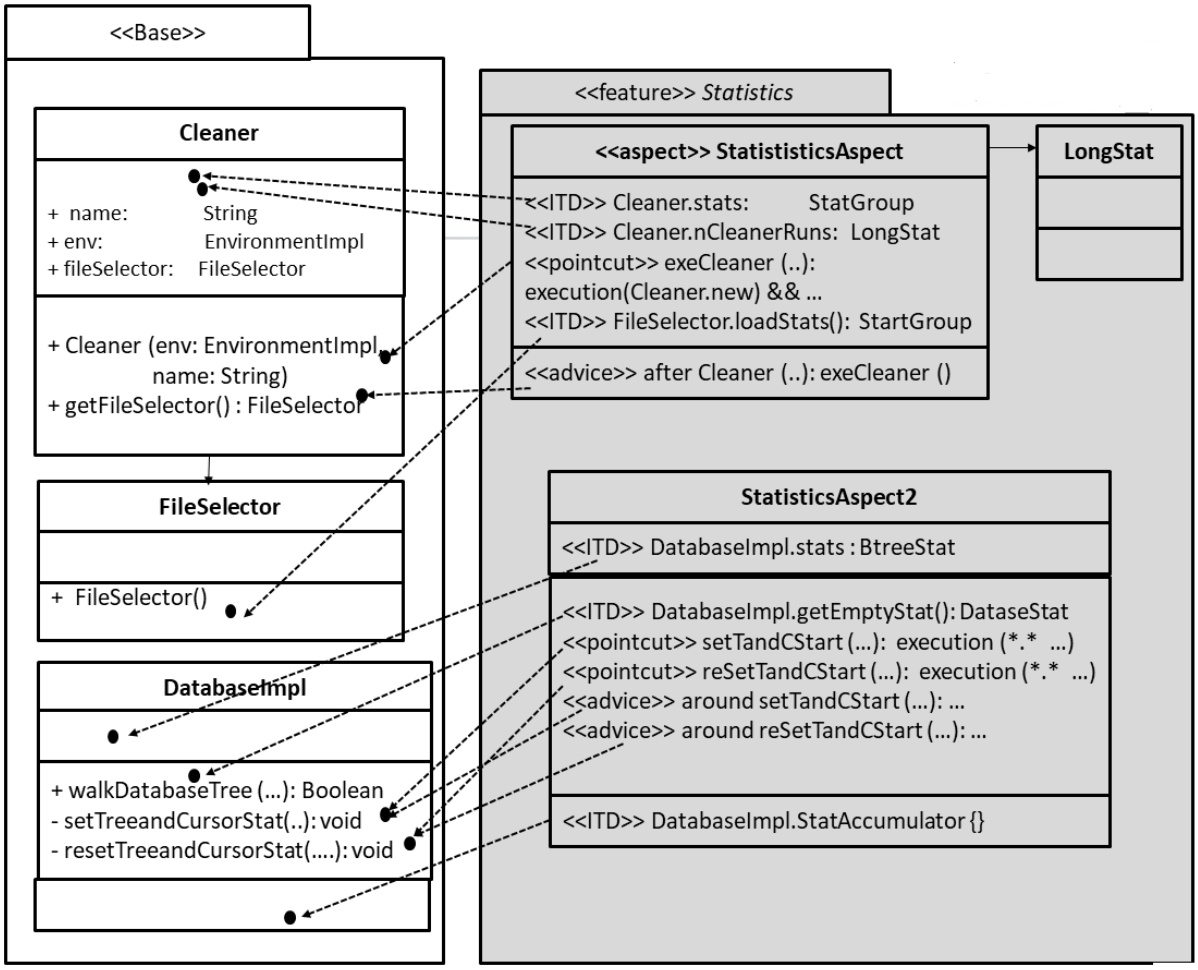


Figure 3.14: injection of variation w with AOP

Table 3.6: Repetitive Viscosity of decoupling of program elements that are exclusive to the *Statistics* feature in the *DatabaseImpl* class with FOP.

Technique	**#C**	**#M**	**#V**	Total
AOP	1	3	3	7

are implemented as a base-program.

AOP construct, *aspect*, is used to encapsulate a concern, which may be a feature in SPLE. Aspect extends (*advice*) the behaviour of a base-program at some points (*join points*). Similarly, an AOP construct, *pointcut*, is used to define the specific join points where the aspect extends (advice) in the base-program. In addition, an aspect may introduce new program elements or new inheritance hierarchies to program elements in the base-program using the *intertype declaration* mechanism.

For AOP, we do not show the structure of program elements, representing the base-program, where the *Statistics* feature is still mandatory because it is the same with the structure depicted in Fig.3.8 in the FOP discussion.

Fig.3.14 shows the use of AOP to inject variation and make *Statistics* optional. In the figure, *StatisticsAspect* (the grey box at the right-hand side of the figure) injects additional attributes (*stats* and *nCleanerRuns*) to the *Cleaner* class and the method, *loadStats*, to the *FileSelector* class. Similarly, *StatisticsAspect2* at the bottom left of the figure injects the attribute *stats*, the method *getEmptyStat()*, and the static inner class *StatAccumulator* to the *DatabaseImpl class*. The injection of attributes and methods is achieved through the intertype declaration mechanism (represented as the $\langle\langle ITD \rangle\rangle$ stereotype in Fig.3.14).

Furthermore, *StatisticsAspect* modifies the construction of the *Cleaner* object (illustrated using $\langle\langle advice \rangle\rangle$ stereotype) through the joint point captured by the pointcut (illustrated using $\langle\langle pointcut \rangle\rangle$stereotype), *exeCleaner*. Similarly, the *StatisticsAspect2* modifies the two empty methods introduced to the *DatabaseImpl* class hooks using two point cut declarations and two around advices. In this work, we use AspectJ [Lad03], one of the popular Java-based AOP.

When selected, the separated feature (*Statistics* in this example) is composed with the base-program using aspect weaving mechanism. The actual weaving may be done at different phases of the product lifecycle. From the feature binding perspective, the AOP approach is also compositional. The original intention of AOP was not to separate features in SPLE. However, there is increasing interest in using AOP to support flexible variations and binding times [KAB07, ARR+16, CRE08, AM04].

Table 3.7: Repetitive Viscosity of decoupling intersections between the base-program and *Statistics* within the *walkDatabaseTree* method and within the constructor of the *Cleaner* class with the AOP technique.

Decoupling scenario	RV	Remarks
intersections within a method	moderate +	requires the creation of extension points and several language constructs.
intersections within a constructor	moderate +	aspect has to have several references to targeting type in intertype declaration.

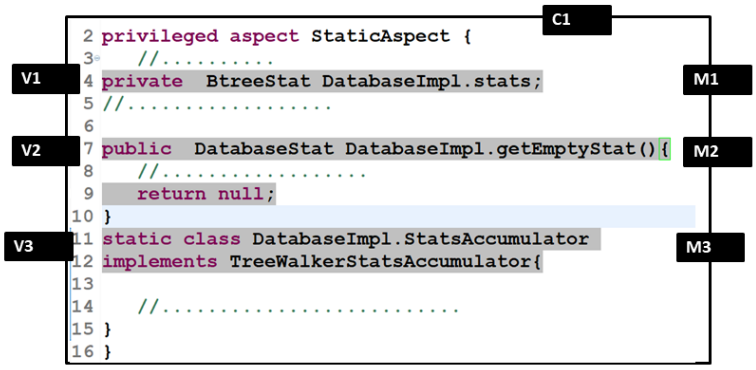


Figure 3.15: Decoupling of exclusive program elements with AOP

Modifiability Case 1: decoupling program elements exclusive to feature with AOP

To decouple program elements that are exclusive to *Statistics,* in the *DatabaseImpl* class, With AOP, a new aspect has to be created which makes #C to be one (1). The three program elements that are exclusive to *Statistics* have to be removed from the base-program and moved to the new aspect: this makes #M to be three (3) (see Fig.3.15). The new aspect of the *Statistics* feature injects (through intertype declaration) each of the moved program elements to the *DatabaseImpl* class using the prefix of the class (e.g. *DatabaseImpl.stats*), which makes #V to be three (3) (see Table 3.6).

Modifiability Case 2: decoupling program intersection within a method with AOP

To decouple intersections with a method with AOP and when the intersections are in the middle of the method, the same extractions of hooks that are required in FOP are also required in AOP. In addition, several language mechanisms have to be used. Fig.3.16 shows the mechanisms used in AOP: two pointcut declarations (V1 and V2 - not always needed when anonymous advices are used); two advices (V5 and V6); movement of program elements related to *Statistics* from the base-program to the new aspect (M1, M2, and M3); reference to *DatabaseImpl* (V3 and V4). Further, 6 additional language constructs are required (*pointcut, execution, target, within, around and proceed*). Thus, similar to

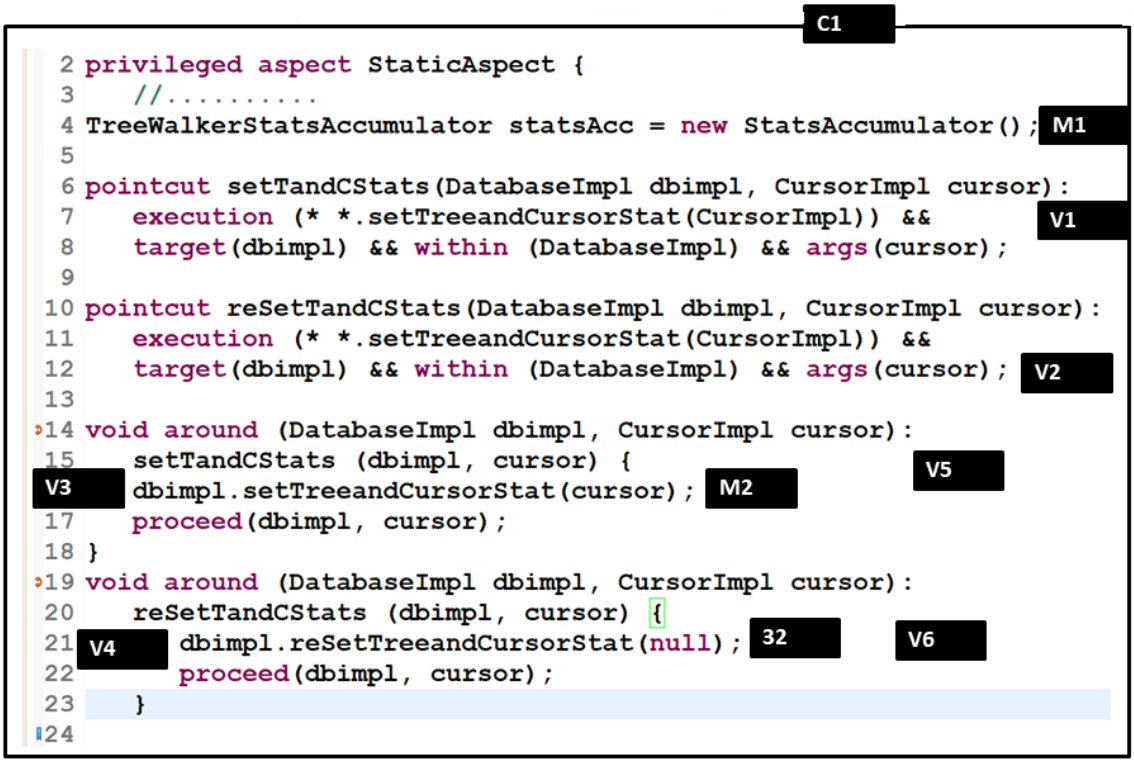


Figure 3.16: Decoupling intersection within a method with AOP

FOP, we rate the RV of FOP for decoupling intersection within a method as moderate +. The + sign means that the moderation is tending towards high (see the first row of Table 3.7 for the summary).

Modifiability Case 3: decoupling intersection within a constructor with AOP

To decouple intersection within the constructor of the *Cleaner* class with AOP, a pointcut and advice have to be created in the new aspect. The program elements that are related to *Statistics* have to be removed from the legacy base-program and moved to the *Statistics* aspect (M1-M.. in Fig.3.17). The aspect of the *Statistics* feature uses the reference to the *Cleaner* object severally (V1-V.. in Fig.3,17) in the intertype declaration. Thus, we rate RV of AOP moderate because of the several uses of the reference to the *Cleaner* class. The plus (+) means that the moderation is closer to high (see the second row of Table 3.7 for the summary).

Modularity and support for variations in feature binding time in AOP

AOP supports vertical modularity, the horizontal modularity can be achieved but it is not strictly enforced (depicted as the dotted-triangle in Fig.3.18). For example, a package system in Java can be used to organize program elements that are exclusive to

```
40 pointcut executingCleaner(Cleaner cl):                    V1
41    execution (Cleaner.new())&&  target(cl);
42 after (Cleaner cl): executingCleaner (cl){
43    //.....................                                 V1
44   V3  cl.stats = new StatGroup(); M1
45   V4  cl.nCleanerRuns = new LongStat(); M2
46   V..    //.....................                  V..
47 }
```

Figure 3.17: Decoupling intersection within a constructor with AOP

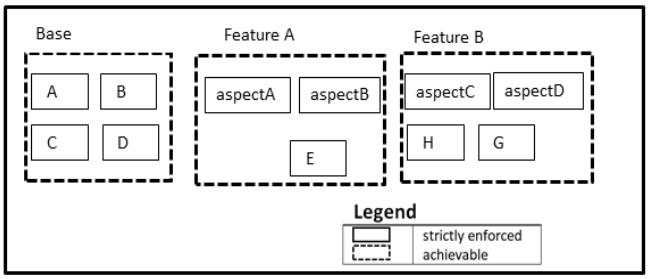


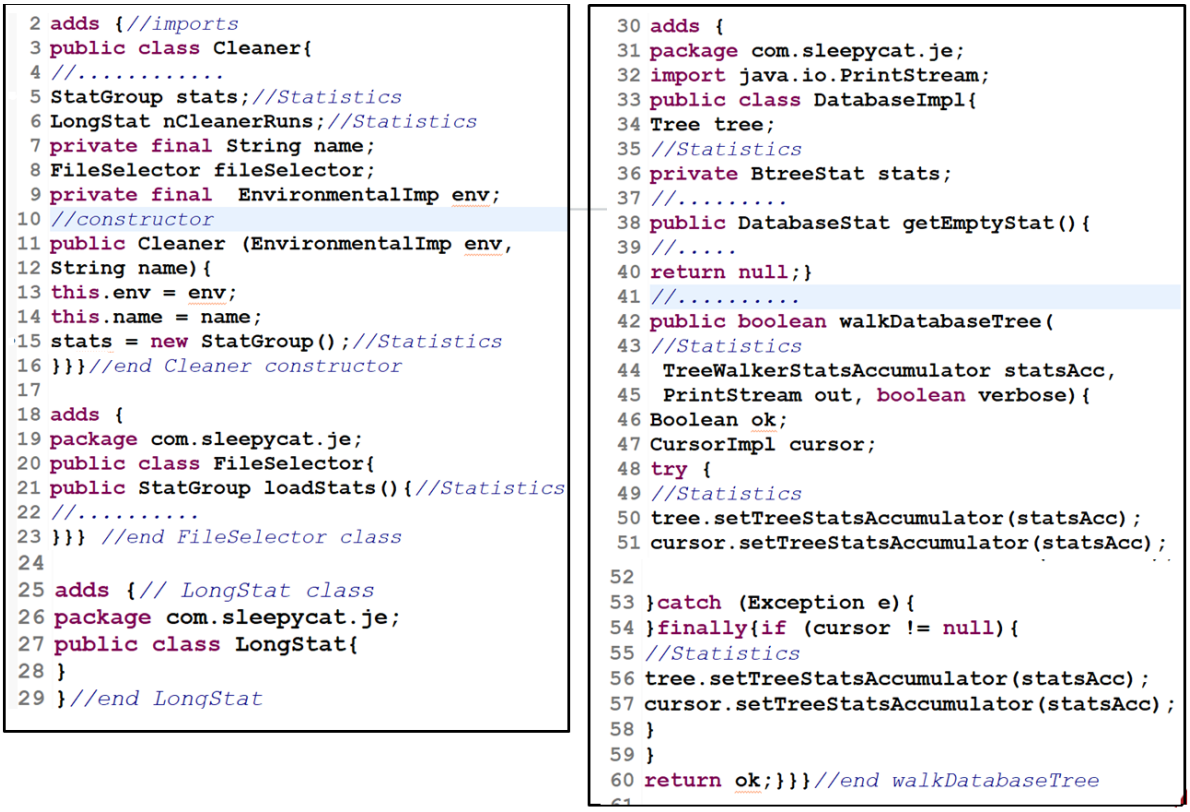
Figure 3.18: Modularity with AOP

feature. Aspects will then have to be used to encapsulate program elements intersecting with other features.

AOP supports binding of feature to a product at two phases: The first phase, *compile-time weaving phase*, corresponds to the binding of a feature at pre-deployment time. The second phase, *load-time weaving phase*, roughly corresponds to the binding of a feature at a deployment phase. Similarly, the binding can also be delayed by externalizing the feature configuration (e.g.using XML file).

3.2.4 Delta Oriented Programming (DOP) with DeltaJ 1.5

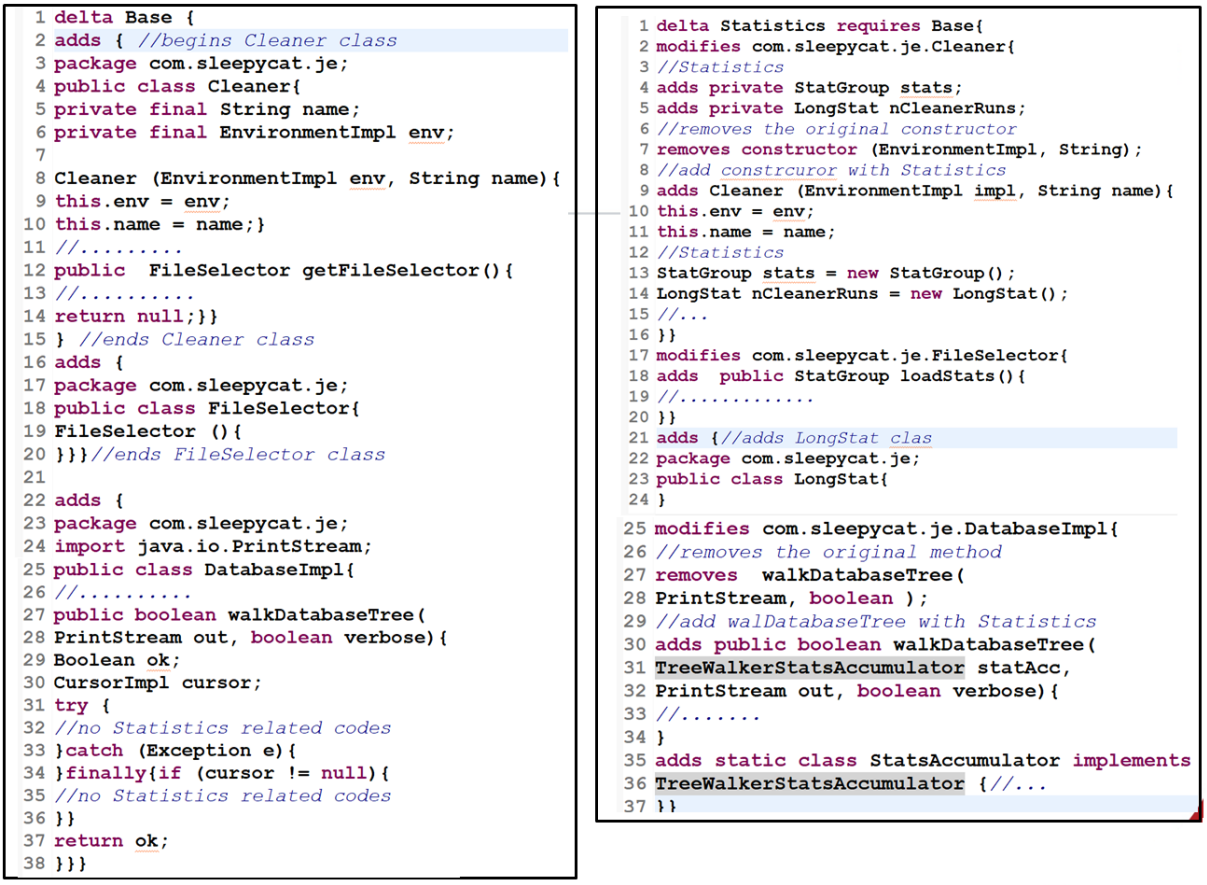
Delta Oriented Programming (DOP) is also a language-based implementation technique proposed to support flexible variations[SBB+10]. In DOP, a basic implementation of a software system is known as a *core*, and features are implemented as *deltas* to the core program. We stick with the term base-program for consistency. A delta is a unit of specification on how the based-program should be changed. Fig.3.19 depicts delta specifications representing the base-program, as in Fig.3.5 and Fig.3.7, and in which the*Statistics* feature is still mandatory. In the figure, the program elements of *Statistics* are shaded in grey.



```
adds {//imports
public class Cleaner{
//............
StatGroup stats;//Statistics
LongStat nCleanerRuns;//Statistics
private final String name;
FileSelector fileSelector;
private final  EnvironmentalImp env;
//constructor
public Cleaner (EnvironmentalImp env,
String name){
this.env = env;
this.name = name;
stats = new StatGroup();//Statistics
}}}//end Cleaner constructor

adds {
package com.sleepycat.je;
public class FileSelector{
public StatGroup loadStats(){//Statistics
//..........
}}} //end FileSelector class

adds {// LongStat class
package com.sleepycat.je;
public class LongStat{
}
}//end LongStat
```

```
adds {
package com.sleepycat.je;
import java.io.PrintStream;
public class DatabaseImpl{
Tree tree;
//Statistics
private BtreeStat stats;
//.........
public DatabaseStat getEmptyStat(){
//.....
return null;}
//..........
public boolean walkDatabaseTree(
//Statistics
 TreeWalkerStatsAccumulator statsAcc,
 PrintStream out, boolean verbose){
Boolean ok;
CursorImpl cursor;
try {
//Statistics
tree.setTreeStatsAccumulator(statsAcc);
cursor.setTreeStatsAccumulator(statsAcc);

}catch (Exception e){
}finally{if (cursor != null){
//Statistics
tree.setTreeStatsAccumulator(statsAcc);
cursor.setTreeStatsAccumulator(statsAcc);
}
}
return ok;}}}//end walkDatabaseTree
```

Figure 3.19: Delta specifications of *Base* before variability injection with DOP



```
delta Base {
adds { //begins Cleaner class
package com.sleepycat.je;
public class Cleaner{
private final String name;
private final EnvironmentImpl env;

Cleaner (EnvironmentImpl env, String name){
this.env = env;
this.name = name;}
//.........
public  FileSelector getFileSelector(){
//..........
return null;}}
} //ends Cleaner class
adds {
package com.sleepycat.je;
public class FileSelector{
FileSelector (){
}}}//ends FileSelector class

adds {
package com.sleepycat.je;
import java.io.PrintStream;
public class DatabaseImpl{
//..........
public boolean walkDatabaseTree(
PrintStream out, boolean verbose){
Boolean ok;
CursorImpl cursor;
try {
//no Statistics related codes
}catch (Exception e){
}finally{if (cursor != null){
//no Statistics related codes
}}
return ok;
}}}
```

```
delta Statistics requires Base{
modifies com.sleepycat.je.Cleaner{
//Statistics
adds private StatGroup stats;
adds private LongStat nCleanerRuns;
//removes the original constructor
removes constructor (EnvironmentImpl, String);
//add constrcuror with Statistics
adds Cleaner (EnvironmentImpl impl, String name){
this.env = env;
this.name = name;
//Statistics
StatGroup stats = new StatGroup();
LongStat nCleanerRuns = new LongStat();
//...
}}
modifies com.sleepycat.je.FileSelector{
adds  public StatGroup loadStats(){
//.............
}}
adds {//adds LongStat clas
package com.sleepycat.je;
public class LongStat{
}
modifies com.sleepycat.je.DatabaseImpl{
//removes the original method
removes  walkDatabaseTree(
PrintStream, boolean );
//add walDatabaseTree with Statistics
adds public boolean walkDatabaseTree(
TreeWalkerStatsAccumulator statAcc,
PrintStream out, boolean verbose){
//.......
}
adds static class StatsAccumulator implements
TreeWalkerStatsAccumulator {//...
}}
```

Figure 3.20: Injection of variation with with DOP

To inject variation in DOP, a feature implementation is specified in a separate delta. The left box of the Fig.3.20 depicts specification of *Base* in DOP, with program elements of *Statistics* removed. The right box of the Fig.3.20 depicts specification of a delta (changes) to *Base* in the implementation of *Statistics*.

Similar to FOP, the composition of program elements of the selected features is gradual modifications of the base-program and the composed program elements are then transformed into target programming language (Java in this case).

DOP is an emerging variability implementation technique that supports changes to a based-program and the changes are not only additions of program elements but also the removal of program elements (positive and negative variability). In this work, we used DeltaJ [Del17] implemented in Xtext - a language implementation framework [Xte17].

Table 3.8: Repetitive Viscosity of decoupling of program elements that are exclusive to the *Statistics* feature in the *DatabaseImpl* class with DOP.

Technique	**#C**	**#M**	**#V**	Total
DOP	1	3	3	7

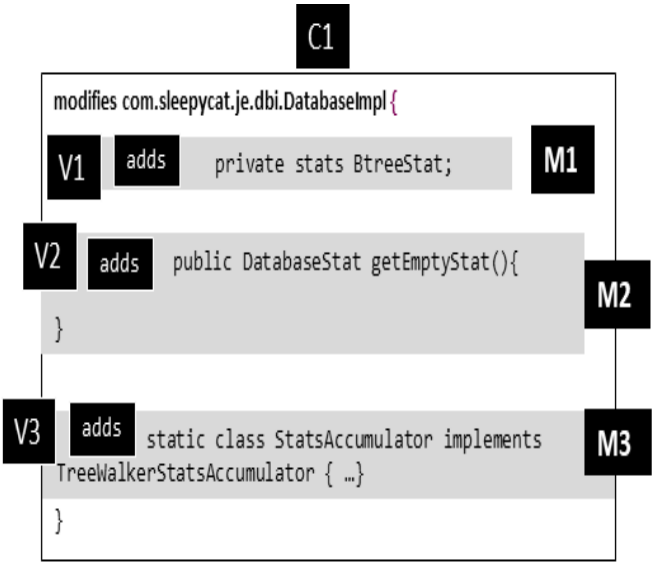


Figure 3.21: Decoupling of exclusive program elements with DOP

Modifiability Case 1: decoupling program elements exclusive to feature with DOP

To decouple program elements that are exclusive to *Statistics* in the *DatabaseImpl* class with DOP, a new delta specification has to be created, which makes the #C to be one (1). The three program elements that are exclusive to *Statistics* have to be removed from the base-program and moved to the new delta: this makes #M to be three (3) (see Fig.3.21). A separate *adds* specification is required for each of the three program elements which makes #V to be three (3) (current DOP implementation does not support the addition of a block of program elements as a single operation). Table 3.8 summarises the RV of decoupling program elements that are exclusive to the *Statistics* feature in the *DatabaseImpl* class.

Modifiability Case 2: decoupling program intersection within a method with DOP

DOP has limited language semantics for decoupling intersections within a method. The technique supports wrapping of an existing method only when both the wrapper and the wrapped methods have the same number of parameters. This is because DOP has only two constructs for modifying a method: (i) the *modifies* keyword to specify the full signature of the method to be changed and (ii) the *original* construct to reuse the

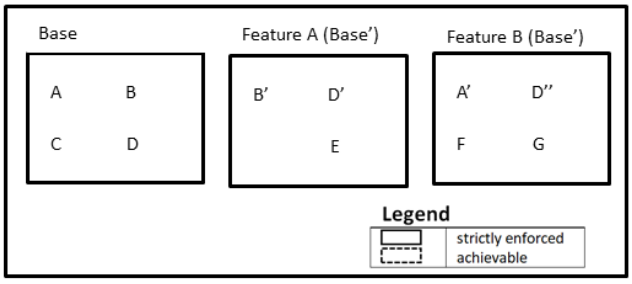


Figure 3.22: Modularity in DOP

Table 3.9: Repetitive Viscosity of decoupling intersections between the base-program and *Statistics* within the *walkDatabaseTree* method and within the constructor of the *Cleaner* class with the DOP technique.

Decoupling scenario	**RV**	**Remarks**
intersections within a method	high	two separate methods have to be created one without *Statistics* in the base-program and one with *Statistics* in a separate delta specification.
intersections within a constructor	high	two separate constructors have to be created one without *Statistics* in the base-program and one with *Statistics* in a separate delta specification.

```
27 public boolean walkDatabaseTree(
28 PrintStream out, boolean verbose){
29 Boolean ok;
30 CursorImpl cursor;
31 try {
32 //no Statistics related codes
33 }catch (Exception e){
34 }finally{if (cursor != null){
35 //no Statistics related codes
36 }}
37 return ok;
38 }}}
```

(a) Re-specified method without intersection in DOP

```
27 //removes the original method
28 removes  walkDatabaseTree(
29 PrintStream, boolean );
30 //add walDatabaseTree with Statistics
31 adds public boolean walkDatabaseTree(
32 TreeWalkerStatsAccumulator statAcc,
33 PrintStream out, boolean verbose){
34 //......
35 Boolean ok;
36 CursorImpl cursor;
37 try {//.....
38 tree.setTreeStatsAccumulator(statAcc);
39 cursor.setTreeStatsAccumulator(statAcc);
40 }catch (Exception e) {}
41 finally{ if (cursor != null){//....
42 tree.setTreeStatsAccumulator(null);
43 cursor.setTreeStatsAccumulator(null);
44 }}
45 return ok;}
```

(b) Re-specified method with the intersection in DOP

Figure 3.23: Decoupling intersection within a method with DOP

```
 6 //removes the original constructor
 7 removes constructor (EnvironmentImpl, String);
 8 //add constrcuror with Statistics
 9 adds Cleaner (EnvironmentImpl impl, String name){
10 this.env = env;
11 this.name = name;
12 //Statistics
13 StatGroup stats = new StatGroup();
14 LongStat nCleanerRuns = new LongStat();
15 //...
16 }}
```

Figure 3.24: Constructor modification with DOP

specification of the old method. Consequently, if a method modification involves changing a parameter in DOP, as in our example, two separate methods have to be created: one without *Statistics* (Fig.23a), in the base-program, and one with *Statistics* (Fig.23b) in a separate delta specification. Consequently, we rate the repetitive viscosity of decoupling intersections within a method with DOP as high(see the first row of Table 3.9)

3.2.4.1 Modifiability Case 3: decoupling interaction within a constructor with DOP

DOP does not support the direct modification of a constructor, and therefore, two separate constructors have to be created: one without *Statistics* in the base-program, and one with *Statistics* in a separate delta specification. Again, we rate the RV of DOP high because two separate constructors have to be specified(see the second row of Table 3.9).

