UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Vander Ramos Alves

Implementing Software Product Line Adoption Strategies

Ph.D. thesis

SUPERVISORS:
Prof. Paulo Henrique Monteiro Borba
Prof. Geber Lisboa Ramalho

Recife, March 2007.

Acknowledgments

I am most grateful to my supervisors, Paulo Borba and Geber Ramalho, for their relentlessly in-depth and most fruitful supervision. Current and previous members of the Software Productivity Group contributed substantially for achieving the results of this work. In particular, Pedro Matos Jr. provided fundamental support during the method definition and in the case studies. Alberto Costa Neto and Ivan Cardim also helped in some case studies and helped substantially with discussions and reviews. Rohit Gheyi and Tiago Massoni collaborated very closely for the definition of feature model refactorings. I would also like to thank the members of the FLIP project, in particular Vilmar Nepomuceno, Fernando Calheiros, Davi Pires, Jorge Leal, Gustavo Santos, and Sérgio Soares for their most valuable participation in the evaluation and tuning of our method. I would also like to thank Meantime Mobile Creations for research collaboration and providing access to the mobile games. Tarcísio Câmara, Alexandre Damasceno, and Pedro Macedo shared a substantial amount of domain knowledge for this work. I am most grateful to them.

Uirá Kulesza has consistently provided most valuable feedback on parts of this work. Also, discussions with him in related topics was most inspiring and an enjoyable experience. Valuable feedback on specific and general parts of this work has also been provided by Rosana Braga, André Santos, and Jacques Robin.

This work was partially supported by CNPq, FINEP, FACEPE.

Abstract

Software Product Line (SPL) is a promising approach for developing a set of products scoped within a market segment and based on common artifacts. Potential benefits are large scale reuse and significant boost in productivity. An incurred key challenge, however, is handling adoption strategies, whereby an organization decides to start the SPL from scratch, bootstrap existing products into a SPL, or evolve an existing SPL. In particular, at the implementation and feature model level, development methods lack adequate support for addressing this issue. In this context, we present an original method addressing the creation and evolution of SPL focusing at the implementation and feature model level. The method first bootstraps the SPL and then evolves it with a reactive approach. The method is systematic because it relies on a collection of provided refactorings at both the code level and at the feature model level. The method was validated in the highly variant domain of mobile games.

Resumo

Linha de Produtos de Software (LPS) é uma aborgadem promissora para o desenvolvimento de um conjunto de produtos focados em um segmento de mercado e desenvolvidos a partir de um conjunto comum de artefatos. Possíveis benefícios incluem reuso em larga escala e significante melhoria em productividade. Um problema chave associado, no entanto, é o tratamento de estratégias de implantação, em que uma organização decide iniciar a partir do marco zero, fazer bootstrap de produtos existentes em uma LPS, ou evoluir uma LPS. Em particular, a nível de implementação e de modelo de features, métodos de desenvolvimento carecem de tratar tal problema. Neste contexto, apresentamos um método original para tratar da implantação e evolução de LPS focando a nível de implementação e modelo de features. O método primeiro faz o bootstrap da LPS e então a evolui com uma abordagem reativa. O método é sistemático porque se baseia numa colecão de refactorings tanto a a nível de implementaçã como de modelo de features. O método foi validado no domínio altamente variável de jógos móveis.

Acronyms

Acronym	Meaning
SPL	Software Product Line
FM	Feature Model
AOP	Aspect-Oriented Programming
AJDP	AspectJ Development Tools
IDE	Integrated Development Environment
CC	Conditional Compilation
API	Application Programming Interface
J2ME	Java 2 Micro Edition
MIDP	Mobile Information Device Profile

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computational systems are becoming ubiquitous [110]. By using a mobile phone with computational power, we can access and manipulate information almost everywhere and anywhere. Similarly, other electronic devices will gain or augment computational power. Indeed, the impact of information technologies in the society will increase significantly. Therefore, in this scenario, applications have to comply with high ever-increasing quality standards, specially availability and usability.

In order to meet these high quality standards, current applications must comply with a series of functional and non-functional requirements such as persistence, concurrency, distribution, and adaptability. This further increases the already complex task of developing these systems. Additionally, the development processes must be productive, and the resulting software must be extensible and reusable [29].