Modularity and support for variations in feature binding time with DOP

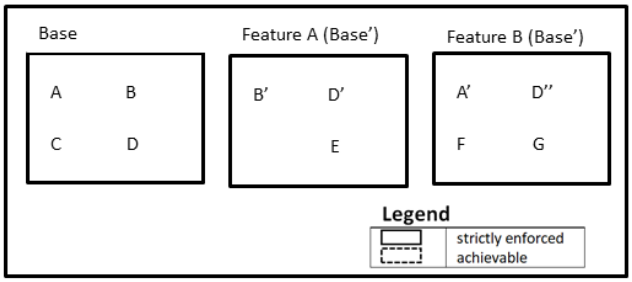


Figure 3.25: Modularity with DOP

DOP supports horizontal modularity as each feature can be traced directly to a separate delta specification. However, there is no hierarchical modularity in the sense that the type elements are flatly organized in a possibly large file of delta specification. The modularity in DOP is conceptually depicted in Fig.3.25. There is no hierarchical modularity because there is no boundary between program elements of different types within the same delta. The lack of hierarchical modularization may be detrimental to maintaining sizable feature specification. For example, when a feature module has more than 10 Kilo Lines of Code (KLOC), it is difficult to navigate to individual program elements of interest as we experienced during refactoring of *Statistics* feature.

On the support of flexible binding time, DOP is similar to PP. That is, feature binding has to be decided at pre-deployment time before the product code is generated but not at any time.

3.2.5 Comparison between the implementation techniques

Table 3.10: Comparison between the 4 techniques on decoupling of exclusive program elements

Technique	**#C**	**#M**	**#V**	Total
PP	0	0	3	3
FOP	1	3	1	5
AOP	1	3	3	7
DOP	1	3	3	7

Table 3.10 presents the Repetitive Viscosity of each of the techniques for decoupling the three program elements that are exclusive to the *Statistics* in the *DatabaseImpl* class. In this context, none of the language-based techniques beats PP in terms of modifiability.

Table 3.11: Pattern of RV of implementation techniques on program elements exclusive to a feature

Technique	**Program element Type**	**RV**	Total
PP	exclusive class	single annotation (#V = 1).	RV =1
	exclusive attribute	single annotation (#V = 1) or becomes part of block annotation (#V = 1/x).	1/x ≤ RV ≤ 1
	exclusive method	single annotation (#V = 1) or becomes part of block annotation (#V = 1/x).	1/x ≤ RV ≤ 1
FOP	exclusive class	move (#M =1) + create (#C=1).	RV =2
	exclusive attribute	move + refined (#M =1, #V =1) or move only(#M =1).	1 ≤ RV ≤ 2
	exclusive method	move + refined (#M =1, #V =1) or move only(#M =1).	1 ≤ RV ≤ 2
AOP	exclusive class	move (#M =1) + create (#C=1).	RV =2
	exclusive attribute	move + inject + create (#M =1, #V =1, #C=1) or move + inject only (#M =1, #C =1).	2 ≤ RV ≤ 3
	exclusive method	move + inject + create (#M =1, #V =1, #C=1) or move + inject only (#M =1, #C =1)(#M =1).	2 ≤ RV ≤ 3
DOP	exclusive class	move (#M =1) + create (#C=1).	RV =2
	exclusive attribute	move + adds + create (#M =1, #V =1, #C=1) or move + adds only (#M =1, #C =1).	2 ≤ RV ≤ 3
	exclusive method	move + adds + create (#M =1, #V =1, #C=1) or move + adds only (#M =1, #C =1).	2 ≤ RV ≤ 3

FOP has moderate modifiability compared to the other techniques while AOP and DOP have the lowest modifiability index.

Through this analysis, we observe a generic pattern of RV of each of the techniques on three types of exclusive program elements (exclusive class, exclusive method, and exclusive attributes). We present this pattern in Table 3.11.

For PP, referring to Table 3.11, the RV for decoupling exclusive class is one (1) (#V = 1). The RV for decoupling exclusive method and exclusive attribute can either be one (1) (#V = 1) or a fraction of one (#V = 1/x). It is one when no two program elements to be annotated are adjacent in the code-assets (i.e. each of the program elements has to have a separate annotation) and a fraction of one when program elements to be annotated are adjacent in the code-assets (i.e. a single annotation covers more than one program

elements).

For FOP, the RV for decoupling exclusive class is two (2): one RV for moving the class from the base program (#M = 1) and one RV for re-creating the class in the target containment hierarchy. The RV for decoupling exclusive method and exclusive attribute in FOP can either be two (2) or one (1). It is two when each of the methods or attributes is in a separate container class - there has to be move operation (#M =1) and the refinement of the container class (#V =1) for each of the program elements. It is one (1) when an element is only moved but no refinement is required(see Table 3.11).

Still on Table 3.11, the RV for decoupling exclusive class with AOP and DOP is two (2): one RV for moving the class from the base program (#M = 1) and one RV for re-creating the class, aspect or delta as the case may be. The RV for decoupling exclusive method and exclusive attribute in both AOP and DOP can either be three (3) or two (2). It is three when each of the methods or attributes is to be moved into a new aspect or delta - there has to be move operation (#M =1); intertype declaration in aspect or *adds* specification in delta (#V =1); the creation of aspect or delta container (#C=1). It is two when no new aspect or delta is to be created as a result of the decoupling.

We will return to the RV for decoupling exclusive program elements when we have an idea about the number of exclusive program elements for the *Statistics* feature in *Custom annotation* section (next section).

3.3 Custom annotations

In this section, we present the definition and application of custom annotations to make sense about the results of the repetitive viscosity discussed in the previous section. We defined custom annotations to use as *metadata* on program elements. We refer to the annotations as custom to distinguish them from the pre-existing reusable annotations (eg. #ifdef annotations). The custom annotations are used to indicate whether a program element is exclusive to a given feature or it is shared with one or more features (involved in an intersection). Program elements that may be annotated exclusively to a feature can be class, method or attribute. The intersection between one or more features can be marked on a class constructor or on a method.

3.3.1 Custom annotation definition

Listing 3.1: List of features marked with custom annotations

```
1: public enum FeatureSet {
```

```
2: BASE, STATISTICS, ......
3: }
```

Listing 3.1 depicts the list of pre-defined features specified as Java enumerated list. A program element can be marked as exclusive to one of the features in the list (see the full list in Table 3.1). We can also mark intersection, on program element, between two or more features in the list. The "BASE" at the beginning of the list represents the base-program.

Listing 3.2: Types of program elements marked with custom annotations

```
1: public enum ProgramEType {
2: EXCLUSIVE_CLASS, INTERSECTION_METHOD,..
3: }
```

Listing 3.2 is also Java enumerated list in which each entry indicates the type of program element and its relation with the feature being annotated. For example, EXCLUSIVE_CLASS indicates that the program element being annotated is a class and is exclusive to a particular feature.

Listing 3.3 depicts the definition of the custom annotation where the attributes are references to the enumerations from listing 3.1 and 3.2: The *name* attribute in line 3 takes one of the predefined features from listing 3.1; The *elementType* attribute in line 4 takes one of the pre-defined types of program elements from listing 3.2; The *intersectingFeatures* attribute in line 5 is an array that takes one or more features defined in listing 3.1 as elements. The default value of *intersectingFeatures* attribute is base-program (i.e. FeatureSet.BASE). That is, a feature intersects with at least the base-program.

Listing 4.3: Specification of the custom annotation

```
1: ...
2: public @interface Feature {
3: FeatureSet name();
4: ProgramEType elementType ();
5: FeatureSet [] intersectingFeatures ()
6: default {FeatureSet .BASE};
7: String [] descriptions() default {};
}
```

Listing 3.4 is an example of marking a program element with the custom annotation. In the listing of 3.4, the constructor of *Cleaner* class is marked as one of the intersection points between *Statistics* feature and the base-program.

Table 3.12: Exclusive and intersections of program elements fro 11 features selected from BDE

Feature name	exclusive class	exclusive method	exclusive attribute	intersection in constructor	intersection in method	Total
Checksum	6	0	16	6	16	**44**
Delete	0	14	18	8	18	**48**
Environment Lock	0	2	6	2	2	**12**
Evictor	4	28	4	0	10	**46**
Incompressor	2	12	8	0	8	**30**
IO	1	0	0	1	0	**2**
Loook ahead cache	1	0	0	0	3	**4**
Memory budget	2	38	14	2	40	**96**
NIO	1	0	0	1	0	2
Statistics	20	70	154	64	2	**310**
Truncate	0	8	0	0	0	**8**
Total	**37**	**172**	**210**	**84**	**99**	**602**

Listing 3.4: Example of marking a program element with the custom annotation

```
1: @Feature(name= FeatureSet.STATISTICS,elementType
2: =ProgramEType.INTESECTION_CONSTRUCTOR)
3: public Cleaner( .){..}
```

3.3.2 Custom annotation application and processing

We marked the eleven (11) features from BDE, summarised in Table 3.1, with the custom annotations.

We implemented a pre-processor for the custom annotations and collected statistics about the annotated code-asset at compile time. We sought to have an idea about the spread of exclusiveness and intersections to make sense of the results of the evaluations obtained from *Study execution* section. Table 3.12 presents the spreads of exclusiveness and intersections about the 11 annotated features. Each row in the table represents an entry for a single feature. Note that the exclusive program elements made-up the first three columns (i.e. exclusive class, exclusive method, and exclusive attribute).

From Table 3.12, observe that *Statistics* feature has the highest number of anno-

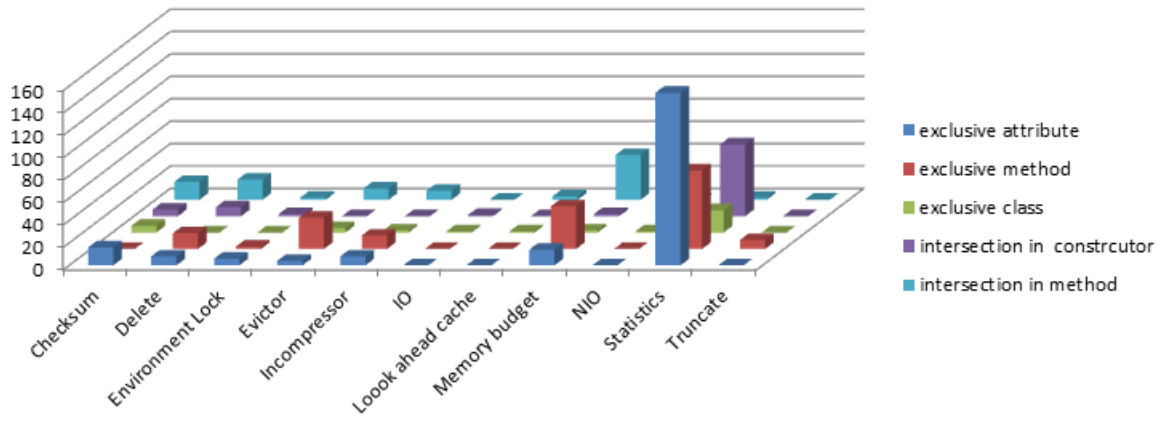


Figure 3.26: Spread of exclusive and intersection from 11 features

tated program elements with a total of 310, constituting about 52% of the total annotated program elements, while *IO* and *NIO* jointly have the least number of program elements, having a total of 2 each. This suggests that a feature in the implementation can be both small and large. Fig.3.26 depicts this information graphically.

Referring to the columns of Table 3.12, exclusive attributes constitute the major part of the annotated program elements with a total of 210, representing 35% of the total program elements annotated. On the other hand, exclusive classes (being the coarse-grained implementation units in OOP) are 37 in total, constituting only 6% of the total program elements in the 11 features annotated. This means that most parts of the features are traced to fine-grained program elements than coarse-grained program elements.

Extrapolating the generic pattern of RV presented in Table 3.11 to the exclusive program elements of the 11 features, we obtain the results of Table 3.13 which are also depicted graphically in Fig.3.27a. Recall that the RV of methods and attributes of all the techniques is not fixed. The RV of these program elements depend on some factors such as whether or not a block of methods or attributes can be annotated at once with PP. The other techniques also have their peculiar factors. This explains the use of less-or-equal sign ($\leq$) to reflect on the patterns presented in Table 3.13.

The same general pattern of RV can also be extrapolated to individual features. For example, Table 3.14 represents the estimated RV of each of the techniques on program elements exclusive to the *Statistics* feature and Fig.3.27b depicts the same information graphically.

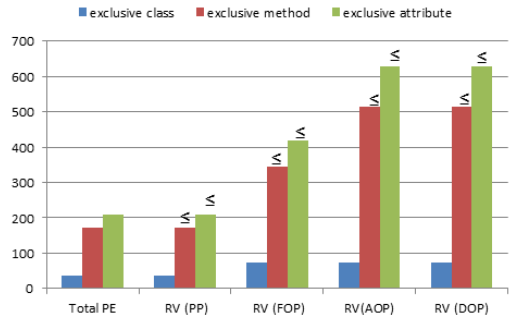
Apart from the extrapolation of exclusiveness, we can also obtain information about intersections between various features. This information can be useful in determining the order of precedence of applying aspects in AOP, for example, or order of refinements and modifications in FOP and delta respectively.

Table 3.13: RV for each of implementation techniques on exclusive program elements of 11 features

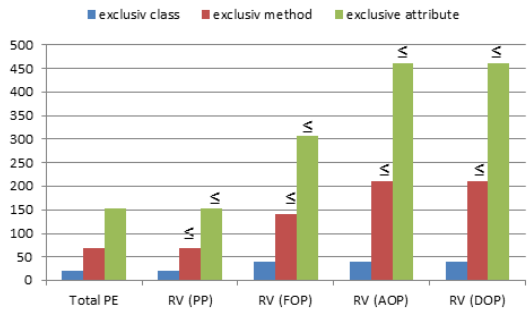
	exclusive class	exclusive method	exclusive attribute
Tota PE	37	172	210
RV (PP)	37	$\leq$ 172	$\leq$210
RV (FOP)	74	$\leq$ 344	$\leq$ 420
RV (AOP)	74	$\leq$ 516	$\leq$ 630
RV (DOP)	74	$\leq$ 516	$\leq$ 630

Table 3.14: RV for each of the techniques on exclusive program elements of *Statistics* feature

	exclusive class	exclusive method	exclusive attribute
Tota PE	20	70	154
RV (PP)	20	$\leq$ 70	$\leq$154
RV (FOP)	40	$\leq$ 140	$\leq$ 308
RV (AOP)	40	$\leq$ 210	$\leq$ 462
RV (DOP)	40	$\leq$ 210	$\leq$ 462



(a) RV for each of implementation techniques on exclusive program elements of 11 features



(b) RV for each of the techniques on exclusive program elements of *Statistics* feature

Figure 3.27: RV for each of the techniques on exclusive program elements

Table 3.15: Intersecting features

Feature	Intersections
Delete	Transaction, Cleaner
Environmental Lock	IO, NIO
Evictor	Incompressor
Memory Budget	Transaction, Evictor
Statitistics	Transaction, Evictor, Incompressor, Cleaner, Log, Cache, Checkpointer

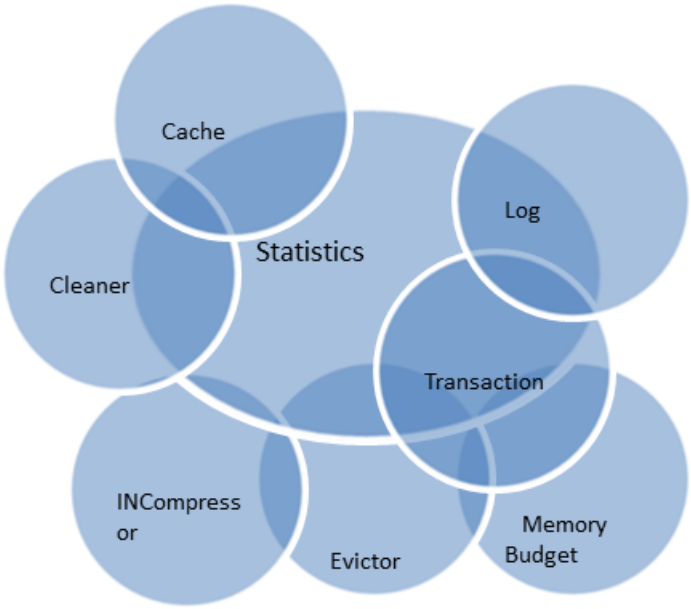


Figure 3.28: Example of intersecting features

Table 3.15 shows the summary of intersecting features. Note, some of the intersecting features in Table 3.15 are not among the 11 selected features but were picked while annotating the 11 features. Also, note that the intersection is reflexive, i.e. if feature A intersects feature B, implies feature B also intersects feature A. Fig.3.28 depicts an example of 7 features intersecting the *Statistics* feature.

Taken the extrapolated results together with the qualitative assessments for the intersection within a method and within a constructor, we can estimate the modifiability of each of the techniques. We can also conclude that none of the language-based implementations techniques is better than PP in terms of modifiability when injecting additional variations. In the same context, the code-asset of DOP is the least modifiable to inject additional variations. This is because, in addition to being comparable to AOP in most respects, it falls short on certain critical language semantics.

To conclude the discussion about the action research, in the next section, we compare our approach with similar action researches from the literature.

3.4 Comparison with similar action researches

In this section, we present similar action researches from the literature to highlight the novelty of our approach.

In an exploratory study, Murphy *et al* [MLWR01] investigated the flexibility of three language-based techniques when used to untangle features from a code-asset. All the three techniques, Hyper/J[TO], AspectJ[Lad03], and the authors' own technique, were designed for advanced separation of concern. The authors qualitatively characterized the effect the different techniques had on the structure of the code-asset and also characterized how to restructure the code-asset to untangle features with each of the techniques. Except for the AspectJ, the other techniques were not reported to be used for product line implementation. In addition, they did not use constituents of flexibility as criteria for the evaluation and did not consider programming languages specifically designed for the product line implementation.

Similarly, previous researchers that analysed features of Berkeley DB [KAB07, ARR+16] focused on the aspect-oriented refactoring of the features and they made no comparison with any other technique. For example, Chakravarthy*et al* [CRE08] proposed a combination of design patterns[GHJ+95] and aspect-oriented programming to achieve the flexibility of feature binding time. In their approach, a pattern encapsulates a variation point; variants features are implemented with a separate set of aspects, each set for a specific binding time.

In programmable logic controller domain, Bayrak *et al* [BOVH17] evaluated flexibility and maintainability of three process-oriented programming notations. The three different programming notations are Activity diagram, Statechart diagram, and Sequential function chart. Although they used a wide-range of criteria, including modifiability, none of the programming notations is used for product line implementation.

In summary, we differ with the previous approaches in the following ways: 1) we studied the properties of feature in the source -codes; 2) we covered more implementation techniques two of which were proposed to implement features in SPLE exclusively ; 3) we compared the modifiability of the techniques when injecting additional variations.

3.5 Chapter Summary and perspective

In this chapter, we explored properties features in the code-asset using Berkeley Database Engine (Java edition), as a case study. In the exploration, we observed that an intersection between program elements of different features is the major property of code-asset that affects the modifiability of injecting additional variations. Another property of a feature in a code-asset is exclusivity. That is, certain program elements exist for implementation of a certain feature exclusively.

The implication of this observation to practice is that, an unstable product line and with the tendency of emerging usage-contexts, should aim for implementation technique that limits the number of intersections between program elements of different features. Similarly, the implication of the observation to research is that, there should be subsequent investigations using multiple case studies in different software domains in order to improve the understanding of our findings.

Also, in this chapter, we report the selection and the evaluation of the flexibility of implementation techniques when injecting additional variations. Two of the selected techniques, FOP and DOP, were specifically designed for product line implementation.

Modifiability of DOP, in its current form, is consistently high because the technique lacks the necessary language semantics to support flexibility. In addition, although it supports horizontal modularity, program elements are flatly organized in a delta specification (no vertical modularity).

AOP trails DOP on modifiability mainly because of the high number of language constructs that have to be used to untangle a feature. Achieving horizontal modularity in AOP depends on the discipline of the developers because the technique does not enforce it. Modifiability of FOP is consistently moderate because few operations are involved when untangling a feature from a code-asset. Similarly, FOP has the best form of modularity because it supports both horizontal as well as vertical modularity.

Reflecting on our hypothesis, none of the techniques is better than PP in terms of modifiability but the approach has no form of modularity. Note, we are not claiming that PP is a better technique for product line implementation but hypothesized that, new techniques that are proposed to support flexible variations should also be better in modifiability when injecting additional variations than the PP - which is a classical implementation technique.

Except for FOP, the language-based initiative for flexible variations and binding time fall short when it comes to adapting code-asset to add additional variations. Another limitation of language-based approaches is the assumption that the variable features always extend a monolithic base-program. A variable feature may extend (intersect) other variable

features, and when only some of the variable features from the intersection are selected, the separation becomes a problem[KAR+09].

The next chapter (*Chapter four*) presents a systematic review of approaches proposed to directly or remotely support variations of feature binding time.

Chapter 4

Approaches for supporting variations of feature binding time: A systematic study

From the results of the action research in the previous chapter (*Chapter three*), we found that the modern language-based implementation techniques are not better than the classical technique in terms of modifiability. Similarly, even the techniques that were specifically proposed to implement features in SPLE (Feature-Oriented Programming and Delta-Oriented Programming) do not support variations in feature binding time even though they have improved modularity.

Given the limitations of the language-based implementation techniques with respect to the adaptability of reusable assets and the support for variations of feature binding time, we expanded our search with a systematic literature review to include approaches that directly or remotely support the variations of feature binding time.

Consequently, we used theories on conducting Systematic Mapping Study (SMS) and Systematic Literature Review (SLR) as guides. From the review, we found that the current support is limited in one of the following ways:

i A feature may have to be implemented more than once, each implementation for a specific binding time. Similarly, where model elements are used to represent a feature, more than one representation may have to be used, each representation for a specific binding time;

ii A product may have to be composed from too fine-grained model slices or source-codes implemented with aspects at a low-level of abstraction. In the former case, the model slices will have different target transformation depending on the target binding time. In the latter case, a separate set of aspects, in aspect-oriented programming, are used to implement a feature and each set is for a specific binding time;

iii Abstracting the two binding modes (static and dynamic) at the model level in model-driven development. The abstraction is limited to supporting only the existence of

alternative execution paths for possible binding of any of the variant features at runtime or absence of it to support static binding of only one of the variant features.

In the next section, we introduce the systematic review in relation to our research question. Subsequently, we discuss the review protocols and provide a broad overview and an in-depth narrative summary of the reviewed works. We end the chapter with the highlights of research gaps that form part of the basis of our research contributions.

4.1 Introduction

Systematic Mapping Study (SMS) is a secondary research aimed at investigating primary studies in a specific topic and broadly categorizing and summarizing the findings with the help of visual charts - otherwise known as maps. The maps highlight different facets of interest such as frequencies of publications, trends of publication over time and other information that help to elucidate the overall overview of the research area [K+07, PFMM08]. A Systematic Literature Review (SLR) is also a secondary study. However, a SLR is a step further when compared to the SMS. With SLR, a researcher provides an in-depth analysis and interpretations of the review findings, in addition to the broad overview. In this thesis, we use both the SMS and the SLR to complement each other as suggested by Petersen *et al*[PFMM08].

The systematic study fulfills our third research objective (O3). The findings from the study are significant for the identification of the research gap, and for the positioning of our contributions in a less partial manner. Equally important, in this chapter, is critical discussions of the pros and cons of the approaches and on what might need to be adjusted to improve them. The discussions can serve as a guide to intending practitioners who wish to select an approach appropriate to their setting. It is also useful to established practitioners who want an objective appraisal to fine-tune their approach.

More specifically, we decompose our third research question (RQ3) into the following review question:

4.1.1 Review questions

RVQ1. How much researches on variations of binding time are available from 1990 to date?

a. In what software domain?

RVQ2. What approaches do the existing researches proposed to manage variations of binding time

a. What outcome they seek to achieve?

b. What are the major limitations of the approaches?

In the next section, we discuss the review protocol.

4.1.2 Review protocol

As a scientific approach, both SMS and SLR have to have a prescribed protocol to aid replication of the study. Without the protocol, it would be impossible to replicate the review. Study replication is encouraged to either consolidate the existing finding or to dispute it. If the results from the repeated study are consistent with the results from the earlier study, then the results from the earlier study are validated. If, however, the results from the repeated study contradict the existing results, another investigation is necessary to attain a certain level of (no) confidence about the contentious results. Next, we describe our review protocol in line with the review questions.

4.1.3 Search terms

We derived the search terms from the review questions; from preliminary random searches into scholarly publications; and from the researcher's prior exposure to the literature that are relevant to supporting variations of binding time. We grouped similar search terms and joined them with logical OR while we joined the groups of dissimilar terms with logical AND to form a search query as follows:

(feature OR variability OR variation) AND ("binding time*" OR "configuration time") AND ("product line*" OR "product family" OR "system family")*

The choice of the search terms in the first group (feature OR variability OR variation*) is to capture papers reporting both feature-oriented and non-feature oriented approaches to supporting variations of feature binding time. Similarly, the search terms in the second group ("binding time*" OR "configuration time") and third group ("product line*" OR "product family" OR "system family") is to reflect the interchangeable terms used in the literature for binding time and product line respectively.

4.1.4 Search databases

We adapted the search query, based on specific database search format, and executed on the following resources:

Table 4.1: Inclusion/exclusion criteria

Inclusion criteria	**Exclusion criteria**
- The paper is written in English	- The paper is not written in English
- The paper is peer-reviewed and published as article in journals, conferences or workshops	- The paper is not peer-reviewed
- The paper is published between 1990-2019	- The paper focuses on behavioural variations at runtime exclusively (no pre-runtime considerations) or the paper focuses on design time variations exclusively.
- The paper discusses some aspects of binding time	- The paper does not discuss binding time

- ACM Digital Library;
- IEEE Xplore Digital Library;
- Science Direct
- Scopus

The four databases listed in this section are repositories of research papers from major workshops, conferences and journals articles of software engineering in general, and of software product line engineering in particular. Search through the databases returned voluminous research papers, as expected, with the majority of the papers irrelevant to the review questions. In the next section, we explain how we filtered the most irrelevant papers.

4.1.5 Selection strategy

We applied the criteria in Table 4.1 for the actual selection. We did the selections in three steps. In each step, we applied the inclusion/exclusion criteria. In the first step, we only read the abstract and sometimes the keywords and then decide whether to include or to exclude the paper for another screening in the second step. The first step is mainly meant to reduce the volume of irrelevant papers. In the second step, we applied an adaptive reading strategy [PFMM08](i.e., the scope of reading depends on the clarity of the paper's presentation and, therefore, differs from one paper to another). In the second step, a decision is made on whether to include or to exclude a paper mostly based on re-reading the abstract, reading the introduction and reading the conclusion section of the paper. However, sometimes we had to read other sections of the paper to make the

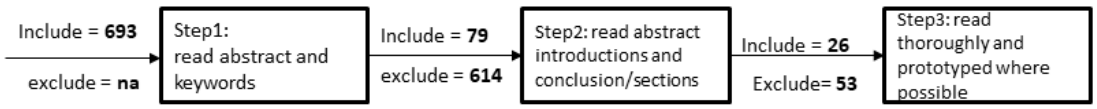


Figure 4.1: Paper selection process

Table 4.2: Papers selection per database

Database	**1st phase**	**2nd phase**	**3rd phase**
ACM Digital Library	15	10	4
IEEE Xplore	211	27	9
Science Direct	73	7	1
Springer	292	18	4
Scopus	92	11	4
Snowballed	10	6	2
Manually added	-	-	2
Total	**693**	**79**	**26**

decision. In the last and final step, we read the selected papers thoroughly, often several times, and prototyped with the proposed approach whenever possible. Fig.4.1 depicts the process of paper selection.

In Fig.4.1, each box represents a step in the selection process and the texts in the box describe the basis for including or excluding a paper to the next step. Also depicted in the figure is the total number of papers included/excluded at each step. Table 4.2 presents the breakdown of the papers selected from each of the sources.

In Table 4.2, each row represents the source of papers included in the review. The *1st phase* column represents the number of papers found in the first execution of the search query before the screenings; the *2nd phase* column represents the number of papers included in the second step of the paper selection; the *3rd phase column* represents the number of papers selected for the final selection step. In addition, we sourced the *snowballed* papers by following-up with references or citations of the paper under review. In case the snowballed paper is either an earlier or an updated version of the paper under review, we included only the latest version. In one case, the later version [TLSPS09] has a different first author and the research was in the context of a different application domain from that of the earlier version[SPLS+06]. In that case, we included both versions of the papers in the review.

We also manually added two (2) papers we know of but the papers were not returned from any of the searches. This is because the papers were published under research themes that are different from SPLE and their keywords and abstracts contain no pointer to variations. Nevertheless, the contents of the papers are relevant to variations of feature

binding time.

4.1.6 Data extractions

We extracted the data fields based on existing review templates and in consideration of the review questions. The following are the fields derived from the review template [PFMM08]: research approach, contributions proposed in the paper; and other auxiliary fields such as publication venues and publishers. Similarly, we derived the following fields from the review questions: proposed approaches; software domain in which the approaches were validated or evaluated; and the goals of the research. In total, we included twenty-six (26) papers for the final review.

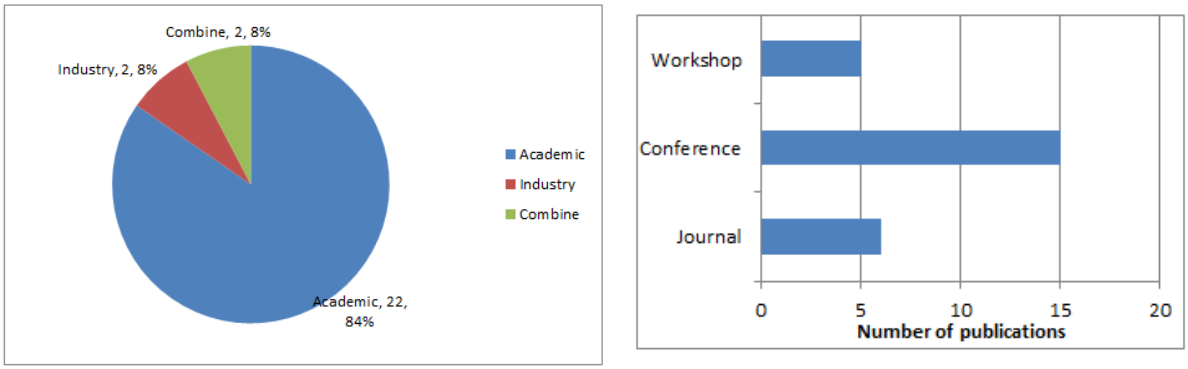
We end this section with an attempt to answer the first review question (RVQ1):

> We were able to find only twenty-six (26) published papers related to supporting variations of feature binding time.