In order to meet the challenge of developing current applications, paradigms such as object orientation and software processes are used. The Object-Oriented Paradigm is implemented by languages and relies on design and architectural patterns [33, 55]. As a result, it offers more effective means to achieve reuse, thereby increasing productivity of future projects, and software maintenance. Object orientation, however, has some shortcomings, such as difficulty in modularizing systemic requirements like non-functional requirements and complex object protocols [100, 101]. In order to overcome these shortcomings, novel extensions of object orientation have been proposed, among which Aspect-Oriented Programming (AOP) [77, 79] is receiving increasing attention [117, 129].

Software development processes also guide application development. These processes define activities to be carried out, the resulting artifacts, and the roles to perform them. Processes thus help to reduce development complexity, promoting its predictability and reproducibility. One shortcoming of existing processes is lack of implementation support [71]. As a result, reuse and extensibility, achieved during design, may be lost during implementation, resulting in quality decrease of the final software. Some extensions of processes focusing on implementation are already being defined [3, 91, 116].

More recently, software processes are being generalized into process frameworks referred to as Software Product Lines (SPL) [39], which focus on the development of a family of products targeting a specific market and based on a common base of artifacts. In a product line, there is a generic architecture which is common to all products in the line; this architecture is adapted for the creation of a particular product. In this

process, variability management [82] plays a key role: the specific products differ in terms of these variations, and thus modelling and implementing them appropriately will translate into higher product line productivity.

Additionally, variability in product lines is more pervasive through development phases than in single-system development. In particular, the design and implementation assets are considerably more variant in the product line setting [65]. Although modelling and design are being intensively discussed in conferences like the Software Product Line Conference, investigation of more suitable implementation techniques cannot be neglected [17].

Additionally, a key orthogonal issue to a particular SPL development process is handling adoption strategies, whereby an organization decides to start the SPL from scratch, extract (bootstrap) existing products into a SPL, or react (evolve) an existing SPL. Particularly, there is lack of support for this at the implementation and at the feature model level, which describes its configurability. For instance, it is often not clear which transformations should be applied to extract which variability. Further, there is lack of correctness discussion regarding this transformations. This is particularly important, since, in the SPL context, testing is considerably expensive [106].

Moreover, during such adoption strategies, it is necessary to evaluate the configurability space of the SPL. For instance, if three isolated applications are to be extracted into a SPL, then, at the end of such process, the resulting feature model describing the SPL should should also have three instances. In the case in which the product line reacts (evolves), the feature model should have more than 3 instances.

To the best of our knowledge, the issues described above have not been addressed. In this context, we propose a systematic method for creating and evolving product lines. Our method first extracts the SPL and then evolves it with a reactive approach. Initially, there may be one or more independent products, which are refactored in order to expose variations to bootstrap the SPL. Next, the SPL scope is extended to encompass another product: the SPL reacts to accommodate the new variant. During this step, refactorings are performed to maintain the existing product, and a SPL extension is used to add a new variant. The SPL may react to further extension or refactoring. Alternatively, there may be an existing SPL implemented with a variability mechanism from which we may want to migrate. During such activities, the feature model as well as the configuration knowledge evolve and need to be handled appropriately.

The method is systematic because it relies on a collection of provided refactorings at both the code level and at the feature model level. Such refactorings are described in terms of templates, which are a concise and declarative way to specify program transformations. In addition, refactoring preconditions (a frequently subtle issue) are more clearly organized and not tangled with the transformation itself. Furthermore, the refactorings can be systematically derived from more elementary and simpler programming laws or feature model transformation laws. These laws are appropriate because they are considerably simpler than most refactorings, involving only localized changes, with each one focusing on a specific language construct. Therefore, they are easier to reason about than the refactorings, increasing correctness confidence in such extractive transformations. This is specially relevant because it reduces the burden on testing, which is potentially expensive in the SPL scenario.

Our program refactorings rely on AOP to modularize crosscutting concerns, which

often occur in SPLs. In cases where variability cannot be handled by AOP, we point to extensions in this paradigm or alternative implementation techniques. Accordingly, we conduct a comparative analysis of SPL variability implementation techniques, which is useful in this respect.

We evaluate our method in existing industrial-strength mobile games, assessing its advantages and drawbacks. The benefits of the evaluation are twofold. First, it contributes to building knowledge of use of non-trivial examples of variability mechanisms such as AOP. Second, it sheds light on how non-trivial industrial issues in the context of SPL can profit from novel uses of emerging techniques.

1.1 Summary of Goals

Our research has the following goals:

- provide systematic method for the extractive and reactive SPL adoption strategies;
- extend the notion of refactoring for SPLs;
- evaluate the proposed method in industrial-strength SPLs.

1.2 Organization

The remainder of the thesis is organized as follows:

- Chapter 2 reviews some essential concepts concerning software product lines and discusses the notion of variability for product lines, contrasting it with single-system development; it then describes a domain example and briefly reviews some SPL methods as well as adoption strategies. Finally, it refines the scope of our proposed method;
- Chapter 3 first reviews techniques for handling variability in software product lines. It then presents a framework for describing these approaches and finally contrast them according to such framework;
- Chapter 4 defines our method for implementing the extractive and reactive SPL adoption strategies. It additionally shows how some elements of such method can be understood formally;
- Chapter 5 first motivates the need for an extended notion of refactoring, where feature models are also considered. It then extends the notion of refactoring to the SPL context and formalizes feature models. Finally, it illustrates an strategy for employing these concepts in the context of a case study in the mobile games domain;
- Chapter 6 describes two case studies, evaluating the proposed method in the mobile games domain. It then presents and addresses some open issues. Finally, it compares the results of both case studies;
- Chapter 7 offers concluding remarks, pointing to future research and related work.

Chapter 2

Software Variability

Variability is the ability to change or customize a system [65]. Improving variability in a system implies making it easier to do certain kinds of changes. It is possible to anticipate some types of variability and construct a system in such a way that it facilitates this type of variability. Reusability and flexibility have been the driving forces behind the development of such techniques as object orientation, object-oriented frameworks, and software product lines.

In this chapter, we first review some essential concepts concerning software product lines (Section 2.1); next, we discuss the notion of variability for product lines, contrasting it with single-system development (Section 2.2); we then describe a domain example where variability issues are discussed (Section 2.3); finally, we briefly review some SPL methods as well as adoption strategies (Section 2.4).

2.1 Historical Notes and Terminology

The concept of software families has early roots in Dijkstra [48] and Parnas [104, 105]. Dijkstra proposed a model of family-based development where differences in design decisions distinguished family members. The original concepts of information hiding separated the common and variable aspects of a module into an implementation and interface, respectively [104]. Parnas later characterized families as groups of items that are strongly related by their commonalities, where commonalities are more important than the variations between family members [105].

Likewise, according to Withey [125], a product family is a group of products that can be built from a common set of artifacts. A product family is defined on the basis of similarities between the structure of its member products. Product family members share at least a common generic architecture. That is, product families are scoped primarily based on technical commonalities between the products.

On the other hand, the same author [125] defines a product line as a group of products sharing a common managed set of features that satisfy the specific needs of a selected market. Thus, this definition of a product line is based on a marketing strategy rather than on technical similarities between its member products. The features defined for a product line might require totally different solutions for different member products. A product line might be well served with one product family; however, it might also

require more than one product family. On the other hand, a product family could be reused in more than one product line.

Clements and Northrop [39] extended the previous definition in order to incorporate technical similarities between member products. Accordingly, a software product line is a set of software-intensive systems sharing a common, managed set of features satisfying the specific needs of a particular market segment or mission *and* that are developed from a common set of core assets (a *core asset* is an artifact used in the production of more than one product in a SPL [39]) in a prescribed way. In the scope of our research and, in particular, during the rest of this document, we assume this definition.

At its essence, a product line involves core asset development (also known as Domain Engineering [45]) and product development (also known as Application Engineering [45]) using the core assets, both under the supervision of technical and organizational management. Core asset development and product development from the core assets can occur in either order: new products are built from core assets, or core assets are extracted from existing products. Often, products and core assets are built in synergy with each other. Figure 2.1 illustrates this triad of essential activities [39].

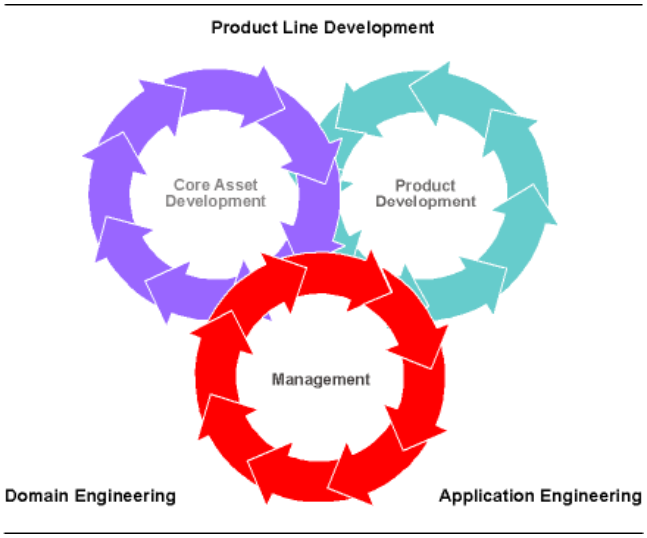


Figure 2.1: The Three Essential Activities for Software Product Lines [39].