4.2 Overview of the publications

Six (6) papers are journals articles, fifteen (15) are conference papers and the remaining five (5) are workshop papers (see Fig.4.2b). The papers appear in twenty-one (21) publication venues (workshops, conferences, and journals). In this context, *Software Product Line Conference (SPLC)* attracted the highest number of publications with a total of five (5) papers; *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)* and Journal of *Science of computer programming* has two (2) papers each. All the other venues have only one (1) paper each (see Fig 4.3).

Based on SCImago Journal Rank (SJR) indicator [GPGBMA10], one of the journals has reputation for being in the top 25% (first Quartile) of its subject category and the remaining journals are in the top 50%(second Quartile). In the conference category, among the 15 papers, 5 were presented at SPLC- the premier conference of SPLE researches; one (1) paper was presented at the top-tier Aspect-Oriented Development conference; four (4) papers were presented at second-tier conference; two (2) papers were presented at the bottom 50% (third Quartile) conference venues; three (3) papers were presented at the bottom 25% (fourth Quartile) conference venues. From the five (5) workshop papers, two (2) papers were presented at the VaMoS, the premier workshop for SPLE; 2 were published in Springer Lecture Note in Computer Science (LNCS) – a second-tier book series for computing; one paper (1) has a reputation of being third tier publication.



(a) authors' affiliation **(b)** paper categories

Figure 4.2: Publications by (a) affiliations and (b) paper categories

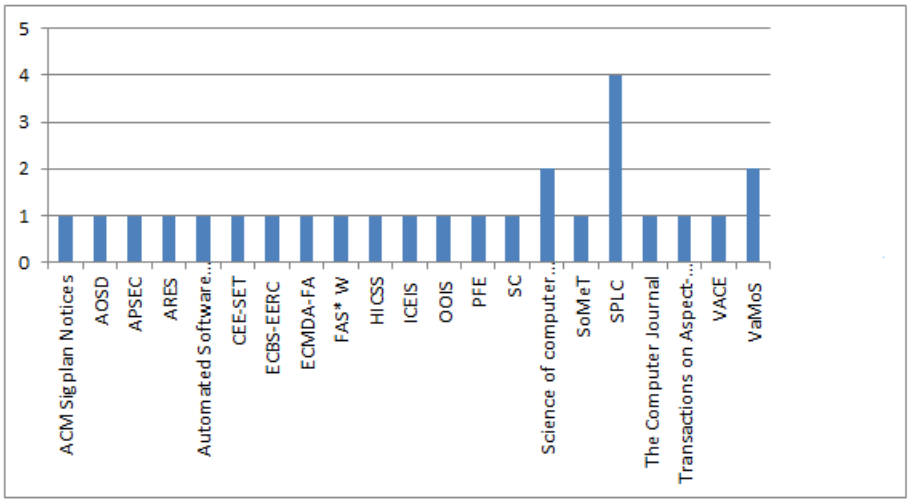


Figure 4.3: Publication venues

The graph in Fig.4.4 shows the publication trends over a period of two decades. Two interesting revelations from the graph are: 1) SPLE community recognized the challenge of variations of binding time long enough. 2) There are sparse publications despite the recognition of the problem. For example, no paper was published in the year 2000, 2003, 2005 and 2010. 2009 is the year with the highest number of published papers (with a total of 5 papers). 2016 is the year with the second-highest published papers (a total of 4 papers).

Using the research categories proposed in[WMMR06], each of the papers reflects one or more of the research approaches in Table 4.3.

Each of the papers reviewed pursues one or more research goals (horizontal axis of Fig.4.5). Fourteen (14) papers share the goal of supporting flexibility of feature binding

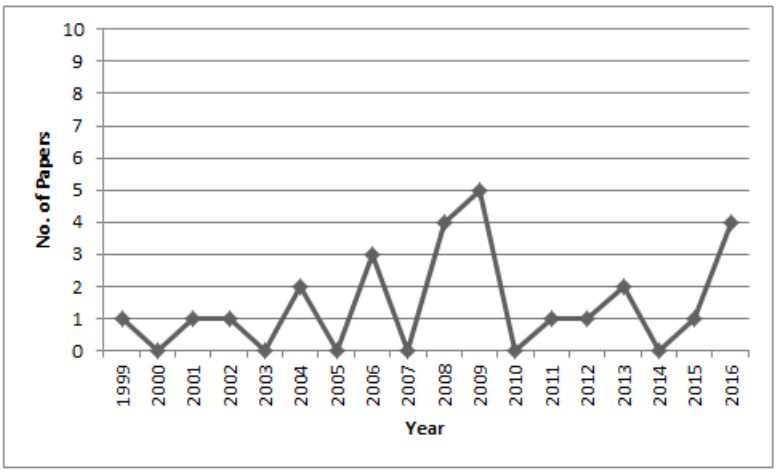


Figure 4.4: Publication trend

Table 4.3: Research approaches

Research approach	Description
Experience report	Research describing personal experience of using a particular technique, often by practitioners in the industry.
Evaluation research	Researches describing detail investigation of a particular technique or groups of techniques and reporting the pros and cons.
Philosophical view	Researches describing a new conceptual framework or new way of looking at existing things.
Solution proposal	Researches proposing a novel or an incremented solution to a particular problem.
Validation research	Validation research may be a solution proposal research but with a detail implementation and demonstration with small example (e.g. work done in the lab).

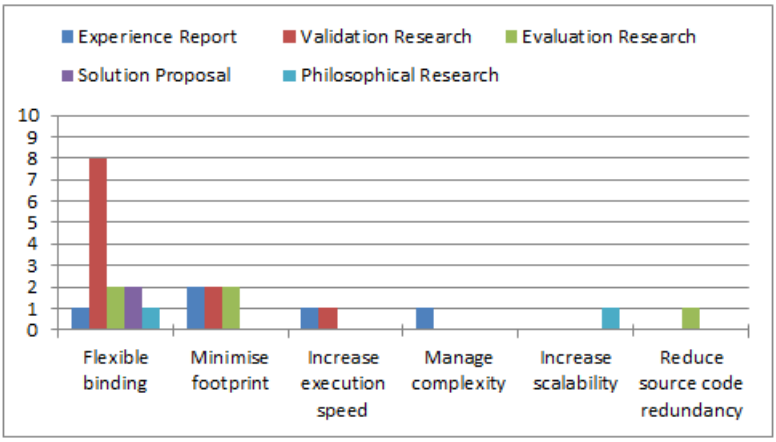


Figure 4.5: Various research goals of the proposed papers vs research approaches

and with eight (8) of the papers reflecting the validation research approach (first clustered bars in Fig.4.5). Next, six (6) papers share the goal of minimising the footprint of executable and two (2) out of the six papers are experience reports, other two (2) are validation research and the remaining two (2) papers are evaluation research (second clustered bars in Fig.4.5). Two (2) papers, one (1) validation research paper and one (1) philosophical research paper, share the goal of increasing the execution speed of the software product (third clustered bars in Fig.4.5). Lastly, the research goals of managing complexity, increasing scalability and reduction of source-codes redundancy were pursued by one paper each.

Thus, the following is the summarized answer to RVQ2a:

> The researches on variations aimed to influence the following outcomes: flexible binding, minimizing footprint, increasing execution speed, managing complexity, increasing scalability and reducing source code redundancy.

In the next section, we present a narrative summary of the proposed techniques

4.3 Narrative summary of the proposed approaches

This section presents a narrative summary of the proposed approaches in the literature to support variations of feature binding time. Narrative summary is a type of interpretive synthesis in qualitative research for aggregating theories[DWAJ+05]. We used narrative summary here in its simple form to guide the description of approaches proposed to support variations of feature binding time.

We derived initial classifications of the approaches from the second step of paper selection. After a thorough reading of the papers in the final step, we refined the classifications with the addition of new approaches and merging of similar ones.

The following is the list of the refined approaches and also a partial answer to the review question 2 (RVQ2):

- Delegation of binding to *aspect weaver*
- Programming language extension
- *Metadata* interpretation
- Model composition
- Delegation of binding to deployment platform
- Abstracting the binding time at model level

Note that the classification is not a hard one as an approach may subsume other approaches. For example, some implementations of aspect weaving use *metadata* interpretation and the actual binding may be delegated to a deployment platform. Similarly, *aspect-oriented* implementation is also considered as a form of language extension. Nonetheless, we consider the characteristic of the approach that was primarily utilized in the research paper in focus. The proposed approaches were used in different application domains as depicted in Fig.4.6.

Fig.4.6 presents a taxonomy of application domains in which the approaches were either validated or evaluated. Thus, Fig.4.6 stands as an answer to the review question RQ1a.

In the subsequent sections, we provide a brief overview of each of the approaches and then followed with a summary of the research contributions of papers in that category. We also assess each of the approaches based on the criteria in Table 4.4.

4.3.1 Delegation of binding to aspect weaver

Aspect Orientation (AO) is an engineering approach for separation of cross-cutting concerns - concerns that would normally be scattered across other concerns. Each of the separated concerns is implemented as one or more *aspects*. *Aspect weaver* is a tool that intertwines the separated concerns with the rest of other concerns. Depending on the weaver, the intertwining of the separated concern (*aspect weaving*) can be done at different phases. Thus, the delegation of the binding decision to aspect weaver is handing over the actual binding to the intertwining process.

Table 4.4: Assessment criteria of proposed approaches

Criterion	Explanation
Binding phases	How the technique support binding at the different phases and whether or not the technique supports both inclusion and activation of new artefacts, realizing a feature, or it simply supports activation/deactivation of a pre-included artefact.
Granularity	Whether the approach supports fine-grained variations (e.g., variation at the level of functions, methods, attributes, and clauses) or coarse-grained variations (e.g., variation at the level of modules, packages, classes, and files) or both.
Level of abstraction	The level of detail in which the product line asset together with the variation is expressed. High-level of abstraction represents less detail representation of the system of which further translation is required to arrive at low-level details (executable source codes). On the other hand, a low-level of abstraction represents a more detail representation of the system and of which no further translation is required.

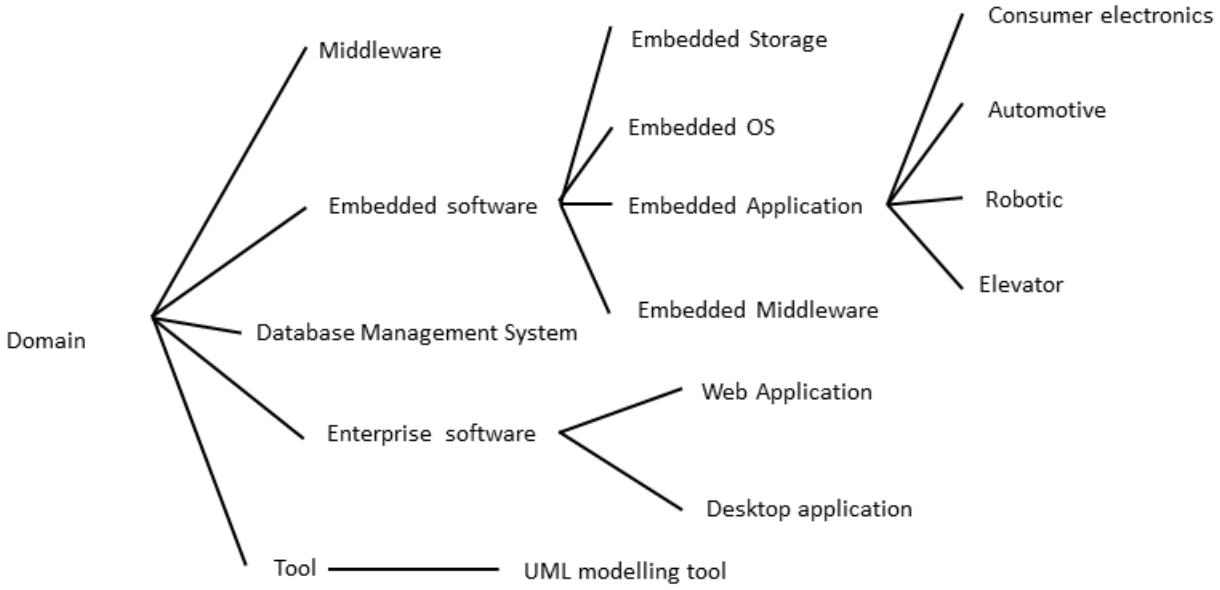


Figure 4.6: Application domains

A feature in SPLE can also be a separable concern to some extent[KAB07, CLK08, FCS+08]. The fundamental assumption is that, variable features are implemented with aspect-oriented mechanisms to extend a monolithic *base-program*. Other program elements (interfaces and classes) may be provided to support aspects of a variable feature. Ideally, none of the supporting program elements would be referenced from anywhere in the base-program or in any other feature. To manage variations of feature binding time, the binding decision of the variable feature (a feature implemented as aspect) can be made at either of the weaving phases supported by the target aspect-based implementation.

4.3.1.1 Existing research contributions

Aspect-oriented related techniques constitute the largest interventions proposed to support variations of feature binding time. We begin with three aspect-oriented design idioms proposed to support flexibility of feature binding time. A design idiom is a style of solving a specific implementation challenge in a specific environment. Chakravarthy *et al* [CRE08] proposed the first idiom- Edict. Andrade e*t al* [ARR+16] proposed the other two idioms: Pointcut redefinition and Layered aspect. All three idioms were evaluated with the earlier version of AspectJ[1] (a Java-based aspect-oriented programming language) but the evaluations were carried out in different application domains.

Edict idiom: with this idiom, a developer separates source codes of a variable feature from the base-program and implements the separated source-codes in one or more aspects. A catalogue of design patterns [GHJ+95] is used to aid the separation. A variable feature is then implementation as aspect in a two-step process. In the first step, one or more abstract aspects are implemented with the necessary *intertype declarations* and *pointcuts definitions*. The intertype declarations are additional structural codes to support the aspect while the pointcut definitions are specifications to capture places in the base-program where the variable feature extends. In the second step, two sets of concrete aspects are implemented: 1) aspects for binding feature to execution context statically, and 2) aspects for binding of a feature to the execution context dynamically. Both sets of concrete aspects are subtypes of their corresponding super abstract aspects. The two sets of aspects differ in the code snippet to enforce either of the two binding modes. The choice of the sets of aspects to include in the product configuration depends on the choice of binding of a feature.

Because of the granularity of variations, using Edict idiom to implement a variable feature may require several abstract aspects and each having multiple concrete subtypes. Since the multiple concrete subtypes of the same abstract aspect differ only in few codes snippets, most of the same source codes have to be duplicated across the multiple sub-types. Hence, using the Edict idiom can breed too much code duplication. Andrade *et*

[1]https://www.eclipse.org/aspectj/

al [ARR+16] proposed to minimise the code duplication and we discuss their approach in the following paragraphs.

Andrade *et al*, [ARR+16]proposed two design idioms to improve on the utility of Edict idiom. Specifically, they aimed to minimize the following shortcomings of Edict: code duplication, code tangling, and code size. The two design idioms share the same philosophy with the Edict: different sets of aspects for the different binding modes. We present the summary of the two idioms in the following paragraphs.

Pointcut redefinition idiom: in this idiom, a developer implements intertype declarations and pointcut definitions in the super abstract aspect(s), as in Edict. Unlike in Edict, the super abstract aspect(s) should also contain a complete implementation of the feature code. In addition, a developer should also implement two separate sets of concrete aspects: 1) an empty concrete *subtype* for each of the abstract aspect to instantiate the super abstract aspect 'as is'. 2) A concrete subtype for each of the abstract aspect and each concrete aspect overrides all the pointcuts of its super abstract aspect. The only reason for the overriding is to link every pointcut with activation condition so that the pointcuts are only *advised* when the feature is activated (through a configuration file or similar techniques). A piece of advice, in aspect-oriented programming, is a unit of implementation for adding behaviour to the base-program. If a binding time is decided before compilation, a developer includes the abstract aspects as well as the set of empty concrete aspects in the product configuration (1 above). Otherwise, a developer includes, in the product configuration, the abstract aspects as well as the set of concrete aspects with the overridden *pointcuts* (2 above).

Since the concrete aspects for the early binding are empty, the code duplication is significantly reduced in comparison to Edict idiom. In addition, the concrete aspect for the late binding contains only the pointcuts with the additional link to the activation conditions and thus reduces the code duplication of Edict. Andrade et al [ARR+16] further proposed to improve this idiom with another idiom as explained in the next paragraph.

Layered aspects idiom: In layered aspects idiom [ARR+16], rather than applying activation condition for every pointcuts, as in pointcut redefinition, a single activation condition is applied to all pointcuts collectively. Technically, the condition is applied to all pointcuts at once using AspectJ special construct, *adviceexcution*. However, the pointcuts intercepted with around *advices* need to be overridden to prevent skipping a code in the base-program when the feature is deactivated.

Similar to the contribution with design idioms, Lee *et al* [LKKP06, CLK08], proposed patterns of aspect-oriented design to implement features and manage their dependencies in order to support flexible binding time. Likewise, Vranić *et al* [VBMD08, MV09, BVD07] proposed patterns for addition and removal of functionality to a web application to implement change requests. The approach may be used to control feature binding by mapping feature implementation patterns to the proposed implementation patterns of

change requests.

The approaches discussed so far, under the delegation of binding to aspect weaver, were reported in the context of a static weaver. Static weaver intertwines aspects to physical artefacts such as source codes and binary and that limits the support of feature binding at post-deployment time. On the other hand, dynamic weaver modifies a program in execution at runtime and that facilitates feature binding at post-deployment time. For the dynamic weaving, the actual binding is often specified as a *metadata* and interpreted by the deployment platform. To fully support binding time, Tartler *et al* [TLSPS09, SPLS+06] integrated the two modes of weaving: static weaving and dynamic weaving in a single aspect weaver. However, the integration had to compromise with fine-grained separation of concerns. This is similar to the compromise made when AspectJ[Lad03] (a Java-based aspect-oriented language with static weaving) was merged with AspectWerkz[2] (also a Java-based aspect-oriented language but with dynamic weaving) and thereby giving up some dynamic capability of AspectWerkz.

Binding phases

The approach in [CRE08] and [ARR+16] support binding of feature to a product in two phases: The first phase, *compile-time weaving phase*, corresponds to the binding of a feature at pre-deployment time. The second phase, *load-time weaving phase*, roughly corresponds to the binding of a feature at a deployment phase. Conversely, they do not support the inclusion of a new feature at post-deployment time. However, a feature can be included at an earlier phase and then supported with activation/deactivation mechanism for post-deployment binding time decision. In other words, approaches in this category have limited support for feature binding at post-deployment time. In contrast, the approach proposed in [TLSPS09, SPLS+06] provides a technical support to include a feature at post-deployment time albeit at a coarse-grained level.

Granularity

Using the design idioms proposed in [CRE08] and [ARR+16], variations at all levels of granularity can be encapsulated as aspect. Generally, an aspect-oriented language with a static weaving provides support for separation of concern at a fine granular level. On the other hand, the integrated weaving in [TLSPS09, SPLS+06] does not support the separation of fine-grained variations.

Level of abstraction

All of the approaches discussed, under *Delegation of binding to aspect weaver*, support specification of product line assets only at low-level of abstraction - at the source code level.

Discussion

Generally, implementing a feature as a separable concern, in terms of aspect of aspect-

[2]https://www.eclipse.org/aspectj/aj5announce.html

oriented programming, requires non-trivial efforts. This is because implementation of a feature in SPL was not the initial intention of aspect-orientation. Specifically, the power and utility of aspect is more apparent when applied to separate a *homogeneous cross-cutting concern* - a single concern that appears uniformly in multiple places in the base-program (e.g. logging). However, a feature in SPL often manifests as *heterogeneous cross-cutting concern* at the source code level. In other words, a single feature may affect several places in the base-program differently. Consequently, several aspects may have to be used to effectively separate a single feature of moderate to large size [KAK08].

We notice that the two authors [CRE08] and [ARR+16] aimed at reduction in footprint. Accordingly, in [CRE08], aspects for the static binding (the aspects to be included if the binding decision is decided at pre-deployment time before compilation) are optimised to make the implementation of deselected feature unreachable and then removed by an optimisation tool. We recommend that the optimisation should be applied in consideration of the overall domain knowledge. For example, it is reasonable to optimise a low-end product with memory constraint and which is unsuitable for an upgrade with a prior deselected feature. However, that type of optimisation may have negative consequences on an optional variation point that may be transformed into an inclusive OR group in the course of product line evolution in which the additional feature would be bound at post-deployment time.

Lastly, the notion of binding time in all but in [TLSPS09, SPLS+06] is limited to binding of a feature to product execution context (i.e. static vs dynamic binding mode) as against binding a feature to a product.

4.3.2 Language extension

Language extension is additions of new constructs or new notations to an existing programming language. There are two broad categories of language extensions. The first category is extending a language with additional constructs that are similar to the constructs of the target language. In this case, a specification with the extension represents the same level of abstraction with the specification in the target language. The second category is extending a language with constructs or notations that are entirely different from that of a target language. In this case, a specification with the extension represents a different level of abstraction from the specification in the target language. What is common among the two categories is that the design specifications with the extended languages are eventually transformed into the target language being extended. The next section presents the research contributions in this category.

Existing research contributions

Philip Consumer Electronics [VO98] developed a purposeful language, Component Description Language (CDL), and used it to describe a family architecture of electronics

software. CDL is an architecture description language (ADL) with its own constructs. However, the language is compiled to C as the target language. The CDL constructs support architectural specification at a high level of abstraction than what is possible with the C programming language. The configuration information and the binding decisions are separated from the component architecture specifications and delegated to the CDL compiler. The compiler evaluates the binding decision expressions and replaces them with the appropriate constructs for the desired binding in the C language. The collective techniques were integrated into a special component model known as *Koala*.

Similarly, Rosumuller *et al* [RSAS11] extended C++ programming language to support step-wise refinements (SWR)- a development approach promoted in GenVoca[BST+94]. GenVoca is an approach to gradually modify program elements such as classes and functions. The extension supports implementing a feature as a series of fragmented modifications of C++ classes and functions. A specific product is generated through composition of the refinements corresponding to the selected features. On one hand, an optimised product is generated from *inlined* refinements (i.e. static binding of feature to execution context). On the other hand, they framed the refinements as decorations, as in the decorator pattern[GHJ+95], to support delaying of feature binding to execution context until at runtime.

Binding phases

With Koala [VO98], it is possible to generate binding mechanism that facilitates delay of binding time up to deployment time. However, there is no explicit support for feature binding at post-deployment time. Similarly, the approach of Rosumuller *et al* [RSAS11] is limited to the selection of how a feature would be bound to execution context- static or dynamic – and the decision has to be made before compilation. The research goal of Rosumuller *et al* [RSAS11] is to optimise source codes (using function *inlining* and hardwiring of implementation components), if the binding time of feature can be decided before compilation (i.e. before deployment).

Granularity

Since the approach of Rosumuller *et al* [RSAS11] is a code-level specification, it is applicable to both fine-grained and coarse-grained variations. Likewise, not only the approach in Koala [VO98] supports variations at different level of granularities but also they do so in an interesting way. We summarized how Koala supports variation at three levels of granularity:

- **Component property level**: Fine-grained variations (called component internal diversity in Koala) are represented as variable properties on components and are exposed to configuration interface as *diversity interface*. When a constant value is assigned to a variable property, Koala compiler completely removed the alternative clauses (alternative execution paths) in the source codes. As a principle in Koala, a variable property should not have Set and Get operations because one end up with

two more functions getX() and SetX() for each property even if the functions will never be used because the property is set as a constant. With this approach, a problem of bloating component with various alternative execution paths is minimised.

- **Component interface level**: A component in Koala can have optional required or provided interface. Koala compiler generates a stub, as in null object pattern[GHJ+95], for each optional provided interface. Similarly, Koala compiler generates Boolean presence condition to guard each optional required interface.

- **Component level**: A connection from one component may be rerouted to one of the several alternative components using an architectural construct called *switch*. The actual rerouting decision can be fixed to one of the several alternatives components at configuration time (i.e static binding to execution context) or can be decided at runtime (i.e. dynamic binding to execution context).

Level of abstraction

Koala supports product line specification at high-level of abstraction using components as building blocks. In contrast, the approach in [RSAS11]supports specification only at low-level of abstraction.

Discussion

The Koala component model is one of the earliest approaches (at least in the public domain) of product line practice. One of its powers is optimisation techniques incorporated in the Koala compiler. For example, in component connection rerouting, if the connection is fixed to one component at compile time, the Koala compiler replaces the connection with a direct function call and the unreachable component is removed from the configuration to reduce footprint. Another example is the removal of alternative clauses when a property is assigned constant value in the diversity interface.

The comment on optimisation in the previous section also applies here. However, of note is, application engineers using Koala are also expert in the product they build.

Binding a feature at post-deployment time may be supported in Koala as one of the configuration decisions. In this case, the decision of feature binding at post-deployment time at CDL level should be replaced with C language constructs supporting field-programmability.

Overall, the approach in Koala may be valuable if a significant investment can be made both in tooling and human resources. This is because even the component composition itself is technically-oriented.

The approach proposed in [RSAS11] introduced development processes that may not be consistent with software engineering practices. The utility of the approach may be

supported through a model at a higher level of abstraction to shield engineers using the approach from the low-level manipulations.

4.3.3 Metadata interpretation

Metadata is a secondary data that describes a primary data. In the context of SPLE, a *metadata* is often used to specify product configuration. For example, business rules such as insurance policy may be stored in a table or as an XML file as a form of *metadata* and different set of rules distinguish one insurance application from the other. Similarly, data about variations as well as their binding times can also be stored as a *metadata* [SE08]. In this case, products of a product line do not differ only on features but also on feature binding times.

Existing research contributions

Yoder *et al* [YBJ01] advocate that, whenever possible, the semantics of a domain should be abstracted and modelled as a set of 'supertypes'. For example, a domain may be modelled as a state machine consisting of a set of 'State', 'Transition', 'Action' as 'supertypes'. The *meta* relationships between the 'supertypes' are explicitly specified. For example, *'State' has one or more 'Transition' to another 'State' and each 'Transition' is triggered by an 'Action'.* Different products in the domain would differ in the concrete 'subtypes' of the 'supertypes'. Thus, a software configuration is the specification of the required 'subtypes' as a *metadata*. While the authors called this approach 'Adaptive Object Model', Fowler called it a 'Semantic Model' style[Fow10]. In this thesis, we go with the 'semantic Model' as the approach has more to do with the logic of the domain than any specific implementation technique. We consider this approach in the review because it may be used to support variations of products of a product line. The specification of the *metadata*, which is comparable to feature selection, is a decision on whether or not to include a specific 'subtype' such as an optional 'State' a system may assume and can be done at the different phases of binding. Bugerili *et al* [BMF09] adapted this approach to model satellite launcher product line using sensors, actuators, control, etc as the 'supertype' in that domain.

Capilla and Bosch [BC12] discussed ideas on how to manage variations of feature binding time. In a separate work, they mapped the ideas to reconfigurations processes in a power plant control product line [CB16]. Subsequently, and in a more concrete work [CVD16], they illustrated the binding of features to a feature model at a post-deployment time. In the implementation, they used a database to store feature with its 'supertype' as a *metadata*. a feature can then be added or removed to a feature model by updating the database. The updates are then interpreted and reflected on the feature model using a matching algorithm that compares the root of the new feature to pre-existing 'supertypes'. The root of the new feature may match an existing 'supertype' and, therefore, a feature can be added as a child of the 'supertype'. Otherwise, the new feature may have to be

added as a new 'supertype'.

Khediri *et al* [KK15], proposed an approach to modelling database using delta-oriented technique[SBB+10] to specify deltas in data description language (DDL) of Structured Query Language (SQL). Each delta is a specification to change the structure of the database to reflect the binding of a specific feature of a database management system. Although this approach is specific to database management domain, we envision the transfer of the concept to product line domains that are implementable in scripting languages such as Hypertext Preprocessor (PHP), Python, Perl.

Binding phases

In Yoder *et al* [YBJ01] and [BMF09], all the features are conceptually included in the product configuration (although variant of this approach exist). The specifications in the *metadata*, possibly at the different binding time, are merely for activation/ deactivation of pre-included features. In the case of the approach in[CVD16], the feature binding is neither to the execution context nor to the product but the binding is to a feature model. Since a feature model is not executable, we can only imagine that the feature model is supported with a repository of implementation components as in [VdH04]. In this case, the updates on the feature model are interpreted to activate or deactivate implementation component realizing the features. Therefore, these approaches offer limited support for binding time even though they support binding a feature at multiple phases. The approach in[KK15] supports the binding of a feature at all phases, subject to the availability of interpretive support at the binding phase.

Granularity

In [YBJ01] and [BMF09], variations are supported only in terms of the semantics of the domain and therefore limited. In the example of the state machine, although different products may be composed with different states and different transitions, an internal variation within a state, for instance, is not explicitly exposed. There is no enough information to assess the approach proposed in [CVD16] with respect to the granularity of variations. Delta modelling, as proposed in [KK15], can be used to specify changes of a database in terms of field, record, database table or the complete database schema. Hence it supports both fine-grained and coarse-grained variations.

Level of abstraction

Both approaches to specifications of product line assets proposed in [YBJ01] and that of [KK15] are at low-level of abstraction. Binding specification proposed in [CVD16] is at the level of feature, and a feature can represent both high-level and low-level of abstraction.

Discussion

The 'semantic model' approach is only applicable to specific classes of system. Fowler, M. [Fow10] provided guidelines on where the approach may be applicable. A variant of this approach is to use the *metadata* to guide code generation or to implement the 'supertype' and only use the *metadata* to specify configuration and reconfigurations.

The search strategy to locate the appropriate position of the new feature in the existing feature model proposed in [CVD16] may be resource-intensive. Nonetheless, the major skeptic about this approach, at least what the concrete implementation demonstrates, is their focus on the feature model exclusively.

4.3.4 Abstracting the binding time at the model level

Abstracting the binding time at a model level is to provide special model properties to represent binding time at a model level in model-driven development. This approach is similar to language extension in the sense that the model and the properties representing the binding times are ultimately transformed into source-codes of the target programming language.