Each rotating circle represents one of the essential activities. All three are linked together and in perpetual motion, showing that all three are essential, are inextricably linked, can occur in any order, and are highly iterative.

The rotating arrows indicate not only that core assets are used to develop products, but also that revisions of existing core assets or even new core assets might, and most often do, evolve out of product development. The diagram in figure 2.1 is neutral in regard to which part of the effort is launched first. In some contexts, already existing products are mined for generic assets (perhaps a requirements specification, an architecture, or software components) which are then migrated into the product line's core asset base. In other cases, the core assets may be developed or procured for later use in the production of products.

There is a strong feedback loop between the core assets and the products. Core assets are refreshed as new products are developed. Use of core assets is tracked, and the results are fed back to the core asset development activity. In addition, the value of the core assets is realized through the products that are developed from them. As a result, the core assets are made more generic by considering potential new products on the horizon. There is a constant need for strong, visionary management to invest resources in the development and sustainment of the core assets. Management must also support the cultural change to view new products in the context of the available core assets. Either new products must align with the existing core assets, or the core assets must be updated to reflect the new products that are being marketed. Iteration is inherent in product line activities, that is, in turning out core assets, in turning out products, and in the coordination of the two.

2.2 Variability in Software Product Lines

Over time, variability in software assets has become increasingly important in software engineering. Whereas software systems originally were relatively static and it was accepted that any required change would demand potentially extensive editing of the existing source code, this is no longer acceptable for contemporary software systems. Instead, newer approaches to software design share as a common denominator that the point at which design decisions concerning the supported functionality and quality requirements are made is delayed to later stages.

A typical example of such delayed design decisions is provided by software product lines. Rather than deciding on what product to build beforehand, in software product lines, a software architecture and set of components is defined and implemented that can be configured to match the requirements of a family of software products. A second example is the emergence of software systems that can dynamically adapt their behavior at run time, either by selecting alternatives embedded in the software system or by accepting new code modules during operation, such as plug-and-play functionality, for instance.

The consequence of the developments described above is that, whereas earlier decisions concerning the actual functionality provided by the software system were made during requirement specification and had no effect on the software system itself, new software systems are required to employ various variability mechanisms that allow the software architects and engineers to delay the decisions concerning the variants to the point in the development cycle that optimizes overall business goals. For example, in some cases, this leads to the situation where the decision concerning some variation points is delayed until run time, resulting in customer or user performed configuration of the software system. In other cases, variability can be handled before compilation, thus removing complexity of the final product.

Figure 2.2 illustrates how the variability of a software system is constrained during development [65]. The space between the arrows of the funnel denotes the amount of variability in the system. When the development starts, there are no constraints on the system. This is visualized in Figure 2.2 by having infinite space between the arrows. During development, the number of potential system decreases (so there is increasingly

less variability) until finally at run-time there is exactly one system, that is, the running and configured system. At each step in the development, design decisions are made. When software product lines are considered, it is beneficial to delay some decisions so that products implemented using the shared product line assets can be varied. These delayed design decisions are referred to as variation points [65].