Existing research contributions
Beuche1, D. and Weiland, J. [BW09] extended Simulink -modelling tool[3] with constructs for modelling variation points and parametrization of the variation points with binding modes. Each of the binding modes, represented as a property at the model level, is transformed into an appropriate snippet at source code level using a model to code transformation. Similar to this approach, Schmid and Eichelberger [SE08] propose to store the binding time properties as part of decision-based variability model.

Hartmann *et al*[HKM+13] proposed an approach to generate a glue component to adapt binding interfaces of components that might have been developed by different organizations. The approach has the potential to support abstracting the binding time and presenting it at the model level. That is, the glue component may embed a parameterized binding time property and the binding time decision can be exposed to a configuration interface.

The work of Andre Vanda hoek [VdH04] is one of the earliest to address variations of feature binding time at the architecture level. In his approach, architectural representation is used as a configuration interface and is mapped to concrete implementation artifacts. Feature binding at pre-deployment time is a selection of architectural elements to form an architectural representation for the desired product; the implementation components for the architectural representation are then activated. At deployment time, a prior configured architecture representation may be updated with additional architectural elements to reflect the binding of additional features. A new architecture representation may as well be formed if no one exists (i.e., no binding decision was made at pre-deployment time). In this case, the architecture representation is translated into calls to concrete implementation components. At post-deployment time, features are bounded to the product in operation by comparing a model of the desired architecture representation with

[3]https://www.mathworks.com/products/simulink.html

the model architecture representation of the deployed product. The difference between the two architecture representations is then applied to the existing product through the activation or deactivation of implementation artifacts.

Binding phases

The Simulink tool extension [BW09] does not include the support for feature binding at post-deployment time. On the other hand, the extended support of the model in [SE08] includes properties for generating aspect code to activate or deactivate a feature at post-deployment time and thereby partially supporting feature binding at post-deployment time. The glue code generation proposed by Hartmann *et al* [HKM+13] supports the binding decision at the code generation time (before deployment) or the binding decision may be pass-on to be resolved at deployment time - no support for feature binding at post-deployment. In Andre Vanda hoek[VdH04], apart from the pre-deployment binding time, the architecture model only facilitates activation and deactivation of implementation components at the different binding time.

Granularity

Both the Simulink extension[BW09] and the glue generation approaches[HKM+13] do not explicitly manage internal component diversity. Hence, they do not support fine-grained variations. In contrast, in [SE08], variations are implemented at both fine-grained and coarse-grained levels as in traditional variability management (e.g. using pre-processing) and then mapped to the decision points at the level of the variability model. In [VdH04], the variations are only explicitly supported at the level of component and interface. However, the approach supports different versions of architectural components and versions may differ in fine-grained variations.

Level of Abstraction

The Simulink tool in itself provides model elements (e.g. subsystems, signal routing, and switch block) for modelling a single system at a high-level of abstraction. Thus, the additional support for variation and binding time proposed in [BW09] supports modelling product line at a high-level of abstraction. Similarly, a glue code generation proposed by Hartmann *et al* [HKM+13] is to bridge the differences between high-level components and hence it is at a high-level of abstraction. The decision-based variability model and the binding time representation in [SE08] abstracts away the low-level implementation details and exposed binding decisions as interactive questions. However, a developer has to implement the low-level implementation details and mapped them with the variability model. The approach in [VdH04] supports specification at architecture which is relatively a high-level of abstraction although the architecture has to be manually mapped to low-level implementation details.

Discussion

Embedded software is the target domain for the glue code generation proposed in [HKM+13]. Support for feature binding at post-deployment time in an embedded system is to support

field-programmability and which may be represented as a model property at the level of glue modelling. The scope of the Simulink tool extension proposed in [BW09] is limited to providing support for binding a feature to the execution context. Consequently, it is difficult to extend the approach to support binding a feature at post-deployment time. Different architectural representations for the different binding time as proposed by Vanda Hoek [VdH04] may hinder the utility of the approach.

4.3.5 Model composition

Model composition is a process of combining two or more model slices to form a single model. In SPLE, a model slice represents an implementation of a specific feature, and the composed model represents an implementation of a specific product. A model or other artefacts generated from the model can be composed either at pre or post-deployment time.

Existing research contributions

Zdun, U. and Strembeck, M. [ZS06] proposed a *metamodel* for modelling introduction of additional structure and interaction of model elements into an existing model (or possibly an empty model). Although the approach was not meant specifically for modelling variations of feature binding time, it could be adapted to model program elements that would be composed to form a product as a result of feature selection. It can also be used to add a feature to an existing product at the different binding time.

Whittle *et al* [WJE+09], proposed the use of graph transformation to specify model slices and to use aspect-oriented techniques to compose the individual slices. Model elements for specific concerns (a concern may be a feature) are separated and composed when needed. In this approach, a modeller slices models such as UML class, sequence, and state diagrams according to requirements for implementing a specific feature. The model slices, for selected features, are then composed to derive a custom model. Each model slice, for the specific feature, is a specification of fine-grained model elements to be added or to be removed from the base-model or another model slice. For example, a model slice to implement a feature may be a specification to add 'states', 'events', 'transitions', and 'regions' to the base state diagram or another model slice. In this approach, feature binding is a selection of the fine-grained model elements to realize features as a model slices and using separate target transformation based on binding time requirement: At pre-deployment, the selected model slices, representing a custom product, is transformed into source-codes for compilation. At deployment and post-deployment time, the selected model slices are transformed into dynamic script/aspect to adapt the product.

Binding phases

The assessment of binding phases does not apply to the approach proposed by Zdun and Strembeck [ZS06], because it is an abstract modelling of features with no concrete

implementation. The approach proposed by Whittle *et al* [WJE+09] supports feature binding at two phases: at pre-deployment time, model slices representing selected features are composed and transformed into executable. At post-deployment binding time, model slices are transformed into AspectWerkz code - an AOP for dynamic weaving.

Granularity
Both the approach in [ZS06] and [WJE+09] support modelling variations at both fine-grained and coarse-grained level.

Level of abstraction
In both [ZS06] and [WJE+09] the models or model fragments, as the case may be, are specified in terms of low-level model elements.

Discussion
Model composition approach proposed by Whittle *et al* [WJE+09] is a powerful approach to support feature binding at pre-deployment and post-deployment times. feature binding at the deployment phase can be supported as well through the generation of a configuration interface, from the model, to control the inclusion and activation of a variable feature at the deployment time. The downside of the approach is the focus on fine-grained model elements which may affect the practical scalability of the technique. Imagine several features, some of which are of moderate to large size, have to be modelled as slices of classes, sequences diagram and so on. The focus on fine-grained model elements also impedes reasoning of the product line at the architecture level.

4.3.6 Delegation to deployment platform

The term software platform, as used in this thesis, may be a middleware that provides support services or an architectural framework that is designed to decouple component interaction. Delegating a feature binding to a deployment platform is to use the support of the target environment to decouple dependency between features such that they could be bound, unbound and rebound. The actual binding is often managed with dynamic loading/linking with the support of *metadata* interpretation. A typical example of a deployment platform, for Java-based enterprise application, is Java Enterprise Edition (JEE)[Tec]. JEE platform supports light-weight messaging (event-driven, point-to-point, and publish-subscribe) to decouple component interaction.

Although many platforms, including those in real system domain [sm118], facilitate feature binding, not many pieces of research are available on this class of technique. This may not be unconnected with the fact that existing platforms mainly support single product development, are hard to adapt to product line development and are costly to be developed from scratch. For example, although JEE supports the decoupling of implementation components, mapping components to features is not straight-forward because

of the granularity of variations.

Existing research contributions

Goedicke *et al* [GPZ02], proposed an architectural framework, 'Central Message Redirector', to support customization of generic components for products of a product line. Specific features of a product are specified as configuration scripts to tailor the generic components to specific product requirements. The framework hides requests and responses between the client and the server components. Both the client and the server components can be customized using interpretive script or XML. The 'generic components' in this approach is what the 'supertypes' is to the 'semantic model'.

Binding phases

Feature binding, in the approach of [GPZ02], is a customisation of the generic components to derive a specific product and the customization may be specified at the different binding phases. However, most or all of the 'generic components' have to be included in each of the products. Hence, the approach supports partial feature binding at multiple phases.

Granularity

Generic component implementations do not normally accommodate fine-grained variations. In the specific case proposed by [GPZ02], lots of configuration parameters have to be supplied as part of the configuration scripts.

Level of abstraction

The component level specification is relatively a high-level of abstraction.

Discussion

One limitation of the approach in [GPZ02] is generic component implementation leads to an over-bloated component with many alternative execution paths to accommodate the different usage of the component within possibly many dissimilar products in the product line. Consequently, excessive dynamic scripts have to be used, as part of configuration parameters, to accommodate fine-grained variations within coarse-grained components. Part of the limitations reported by the author is the performance penalty of the approach which makes it unsuitable for other domains such as real-time systems.

Table 4.5: Summary of pros and cons of the proposed approaches

Approach	Pros	Cons
Delegation of binding to aspect weaver	- Variation at all levels of granularities can be implemented as a feature when a static weaver is used. - Supports post-deployment binding time when a dynamic weaver is used.	- Separate sets of aspects may have to be used, each for a specific binding time. - Limited support of fine-grained variations at post-deployment time. A feature will have to be included in the product configuration prior to deployment. - The support is at low-level of abstraction.
Programming language extension	- Supports the two binding modes(static and dynamic) - Complexity can be reduced by introducing domain abstractions in the language extension.	- No support for feature binding at post-deployment time. - Requires different expertise in language and tool design.
Metadata interpretation	- Different products, with arbitrary variations in features and workflows, can be created.	- Limited support for fine-grained variations. - The support is limited to some specialized domains. - Limited support for feature binding at post-deployment time. A feature will have to be included in the product configuration prior to deployment.
Abstracting the binding time at model level	- Supports both fine-grained and coarse-grained variations and at a high-level of abstraction. - Complexity can be reduced by introducing domain abstractions as model elements or properties of model elements.	- Current supports are limited to supporting only dynamic and static binding modes. - Modelling infrastructure has to be created and maintained.
Model composition	- Supports flexible binding at both pre-deployment and post-deployment time. - Supports both fine-grained and coarse-grained variations.	- Current support focuses on too fine-grained model elements, hence, the approach may not scale in practice and may be difficult to reason about the architecture of the product.
Delegation of binding to deployment platform	- Supports flexible binding at both pre-deployment and post-deployment time and at higher level of abstraction	- Limited support for fine-grained variations. - Excessive dynamic scripts may have to be used to bind a feature at post-deployment time.

4.4 Summary of the proposed approaches

In this section, we present a summary of the pros and cons of each of the approaches as depicted in Table 4.5. We also present the overall summary in the following paragraphs.

In the category of approaches proposed around the delegation of binding to aspect weaver, in all but two cases, the support of variations of feature binding time stopped at the deployment time. At post-deployment time, feature binding is limited to activation or deactivation of the pre-included feature. In the two cases of [TLSPS09] and [SPLS+06], both inclusion and activation of a feature at post-deployment time are supported but only at a coarse-grained unit of implementation. Similarly, since the aspect-orientation, as proposed in the approaches, focus on the source-codes level exclusively, all the approaches in this category of approaches are only applicable at low-level of abstraction.

In the category of approaches proposed based on the language extension, none support feature binding at post-deployment time. Similarly, in one of the two proposed approaches to the language extension [RSAS11], the notion of feature binding is limited to the scope of binding to the execution context. All the approaches proposed under the *metadata* interpretation are only applicable to specific classes of systems (database management system[KK15], graph[CVD16], and domains whose semantic can be precisely captured as a semantic model [YBJ01, BMF09]).

Although the approaches proposed under abstraction of binding time at the model level are at high-level of abstraction, only one approach [BW09] explicitly supports variations of feature binding and only in a limited scope of binding to the execution context. In the category of model composition approaches, Whittle *et al*[WJE+09] supports feature binding at both pre and post-deployment time but only at low-level of abstraction (through the support of flexible composition of fine-grained model elements).

Other papers only discussed the pros and cons of feature binding techniques based on experience (e.g. [BC12, CB16] .)

Overall, apart from approaches that remotely support variations of feature binding time (*metadata interpretation* and *language extension*), current support for variations in feature binding times is through one of the following:

i using separate architectural representations for the different binding time;

ii composition of fine-grained model elements to realize a feature as a model slice, and using separate target transformation based on binding time requirement;

iii implementing separate set of aspects, in aspect-oriented programming, each set for a specific binding time;

iv abstracting the different binding modes at model level.

All the approaches share one or more of the following limitations:
multiple representations/implementations (i, ii, iii);
composition of too fine-grained model/program elements(ii, iii);
low-level of abstraction (iii);
limited in scope (iv).

In this thesis, we proposed PLA modeling language, that is aware of binding time, to overcome some of the existing limitations in the following ways:

- To abstract the feature binding to a product, as against feature binding to the execution context of a product, at a model level, and, thus, increase the scope of support for feature binding.
- To use architectural elements as the dominant representation, as against fine-grained model elements. Hence we raise the level of abstraction of which the binding time is supported.
- To take binding time requirement into account when transforming architectural elements into executable, thereby abolishing the need for multiple implementations/representations.

We present the details of our approach in the next chapter (*Chapter five*).

Chapter 5

Binding time aware modelling language: design and implementation

This chapter discusses the design and implementation of binding time aware modelling language as an improved approach to supporting variations of feature binding time. Our approach also contributes to supporting the modifiability of reusable assets.

Recall, from *Chapter three*, language-based implementation techniques specifically proposed to implement features in SPLE (Feature-Oriented Programming and Delta-Oriented Programming) might have better modularity but do not support variations in feature binding time. Although Aspect-Oriented Programming, as an adapted language-based approach for product line implementation, has better support for variations of feature binding time, the support is only at a low-level of abstraction.

Also, recall from the systematic study in *Chapter four*, we classified approaches that are proposed to support variations of feature binding time into four: 1) using separate architectural representations for the different binding time; 2) composition of fine-grained model elements to realize a feature as a model slice, and using separate target transformation based on binding time requirement; 3)implementing a separate set of aspects, in aspect-oriented programming, each set for the different binding time; 4)abstracting different binding modes at a model level in model-driven Development.

We argue that using different architectural representations, that are separately managed, for the different binding time hinders adoption by practitioners because different expertise and different tools are required. This particular approach [VdH04] was introduced long-enough but has not been traced to the industry yet.

For the composition of model slices, unless the code-asset would be fully generated from the model - which is hardly the case, except for few specialized domains - we argue that the approach is too fine-grained and may not scale in practice.

For the aspect weaving category, at a low-level of abstraction, it would have been a normalized practice if only a few features and few products are to be produced from the reusable assets. However, if the feature and product spaces of a product line are

substantially large, implementing every feature with multiple aspect-weaving, based on binding time requirement, will lead to a combinatorial explosion.

For the abstracting of binding time at the model level, current support is limited to variations of feature binding to execution context - which is limited in scope.

Given the limitations of the existing approaches, we proposed a binding time aware modelling language as a new approach to abstract the binding time at the architecture model level in model-driven development. The approach is a novel in the sense that existing approaches to abstract the binding time at the model level are limited in scope (i.e. restricted to the two modes of feature binding to execution context). We do not claim, however, that each of the constituents of our techniques is novel. We present an overview of our approach in the next section.

5.0.1 Process overview to supporting variations of feature binding time

To support variations of feature binding time, we pre-plan for it as we pre-plan for the variations of features. Thus, we tackle all binding time by design. We also provide tool support for the key aspects of the support. Fig.5.1 highlights the main activities of supporting variations of feature binding time at both domain engineering and application engineering as proposed in this thesis. In the figure, the activities supported by our tool suite are shaded in grey.

At the top of Fig.5.1, from left to right, the first two outer boxes illustrate the usual high-level activities at the domain engineering phase: Domain analysis and domain design. Domain analysis activity, the first outer box, is a systematic identification and organization of product features. We discussed domain analysis in details in *Feature-oriented domain analysis* section of *Chapter two*. Supporting variations of feature binding time at the domain analysis level is to analyse, further, the identified features and group them into units that must be bound together for the correct function of the product (depicted as sub-activity beneath use case modelling in the domain analysis box). We elaborate on the grouping of features into units as the extension of domain analysis in *Supporting variations of feature binding time at domain analysis* section.

Domain design activity, the middle outer box at the top of Fig.5.1, is the construction of the product line architecture that accommodates diversities of the products in the domain. We also discussed this in details in *Domain design* section of *Chapter two*. However, since a feature manifests as major or minor implementation units, it follows that, at the design level, a feature may translate into one or more coarse-grained exclusive architectural elements(e.g. component or connector). A feature may also translate into many fine-grained architectural elements (e.g attribute or operation on architectural

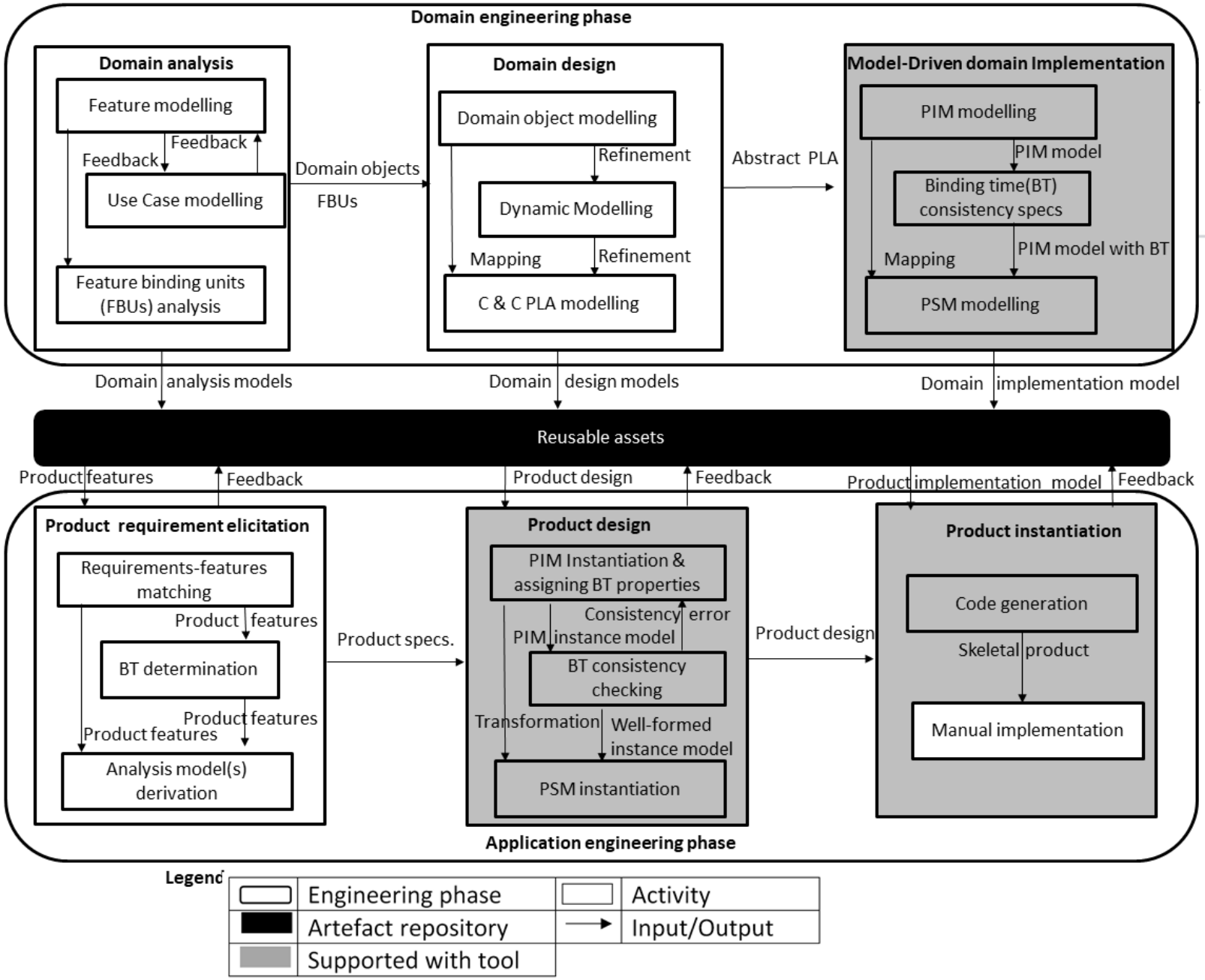


Figure 5.1: Overview of supporting variations of feature binding time

component). As discussed in *Chapter two,* granularities of variations affect how a feature is bound to a product. To support variations of feature binding time at the domain design level, we propose, to re-organize the architecture model so that architectural elements of the fine-grained variations are attached to coarse-grained elements that are in the same binding unit. Fine-grained variations may also be aggregated into an aspectual component. Hence, as illustrated in Fig.5.1, with the arrow from domain analysis to domain design, feature binding units (FBUs) should be taken into account when designing product line architecture. We elaborate on this in *Supporting variations of feature binding time at domain design* section.

The right-most outer box at the top of Fig.5.1, at the domain engineering phase, represents model-driven domain implementation which is one of the approaches to implement a domain as introduced in *Chapter two.* We adapted model-driven because of its flexibility. In our approach, the specific output that comes from domain design and goes to the model-driven domain implementation is the abstract design of PLA (see the arrow from the Domain design to model-driven Domain Implimentation. As with the OMG recommendation we, specify and implement a platform-independent model(PIM) (the top sub-activity in the model-driven domain implementation box) separate from the platform-specific model(PSM) (the bottom sub-activity in the model-driven domain implementation box). PIM is a model that is independent of platform technology and PSM is a model that is specific to particular platform technology.

To manage variations of feature binding time at the PIM level, we assign binding time properties to the model elements. We then define one-way transformation mapping from the PIM to PSM. To manage variations of feature binding time at the PSM level, a model element of the PIM is transformed into its corresponding model element of the PSM based on its value of binding time (see details in *PIM to PSM mapping* section). At this stage, the transformation is only implemented but the model elements of the PSM are not instantiated until at application engineering phase. The details of these activities are presented in *Supporting flexible feature binding at the implementation level* section.

So far we have been outlining engineering activities at the domain engineering phase - the top boxes of Fig.5.1. We also present the high-level overview of the engineering activities at the application engineering phase (the bottom boxes of Fig.5.1) in the subsequent paragraphs beginning from product requirement elicitation.

Product requirement elicitation in our approach is depicted at the left bottom of Fig.5.1 and in the application engineering phase. The first activity in requirement elicitation is matching the product requirements to the pre-planned features (the top sub-activity of requirement elicitation). Desired binding time of the variable features are also determined at this stage (the middle sub-activity of requirement elicitation). The product analysis model may also be derived in our approach (the bottom sub-activity of the requirement elicitation).

As in the traditional approach, specific product requirement may have to be submitted to the reusable assets as the form of feedback. The derived PLA is then instantiated with the desired binding times at the product design phase.

After gathering requirement of the specific product, corresponding abstract PLA, implementing the matched features and the derive analysis model, is then derived from reusable assets and passed-one to product design activity.

Product design in our approach is depicted in the middlebox of the Application engineering phase and constitutes the following sub-activities: instantiation of platform independent PLA and assigning binding time properties to the model elements (the top sub-activity of Product design in Fig.5.1). Since many model elements may contribute to implement a single feature their aggregate binding times have to be checked for consistency with the desired binding time of the feature (middle sub-activity of product design). Binding time consistency should also be checked accross multiple features. Thus, the model elements are subjected to binding time consistency checking.

If the instantiated PIM elements are well-formed with respect to the binding time requirement (i.e. they have passed have binding time consistency checking), the elements are then transformed into elements of another model that is specific to the target platform technology (the bottom sub-activity of product design). The product design is then completed. The next activity is to generate source-codes from the product-specific PSM model.

The last stage in the application engineering is product instantiation (the rightmost outer box at the bottom of Fig.5.1). At this state, a skeletal product is generated from the PSM model of the specific product is generated which will then have to be implemented or to reuse the implementation of the already developed product (the bottom sub-activity of product instantiation at the bottom of Fig.5.1). The key design decisions (architectural pattern/style and the binding time) are enforced in the skeletal product.

In the next sections, we present the details of our approach to supporting variations of feature binding time at the domain engineering phase.

5.1 Supporting variations of feature binding time at domain engineering phase

This section presents details of our approach to supporting variations of feature binding time at the lifecycle activities of domain engineering. Part of the activities is model-driven domain implementation -an approach to implement adaptable infrastructure for a product line.

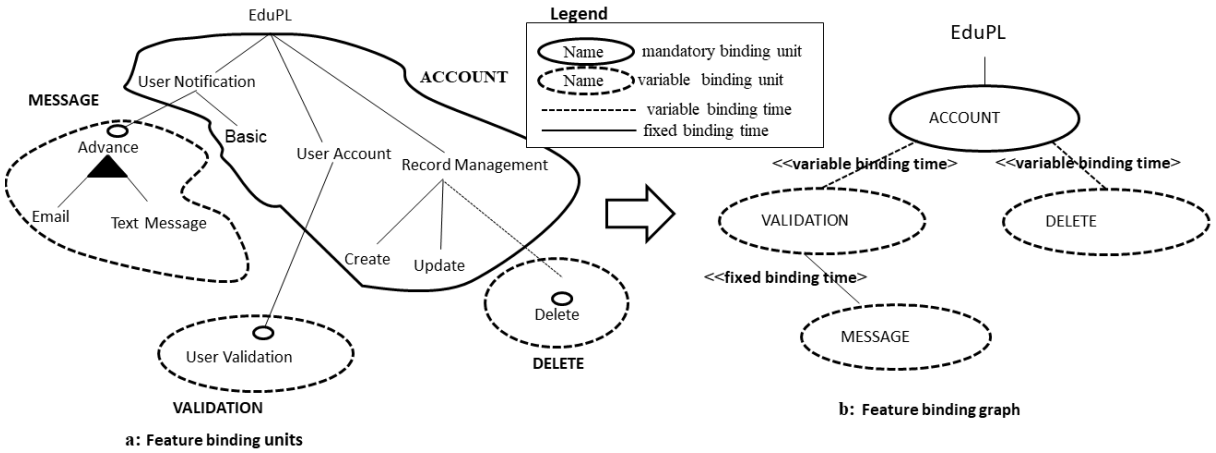


Figure 5.2: Grouping features into units that most be bound together for the correct function of the product

5.1.1 Supporting variations of feature binding time at domain analysis

Recall in the feature-oriented domain analysis section of *Chapter two*, we discussed the traditional approach to domain analysis in which product features are systematically identified and organized based on their types and relationship. To support feature binding time, domain analysis should be extended with binding unit analysis which is an idea conceived by Lee and Kang [LK03]. based on the observation that rather than a single feature, a group of features is often bound together for meaningful service.

A feature binding unit (FBU) is a set of features that share the same binding time. All the features in the same binding unit must be bound together for the correct behaviour of the product [LK03]. Feature binding unit identification starts with the identification of a service feature. Each service feature represents a major functionality of the system that may be added/removed as a service unit.

To derive feature binding units, a domain designer should identify service features. Beginning from the service feature, the designer should trace the constituents of a binding unit by traversing the feature model along the feature relationships and its composition rules (i.e., cross-tree constraints such as require/mutual exclusion). Each binding unit should be assigned a name, in a capital letter, similar to the name of the major feature in the unit. Fig.5.2 illustrates the derivation of feature binding units from EduPL feature model. In the figure, ACCOUNT and VALIDATION are the names assigned to the binding units containing service features *User Account* and User *Validation* respectively (see Fig.5.2a). Within a feature binding unit, there may exist optional or alternative features that should be selected based on customer's needs. These features impose variations on the component design and, therefore, they have to be identified as separate feature binding

units [LK03].

Feature binding units and the relationship between them constitute nodes and edges, respectively, of a binding unit graph. For example, Fig.5.2b depicts a feature binding unit graph with ACCOUNT, VALIDATION, MESSAGE, and DELETE as binding units. For the scope of this thesis, a binding relation between binding units can be fixed or variable. A fixed relation denotes non-varying binding relation while a variable relation represents a binding relation that may differ from one configuration to another. For example, in Fig.5.2b, the connection between VALIDATION and MESSAGE is fixed because the units must always be bound together, while the connection between ACCOUNT and VALIDATION may vary from one product configuration to other product configuration.

Binding unit graph serves as an input to components and connectors product line architecture. In the next section, we discuss the engineering activities of supporting variations of feature binding time at the domain design.

5.1.2 Supporting variations of feature binding time at domain design

To support variations of feature binding time at the domain design level, architectural elements should be organized based on the feature binding unit graph obtained at the domain analysis phase. For coarse-grained variations, it is often a straight forward if the architectural elements have been derived from the gradual refinements of the domain object model. For example, in the *Account Management* sub-system of EduPL, *User Validation* is an optional feature that may be used to validate an applicant as the initial realization of the mandatory *User Account* feature. As such, to organize architectural elements based on the feature binding unit graph, the variable components, *Validation Controller* and *User Validator* should be created separate from the *Account Controller* and the *Account Manager* (see Fig.5.3).

For the fine-grained variations, depending on whether the fine-grained model element is exclusive or is part of an intersection between features, the relevant model elements can be organized in two ways :

i Moving the model element of the fine-grained variation to a coarse-grained model element of the same binding unit. This is the case of a fine-grained model element that is exclusive to a feature but falls into a coarse-grained model element of a different feature.

ii Aggregation of the model elements of the fine-grained variation into an aspectual component. This is often the case of the points of intersection between features but

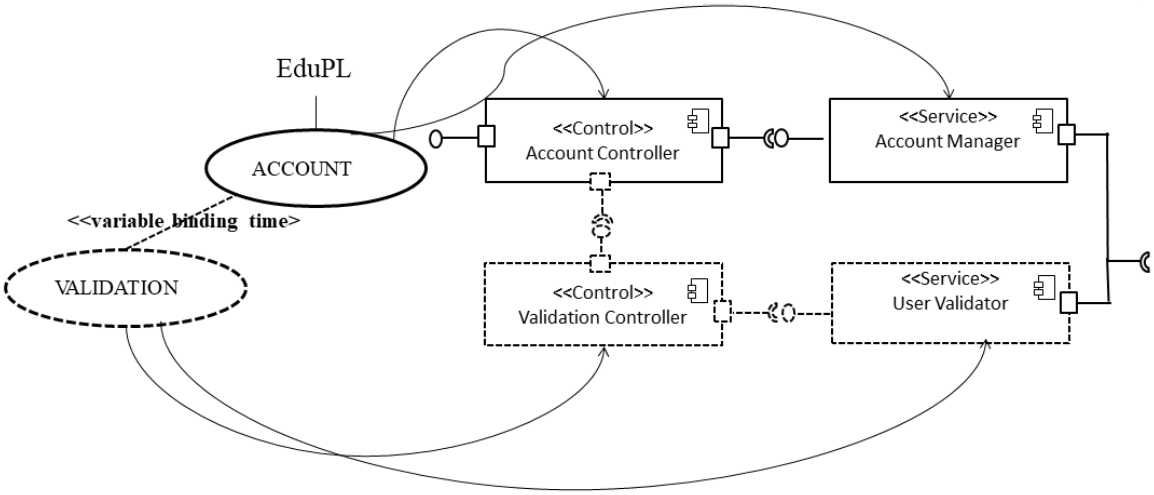


Figure 5.3: Architectural elements (right) organized based on binding unit graph(left)

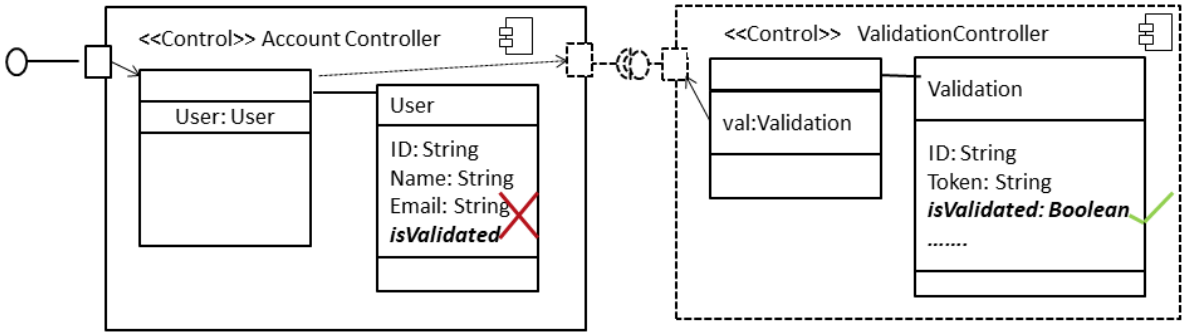


Figure 5.4: *isValidated* as fine-grained model element moved to *Validation Controller* from *Account Controller*.

can also be the case of a fine-grained model element that is exclusive to a feature as in (i) above.

To illustrate (re-)organizing model elements based on the feature binding unit graph obtained at the domain analysis phase, we have to open some of the coarse-grained components (see Fig.5.4 and Fig.5.5).

Fig.5.4 depicts the case of variable Boolean attribute, *isValidated*, which is an element of a fine-grained model elements exclusive to the *User Validation* feature and that could be on *User* object in the implementation of *Account Controller* component in the ACCOUNT binding unit but should be move to *Validation* object in the implementation of *Validation Controller* component of the VALIDATION binding unit.

Fig.5.5 depicts the case of variable operations, *crtlDeleteAccount(user: User)* and

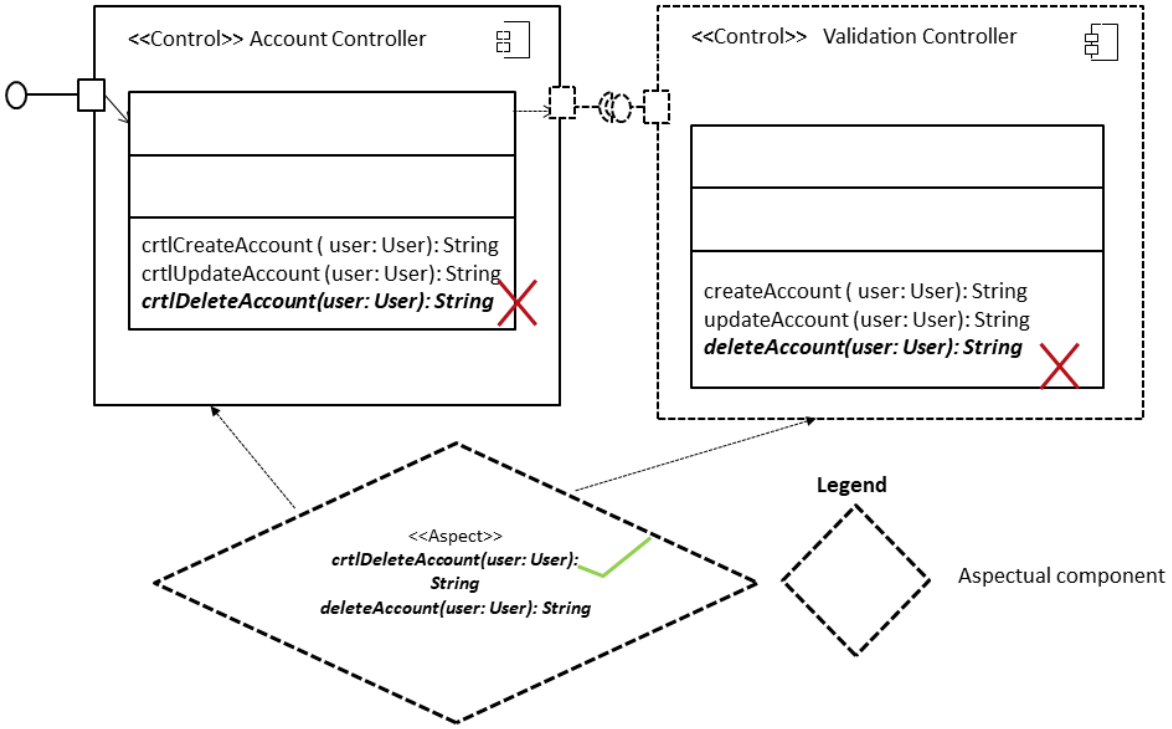


Figure 5.5: *crtlDeleteAccount(user: User)* and *deleteAccount (user: User)* as fine-grained model elements aggregated into *aspectual* component of the same binding unit.

deleteAccount (user: User) that could have been on *Account Controller* and *Account Manager* respectively. These operations should be aggregated and assigned to aspectual component since there is no coarse-grained model element to attach them in the DELETE binding unit.

Even though architectural elements are re-organized based on the feature binding unit graphs, relationships from features to architectural elements is in most cases one-to-many. Thus, each individual model element should be assigned variation type and binding time corresponding to that of its feature. Consequently, the variation type and binding time of a feature should be reflected on the corresponding architectural elements of PLA. At the end of the domain design, abstract PLA should be produced and pass on to domain implementation.

In the next section, we discuss the engineering activities of supporting variations of feature binding time at the domain implementation level.

5.1.3 Supporting flexible feature binding at the domain implementation

Our concept of supporting variations of feature binding time at the implementation level is to switch connection modes between components (SCMBC): The components are connected directly when the binding time is decided before compilation at pre-deployment time (Fig.5.6a). In that case, components should support compile-time and build time composition, do not have to follow separation between implementation and interface and may be followed with a static optimization using a specialized tool. If, however, the binding time cannot be decided at pre-deployment time, dependencies from mandatory components to variable components are removed. The components are connected dynamically, if needed, via a deployment platform(Fig.5.6b). We can think of the deployment platform as a concrete mediator of a mediator pattern[GHJ+95].

To realize the concept of switching connection modes between components, model elements at the PIM level will have to have binding time as properties and should be transformed into model elements of PSM based on the values of the binding time. Consequently, the removal of the dependencies between mandatory and variable components, if the binding time of the variable component is later than pre-deployment, is delegated to model transformation engine as part of MDD process. Specifically, the transformation engine selects target model elements, specific to the deployment platform, based on the binding time decision. Therefore, the final source-codes of a specific product is generated with the binding time decision enforced.

Fig.5.7 depicts the overview of the lifecycle activities to realize the concept of switching connection modes between components. The topmost box in the figure represents *metamodelling* stage which has abstract PLA as input. Ordinarily, *metamodelling* is the specification of the abstract PLA. In our approach, *metamodelling* is extended with a specification of binding time properties as well as binding time constraints. At the *metamodelling* stage (topmost box of Fig.5.7), a *metamodel* of Platform Independent Model (PIM) is developed. Binding time and binding time constraints are inputs to PIM *metamodelling*. PIM is a model that is 'independent' of specific implementation technology. The *metamodel* of a Platform Specific Model (PSM) is also developed at the *metamodelling* stage. PSM is a model that is specific to a particular implementation technology.

The middle box in Fig.5.7 represents the modelling stage which comprises the activity of instantiating elements of PIM and that of PSM. A modeler does not need to instantiate the elements of PSM *metamodel* manually but the PSM model elements are instantiated using model-to-model (M2M) transformation. The transformation takes into account the binding time properties of PIM model elements when selecting elements to be instantiated in the target PSM model.

The bottommost box, in Fig.5.7, contains a single activity: code generation. That

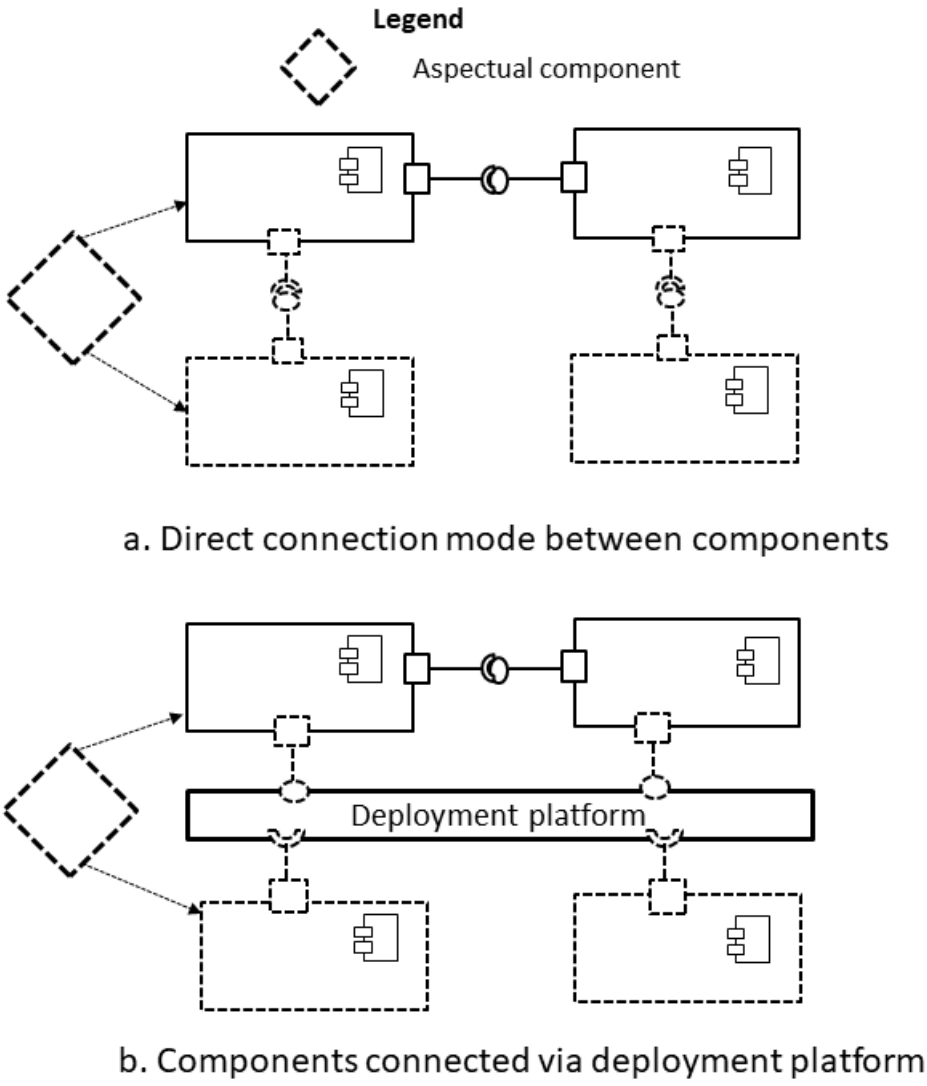


Figure 5.6: Switching connection mode between components

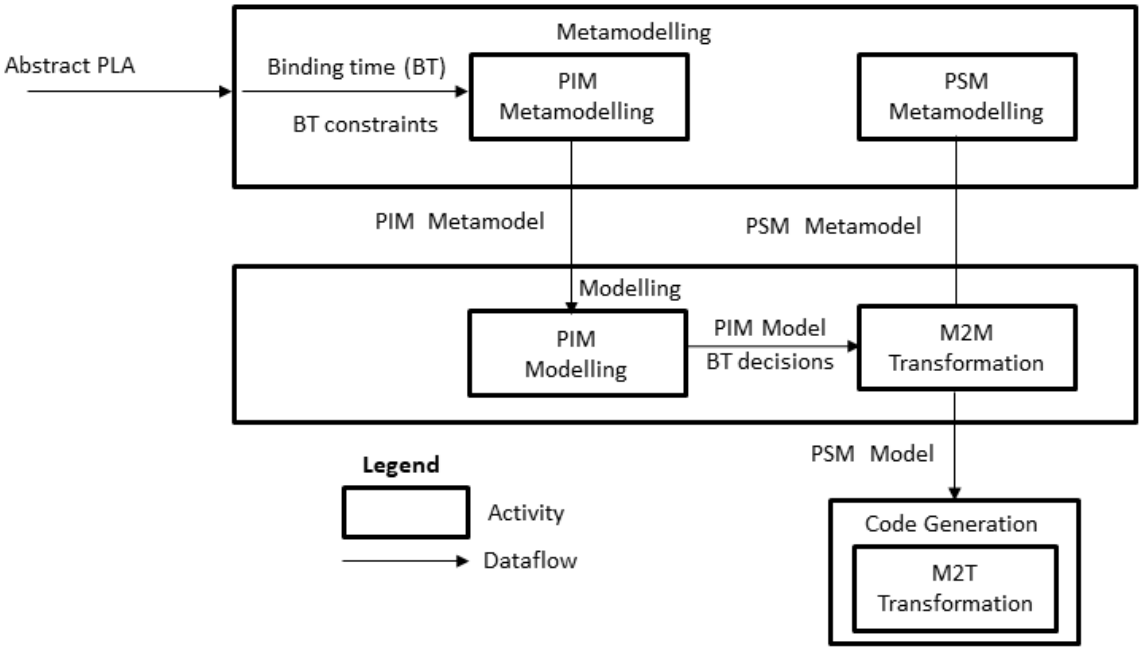


Figure 5.7: Lifecycle activities for supporting variations of feature binding time in Model-Drive domain implementation

is, generating executable and other artifacts from the PSM using model-to-text (M2T) transformation.

The following sections present the details of the MDD processes beginning from platform independent *metamodelling*.

5.1.3.1 Platform independent metamodelling with binding time

Fig.5.3 is a simplified *Component* and *Port* view of the *metamodel* for the proposed modelling language. From the top right of Fig.5.3, a model of PLA is composed of many elements of *ArchitecturalElement*.

To manage variations of feature binding time, the *metaclass ArchitecturalElements* has two enumerated properties: *BindingTime* and *ElementType*. *BindingTime* takes one of five possible values: *asset_dev*, *pre-deployment*, *deployment*, *post-deployment* or *undecided*. If an architectural element is assigned binding time properties of *asset_dev or pre-deployment*, then it has to be transformed into PSM elements that have to be included in the build path for compilation. In other words, other architectural elements can safely depend on program elements with binding time properties of *asset-dev or pre-deployment* because their final transformation into source codes has to support the static

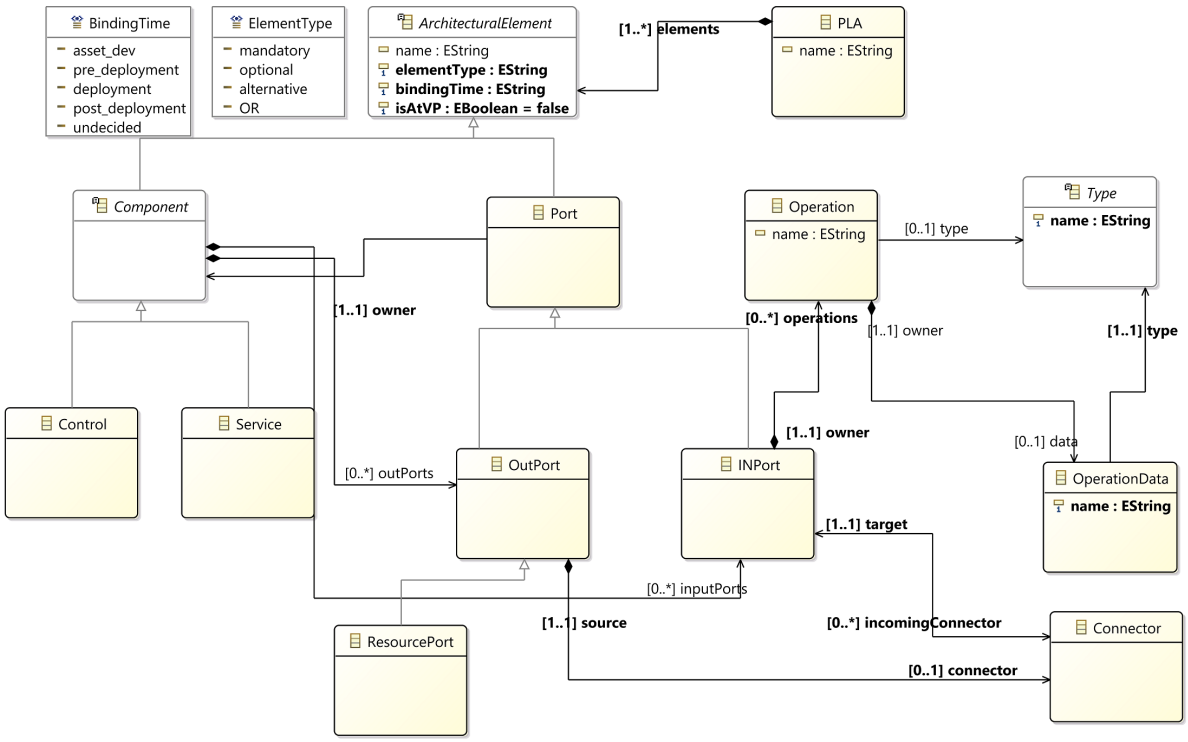


Figure 5.8: A simplifed view of the platform independent metamodel

connection between components.

If, however, an architectural element is assigned binding time property that is different from *asset_dev and pre-deployment* (including *undecided*), then no other architectural element should depend on it. The final transformation of dependencies with architectural elements in this category has to support putting and getting data or control from the deployment platform (i.e., the dynamic connection between components with no explicit dependencies). The binding time properties of *deployment* and *post-deployment* are used for generation of configuration interfaces for the *deployment* and *post-deployment* binding respectively.

ArchitecturalElement also has a Boolean property, *isatVP*, for checking whether or not an element is at a variation point. *EString* and *EBoolean* represent String and Boolean values, respectively, in *ECore* Language (the language we used to specify the *metamodel*). *Component* and *Port* extend *ArchitecturalElement* to inherit the *BindingTime* and *ElementType* as their properties.

Since at the domain design, the N-Tier architecture pattern have been adopted and passed on to the domain implementation as part of abstract PLA, *Component* has subtypes: *Control* and *Service* for modelling different kinds of components. *Service* is for modelling business logic captured from the analysis of the domain[LKCC00]. *Control* is for

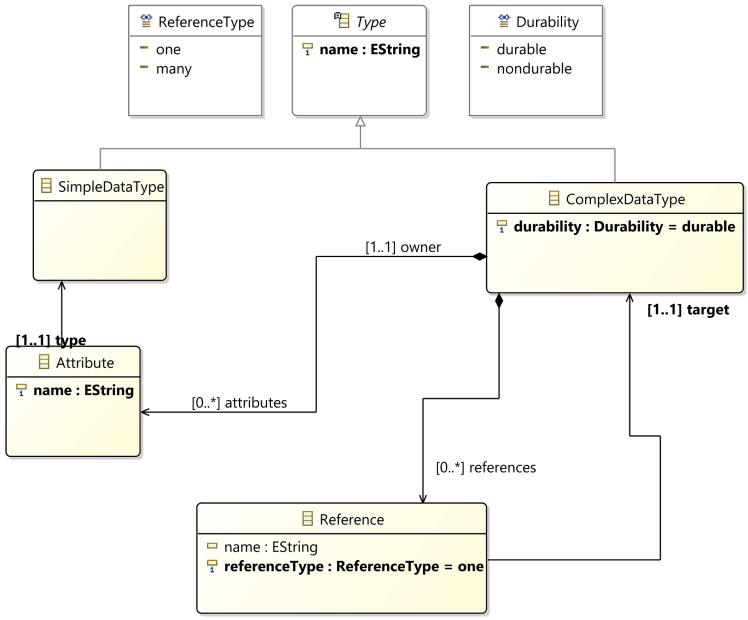


Figure 5.9: Type view of the PIM *metamodel*

modelling interaction between *Service* components and what will otherwise be component composition or gluing in non SPLE component-based development.

The *metaclass Port* is for modelling architectural ports on *Component* [WM98]. A *Port* is a gateway to a request either coming in or going out of component; encapsulates *Interfaces* and their *Operations*; can have additional properties such as synchronous or asynchronous. The *Port* in our model is a logical one; it is transformed into concrete port (or similar logic) in the target chosen platform based on its value of binding time. We introduced a *ResourcePort* as a subtype of *Port* for modelling connection to other resources such as connection to a file or a database.

Still on Fig.5.8, a *metaclass Connector* is for modelling connection between (compatible) ports[WM98]. Not shown in the model, to prevent cluttered diagram, is the inheritance relation from the *metaclass Connector*, to the *metaclass ArchitecturalElement*. According to Mehta *et al* [MMP00], a connector transfers, to other component(s), data or control or both. Hence, although not shown in Fig.5.8, *Component* has properties such as *sink* which is type of *Operation* to which the control or the data is passed to; *source*, which is also a type of *Operation* passing the data or the control; and *datapass* which is a data type (see Fig.5.9 for the type view of the PIM *metamodel*). A connector is defined from *OutPort* to *INPort*.

For completeness, we present the type view of the PIM in Fig.5.9. In the figure, *Type* is an abstract super type which has *ComplexDataType* and *SimpleDataType* as subtypes. *ComplexDataType* is a data type that either contains *SimpleDataDype* as attributes

or other *ComplexDataType* as references; has enumerated property, *Durable*, which indicates whether or not an instance will be stored for the application lifetime (represented as durable and non-durable respectively). *Reference* also has enumerated property, *ReffernceType*, representing cardinality of the instances being referenced.

Since each model elements has the variation type and binding time of its feature, assigning arbitrary values to the *BindingTime* properties of model elements may result in an ill-formed model. In the next section, we discuss how the consistency of binding time can be maintained on the architectural model.

5.1.3.2 Binding time constraints on PIM *metamodel*

Consistency constraints, to enforce valid combinations of features, can be specified on a feature model using prepositional formula [Bat05a]. Binding time consistency may also be added on a features using the same approach of propositional formula, and then propagated to architectural elements. However, we choose to add the binding time constraints on architectural elements to allow independent testing of the architecture model without depending on consistency constraints from the variability model. The Listings from Listing 5.1 through Listing 5.4 are examples of pseudo codes, representing constraints that are relevant to the binding time of architectural elements.

Listing 5.1: Restriction of changing binding time on mandatory *ArchitecturalElement*

```
1: self : ArchitecturalElement
2: invariant btRestriction :
3: if self.elemenType = mandatory
4: self .bindingTime = readOnly
6: end if
```

Listing 5.1 is a specification to prevent changing the binding time of a mandatory architectural element. A binding time of an architectural element can only be changed when the variation type of its corresponding feature is changed from mandatory to variable.

Listing 5.2: Connected ports must have the same binding time

```
1: self : Connector
2: invariant btRCompatible :
3: self.bindingTime =
4: self.source.bindingTime
5: = self.target.bindingTime
```

Listing 5.2 is a constraint defined on *Connector* to enforce the same binding time on its source and target *Ports*. This, to prevent a situation where the source codes, eventually generated from the model, fail compilation because certain program elements that should be in the build path are not included.

Listing 5.3: *OutPort* must have reference to *Connector*

```
1: self : OutPort
2: invariant btmustHavewire:
3: self.connector != null and
4: self.bindingTime =
5: connector.bindingTime
```

Listing 5.3 is a constraint defined on *OutPort* to enforce the presence of a reference to *Connector*. Since *OutPort* encapsulates an interface requirement, it must have a reference to a provided interface. However, the reference can represent a direct static connection or putting data/control to the deployment platform.

Listing 5.4: *Port* should not have binding time earlier than its owner *Component*.

```
1: self : Port
2: invariant notEalierthanOwner:
3: if self.isVariable() and
4: self.owner.isVariable() then
5: if self.bindingTime =
6: 'product_dev' then
7: owner.bindingTime <> 'undecided' and
8: owner.bindingTime <> 'deployment' and
9: ...
```

Listing 4.4 is a constraint on *Port* not to assume binding time earlier than its owner *Component*. If a *Port* is variable while its owner *Component* is mandatory, there should be no problem of binding time inconsistency because a binding time of a mandatory architectural element is fixed by default. However, if a *Port* is variable and its owner component is also variable, binding time consistency will have to be enforced to ensure that the binding time of the port is not earlier than that of its owner component.

Next section presents the specification of platform-specific model.

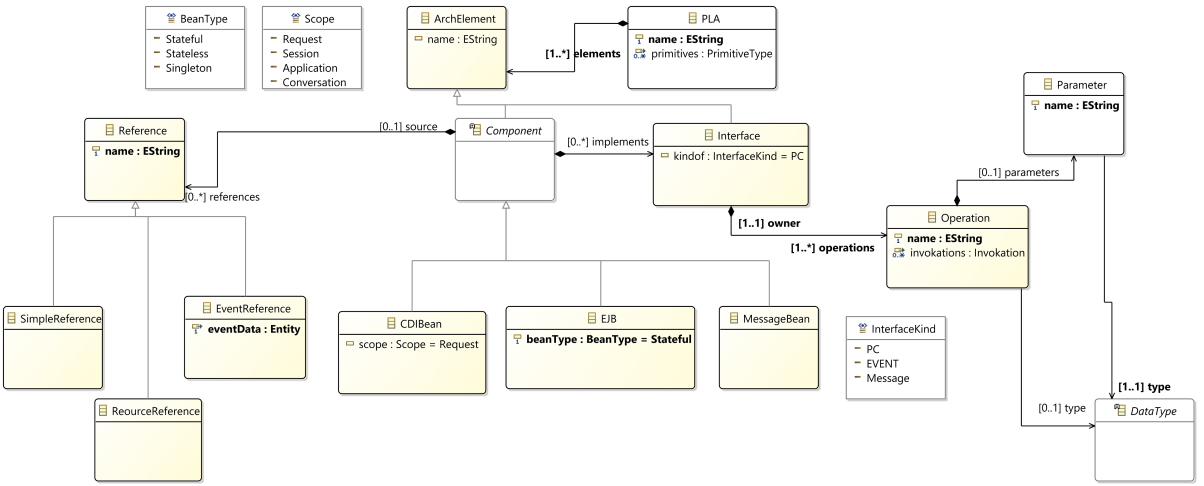


Figure 5.10: A simplifed metamodel of Java Enterpriese Edition (JEE)

5.1.3.3 Platform specifc (PSM) *metamodelling* with Java Enterprise Edition (JEE)

We selected Java Enterprise Edition (JEE)[Tec], a platform for developing distributed server-side applications as an example of a specific platform. JEE offers infrastructure, in the form of Application Programming Interfaces (APIs), to simplify enterprise system development. JEE 'container' manages components and provides commonly used services such as multi-threading, resource pooling, and security. Recently, the support for light-weight event-driven messaging is one of the services added to the JEE container. As there is no official Unified Modelling Language (UML) profile for the recent version of JEE, we used a subset of the JEE APIs that are sufficient to illustrate the binding concept in the PSM metamodel.

Fig.5.10 depicts a simplified *Component* and *Interface* view of JEE *metamodel*. From the top right of Fig.5.10, PLA consists of elements of *ArchELement*. *Component* and *Interface* extend *ArchElement*. *Component* implements 0 or more *Interfaces* and each *Interface* has 1 or more *Operations*. *CDIBean* is a subtype of *Component* used for modelling control component that can either be coupled with the user interface (UI) or other client components. Scope on *CDIBean* is specific to a web application lifecycle: a state of request to a web component may last only a single request (request scope), spans across multiple requests for a single user session (session scope), maintains conversation across multiple requests (conversation scope) or lasts for the entire application scope (application scope). *EJB* is for modelling business logic that runs on the application server. *MessageBean* is for modelling component that creates and process messages in a loosely coupled and asynchronous manner.

As JEE components have no notion of *Port*, we introduced *InterfaceKind* as a prop-

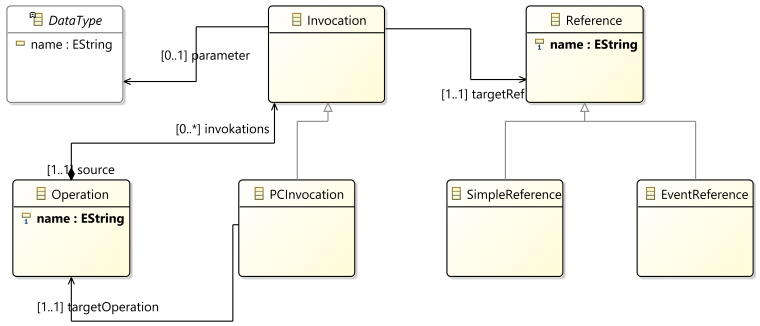


Figure 5.11: Java Enterpriese Edition (JEE) Invocation *metamodel*

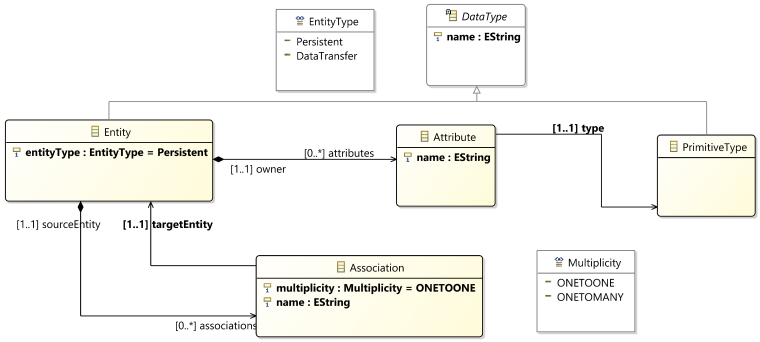


Figure 5.12: Java Persistence API (JPA) metamodel

erty of *Interface*, with three enumerated values: EVENT, PC, and *Message*. EVENT is for modelling even driven invocation, PC is for modelling invocation using procedure call, and *Message* for modelling message processing. The distinguishing properties are propagated down to the operations contained within an interface. For example, if *InterfaceKind* = EVENT, its operations will only support event-driven invocations. Synchronous or asynchronous property may be modelled in a similar manner.