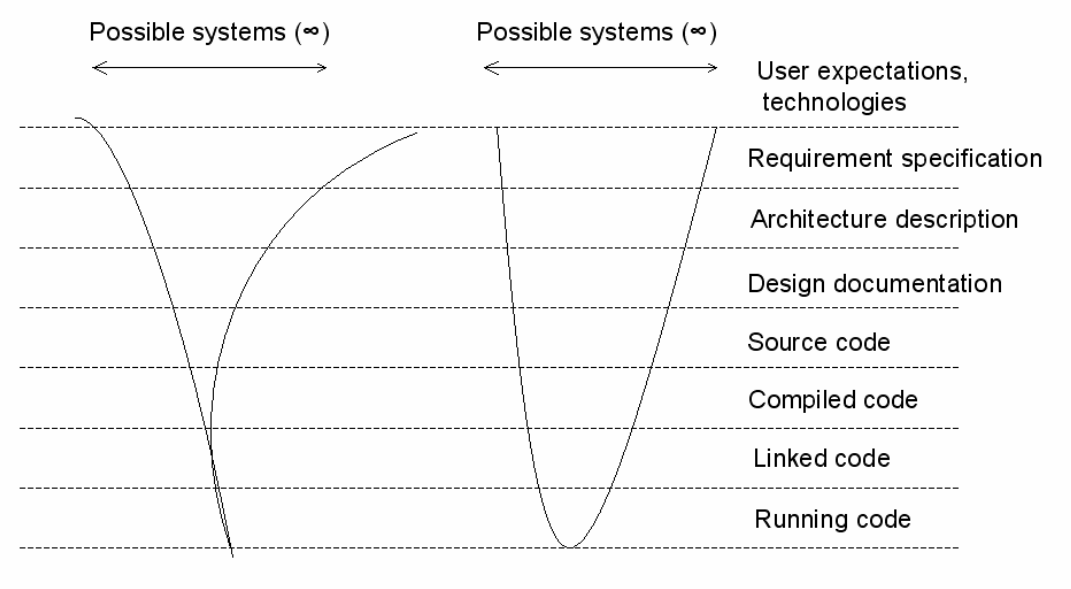


Figure 2.2: Variability funnel with early and delayed variability [65].

Figure 2.2 displays two stereotypical variability funnels. The left one represents a situation where a lot of variability is removed from the system early on; the right one represents a situation where significant effort has been made to preserve variability until very late in the development process. Arguably, the left funnel is easier to develop and is more typical of conventional software development; the right funnel, however, provides greater reusability and is more typical of product line development. Indeed, reusable software contains inherently more variability than concrete applications and such variability spans all development phases as shown on the right funnel of Figure 2.2.

Accordingly, variation management is the key discriminator between conventional software engineering and software product line engineering. In conventional software engineering, variation management deals with software variation over time and is commonly known as configuration management. In software product line engineering, variation management is multi-dimensional. It deals with variation in both time and space [82]. In this context, managing variation in time refers to configuration management of the product line software as it varies over time, while managing variation in space refers to managing differences among the individual products in the domain space of a product line at any fixed point in time. Managing variation in space itself is challenging. Although the variants share a core architecture and other reusable assets, the more diverse the domain, the harder it is to handle variants consistently, which in some cases may outweigh the cost of developing the product line core itself. For instance, as we have analyzed [2], the mobile device game domain, due to portability issues, is

highly variant, where the nature of variation can have different levels of granularity, and the implementation of such variation usually crosscuts a number of artifacts. The scope of our work is handling implementation of product line variability in space as well as handling incurred feature model transformations

2.3 Domain Example: Mobile Games Product Lines

Game development is usually regarded simpler for mobile devices than for desktop platforms [5]. Indeed, the resources provided by the latter support more complex applications, and the development cycle tends to be longer. On the other hand, mobile device games (and mobile applications, in general) must adhere to a stronger portability requirements [5, 34]. In fact, service carriers [1] typically demand from developers that a single application be deployed in a dozen or more platforms. In a more demanding case, a single game had to be ported to 69 different devices [52]. In fact, addressing the porting issue effectively in this context is directly related to the ability to build a product line with highly effective variability management.

Porting stems from a combination of technical and business constraints. Manufactures release different devices targeting diverse customer profiles, in ever-shortening time periods. Besides, operators and publishers need the developed games to be delivered to the greatest possible number of users, forcing the developer to provide multiple versions of the application, each optimized to a specific device. The demand of porting mobile device games is so critical in the industry that there are currently specialized companies in providing such service [124].

Despite being a known critical problem in industry, most current practices only address the portability issue superficially. In fact, the presented solutions are more descriptive than prescriptive; additionally, they present many hypothesis that restrict their applicability, and very few have been validated in industry such as work described elsewhere [96, 51, 49, 54, 36].

A significant amount of different mobile devices is produced and sold because there are segments of the market with distinct needs and financial resources. Therefore, game developers need to adapt the games so that they comply with the specific requirements of each target device.

J2ME [94] is the edition of the Java platform targeted at mobile devices such as mobile phones and personal digital assistants and is currently the most used platform for developing mobile device games [5]. In J2ME and in other platforms, porting demands efforts from the development team due to several variability issues [5]. In J2ME, in particular, the main variability challenges, according to our experience [2, 111], are as follows:

- Different features of the devices regarding user interface, such as screen size, number of colors, pixel size, sounds, and keyboard layout;
- Different execution memory availability and maximum application size;
- Proprietary Application Programming Interface (API) and optional packages;

[1] A service carrier is a telephone company providing local, long distance, or value-added service.