A connection model among JEE components is either through procedure call (remote or local) or messaging (event, or message). As such, we model connection as *Invocation* (see Fig.5.11). *Invocation* has *Operation* as its source and *Reference* as its target. *SimpleRefence* and *EventReference* are subtypes of *Reference*. *SimpleReference* is a reference to an interface upon which an operation will be invoked in the case of a procedure call. *EventReference* is a reference to an event that will be fired by the invocation in the case of event invocation. *EVentReference* has an additional property, *eventdata*, which may hold data to be emitted when the event is fired.

Again, for completeness, the *metamodel* of JEE datatype is depicted in Fig.12. This view is compliant with the Java Persistence API (JPA) - an API for persisting data of Java application into a permanent storage using object-relational mapping technology.

Table 5.1: PIM *Port* to JEE *Interface* mapping

PIM	JEE	Condition
INPort	*Interface*, *InterfaceKind* = *PC*	binding time is decided and no later than pre-deployment
INPort	*Interface*, *InterfaceKind* = *EVENT*	binding time is either undecided or later than pre-deploymentt
OutPort	*SimpleReference*	binding time is decided and no later than pre-deployment
OutPort	*EventReference*	binding time is either undecided or later than pre-deployment

A JPA *Entity* is a data object that is persisted to permanent storage. In our model, we introduced *DataTransfer* as a type of an entity that holds data to be transferred between components(see Fig.5.12).

The full specification of JEE *metamodel*in XMI can be found in *Appendix B.*

5.1.4 PIM to PSM mapping

As mention earlier, instances of PSM are not instantiated manually but by model-to-model (M2M) transformation that takes an instance of PIM as input and produces an instance of PSM as output. Table 5.1 through Table 5.3 present the mapping from PIM model elements to JEE model elements. We mapped each model element of PIM to model element in PSM in consideration of its binding time property. In general, a mandatory *Component* is not made to depend on a variable *Component* if the binding time of the variable *Component* is later than pre-deployment time. This translates into support for independent compilation if the binding time is later than product development time.

Table 5.1 presents the mapping from PIM *Port* to JEE *Interface*. Each instance of *InPort* in PIM is transformed into an instance of *Interface* in JEE. The *IntefaceKind* attribute is set to PC if the binding time is decided and its value is not later than *pre-deployment* (first row in Table 5.1), otherwise, the *InterfaceKind* is set to EVENT (second row in Table 5.1). Similarly, each instance of *OutPort* in PIM is transformed to an instance of *SimpleReference* in JEE if the binding time is decided and its value is not later than pre-deployment (3rd row in Table 5.1), otherwise the instance of the *OutPort* is transformed into *EventRefence* (last row in Table 5.1)

Table 5.2 represents a straight-forward mapping transformation, without any condition, from *Control* in PIM *to CDIBean* in JEE, *Operation* in PIM to *Operation* in JEE, and *OperationData* in PIM to *Parameter* in JEE. In addition, an instance of *Connector* in PIM is transformed to an instance of *PCInvokation* in JEE if the binding time is de-

Table 5.2: PIM *Component* to JEE *Component* mapping

Control	*CDIBean*	None
Connector	*PCInvokation*	Connector binding time no later than pre-deployment
Connector	*EventInvokation*	Connector binding time is either undecided or later than pre-deployment
Operation	*Operation*	None
OperationData	*Parameter*	None

Table 5.3: PIM *Type* to JEE *DataType*

PIM	**JEE**	**Condition**
SimpleDataType	*PrimitiveType*	None
ComplexDataType	*PersistenEntity*	Durability is durable
ComplexDataType	*DataTransfer*	Durability is nondurable
Attribute	*Attribute*	None
Reference	*Association*, Multiplicity = ONETOONE	Reference type is one
Reference	*Association*, Multiplicity = ONETOMANY	Reference type is many

cided and its value is not later than *pre-deployment* (2nd row in Table 5.2), otherwise the instance of the *Connector* is transformed to *EventInvokation* (3rd row in Table 5.2).

Table 5.3 presents mapping from *Type* in PIM to *DataType* in JEE. An instance of *SimpleDataType* in PIM is transformed into an instance of *PrimitiveDataType* in JEE; an instance of *ComplexDataType* in PIM is transformed to an instance of *PersistentEnitity* in JEE if its *Durability* = *durable*, otherwise it is transformed to *DataTransfer* in JEE; instance of *Attribute* in PIM is transformed to instance of *Attribute* in JEE; instance of *Reference* in PIM is transformed to *Association* in JEE. The *Multiplicity* of *Association* is set to ONETOONE if the corresponding *ReferenceType* is one and is set to ONETOMANY if the corresponding reference is many.

So far we have been discussing the detail approach to supporting variations of feature binding time at domain engineering phase. In the next section, we briefly explain the application engineering with the choice of feature binding time.

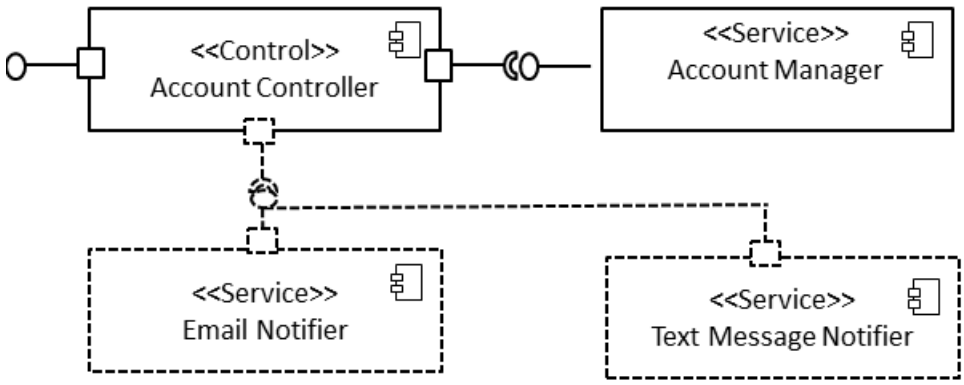


Figure 5.13: Architecture of *Account Management* sub-system when *Advance* child feature of *User Notification* is selected.

5.2 Application engineering with feature binding time

This section briefly highlights the feature binding time activities while deriving a specific product of a product line at the application engineering phase. We begin with the requirement elicitation in the next section.

5.2.1 Product requirement elicitation

To elicit requirements of a specific product, in feature-oriented SPLE, is to match the user-requirements to the pre-planned features and to derive analysis model based on the selected features. We illustrated product requirement elicitation in detail in *Application engineering example* section of *Chapter two* and, therefore, we omit the examples of how to derive the analysis models here.

In our approach, in addition to matching the user requirements to features, the binding time requirements of the selected variable features will have to be determined at this stage in consideration with the product usage-context. If the binding time of some variable features cannot be determined, pre-deployment binding time is eliminated from the possible binding time.

The abstract product architecture is then derived based on the matched features. Recall using the example of *Account Management* sub-system where the user selected *Email* and *Text Message* as variable features, we obtain the abstract product architecture depicted in Fig.5.13. The specification of the binding time together with the derived product architecture becomes part of product specification which will then be passed-on to Product design activity.

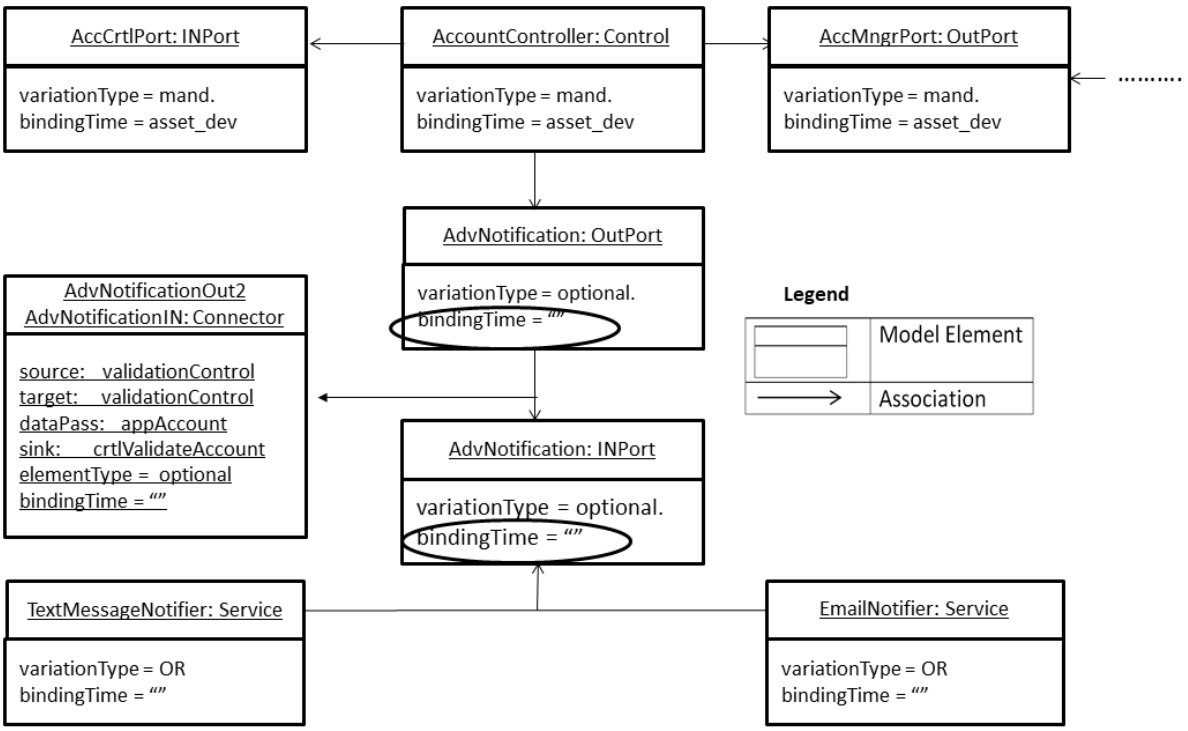


Figure 5.14: Instance of PIM representing partial architecture of *Account Management* sub-system when *Advance* child feature of *User Notification* is selected

5.2.2 Product design

Product design in our approach is comprised of three activities: (i) Instantiation of the abstract architecture, derived from the requirement analysis, using the implemented PIM. (ii) Checking binding time consistencies of the instantiated PIM model elements. (iii) Transformation of the well-formed PIM to PSM.

To instantiate the product architecture of Fig.5.13 using the PIM we specified in the PIM *metamodelling* section, we obtain the instance of PIM architecture model in Fig.5.14. In the figure, each box represents a model element. The text before the colon in the top compartment of each model element is the name of the model element as assigned by the engineer; the text after the column is the name of PIM class from which the model element is instantiated.

In Fig.5.14 the middle model element at the top of the figure, *AccountController*, represents A*ccount Controller* component and its associated model elements at its left and right represent instances of *INPort* and *OutPort* respectively. Note that the instance of the service component *Account Manager* is omitted in the figure.

The model element beneath *AccountController* represents the optional *OutPort* of

the *Account Controller* and the port is one of the points of intersection between the *User Account* feature and the *Advance (User Notification)* feature. In this case, no need to remove the optional *OutPort* because it suffices to remove the dependency between the intersecting features by making the port a gateway to putting and getting data but not tightly coupled connection should the binding time be later than pre-deployment time. Had it been it is an operation or attribute, it may have to be re-align or becomes part of an aspectual component.

The bottom two model elements and from left to right, *TextMessageNotifier* and *EmailNotifier* represent *Text Message Notifier* and *Email Notifier* service components respectively and both of which provides the *AdVNotification* instance of *INPort*. The *AdvNotification* instance of *OutPort*, that is on *AccountController*, is connected to *AdvNotification* INPort, that may be on *TextMessageNotifier* or *EmailNotifier*, using *AdvNotificationOut2AdvNotificationIN* connector. That is, *AdvNotificationOut2AdvNotificationIN* is an instance of *Connector* that joins *Account Controller* with either *Message Notifier* or *Email Notifier* or both.

In our approach, the binding time of the variable features will have to be reflected on the model elements representing the architectural model. As indicated earlier, this often means that a binding time of a single feature has to be reflected on many model elements. For example, all three model elements at the top of the figure are exclusive to the *User Account* feature. Thus, binding time property of the *User Account* feature is reflected on each of the three model elements at the top of the figure.

The instantiated model elements are then subjected to binding time consistency checking and we discuss this in the next section.

5.2.3 Consistency checking and PIM to PSM transformation

Consistency checking is a check for well-formedness of model elements with respect to binding time. The checking can be in relation to the aggregate binding time of model elements of a single feature. For example, in the simple example of product architecture of Fig.5.14, we would want to check if none of the mandatory architectural elements, for the *User Account* feature, at the top of the figure has been deliberately or accidentally changed from *pre-deployment* to something else. The consistency checking can also be in relation to compatibility of binding time of model elements across multiple features. For example, the *AdvNotification OutPort* and *AdvNotification INPort* must have binding time that is compatible with the *AdvNotificationOut2AdvNotificationIN* connector.

The transformation of the PIM architecture into its corresponding PSM architecture in our approach is a simple push-button. The transformation engines selects model element of the PSM based on the value of binding time properties of the PIM element.

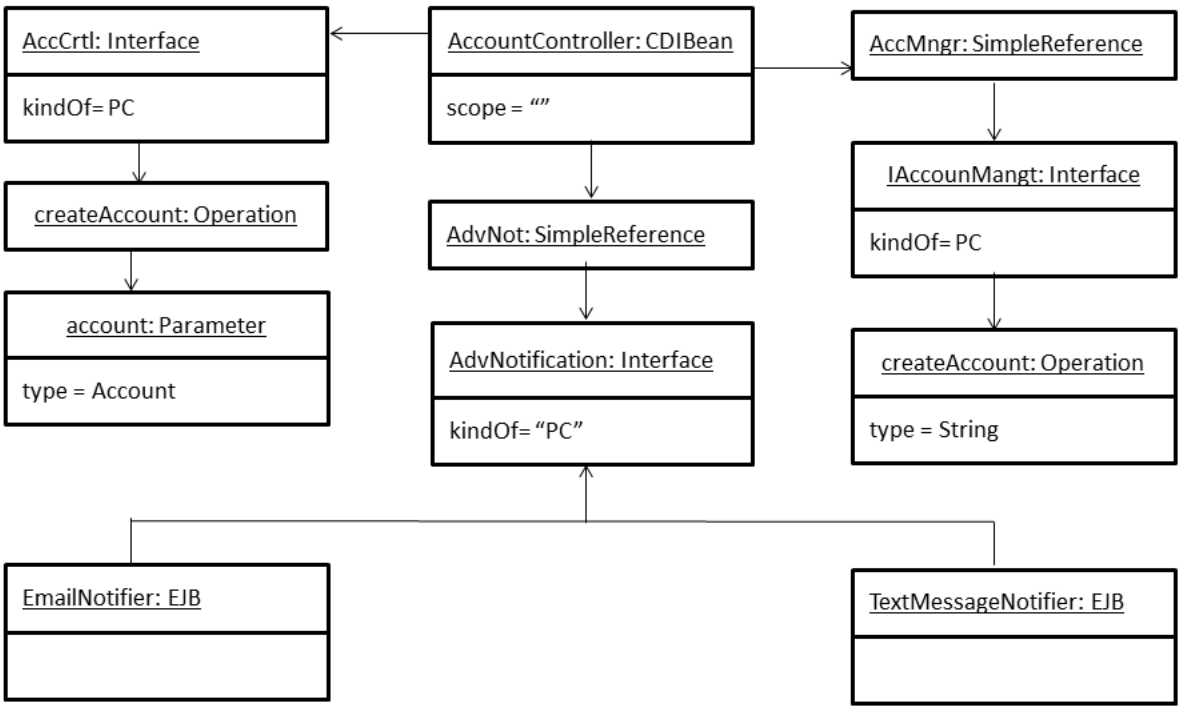


Figure 5.15: JEE platform specific instance model in which the binding time of *Advacnce* (*User Notification* feature is set to *pre-deployment*

Fig.5.15 depicts Java EE instance model as the target platform-specific model of Fig.5.14 platform independent model when the binding time of the *Advance*(*User Notification*) feature is set to *pre-deployment*. In the middle of the figure, the instance of the *SimpleReference*, *AdvNot*, is the result of transforming *AdvNotification* instance of *OutPort* of the model in Fig.14; The *Kindof = "PC"* property of *AdvNotification* instance of *Interface* is the result of transforming *AdvNotification* instance of *INPort* of the model in Fig.14. Not shown in the figure is the procedure call invocation which is a result of transforming*AdvNotificationOut2AdvNotificationIN* instance of *Connector*

Similarly, Fig.5.16 depicts Java EE instance model as the target platform-specific model of Fig.5.14 platform independent model when the binding time of the *Advance*(*User Notification*) feature is later than *pre-deployment* or left *undecided*. In the middle of the figure, the instance of the *EventReference*, *AccEvent*, is the result of transforming *AdvNotification* instance of *OutPort* of the model in Fig.14; The *Kindof = "Event"* property of *AdvNotification* instance of *Interface* is the result of transforming *AdvNotification* instance of *INPort* of the model in Fig.14. Not shown in the figure is the event firing as a result of transforming*AdvNotificationOut2AdvNotificationIN* instance of *Connector*.

At this stage, the PSM should be well-formed and with the binding time decisions reflected (both the selected binding time and the ones that are left open).

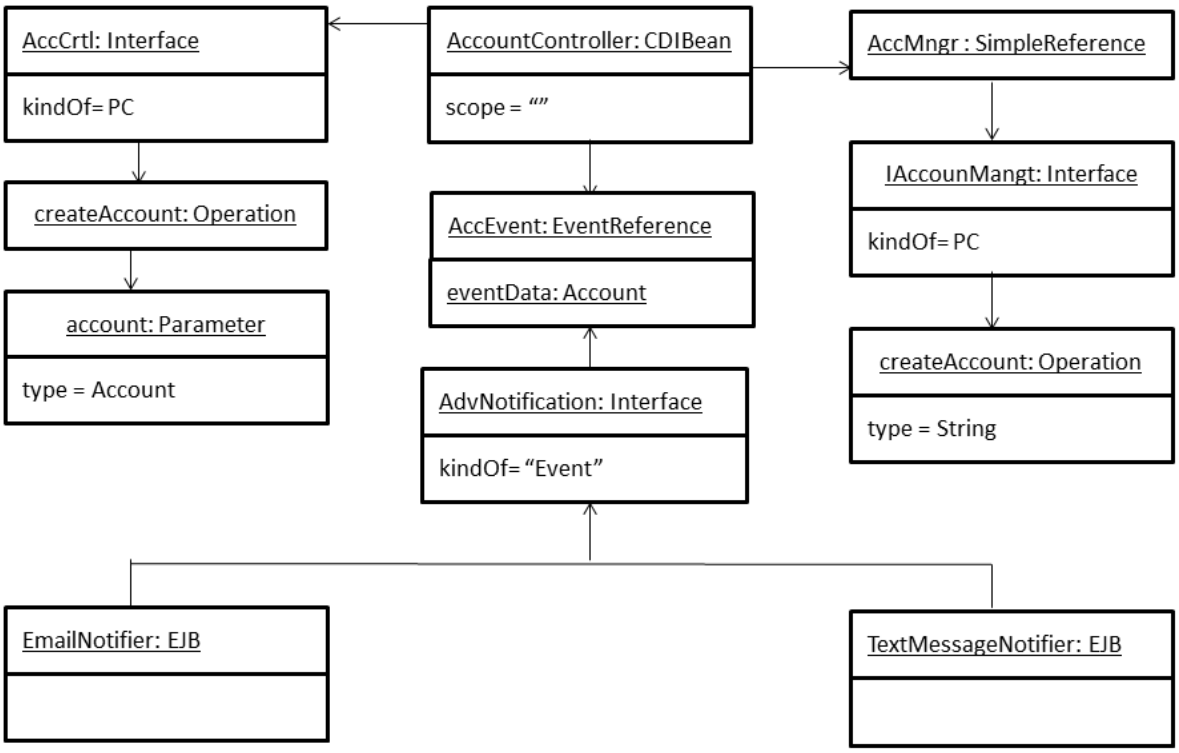


Figure 5.16: JEE platform specific instance model in which the binding time of *Advacnce* (*User Notification* feature is set to *pre-deployment*

5.2.4 Product instantiation

At the product instantiation stage, the skeletal product is generated from the well-formed platform specific model of the specific product. The key design decisions, including the architectural pattern/style and the binding time, are enforced in the skeletal product. The product will then have to be implemented manually or reuse from an existing implementation of similar products. The manual implementation may also be incorporated in the specification of the code generation and preserve against loss in the subsequent re-generation[GHK+15]. Similarly, product-specific extensions that are not fed back to the reusable asset are also implemented and integrated.

In the generated skeletal product, connection modes between the generated components depend on the values of the binding time. The components are connected directly when the binding time is decided before compilation at pre-deployment time. In that case, components should support compile-time and build time composition, do not have to follow separation between implementation and interface and may be followed with a static optimization using a specialized tool. If, however, the binding time cannot be decided at pre-deployment time, dependencies from mandatory components to variable components are removed. The components are connected dynamically, if needed, via a deployment platform.

The process of supporting variations of feature binding time in our approach should be supported with a tool suite. In the next section, we discuss the tooling aspect of our approach.

5.3 Tool Support

Supporting variations of feature binding time in model-driven development has some inherent exigencies that should be supported with tool for greate utility of the approach. In this section, we highlight the implementation and the utility of tooling aspect of our approach.

5.3.1 PIM implementation and instantiation

We used *ECore*, a simplified implementation of meta object facility (MOF)[Gro16] of Eclipse Modelling Framework[1] project, to implement the *metamodel* as an Eclipse plugin. *Ecore* is a *metametamodel* - a language for creating a *metamodel*. A light-weight *metamodel* can as well be derived from a custom extension of UML *metamodel* and the

[1]https://www.eclipse.org/modeling/emf/

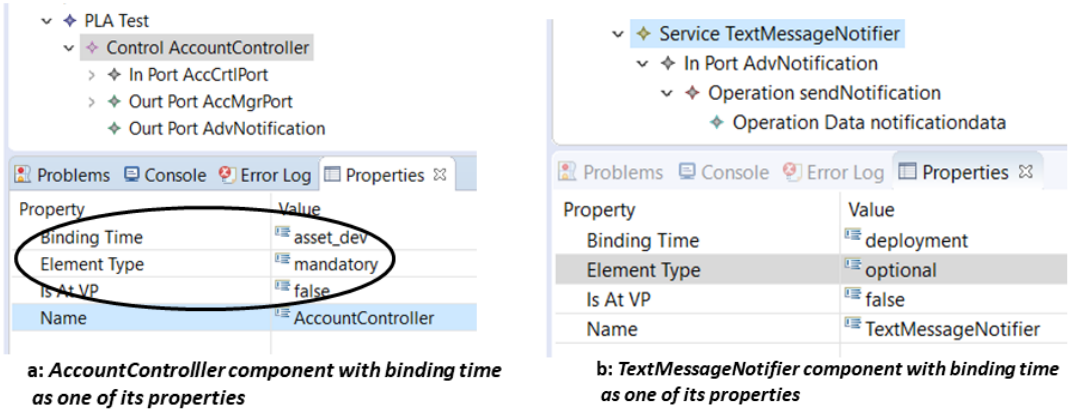


Figure 5.17: Components modelled with the eclipse-based plugin for PIM modelling

extension is known as UML Profile[LWWC11]. The full specification of the *metamodel* of Fig.5.8 in *Platform independent metamodelling* section with binding time can be found in *Appendix A*. Another approach we could have taken is to specify a grammar and implement a compiler for the binding-time aware modelling language. However, the *metamodelling* approach is preferable for interoperability with other tools.

Fig.5.17 is a screenshot of partial PIM architectural elements modelled with the implemented *metamodel* of which the specification is presented in *Appendix A*. The graphical interface is Reflective Ecore Model Diagram Editor which is a plugin from Eclipse Modelling Project (EMP) that provides a graphical editor for any EMF model file, using only the meta-model such as .ecore and .xsd file. Fig.5.17a, and Fig.5.17b, highlight *AccountController* and *TextMessageNotifier* intances of *Component*. The circled parts show some of the components' properties that include *BindingTime* and *ElementType*.

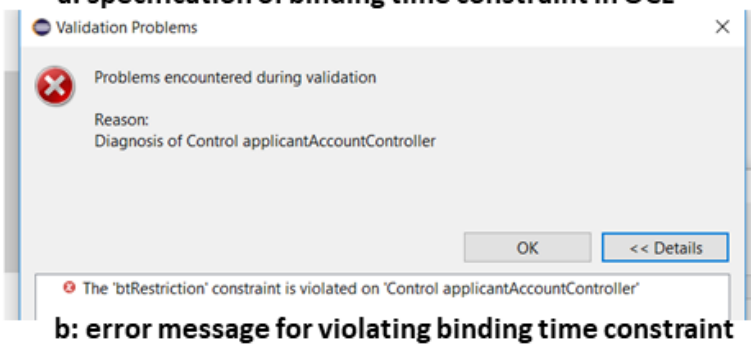
5.3.2 Consistency checking

We implemented the binding time consistency specification as constraints in Object Constraint Language (OCL)[Gro10] that are embedded in the *ECore metamodel*. For example, the code snippet in Fig.5.18a shows an implementation of a Boolean operation on *ArchitecturalElement* to check whether an element is variable or not and this operation is inherited by *Component*, *Port* and *Connector*. Lines 17 to 21 in Fig.5.18a show the implementation of the constraint in listing 1 (a mandatory *ArchitecturalElement* should not have binding time different from core asset development time). Fig.5.18b shows an example of the validation error when the constraint in listing 1 is violated. Full specification of the binding time constraints can be found embedded in the PIM metamodel of *Appendix A*.

A successfully validated model is then saved in an exchangeable format(i.e., in

```
operation isVariable() : Boolean[?]
    {
body: if self.elementType <> 'mandatory' then true
    else false  endif;
    }

invariant       btRestriction:
self.elementType = 'mandatory' implies
self.bindingTime <> 'pre_deployment' and self.bindingTime <>
'deployment' and self.bindingTime <> 'post_deployment'
and self.bindingTime <> 'undecided';
```

a: specification of binding time constraint in OCL



b: error message for violating binding time constraint

Figure 5.18: OCL implementation

XML *Metadata* Interchange (XMI)) for the next transformation. We illustrate the model to model transformation in the next section.

5.3.3 PIM to JEE model transformation and code generation

We implemented the model-to-model (M2M) transformation in Atlas Transformation Language (ATL) [JABK08]. Fig.5.19 shows thespecification for transforming *INPort* in PIM, based on the *INPort*'s *bindingTime* property, to either *PC* or *EVENT Interface* in the target JEE PSM. The full transformation definition can be found in *Appendix C*. The output of the transformation is JEE model from which we generate the skeletal product with the key design decisions such as binding time and architectural style enforced.

***PSM to source code transformation*:**
We implemented the model-to-text transformation (M2T), aka code generation, in Xtend[2]. Xtend is a statically-typed programming language optimised for template-based code generation. The transformation from the PSM to code in Java is a straight-forward and is illustrated in Fig.5.20.

[2]https://www.eclipse.org/xtend/

```
helper context PIM!INPort def: isPCPort(): Boolean =
    if self.bindingTime = 'asset_dev' or
    self.bindingTime = 'pre_deployment' then
        true
    else
        false
    endif;

rule Port2Interface {
    from pt: PIM!INPort(pt.isPCPort())
    to it: JEE!Interface (name <- pt.name, kindof <- 'PC',
        operations  <- pt.operations)
}
rule Port2EventInterface {
    from pt: PIM!INPort (not pt.isPCPort())
    to it: JEE!Interface (name <- pt.name,  kindof <- 'EVENT',
        operations <- pt.operations)
}
```

Figure 5.19: PIM to JEE transformation

```
GeneratorWorkFlow.java   RefinedPIM2JEE.atl   *BusinessComponentGenerator.xtend
def compilOperation(Operation o) {
        '''
public «o.type?.name.toFirstUpper ?: 'void'»
 «o.name»(«o.owner.kindof.interfaceType» «IF o.parameters != null »
«o.parameters.type.name.toFirstUpper» «o.parameters.name»«ENDIF»){
 «FOR inv : o.invokations»
   «inv.compileInvokation»
 «ENDFOR»
    «IF o.type  != null»
    return null;
    «ENDIF»
    }
        '''
    }

def String interfaceType(InterfaceKind itf) {
 if (itf.toString.equals('EVENT')) {
    '@observes'
} else {
            ''
        }
    }

    def dispatch compileInvokation(Invokation invk) {
        '''
«invk.targetRef.name.toFirstLower».fire(«invk.parameter.name»);
'''
    }

    def dispatch compileInvokation(PCInvokation pInvk) {
        '''
```

Figure 5.20: PSM to source code transformation

5.4 Evaluation an discussion

In this section, we present the evaluation of our approach with respect to performance. Also, we discuss the approach with respect to the modifiability of the implementation artefact.

5.4.1 Performance

Recall that we rely on the deployment platform and the removal of dependencies between components to delay the binding if it cannot be decided at pre-deployment time. We can also use the same connection to the deployment platform for the early binding, at pre-deployment time, but we then lose the opportunity to optimise statically and, therefore, compromise performance. In other words, we use the direct static connection to improve performance when the additional flexibility is not required because the binding can be decided before compilation. Consequently, the evaluation is to test how significant is the performance gain for the direct connection.

We set up an experimental environment[3] to compare the performance of the two modes of components connection: i) Direct mode and ii) Platform mode. For the platform mode, we used the JEE event model which supports components communication without any compile-time dependency. The test case for the experiment is: "a user requests to create an account and the server responds with a sign-up form. Once the user fills the form and submits, the server creates a record in the database for the user. The server also generates and sends a token to the user, via email and text message, for confirmation" The test case involves the component implementation of the following features: *Applicant Account* (mandatory), *Applicant Validation* (optional), *Message Service* (optional), *Email* (OR-group) and *Text Message*(OR-group).

In the result tables, the *Users* column indicates the number of concurrent users per request and the *Ramp up* column represents the time interval (in seconds) between requests in the case of multi-user. The two *Res.* columns specify the average response time (delay) of Direct and Platform modes, also in seconds. The *Diff.* column represents the difference response time, in seconds, between the two modes. The *%Diff.* represents the percentage differences of response time between the two modes. Note that the lower values of *Res.* means less delay- which implies better performance.

Table 5.4 presents the results of the comparison between the Direct mode and the Platform mode when the components are configured to execute synchronously. Referring

[3]The components were deployed on Glassfish 4.1, an open-source implementation of the JEE container. A load test was run with *Apache JMeter* https://jmeter.apache.org/ testing tool on Dell Computer with 2.9 GHz Intel® Core ™ i7 and 16GB of RAM.

Table 5.4: Response time comparison of Direct and Platform modes for synchronous component implementation

Users	**Ramp up**(sec)	**Res. (Direct)(sec)**	**Res. (Platform** (sec)	**Diff.**(sec)	**%Diff.**
1	na	1.22	1.01	-0.22	-17.43
20	20	0.95	3.27	2.32	243.23
50	25	4.94	7.44	2.50	50.63
100	30	11.33	16.02	4.682	41.31

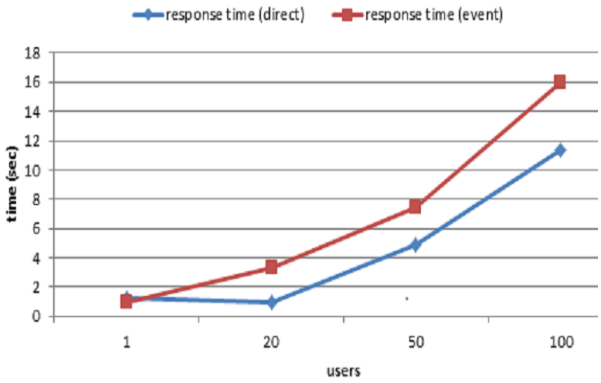


Figure 5.21: Comparison between Direct and Platform in synchronous mode

to Table 5.4, the response time increases with the increase in the number of users for both modes (see Fig.5.21). With a single user (the first row in Table 5.4, the response time was slightly lower (which means faster response and less delay) for the Platform mode by 0.22 seconds (-17.43%) [4]. This is because of the components for sending a text message and email react to the Platform event simultaneously. In all other cases, the response time is slightly lower (which means faster and less delay) in the Direct mode because of the absence of Platform involvement. Fig.5.21 presents the graph of the result of Table 5.4. Overall, the Direct mode offers performance benefit in an execution environment that requires synchronous component connection.

Next, we configured asynchronous component connections on all ports at the variation points and this means the components can be executed simultaneously. Table 5.5 presents the result with the asynchronous component connections.

It can be observed, from Table 5.5 and Fig.5.22, that the response time remains fairly stable with the Platform mode, even with the increase of requests, but increases drastically from an average of 0.90 sec for one user to an average of 13.04 for 100 users with the Direct mode. The results show that the Direct mode limits multi-processing capability in the asynchronous case. Put in another way, when asynchronous communication between components is configured, the Platform mode offers more multi-threading capability than

[4]In this case, we averaged the results of 5 test runs to increase the accuracy of testing.

Table 5.5: Response time comparison of Direct and Platform modes for asynchronous component implementation

Users	**Ramp up**(sec)	**Res. (Direct)(sec)**	**Res. (Platform** (sec)	**Diff.**(sec)	**%Diff.**
1	na	0.90	0.38	-0.52	-57.95
20	20	1.66	0.36	-1.302	-78.56
50	25	6.96	0.40	-6.57	-94.32
100	30	13.04	0.35	-12.69	-97.32

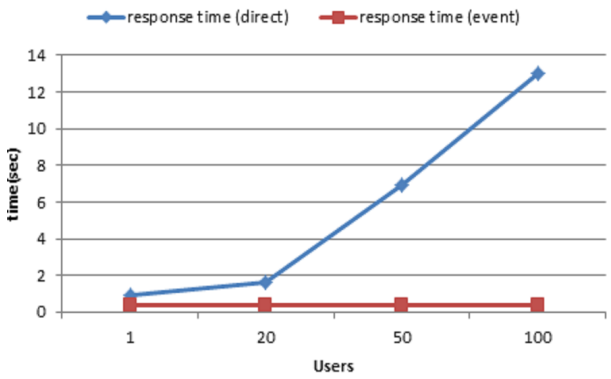


Figure 5.22: Comparison between Direct and Platform in asynchronous mode

the Direct mode.

As such, switching to the Direct mode for the early binding offers benefit only when the synchronous mode of communication between components is desirable. In contrast, a complete loose coupling, using the Platform mode offers more flexibility and performance because the platform-induced delay is compensated with an optimized multi-threading capability. Thus, usage-contexts should also be parametrized and apply the switching only where it is needed and go with full flexibility in where it is not needed.

5.4.2 Modifiability

Recall from chapter four (*Language-based approaches to flexible variations: Action research*), adapting code-asset to inject additional variations using both the language-based implementation techniques and the pre-processing of annotations require making changes to several places in the code-asset. In the case of language-based implementation techniques, often the code-asset has to be re-organized. In contrast, when binding time aware modelling language is used to implement a product line based on our proposed approach, the changes will be made at fewer places (at the level of architectural elements and in the transformation definition). Therefore, our approach may be more modifiable than language based variability implementation techniques.

In addition, the support for variations of feature binding time with language-based implementation techniques has been either through several clones of code-asset (in practice [DRB+13]) or through multiple implementations of features as proposed in academic researches [CRE08, ARR+16].For example, to support variable feature binding time, Chackravathy *et al* [CRE08] proposed separate implementations of features in aspect-oriented programming. In their approach, a single variant feature would have multiple implementations and each of them is the same variant feature with different binding times. Andrade *et al* [ARR+16] worked along the same line, in direct comparison to the approach in [CRE08], and claimed to have reduced only the code redundancy but retained the multiple implementations of a feature.

In contrast to existing approaches, when binding time aware modelling language is used to implement a product line based on our proposed approach, each feature has single representation at the PIM level, the variations of feature binding time only affects how the PIM is transformed into PSM model elements but no multiple representations of a feature is required. Rather, the mode of component-interaction mechanisms are interchangeable based on the chosen binding time and this creates fewer modifiability problems than the case of [CRE08] and [ARR+16]. Our approach could achieve this as we abstract away the actual binding mechanism and manages it at the architecture level.

5.5 Chapter summary

In this chapter, we presented the design and implementation of the binding-aware modelling language to support variations of feature binding time. To demonstrate the feasibility of our approach, we validated the language with a case study of EduPL. However, more works need to be done to enrich the semantics for the *metamodels*. For example, semantics to aggregate the fine-grained variations into aspectual components should be integrated into the PIM and PSM *metamodels*. Other architectural concerns should not only be supported but should also be verifiable for violations.

Chapter 6

Conclusion

This chapter presents a summary of the previous chapters and puts our contributions in perspective. The chapter also highlights the inherent limitations of the thesis and discusses grounds for future works.

6.1 Thesis summary

The *Chapter one* provided a general overview to set the context of the thesis. We began from the background theory of the thesis, which is a reuse from the perspective of software engineering, and progressed to the focal theory, which is adaptable variations and feature binding time in software product line engineering. Also, the chapter highlighted the research motivation, spelled out the research objectives, and followed with the research questions. Finally, the chapter outlined the research contributions and presented an overview of the research methods that bound the research processes.

Chapter two elaborated on the feature-oriented approach to the lifecycle activities of both domain engineering and application engineering. The illustrated, with examples, the concepts of adaptable variations, feature binding time in software product line engineering, and the research challenges associated with both.

Chapter three presented the details of the action research that was executed to investigate the adaptability of reusable assets when adding variations. Part of the investigation is an exploration of the properties of features, in the code-asset, that affect modifiability when injecting additional variations. In the exploration, we observed that an intersection between program elements of different features negatively affects the modifiability of code-assets when injecting additional variations.

In addition to the exploration, the chapter reported the selections and comparison of the following three language-based implementation techniques: (i) Feature-Oriented Programming (FOP), (ii) Aspect-Oriented Programming (AOP), and (iii) Delta-Oriented Programming (DOP). We compared the techniques relative to Pre-Processing (PP) of annotations (a classical variability implementation technique) using modifiability metrics.

The summary of the comparison is none of the language-based implementation techniques is better than pre-processing in terms of modifiability. Similarly, language-based implementation techniques specifically proposed to implement features in SPLE (Feature-Oriented Programming and Delta-Oriented Programming) have better modularity but do not support variations in feature binding time.

Chapter four presented a systematic mapping study and systematic literature review on the approaches proposed to support variations of feature binding time. The chapter presented visual maps of published works, from 1999 through 2016, on the proposed interventions. In the chapter, we also presented the following six categories of approaches that emerged from the systematic study:

- Delegation of binding to *aspect weaver*
- Programming language extension
- *Metadata* interpretation
- Model composition
- Delegation of binding to deployment platform
- Abstracting the binding time at model level

We also presented a narrative summary of each of the approaches in all the categories and qualitative assessments of the proposed interventions through the lenses of binding phases, granularity, and level of abstraction.

Chapter five presented the detail design and implementation of our proposed binding-time aware modelling language. We validated the modelling language with an example of an enterprise software product line.

6.2 Revisiting the Contributions

In this section, we highlight the significance of our contributions in the context of the existing body of knowledge.

In the literature, there are two categories of actions researches on transforming variations into features: 1) the exploration of granularity of program elements forming the intersection between separate features [KAK08], and 2) exploration on the structure of code-asset and possible transformations that are required to untangle features, using one or more language-based techniques for advanced separation of concern [MLWR01, KAB07,

CRE08, ARR+16, MP02]. The two categories of studies had resulted in new knowledge that may be used as an initial benchmark for language semantics that should be considered when proposing/improving programming languages for implementing variations as features.

Our first contribution, C1, also emanated from the exploration of properties of features in the code-asset as one of the aspects of our action research. However, our contribution provides a new perspective (i.e., modifiability to inject additional variations) to the previous studies. Hence, the findings from our action research should also be taken into consideration when proposing or improving languages for SPL implementation. For example, the lack of modifiability when untangling features from the code-asset using AOP due to the high number of constructs that have to be used or the lack of support for method and constructor wrapping in DOP should be useful to designers of programming languages meant for implementing variations as features.

The second Contribution, C2, stemmed from the systematic study to survey approaches that are proposed to support variations of feature binding time. The study is a new addition to SPLE literature since to the best of our knowledge, no similar study exists. Further, we presented, as a narrative summary, detail insight for each of the surveyed approaches and highlighted where the proposed intervention is relevant and where it may be inadequate and in some cases suggested how the approach might be improved. The narrative summary complements existing knowledge of variability resolution and binding time [KK16, Kre15, SvGB05].

The third and fourth contributions, C3 and C4, are the binding time aware modelling language and the toolsets to encode the binding time and to check consistency at the architecture model level. We consider our approach as an improvement over the existing approaches in one of the following ways:

- Abstracting the binding time at the model level in our approach is wider in scope than the current approaches because we did not limit the feature binding to only execution context as in the existing approaches[BW09, HKM+13]. Hence, our approach may be more applicable in the cases where practitioners have to deal with the wider scope of binding that what the current approaches support.
- Modelling the binding time at the architecture level, to circumvent scalability concern, as against current approaches where either the binding time is marked on fine-grained model elements [DRO19] or composition of the fine-grained model elements to realize a feature as a model slice.
- Using the binding time property as a condition for model transformation and thereby eliminating the need for multiple implementations attendant to AOP design idioms[ARR+16, CRE08].

Overall, our approach supports variations of feature binding time by design and improves the modifiability of reusable assets of a product line.

6.3 Limitations

In the action research, the specific cases we considered are by no means generalization of every single case of exclusivity and intersection. Inherently, the level of modifiability differs with the level of tangling between program elements of different features. In addition, the study is from a single product line domain. Other product lines may have different structures of code-asset.

Likewise, the evaluated implementation techniques are generally not mutually exclusive to each other. AOP, in particular, is used to support other techniques. However, in the evaluation, we reduced the feature union into its atomic constituents to enable the reasoning of individual techniques against these atomic units.

The switching of connection mode between components (SCMBC) in our approach may not be relevant where a target middleware or platform does not exist. Even if a target platform exists, it may not support the delegation of binding as proposed in this thesis. However, many domains, including real-time domain, are increasingly adopting a container technology or incorporate a special library that can be adapted for the binding [sm118, ER18]. Likewise, an architectural framework [TMO09] can be developed or adapted to support the delegation of binding. Similarly, a light-weight component manager may also be used for the same purpose.

In the implementation of the modelling language, we extended Ecore *metamodelling* language – an approach that is considered as heavyweight [BGD+15]. A lightweight extension such as UML profiling mechanism might have been easier and could have been made reusable to other domains[LWWC11, BGD+15].

The approach we proposed is most appropriate in *proactive* SPLE [Cle02], where the software product line is developed from scratch since a certain degree of freedom is required since the support of variations of feature binding time has to be pre-planned. Overall, the proposed approach mainly demonstrated that it is feasible to manage the variation of feature binding time at a higher level of abstraction in MDD PLA.

6.4 Future research directions

In this section, we outline the future works to either consolidate the findings in this thesis or to extend our approach to supporting variations of feature binding time. The following some of the grounds for future work:

- One of the immediate extensions to our proposed binding time aware modelling language is to add, to the PIM *metamodel*, a means to aggregate the fine-grained variations into aspectual components.
- The M2M transformation should also be extended to support parameterization in many fronts (e.g., parameterize ports with synchronicity property, parameterize component interaction mechanisms) and to check product-wide effects of the instantiated properties (e.g., to check possible violation for coordination requirements of parallel tasks).
- From the findings in our systematic study on the interventions to support variations of feature binding time, AOP related techniques where the dominant interventions proposed. One of the interventions was the integration of static and dynamic weaver to have the best from the two classes of weavers. Interestingly, the integration had to sacrifice the inter-type declaration of fine-grained program elements such as attributes similar to the compromise made when AspectJ and AspectWerkz weavers were merged. It will be interesting, as future research, to use the two weavers in the same assets and switch between the two types of weavers depending on the binding time requested. Again the switching between the two weavers shall be abstracted at the model level. In this regard, it will also be interesting to compare the integration and the switching between the two classes of weavers in terms of efforts and practical utility.
- It will also be interesting to extend our proposed approach to supporting variations of feature binding time to cloud-based and micro-service deployment platforms such as Docker container[And15].
- Since we use only a single domain in our exploration of feature in the code-asset, further investigations are necessary with multiple case studies in different software domains in order to improve the understanding of our findings.
- When the proposed modelling language attained a certain level of maturity, it imperative to provide empirical evidence when the approach is directly compared to the composition of fine-grained model elements in terms of modifiability and similarly to compare with the use of aspect idioms in terms of code duplication.

6.5 Closing remarks

This thesis has established that modern language-based implementation techniques, including those that are specifically proposed to implement variations in the form of features, have fallen short in supporting the modifiability of reusable assets and variations of feature binding time in SPLE. Similarly, the various engineering approaches that are proposed to support variations of feature binding time have also fallen short in terms of the scope of their coverage and focus. Given the established limitations of the existing approaches, this thesis presents binding time aware modelling language that supports variations of feature binding time by design and improves the modifiability of reusable assets of a product line.

Bibliography

[AABZ14] Sven Apel, Joanne M. Atlee, Luciano Baresi, and Pamela Zave. Feature Interactions: The Next Generation (Dagstuhl Seminar 14281). *Dagstuhl Reports*, 4(7):1–24, 2014.

[ABM00] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component-based product line development: the kobra approach. In *Software Product Lines*, pages 289–309. Springer, 2000.

[AE92] William W. Agresti and William M. Evanco. Projecting software defects from analyzing ada designs. *IEEE Transactions on Software Engineering*, 18(11):988–997, 1992.

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[AKS+13] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, pages 1–8. ACM, 2013.

[AM04] Michalis Anastasopoulos and Dirk Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *International Conference on Software Reuse*, pages 141–156. Springer, 2004.

[And15] Charles Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, 2015.

[ARR+16] Rodrigo Andrade, Marcio Ribeiro, Henrique Rebelo, Paulo Borba, Vaidas Gasiunas, and Lucas Satabin. Assessing idioms for a flexible feature binding time. *The Computer Journal*, 59(1):1–32, 2016.

[Bat05a] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005.

[Bat05b] Don Batory. A tutorial on feature oriented programming and the ahead tool suite. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 3–35. Springer, 2005.

[BBM96] Victor R Basili, Lionel C Briand, and Walcélio L Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):104–116, 1996.

[BC12] Jan Bosch and Rafael Capilla. Dynamic variability in software-intensive embedded system families. *Computer*, 45(10):28–35, 2012.

[BD95] Jean-Marie Burkhardt and Françoise Détienne. An empirical study of software reuse by experts in object-oriented design. In *Human—Computer Interaction*, pages 133–138. Springer, 1995.

[BFK+99a] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. Pulse: A methodology to develop software product lines. *SSR*, 99:122–131, 1999.

[BFK+99b] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. Pulse: A methodology to develop software product lines. *SSR*, 99:122–131, 1999.

[BGD+15] Hugo Bruneliere, Jokin Garcia, Philippe Desfray, Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, and Jordi Cabot. On lightweight metamodel extension to support modeling tools agility. In *European Conference on Modelling Foundations and Applications*, pages 62–74. Springer, 2015.

[BHK11] Don Batory, Peter Höfner, and Jongwook Kim. Feature interactions, products, and composition. In *ACM SIGPLAN Notices*, volume 47, pages 13–22. ACM, 2011.

[Big98] Ted J Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5(1):169, 1998.

[BLL+14] Johannes Bürdek, Sascha Lity, Malte Lochau, Markus Berens, Ursula Goltz, and Andy Schürr. Staged configuration of dynamic software product lines with complex binding time constraints. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, page 16. ACM, 2014.

[BMF09] Luciana Akemi Burgareli, Selma Shin Shimizu Melnikoff, and Mauricio G Vieira Ferreira. A variation mechanism based on adaptive object model for software product line of brazilian satellite launcher. In *2009 First IEEE Eastern European Conference on the Engineering of Computer Based Systems*, pages 24–31. IEEE, 2009.

[BOVH17] Gülden Bayrak, Felix Ocker, and Birgit Vogel-Heuser. Evaluation of selected control programming languages for process engineers by means of cognitive effectiveness and dimensions. *Journal of Software Engineering and Applications*, 10(05):457, 2017.

[BST+94] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The genvoca model of software-system generators. *IEEE Software*, 11(5):89–94, 1994.

[BVD07] Michal Bebjak, Valentino Vranic, and Peter Dolog. Evolution of web applications with aspect-oriented design patterns. In *AEWSE*, 2007.

[BW09] Danilo Beuche and Jens Weiland. Managing flexibility: Modeling binding-times in simulink. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 289–300. Springer, 2009.

[BW14] Thomas Buchmann and Bernhard Westfechtel. Mapping feature models onto domain models: ensuring consistency of configured domain models. *Software and Systems Modeling*, 13(4):1495—1527, 2014.

[CA05] Krzysztof Czarnecki and Michał Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *International conference on generative programming and component engineering*, pages 422–437. Springer, 2005.

[CB13] Rafael Capilla and Jan Bosch. Binding time and evolution. In *Systems and Software Variability Management*, pages 57–73. Springer, 2013.

[CB16] Rafael Capilla and Jan Bosch. Dynamic variability management supporting operational modes of a power plant product line. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, pages 49–56. ACM, 2016.

[CCG+16] Bruno BP Cafeo, Elder Cirilo, Alessandro Garcia, Francisco Dantas, and Jaejoon Lee. Feature dependencies as change propagators: an exploratory study of software product lines. *Information and Software Technology*, 69:37–49, 2016.

[CE00] K Czarnecki and U Eisenecker. Generative programming: Methods, tools, and applications., 2000.

[CEM04] Robert Chatley, Susan Eisenbach, and Jeff Magee. Magicbeans: a platform for deploying plugin components. In *International Working Conference on Component Deployment*, pages 97–112. Springer, 2004.

[Cle02] Paul Clements. Being proactive pays off. *IEEE Software*, 19(4):28, 2002.

[CLK08] Hojin Cho, Kwanwoo Lee, and Kyo C Kang. Feature relation and dependency management: An aspect-oriented approach. In *2008 12th International Software Product Line Conference*, pages 3–11. IEEE, 2008.

[Coo17] Oracle Coorporation. Oracle berkeley db java edition — oracle berkeley db, 2017.

[CRE08] Venkat Chakravarthy, John Regehr, and Eric Eide. Edicts: implementing features with flexible binding times. In *Proceedings of the 7th international conference on Aspect-oriented software development*, pages 108–119. ACM, 2008.

[CVD16] Rafael Capilla, Alejandro Valdezate, and Francisco J Díaz. A runtime variability mechanism based on supertypes. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 6–11. IEEE, 2016.

[Del17] DeltaJ. a delta oriented programming language with core and delta modules, 2017.

[DFV03] Eelco Dolstra, Gert Florijn, and Eelco Visser. Timeline variability: The variability of binding time of variation points, 2003.

[DRB+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 25–34. IEEE, 2013.

[DRO19] Udo Kelter Dennis Reuling, Christopher Pietsch and Manuel Ohrndorf. Flexiple - a tool for flexible binding times in annotated modelbased spls. In *In Proceedings of 23rd International Systems and Software Product Line Conference, Paris, France, 9–13 September, 2019 (SPLC'19)*, pages 52–67. ACM, 2019.

[DSW14] Ferruccio Damiani, Ina Schaefer, and Tim Winkelmann. Delta-oriented multi software product lines. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 232–236. ACM, 2014.

[DWAJ+05] Mary Dixon-Woods, Shona Agarwal, David Jones, Bridget Young, and Alex Sutton. Synthesising qualitative and quantitative evidence: a review of possible methods. *Journal of health services research & policy*, 10(1):45–53, 2005.

[ER18] VF Emets and Jan Rogowski. Comparative analysis of event propagation methods on android platform. In *2018 IEEE 13th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT)*, volume 1, pages 227–230. IEEE, 2018.

[ESSD08] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In

Guide to advanced empirical software engineering, pages 285–311. Springer, 2008.

[FCS+08] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiron, Uira Kulesza, Alessandro Garcia1, Sergio Soares3, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *Proceedings of the 30th international conference on Software engineering*, pages 261 –270, Leipzig, Germany, 2008. ACM.

[Fli09] Shayne Flint. A conceptual model of software engineering research approaches. In *2009 Australian Software Engineering Conference*, pages 229–236. IEEE, 2009.

[Fow10] Martin Fowler. Alternative computational models. In *Domain-specific languages*, pages 113–120. Pearson Education, 2010.

[FPF+98] William Frakes, Ruben Prieto, Christopher Fox, et al. Dare: Domain analysis and reuse environment. *Annals of software engineering*, 5(1):125–141, 1998.

[Fra03] David S Frankel. *Model driven architecture applying MDA*. John Wiley & Sons, 2003.

[FT96] William Frakes and Carol Terry. Software reuse: metrics and models. *ACM Computing Surveys (CSUR)*, 28(2):415–435, 1996.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE software*, 12(6):17–26, 1995.

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. Elements of reusable object-oriented software. *Design Patterns. massachusetts: Addison-Wesley Publishing Company*, 1995.

[GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, et al. A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 74–85. IEEE, 2015.

[GKS+96] Hassan Gomaa, Larry Kerschberg, Vijayan Sugumaran, C Bosch, I Tavakoli, and L O'Hara. A knowledge-based software engineering environment for reusable software requirements and architectures. *Automated Software Engineering*, 3(3-4):285–307, 1996.

[Gla94] Robert L Glass. The software-research crisis. *IEEE Software*, 11(6):42–47, 1994.

[Gla01] Robert L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE SOFTWARE*, 18(3):36–44, May/June 2001.

[Gom05] Hassan Gomaa. *Designing software product lines with UML*. IEEE, 2005.

[GPGBMA10] Borja González-Pereira, Vicente P Guerrero-Bote, and Félix Moya-Anegón. A new approach to the metric of journals' scientific prestige: The sjr indicator. *Journal of informetrics*, 4(3):379–391, 2010.

[GPZ02] Michael Goedicke, Klaus Pohl, and Uwe Zdun. Domain-specific runtime variability in product line architectures. In *International Conference on Object-Oriented Information Systems*, pages 384–396. Springer, 2002.

[Gro10] Object Management Group. *About the Object Constraint Language Specification Version 2.2*. `https://www.omg.org/spec/OCL/2.2/About-OCL/`, 2010.

[Gro16] Object Management Group. *About the Meta Object Facility Specification Version 2.5.12*. `https://www.omg.org/mof/`, 2016. Accessed: 2018-04-03.

[HKM+13] Herman Hartmann, Mila Keren, Aart Matsinger, Julia Rubin, Tim Trew, and Tali Yatzkar-Haham. Using mda for integration of heterogeneous components in software supply chains. *Science of Computer Programming*, 78(12):2313–2330, 2013.

[HMT09] Herman Hartmann, Aart Matsinger, and Tim Trew. Supplier independent feature modelling. In *Proceedings of the 13th International Software Product Line Conference*, pages 191–200, San Francisco, California, USA, 2009. ACM.

[HT08] Herman Hartmann and Tim Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *2008 12th International Software Product Line Conference*, pages 12–21. IEEE, 2008.

[HZS+16] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*, 21(2):449–482, 2016.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[K+07] Staffs Keele et al. Guidelines for performing systematic literature reviews in software engineering. Technical report, Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

[KA09] Christian Kastner and Sven Apel. Virtual separation of concerns – a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, September/October 2009.

[KAB07] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *11th International Software Product Line Conference (SPLC 2007)*, pages 223–232. IEEE, 2007.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, pages 311–320. ACM, 2008.

[KAR+09] Christian Kästner, Sven Apel, Marko Rosenmüller, Don Batory, Gunter Saake, et al. On the impact of the optional feature problem: Analysis and case studies. In *Proceedings of the 13th International Software Product Line Conference*, pages 181–190. Carnegie Mellon University, 2009.

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[KK98] Dirk O Keck and Paul J Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, 1998.

[KK15] Niloofar Khedri and Ramtin Khosravi. Incremental variability management in conceptual data models of software product lines. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pages 222–229. IEEE, 2015.

[KK16] Michael Krisper and Christian Kreiner. Describing binding time in software design patterns. In *Proceedings of the 21st European Conference on Pattern Languages of Programs*, page 25. ACM, 2016.

[KKL+98] Kyo C Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-; oriented reuse method with domain-; specific reference architectures. *Annals of Software Engineering*, 5(1):143, 1998.

[Kre15] Christian Kreiner. A binding time guide to creational patterns. In *Proceedings of the 18th European Conference on Pattern Languages of Program*, page 14. ACM, 2015.

[Kru95] Philippe B Kruchten. The 4+ 1 view model of architecture. *IEEE software*, 12(6):42–50, 1995.

[Kru06] Charles W Krueger. New methods in software product line development. In *10th International Software Product Line Conference (SPLC'06)*, pages 95–99. IEEE, 2006.

[Lad03] Ramnivas Laddad. *AspectJ in action: practical aspect-oriented programming*. Dreamtech Press, 2003.

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 105–114. ACM, 2010.

[Lee] Kwanwoo Lee. Aspect-oriented patterns for the realization of flexible feature binding. In *1st International Workshop on Model-Driven Approaches in Software Product Line Engineering Goetz Botterweck, Iris Groher, Andreas Polzer*, page 51. Citeseer.

[LHKS92] John A Lewis, Sallie M Henry, Dennis G Kafura, and Robert S Schulman. On the relationship between the object-oriented paradigm and software reuse: An empirical investigation. Technical report, Department of Computer Science, Virginia Polytechnic Institute & State ..., 1992.

[LK03] Jaejoon Lee and Kyo C Kang. Feature binding analysis for product line component development. In *International Workshop on Software Product-Family Engineering*, pages 250–260. Springer, 2003.

[LK10] Kwanwoo Lee and Kyo C Kang. Usage context as key driver for feature selection. In *International Conference on Software Product Lines*, pages 32–46. Springer, 2010.

[LKCC00] Kwanwoo Lee, Kyo C Kang, Wonsuk Chae, and Byoung Wook Choi. Feature-based approach to object-oriented engineering of applications for reuse. *Software: Practice and Experience*, 30(9):1025–1046, 2000.

[LKKP06] Kwanwoo Lee, Kyo Chul Kang, Minseong Kim, and Sooyong Park. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In *10th International Software Product Line Conference (SPLC'06)*, pages 10–pp. IEEE, 2006.

[LKL02] Kwanwoo Lee, Kyo C Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *International Conference on Software Reuse*, pages 62–77. Springer, 2002.

[LWWC11] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. From uml profiles to emf profiles and beyond. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 52–67. Springer, 2011.

[MLWR01] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: An exploratory study. In *Proceedings of the 23rd International Conference on Software Engineering(ICSE 2001)*, pages 275– –284, Toronto, Ontario, Canada, 2001. IEE.

[MMP00] Nikunj R Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, pages 178–187. ACM, 2000.

[MP02] Dirk Muthig and Thomas Patzke. Generic implementation of product line components. In *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, pages 313–329. Springer, 2002.

[MPH+07] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 243–253. IEEE, 2007.

[MV09] Radoslav Menkyna and Valentino Vranić. Aspect-oriented change realization based on multi-paradigm design with feature modeling. In *IFIP Central and East European Conference on Software Engineering Techniques*, pages 40–53. Springer, 2009.

[Nau68] Peter Naur. Software engineering-report on a conference sponsored by the nato science committee garimisch, germany. *http://homepages. cs. ncl. ac. uk/brian. randell/NATO/nato1968. PDF*, 1968.

[Nei84] James M Neighbors. The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, (5):564–574, 1984.

[PBvDL05] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media, 2005.

[PCH93] Jeffrey S. Poulin, Joseph M. Caruso, and Debera R. Hancock. The business case for software reuse. *IBM Systems Journal*, 32(4):567–594, 1993.

[PFMM08] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *Ease*, volume 8, pages 68–77, 2008.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997.

[Ran98] Alexander Ran. Architectural structures and views. In *Foundations of Software Engineering: Proceedings of the third international workshop on Software architecture*, volume 1, pages 117–120, 1998.

[RK12] Baishakhi Ray and Miryung Kim. A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 53. ACM, 2012.

[RKS00] C. R. Roast, B. Khazaei, and J. I. Siddiqi. Formal comparisons of program modification. In *Proceeding 2000 IEEE International Symposium on Visual Languages*, pages 165–171, Seattle, WA, USA, 2000. IEE.

[RSAS11] Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible feature binding in software product lines. *Automated Software Engineering*, 18(2):163–197, 2011.