- Different profiles (MIDP 1.0 and MIDP 2.0)[2];
- Different implementations of a same profile in J2ME [93];
- Device-specific bugs;
- Internationalization.

J2ME technology is evolving with the release of version 2.0 of its specification [108] and the optional libraries specification, which can be present in the devices [93]. Moreover, most device manufacturers supply proprietary APIs which extend standard J2ME functionalities. In principle, these innovations could be ignored in favor of porting, so that all games would be implemented using the same API. However, industrial-strength games frequently rely on such APIs, optional packages, and more advanced profiles like MIDP 2.0. Likewise, some carriers require the inclusion of their proprietary APIs in the telephones they commercialize and demand that developers use these libraries, further compromising portability. This myriad of resources, of which the developer should take advantage to build professional games, makes porting very expensive and complex.

Despite manufacturers efforts to make their devices totally compatible with the J2ME standard specification, some devices have known bugs, requiring a number of device-specific work arounds from the programmer when he or she has to use the defective libraries. Once again, porting is compromised. Lastly, there is the language issue: developers and publishers which operate globally inexorably need to translate their games to a great variety of languages. In some cases, several languages can be included in a single SPL instance; however, most of the time, it is more convenient and efficient, in terms of final size of the application, to have several SPL instances, one for each language.

As a result of these factors, developers are frequently forced to develop up to dozens of variations of a single game, optimized for different types of devices, operators, and languages. This further complicates the game development process, thus very likely having a negative impact on the quality of the resulting software, because these variations usually involve modifications scattered across various artifacts. Accordingly, providing consistent maintenance of these variations becomes a more expensive and error-prone task, as the functional common core is normally dispersed across such variations.

In order to illustrate the impact on the resulting code, we considered the porting of a game developed (Rain of Fire[3]) from Motorola's platform T720 to Nokia's Series 60, both J2ME-compliant, but the latter relying on proprietary API offering advanced graphics manipulation [111]. Despite the apparent functional game simplicity, the differences between the devices prompted changes in almost all application classes, adding up to 79 modifications. The average size of each modification was 2 lines, which revealed the fine granularity of these changes. Table 2.1 illustrates the types of variations handled in the porting.

[2]A profile is a set of APIs focusing on one domain of application. The Mobile Information Device Profile (MIDP) is the most used profile, but there are differences across its versions, such as MIDP 1.0 and MIDP 2.0.

[3]In cooperation with Meantime Mobile Creations, under FACEPE/PAPPE and CNPq/Universal research projects.

Type of Variation	Frequency(%)
Argument in method call	30
Class constant value and definition	20
Local variable value and definition	11
Attribute definition	9
Graphics API	8
Additional softkey expression	5
Different expression	4
Drawing calculation expression	7
Class hierarchy	3
Method definition	3

Table 2.1: Effects of porting in the source code, listing the types of variation and corresponding frequency.

We can visualize some of these variations in Figure 2.3. This figure shows screen shots of the difference of a few game classes in two different platforms. The shaded patterns denote code specific to a platform; code in white background is common to both platforms. The goal here is not to understand the individual lines of code, but rather to notice variability patterns. For example, the top leftmost screen shot shows that the game in one platform incorporates additional behavior. The screen shot just below it indicates isomorphic variations in the game in both platforms. The patterns in remaining screen shots lie somewhere between these two. In general, we notice that platform variations are highly crosscutting, i.e., they affect a large number of classes [2, 5, 34]. Furthermore, a significant body of research supports that the implementation of variant features is inherently crosscutting [85, 23, 92, 89, 17].

2.4 Software Product Line Approaches

In order to handle variability and effectively build product lines, a number of approaches have emerged. We describe some SPL development methods (Sections 2.4.1, 2.4.2, and 2.4.3). The description is brief and not intended to be exhaustive for the following reasons. First, we mention some fundamental approaches, from which we borrow essential concepts (for example feature models from FODA); second, as Section 2.4.4 explains, an orthogonal issue to such methods is adoption strategy, which is the focus of our research, as Section 2.4.5 describes.

2.4.1 Feature-Oriented Domain Analysis

In Section 2.2, we discussed that a key activity of a SPL development is core asset development. Within that activity, domain analysis is an important activity, which defines a set of reusable requirements for the applications in the SPL domain. In this context, FODA (**F**eature-**O**riented **D**omain **A**nalysis) is domain analysis method focused on the description of variabilities and commonalities by means of features, where a feature is

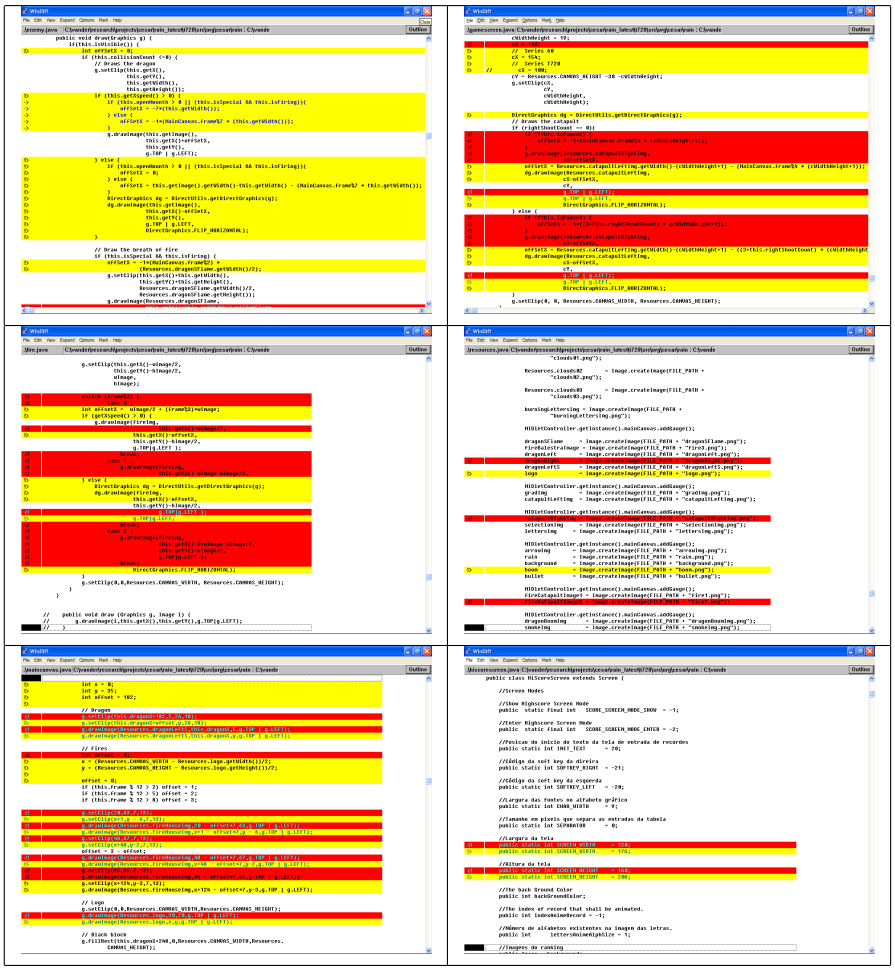


Figure 2.3: Visualizing the effects of porting in the source code.

a prominent and user-visible aspect, quality, or characteristic of a software system or systems [75] (shortly ahead will we present more complete definitions, which we will use for the remainder of this work). The FODA process consists of two phases: 1) Context Analysis, whose purpose is to scope the domain to be analyzed, by considering project constraints and availability of domain expertise; 2) Domain Modeling, which identifies and models the main commonalities and variabilities between the applications in the domain, representing them in a feature model.

A FODA feature model comprises the following elements:

- Feature diagram. The diagram depicts a hierarchical decomposition of features with mandatory (must have), alternative (selection from many) and optional (may or may not have) relationships;

- Feature definitions. Description of all features and its binding time (preprocessing, compilation, deployment, or runtime);

- Composition rules (also referred to as constraints). These rules indicate which feature combinations are valid and which are not;

- Rationale for features. The rationale for choosing or not choosing a particular feature, indicating the trade-offs.

A feature diagram models the configurability aspect of the product line, thereby leaving other aspects such as structural and behavioral relationships to other models. The very advantage of feature diagrams is that they avoid cluttering the configurability aspect with other aspects.

Figure 2.4 depicts a feature diagram for a product line in the domain of dictionary applications for embedded devices [7, 47]. `Dictionary` is the root feature of the product line. Features `Translation`, `Screens`, and `Search mechanism` are mandatory (filled circle); feature `Dynamic customization` is optional (open circle); features `Dynamic screens`, `Colorized screens`, and `Internationalized screens` are or-features (filled arc); `Server` and `Memory` are alternative features (open arc). An example of constraint on this model (not shown in Figure 2.4) is that if feature `Dynamic screens` is selected, then feature `Dynamic customization` should also be selected, since the latter supports the former.