[SB99] Mikael Svahnberg and Jan Bosch. Evolution in software product lines: Two cases. *Journal of Software Maintenance: Research and Practice*, 11(6):391–422, 1999.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *International Conference on Software Product Lines*, pages 77–91. Springer, 2010.

[SDS10] Arnon Sturm, Dov Dori, and Onn Shehory. An object-process-based modeling language for multiagent systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews),*, 40:227–241, March 2010.

[SE08] Klaus Schmid and Holger Eichelberger. Model-based implementation of meta-variability constructs: A case study using aspects. *VaMoS*, 8:63–71, 2008.

[sm118] SmartSoft : *SmartSoft-Robotics*. `https://sourceforge.net/projects/smart-robotics/`, 2018.

[Som11] Ian Sommerville. Software engineering 9th edition. *ISBN-10*, 137035152, 2011.

[SPLS+06] Wolfgang Schroder-Preikschat, Daniel Lohmann, Fabian Scheler, Wasif Gilani, and Olaf Spinczyk. Static and dynamic weaving in system software with aspectc++. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, volume 9, pages 214a–214a. IEEE, 2006.

[SSS17] Christoph Seidl, Sven Schuster, and Ina Schaefer. Generative software product line development using variability-aware design patterns. *Computer Languages, Systems & Structures*, 48:89–111, 2017.

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, July 2005.

[Tec] Oracle Technology. *Java™ EE at a Glance*. `https://www.oracle.com/technetwork/java/javaee/overview/index.html`.

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[TLSPS09] Reinhard Tartler, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Dynamic aspectc++: Generic advice at any time. In *SoMeT*, pages 165–186. Citeseer, 2009.

[TMD10] Richard N Taylor, Nenad Medvidovic, and Eric Dashofy. *Software architecture: foundations, theory, and practice*. John Wiley & Sons, 2010.

[TMO09] Richard N Taylor, Nenad Medvidovic, and Peyman Oreizy. Architectural styles for runtime software adaptation. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, pages 171–180. IEEE, 2009.

[TO] Peri Tarr and Harold Ossher. Hyperj user and installation manual. ibm corporation 2000.

[TSCS16] Nguonly Taing, Thomas Springer, Nicolás Cardozo, and Alexander Schill. A dynamic instance binding mechanism supporting run-time variability of role-based software systems. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 137–142. ACM, 2016.

[Uni18] European Union. *The EU General Data Protection Regulation (GDPR).* `https://eugdpr.org/`, 2018.

[VBMD08] Valentino Vranić, Michal Bebjak, Radoslav Menkyna, and Peter Dolog. Developing applications with aspect-oriented change realization. In *IFIP Central and East European Conference on Software Engineering Techniques*, pages 192–206. Springer, 2008.

[VdH04] André Van der Hoek. Design-time product line architectures for any-time variability. *Science of computer programming*, 53(3):285–304, 2004.

[VG91] Michiel Van Genuchten. Why is software late? an empirical study of reasons for delay in software development. *IEEE Transactions on software engineering*, 17(6):582–590, 1991.

[VO98] ROb Van Ommering. Koala, a component model for consumer electronics product software. In *International Workshop on Architectural Reasoning for Embedded Systems*, pages 76–86. Springer, 1998.

[VV11] Markus Voelter and Eelco Visser. Product line engineering using domain-specific languages. In *2011 15th International Software Product Line Conference*, pages 70–79. IEEE, 2011.

[WG18] Bernhard Westfechtel and Sandra Greiner. From single-to multi-variant model transformations: Trace-based propagation of variability annotations. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 46–56. ACM, 2018.

[WJE+09] Jon Whittle, Praveen Jayaraman, Ahmed Elkhodary, Ana Moreira, and João Araújo. Mata: A unified approach for composing uml aspect models based on graph transformation. In *Transactions on Aspect-Oriented Software Development VI*, pages 191–237. Springer, 2009.

[WM98] Guijun Wang and H Alan MacLean. Architectural components and object-oriented implementations. In *A position paper presented at the 1998 International Workshop on Component-Based Software Engineering, ICSE*, 1998.

[WMMR06] Roel Wieringa, Neil Maiden, Nancy Mead, and Colette Rolland. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements engineering*, 11(1):102–107, 2006.

[Xte17] Xtext. Language engineering for everyone, 2017.

[YBJ01] Joseph W Yoder, Federico Balaguer, and Ralph Johnson. Architecture and design of adaptive object-models. *ACM Sigplan Notices*, 36(12):50–60, 2001.

[ZBP+13] Bo Zhang, Martin Becker, Thomas Patzke, Krzysztof Sierszecki, and Juha Erik Savolainen. Variability evolution and erosion in industrial product lines: a case study. In *Proceedings of the 17th International Software Product Line Conference*, pages 168–177. ACM, 2013.

[ZS06] Uwe Zdun and Mark Strembeck. Modeling composition in dynamic programming environments with model transformations. In *International Conference on Software Composition*, pages 178–193. Springer, 2006.

Appendices

Appendix A Platform Indipendent Model with OCL Embedded

```
package PIM : moonlit = 'http://www.moonlit.com/rpim'
{
//product line architecture has many ArchitecturalElements
        class PLA
        {
                property elements : ArchitecturalElement[*|1] { ordered
                    ↪ composes };
                attribute name : String[1];
                property datatype : Type[+|1] { composes };
        }
        //ArchitecturalElement
        abstract class ArchitecturalElement
        {
        //reusable Boolean operation to check whether or not an element
            ↪ is mandatory
                operation isVariable() : Boolean[1]
                {
                        body: if self.elementType <> 'mandatory' then true
        else false endif;
                }
                //attributes of ArchitecturalElement
                attribute name : String[1];
                attribute elementType : String[1];
                attribute bindingTime : String[1];
                attribute isAtVP : Boolean[1];
                //OCL constraint to restric changing binding time of
                    ↪ mandatory element
invariant   btRestriction:
self.elementType = 'mandatory' implies
self.bindingTime <> 'pre_deployment' and self.bindingTime <>
'deployment' and self.bindingTime <> 'post_deployment'
and self.bindingTime <> 'undecided';
        }

        enum BindingTime { serializable }
        {
                literal asset_dev = 1;
                literal pre_deployment;
                literal deployment;
```

```
        literal post_deployment;
        literal undecided;
}
enum ElementType { serializable }
{
        literal mandatory = 1;
        literal optional;
        literal alternative;
        literal OR;
}
//Abstract componnent
abstract class Component extends ArchitecturalElement
{
        operation notAtPre-Deployment() : Boolean[1]
        {
                body: inputPorts.incomingConnector.source->forAll(
                    ↪ bindingTime <> '');
        }
        //ports
        property inputPorts : INPort[*|1] { composes };
        property outPorts : OutPort[*|1] {composes };
}
//ComplexDataType
class ComplexDataType extends Type
{
        property attributes#owner : Attribute[*|1] { composes };
        attribute durability : Durability[1];
        property references : Reference[*|1] { composes };
}
//Service and Control extend Component
class Service extends Component;
class Control extends Component;

//Operation
class Operation
{
        property type : Type[1];
        property data#owner : OperationData[1] { composes };
        property owner#operations : INPort[1];
        attribute name : String[1];
        property sourcinTo#passer : Connector[*|1] { };
}
//Port
```

```
class Port extends ArchitecturalElement
{
        property owner : Component[1];
        //OCL Constraint prohibing connector have binding time
            ↪ earlier than its owner Component
        invariant
        notEalierthanOwner: if self.isVariable() and self.owner.
            ↪ isVariable() then
                self.bindingTime = 'pre_deployment' implies owner.
                    ↪ bindingTime <> 'undecided' and owner.
                    ↪ bindingTime <> 'deployment' and owner.
                    ↪ bindingTime <> 'post_deployment'
                else
                true
                endif;
}
class SimpleDataType extends Type;
class OperationData
{
        property type : Type[1];
        attribute name : String[1];
        property owner#data : Operation[1];
}
abstract class Type
{
        attribute name : String[1];
}
class Attribute
{
        attribute name : String[1];
        property type : SimpleDataType[1];
        property owner#attributes : ComplexDataType[1];
}
class ResourcePort extends OutPort;
class DataItem
{
        attribute name : String[1];
        property type : ComplexDataType[1];
        property owner : Component[1];
}
enum ReferenceType { serializable }
{
        literal one;
```

```
        literal many;
    }
    enum Durability { serializable }
    {
        literal durable;
        literal nondurable;
    }
    //Connector
    class Connector extends ArchitecturalElement
    {
        property target#incomingConnector : INPort[1];
        property datapass : Type[1];
        property sink : Operation[1];
        property source#connector : OutPort[1];
        property passer#sourcinTo : Operation[1];
        //OCL contraint: OutPort must have reference to Connector
        invariant
        btCompatability: self.bindingTime.toString() = self.source.
            ↪ bindingTime.toString() and self.target.bindingTime.
            ↪ toString() = self.bindingTime.toString();
    }
    class Reference
    {
        attribute name : String[1];
        property target : ComplexDataType[1];
        attribute referenceType : ReferenceType[1];
    }
    class INPort extends Port
    {
        property operations#owner : Operation[*|1] { ordered
            ↪ composes };
        property incomingConnector#target : Connector[*|1] {
            ↪ ordered };
    }
    class OutPort extends Port
    {
        property connector#source : Connector[?] { composes };
        //OCL constraint: OutPort must have reference to Connector
        invariant mustHaveCon: not self.connector.oclIsUndefined()
            ↪ and self.connector.bindingTime = self.bindingTime;
    }
}
```

Appendix B JEE Platform Specific Model in XMI with Ecore Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns
   ↪ :xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="JEEPSM"
      ↪ nsURI="http://www.moonlit.com/rjee" nsPrefix="psm">
  <eClassifiers xsi:type="ecore:EClass" name="PLA">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
       ↪ lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf
       ↪ /2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="elements"
       ↪ lowerBound="1"
       upperBound="-1" eType="#//ArchElement" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="primitives"
       ↪ upperBound="-1"
       eType="#//PrimitiveType" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Entity" eSuperTypes="#//
     ↪ DataType">
    <eStructuralFeatures xsi:type="ecore:EReference" name="associations"
       ↪ upperBound="-1"
       eType="#//Association" containment="true" eOpposite="#//
          ↪ Association/sourceEntity"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="attributes"
       ↪ upperBound="-1"
       eType="#//Attribute" containment="true" eOpposite="#//Attribute/
          ↪ owner"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="entityType"
       ↪ lowerBound="1"
       eType="#//EntityType"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Association">
    <eStructuralFeatures xsi:type="ecore:EReference" name="targetEntity"
       ↪ lowerBound="1"
       eType="#//Entity"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="multiplicity"
       ↪ lowerBound="1"
       eType="#//Multiplicity"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
       ↪ lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf
```

```
    ↪ /2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="sourceEntity"
    ↪ lowerBound="1"
    eType="#//Entity" eOpposite="#//Entity/associations"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EEnum" name="Multiplicity">
  <eLiterals name="ONETOONE"/>
  <eLiterals name="ONETOMANY"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Component" abstract="true"
  ↪ eSuperTypes="#//ArchElement">
  <eStructuralFeatures xsi:type="ecore:EReference" name="references"
    ↪ upperBound="-1"
    eType="#//Reference" containment="true" eOpposite="#//Reference/
      ↪ source"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="implements"
    ↪ upperBound="-1"
    eType="#//Interface" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="EJB" eSuperTypes="#//
  ↪ Component">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="beanType"
    ↪ lowerBound="1"
    eType="#//BeanType"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EEnum" name="BeanType">
  <eLiterals name="Stateful"/>
  <eLiterals name="Stateless"/>
  <eLiterals name="Singleton"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="MessageBean" eSuperTypes
  ↪ ="#//Component"/>
<eClassifiers xsi:type="ecore:EClass" name="Operation">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
    ↪ lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf
    ↪ /2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="parameters"
    ↪ eType="#//Parameter"
    containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="type" eType
    ↪ ="#//DataType"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="owner"
    ↪ lowerBound="1" eType="#//Interface"
```

```
      eOpposite="#//Interface/operations"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="invokations"
      ↪ upperBound="-1"
      eType="#//Invocation" containment="true" eOpposite="#//Invocation/
         ↪ source"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="SimpleReference" eSuperTypes
    ↪ ="#//Reference"/>
<eClassifiers xsi:type="ecore:EClass" name="ArchElement">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="
      ↪ ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString
      ↪ "/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="CDIBean" eSuperTypes="#//
    ↪ Component">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="scope" eType
      ↪ ="#//Scope"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EEnum" name="Scope">
  <eLiterals name="Request"/>
  <eLiterals name="Session" literal="Session"/>
  <eLiterals name="Application"/>
  <eLiterals name="Conversation"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="EventReference" eSuperTypes
    ↪ ="#//Reference">
  <eStructuralFeatures xsi:type="ecore:EReference" name="eventData"
      ↪ lowerBound="1"
      eType="#//Entity"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Attribute">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      ↪ lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf
      ↪ /2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="type"
      ↪ lowerBound="1" eType="#//PrimitiveType"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="owner"
      ↪ lowerBound="1" eType="#//Entity"
      eOpposite="#//Entity/attributes"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Reference">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      ↪ lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf
```

```
      ↪ /2002/Ecore#//EString"
      defaultValueLiteral=""/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="source" eType
      ↪ ="#//Component"
      eOpposite="#//Component/references"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="target"
      ↪ lowerBound="1"
      eType="#//Interface"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EEnum" name="EntityType">
  <eLiterals name="Persistent"/>
  <eLiterals name="DataTransfer" value="1"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EEnum" name="InterfaceKind">
  <eLiterals name="PC"/>
  <eLiterals name="EVENT"/>
  <eLiterals name="Message"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Parameter">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      ↪ lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf
      ↪ /2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="type"
      ↪ lowerBound="1" eType="#//DataType"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ReourceReference"
    ↪ eSuperTypes="#//Reference"/>
<eClassifiers xsi:type="ecore:EClass" name="PrimitiveType" eSuperTypes
    ↪ ="#//DataType"/>
<eClassifiers xsi:type="ecore:EClass" name="DataType" abstract="true">
  <eAnnotations source="http://www.obeo.fr/dsl/dnc/archetype">
    <details key="archetype" value="MomentInterval"/>
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="
      ↪ ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString
      ↪ "/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Interface" eSuperTypes="#//
    ↪ ArchElement">
  <eStructuralFeatures xsi:type="ecore:EReference" name="operations"
      ↪ lowerBound="1"
      upperBound="-1" eType="#//Operation" containment="true" eOpposite
          ↪ ="#//Operation/owner"/>
```

```
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="kindof" eType
       ↪ ="#//InterfaceKind"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="PCInvocation" eSuperTypes
     ↪ ="#//Invocation">
    <eStructuralFeatures xsi:type="ecore:EReference" name="targetOperation
       ↪ " lowerBound="1"
       eType="#//Operation"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Invocation">
    <eStructuralFeatures xsi:type="ecore:EReference" name="targetRef"
       ↪ lowerBound="1"
       eType="#//Reference"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="parameter"
       ↪ eType="#//DataType"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="source"
       ↪ lowerBound="1"
       eType="#//Operation" eOpposite="#//Operation/invokations"/>
  </eClassifiers>
</ecore:EPackage>
```

Appendix C PIM to JEEE transformations

```
----sources of the metamodels
--@nsURI PIM = http://www.moonlit.com/rpim
-- @path PIM=/RefinedPIM2JEE/model/PIM.ecore
-- @path JEE=/RefinedPIM2JEE/model/JEEPSM.ecore

module RefinedPIM2JEE;
create OUT : JEE from IN : PIM;

---- Data type mapping
 rule SimpleDataType2Primitive {
       from
              s: PIM!SimpleDataType
       to
              p: JEE!PrimitiveType (
                     name <- s.name
              )
}
----complex data type to persitent entity
rule ComplexDataType2PersistentEntity {
       from
              c: PIM!ComplexDataType (
                     c.durability.toString() = 'durable'
              )
       to
              j: JEE!Entity (
                     name <- c.name,
                     entityType <- 'Persistent',
                     attributes <- c.attributes,
                     associations <- c.references
              )
}
----complex data type to data transfer object
rule ComplexDataType2DataTransferObject {
       from
              c: PIM!ComplexDataType (
                     c.durability.toString() = 'nondurable'
              )
       to
              j: JEE!Entity (
                     name <- c.name,
                     entityType <- 'DataTransfer',
```

```
                attributes <- c.attributes,
                associations <- c.references
        )
}

rule Attribute2Attribute {
        from
                a: PIM!Attribute
        to
                ja: JEE!Attribute (
                        name <- a.name,
                        type <- a.type
                )
}

rule Reference2OneAssociation {
        from
                r: PIM!Reference (
                        r.referenceType.toString() = 'one'
                )
        to
                a: JEE!Association (
                        name <- r.name,
                        targetEntity <- r.target,
                        multiplicity <- 'ONETOONE'
                )
}

rule Reference2ManyAssociation {
        from
                r: PIM!Reference (
                        r.referenceType.toString() = 'many'
                )
        to
                a: JEE!Association (
                        name <- r.name,
                        targetEntity <- r.target,
                        multiplicity <- 'ONETOMANY'
                )
}
----helper operation to determine binding time of port
helper context PIM!INPort def: isPCPort(): Boolean =
        if self.bindingTime = 'asset_dev' or
```

```
        self.bindingTime = 'pre_deployment' then
                true
        else
                false
        endif;

rule Port2Interface {
        from pt: PIM!INPort(pt.isPCPort())
        to it: JEE!Interface (name <- pt.name, kindof <- 'PC',
                operations <- pt.operations)
}
rule Port2EventInterface {
        from pt: PIM!INPort (not pt.isPCPort())
        to it: JEE!Interface (name <- pt.name, kindof <- 'EVENT',
                operations <- pt.operations)
}

rule Operation2PCOperation {
        from po: PIM!Operation (not po.owner.oclIsTypeOf(PIM!OutPort))
        to op: JEE!Operation (name <- po.name, parameters <- po.data,
           ↪ invokations <- po.sourcinTo, type <- po.type)



}


rule OperationData2Parameter {
        from pi: PIM!OperationData (not pi.owner.owner.oclIsKindOf(PIM!
           ↪ OutPort))
        to pj:JEE!Parameter(name <- pi.name, type <- pi.type)
}
helper context PIM!Service def: isStateful(): Boolean =
        if self.outPorts -> select(r | r.oclIsTypeOf(PIM!ResourcePort)).
           ↪ notEmpty() then
                true
        else
                false
        endif;


rule Controller2CDIBean {
        from
```

```
                ct: PIM!Control
        to
                cd: JEE!CDIBean (
                        name <- ct.name,
                         implements <- ct.inputPorts, references <- ct.
                            ↪ outPorts
                )
}

rule Service2Stateless {
        from
                sv: PIM!Service (
                        not sv.isStateful()
                )
        to
                st: JEE!EJB (
                        name <- sv.name,
                        beanType <- 'Stateless', implements <- sv.inputPorts
                            ↪ , references <- sv.outPorts

                )
}


rule Service2Stateful {
        from
                sv: PIM!Service (
                        sv.isStateful()
                )
        to
                st: JEE!EJB (
                        name <- sv.name,
                        beanType <- 'Stateful', implements <- sv.inputPorts,
                            ↪  references <- sv.outPorts)

}
---- OutPort to SimpleReference
rule OutPort2SimpleReference {
        from outpt: PIM!OutPort (not outpt.oclIsTypeOf(PIM!ResourcePort)
            ↪ and
                (outpt.bindingTime ='asset_dev' or outpt.bindingTime ='
                    ↪ pre_deployment'))
        to sref:JEE!SimpleReference(name <- outpt.name, target <- outpt.
```

```
            ↪ connector.target)
}

----OutPort to event reference
rule OutPortEventReference {
       from outpt: PIM!OutPort (not outpt.oclIsTypeOf(PIM!ResourcePort)
            ↪ and
                (not(outpt.bindingTime ='asset_dev' or outpt.bindingTime ='
                    ↪ pre_deployment')))
       to sref:JEE!EventReference(name <- outpt.name, eventData <- outpt.
            ↪ connector.datapass)
}
--
----connection to direct invokation
rule Connection2Invoke{
       from cn: PIM!Connector (cn.bindingTime = 'asset_dev' or cn.
            ↪ bindingTime ='pre_deployment')
       to iv:JEE!PCInvocation(parameter <- cn.datapass, targetOperation
            ↪ <- cn.sink, targetRef <- cn.source)
}
----connection to event invokation
rule Connection2EventInvoke{
       from cn: PIM!Connector (not (cn.bindingTime = 'asset_dev' or cn.
            ↪ bindingTime ='pre_deployment'))
       to iv:JEE!Invocation(parameter <- cn.datapass, targetRef <- cn.
            ↪ source)
}


--
```

Appendix D Code generation with Xtend

```
package genetaor

import org.eclipse.emf.ecore.EPackage
import org.eclipse.emf.ecore.resource.Resource
import JEEPSM.JEEPSMPackage
import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl
import org.eclipse.xtext.generator.JavaIoFileSystemAccess
import JEEPSM.Entity
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl
import org.eclipse.emf.common.util.URI
import org.eclipse.xtext.resource.impl.ResourceServiceProviderRegistryImpl
import org.eclipse.xtext.parser.IEncodingProvider
import JEEPSM.Attribute
import JEEPSM.Association
//
class EntityGenerator {
	JavaIoFileSystemAccess fsa;
//Generator Workflow calls this method and passed file
	//of PSM model in xmi
	def generateEntity(String file) {
		doEMFSetup
		val resourceSet = new ResourceSetImpl
		val resource = resourceSet.getResource(URI.createURI(file),
		    ↪  true)
		//Iterate through JEE Entities in the passed file
		for (e: resource.allContents.toIterable.filter(Entity)){
		fsa = new JavaIoFileSystemAccess(new
		    ↪ ResourceServiceProviderRegistryImpl(),
		        new IEncodingProvider.Runtime()
		)
		//set the absolute path for the generated output
			fsa.setOutputPath("C:\\Users\\umar_\\OneDrive\\
			    ↪ Desktop\\Box␣Sync\\PHD\\dsl\\RefinedPIM2JEE\\
			    ↪ gen\\entity")
			fsa.generateFile(e.name.toFirstUpper+".java", e.
			    ↪ compileAttributes)
		}
	}

	def compileAttributes(Entity e) {
		'''
```

```
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣public␣class␣e.name.toFirstUpper{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣FOR␣a:␣e.attributes
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣a.compileAttribute
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ENDFOR
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣FOR␣ass:␣e.associations
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ass.compileAssociation
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ENDFOR
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣''''

        }
        //generate attributes
def compileAttribute (Attribute a){'''

private␣a.type.name␣a.name;
public␣a.type.name␣geta.name.toFirstUpper(){
␣␣␣␣␣␣␣␣return␣a.name;
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣public␣void␣seta.name.toFirstUpper(a.type.name␣a.name.toFirstLower
    ↪ ){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣}
'''
}

//generate associations
def compileAssociation (Association ass){'''
IF␣ass.multiplicity.literal␣==␣'ONETOONE'
private␣ass.targetEntity.name␣ass.targetEntity.name.toFirstLower;

ENDIF
IF␣ass.multiplicity.literal␣==␣'ONETOMANY'
private␣java.util.Collection<ass.targetEntity.name>␣␣ass.targetEntity.name
    ↪ .toFirstLower;

ENDIF
'''
```

```
}
	def doEMFSetup() {

		EPackage$Registry.INSTANCE.put(JEEPSMPackage.eINSTANCE.
		    ↪ nsURI, JEEPSMPackage.eINSTANCE)
		Resource$Factory.Registry.INSTANCE.extensionToFactoryMap.
		    ↪ put("xmi", new XMIResourceFactoryImpl);
	}
}

package genetaor

import org.eclipse.xtext.generator.JavaIoFileSystemAccess
import org.eclipse.emf.ecore.EPackage
import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl
import JEEPSM.JEEPSMPackage
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl
import org.eclipse.emf.common.util.URI
import org.eclipse.xtext.parser.IEncodingProvider
import JEEPSM.Interface
import org.eclipse.xtext.resource.impl.ResourceServiceProviderRegistryImpl
import JEEPSM.Operation

class BusinessInterfaceGenerator {
	JavaIoFileSystemAccess fsa;
	//Generator Workflow calls this method and passed file
	//of PSM model in xmi
	def generate (String file){
		doEMFSetup()

		val resourceSet = new ResourceSetImpl
		val resource = resourceSet.getResource(URI.createURI(file),
		    ↪  true)
		//Iterate through JEE PSM interfaces in the passed file
		for (intf : resource.allContents.toIterable.filter(
		    ↪ Interface)) {
			fsa = new JavaIoFileSystemAccess(
				new ResourceServiceProviderRegistryImpl(),
				new IEncodingProvider.Runtime()
			)

//set the absolute path for the generated output
```

```
fsa.setOutputPath("C:\\Users\\umar_\\OneDrive\\Desktop\\Box␣Sync\\PHD\\dsl
    ↪ \\RefinedPIM2JEE\\gen\\interface")

fsa.generateFile("I"+intf.name + ".java", intf.compileBusInterface)

        }


        }
        //White space preserving template
        def compileBusInterface(Interface intf){'''
␣␣␣␣␣␣␣␣public␣interface␣Iintf.name{
␣␣␣␣␣␣␣␣FOR␣ops␣:␣intf.operations
␣␣␣␣␣␣␣␣public␣IF␣ops.type␣!=␣nullops.type.nameELSE␣voidENDIF␣ops.name(IF␣
    ↪ ops.parameters␣!=␣nullops.parameters.type.name␣ops.parameters.
    ↪ nameENDIF);
␣␣␣␣␣␣␣␣ops.compilOperation
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣ENDFOR
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣'''

        }
        //this does nothing
        def void compilOperation(Operation operation){

        }


        def doEMFSetup() {

                EPackage$Registry.INSTANCE.put(JEEPSMPackage.eINSTANCE.
                    ↪ nsURI, JEEPSMPackage.eINSTANCE)
                Resource$Factory.Registry.INSTANCE.extensionToFactoryMap.
                    ↪ put("xmi", new XMIResourceFactoryImpl);
        }
}

package genetaor

import org.eclipse.xtext.generator.JavaIoFileSystemAccess
import org.eclipse.emf.ecore.EPackage
import JEEPSM.JEEPSMPackage
```

```
import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl
import org.eclipse.emf.common.util.URI
import JEEPSM.Component
import org.eclipse.xtext.resource.impl.ResourceServiceProviderRegistryImpl
import org.eclipse.xtext.parser.IEncodingProvider
import JEEPSM.Interface
import JEEPSM.SimpleReference
import JEEPSM.EventReference
import JEEPSM.Operation
import JEEPSM.Invocation
import JEEPSM.PCInvocation
import JEEPSM.InterfaceKind
//component generator
class BusinessComponentGenerator {
	JavaIoFileSystemAccess fsa;
//Generator Workflow calls this method and passed file
	//of PSM model in xmi
	def generateBusinessComponent(String file) {
		doEMFSetup
		val resourceSet = new ResourceSetImpl
		val resource = resourceSet.getResource(URI.createURI(file),
		    ↪  true)
		//Iterate through componenents in the passed file
		for (comp : resource.allContents.toIterable.filter(
		    ↪ Component)) {
			fsa = new JavaIoFileSystemAccess(
				new ResourceServiceProviderRegistryImpl(),
				new IEncodingProvider.Runtime()
			)
//set the absolute path for the generated output
			fsa.setOutputPath("C:\\Users\\umar_\\OneDrive\\
			    ↪ Desktop\\Box␣Sync\\PHD\\dsl\\RefinedPIM2JEE\\
			    ↪ gen\\session")

			fsa.generateFile(comp.name.toFirstUpper + ".java",
			    ↪ comp.compileComponent)
		}
	}

	def compileComponent(Component comp) {
		'''
```

```
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣public␣class␣comp.name.toFirstUpper␣IF␣comp.
   ↪ implements.size␣!=0␣implementsFOR␣imp:␣comp.implements␣Iimp.name␣
   ↪ ENDFOR␣ENDIF{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣FOR␣r␣:␣comp.references
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣r.compilereference
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ENDFOR
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣IF␣comp.implements␣!=␣null
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣FOR␣imp:␣comp.implements␣␣imp.
   ↪ compiteInterfaceENDFOR
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ENDIF
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'''
        }

        def compiteInterface(Interface intf) {
               '''
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣FOR␣ops␣:␣intf.operations
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ops.compilOperation
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ENDFOR
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'''

        }
//generate operations
def compilOperation(Operation o) {
               '''␣␣␣␣␣␣␣␣␣␣␣␣␣
public␣o.type?.name.toFirstUpper␣?:␣'void'␣␣o.name(o.owner.kindof.
   ↪ interfaceType␣IF␣o.parameters␣!=␣null␣o.parameters.type.name.
   ↪ toFirstUpper␣o.parameters.nameENDIF){
␣FOR␣inv␣:␣o.invokations
␣␣␣inv.compileInvokation
␣ENDFOR
␣␣␣␣␣␣␣␣IF␣o.type␣␣!=␣null
␣␣␣␣␣␣␣␣return␣null;
␣␣␣␣␣␣␣␣ENDIF
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'''
        }

def String interfaceType(InterfaceKind itf) {
```

```
 if (itf.toString.equals('EVENT')) {
       '@observes'
} else {
                      ''
               }
       }
       def dispatch compileInvokation(Invocation invk) {
               '''
invk.targetRef.name.toFirstLower.fire(invk.parameter.name);
'''
       }

       def dispatch compileInvokation(PCInvocation pInvk) {
               '''

pInvk.targetRef.name.toFirstLower.pInvk.targetOperation.name(pInvk.
   ↪ parameter.name);
'''

       }

       def dispatch compilereference(SimpleReference rs) {
               '''
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Irs.target.name␣rs.name.toFirstLower;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'''

       }

       def dispatch compilereference(EventReference rs) {
               '''
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Event␣<rs.eventData.name>␣rs.name.toFirstLower;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'''

       }

       def doEMFSetup() {

               EPackage$Registry.INSTANCE.put(JEEPSMPackage.eINSTANCE.
                  ↪ nsURI, JEEPSMPackage.eINSTANCE)
               Resource$Factory.Registry.INSTANCE.extensionToFactoryMap.
                  ↪ put("xmi", new XMIResourceFactoryImpl);
```

```
        }
}

package genetaor
//Generator WorkFlow
class GeneratorWorkFlow {
        def static void main (String [] args){
         new EntityGenerator().generateEntity("Jee.xmi")
                new BusinessInterfaceGenerator().generate("Jee.xmi")
                 new BusinessComponentGenerator().generateBusinessComponent
                    ↪ ("Jee.xmi")


        }
}
```