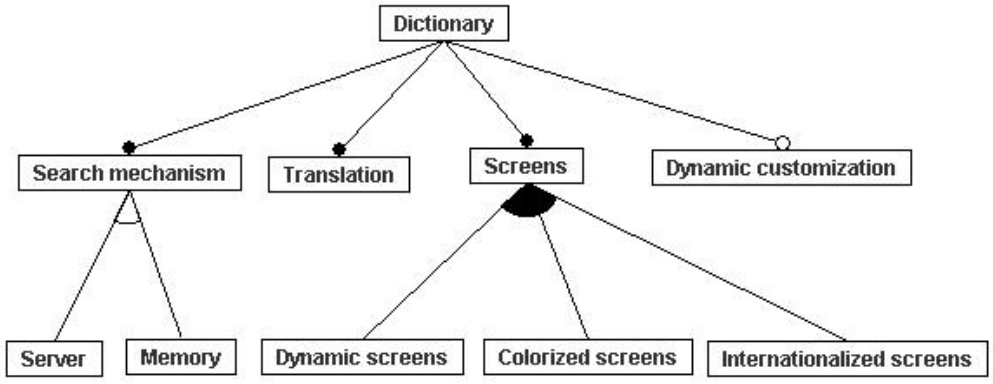


Figure 2.4: Feature diagram for Dictionary domain for embedded devices.

Variability in feature diagrams is expressed using optional, alternative, and or-features. These features are referred to as variant features. The nodes to which variant features are attached are referred to as variation points. In Figure 2.4, all features except `Translation`, `Screens`, and `Search mechanism` are variant features; features `Dictionary`, `Screens` and `Search mechanism` are variation points, which are clearly pinpointed in the diagram.

A number of approaches are based on FODA or combine it with other techniques:

- Feature Oriented Reuse Method (FORM) [76] is a layered approach to the original FODA and extends the latter to the software design phase and prescribes how the feature model is used to develop domain architectures and components for reuse; in particular, FORM provides a mapping from feature models to implementation artifacts. In order to refer this mapping, we employ Czarnecki's [45] term *configuration knowledge*.

- Featuring RSEB (FeatRSEB) [63] combines FODA and the Reuse-Driven Software Engineering Business (RSEB) method [72], by including the domain an engineering and feature modeling steps into RSEB, since this later provides no explicit feature models. FeatRSEB also extends the feature model diagram, which is then changed into a tree or a network of features linked together by UML dependencies or refinements. Another distinguishing addition to the feature model is the explicit representation of variation points.

- Bosch [32] defines a feature as a logical unit of behavior that is specified by a set of functional and quality requirements. The idea is that a feature is a construct used to group related requirements. We adopt this definition in our work.

- Benavides et al [25] extend feature models with extra-functional features and relations amongst attributes; they can automatically analyze five properties in this language, such as number of instances of a FM. However, they do not propose a set of refactorings for FM and use them to refactor SPL.

2.4.2 FAST

FAST (**F**amily-**O**riented **A**bstraction, **S**pecification, and **T**ranslation) [123] is a SPL development process conceived in the context of telecommunication infrastructure and real-time systems. It relies on a set of assumptions suggesting that SPL development it is worthwhile in a domain. The first assumption is that software development is often redevelopment, mostly creating new variations on existing software systems. Usually, there are more similarities than differences between variations. The second assumption is that it is possible to predict the changes that are likely in a software system, thereby deriving future changes from earlier changes. In the last hypothesis, both software and software development organization can be structured to take handle predicted changes, which can be made independently of other type of changes. The purpose is to confine changes refer only to a minimal subset of system modules.

FAST comprises three sub processes: domain qualification, domain engineering, and application engineering. Domain qualification provides an economic model relating the other two sub processes, by showing to which extent investment in domain engineering pays off in application engineering. In particular, domain qualification outputs an economic model estimating the number and value of family members and the cost to produce them.

In domain engineering, commonality & variability analysis is performed. Unlike FODA, there is no use of feature model nor any graphical notation displaying the configurability space of the product line. Additionally, core assets are designed and implemented. Two important such assets are the definition of Application Modeling Language (AML) and of the the application engineering environment. The former provides support for abstract specifications of applications. The latter enables analyzing the specifications written in AML and generating code from the abstract specifications. The synthesis of SPL instances can be either by composition or compilation, the choice of which is domain dependent.

Finally, in application engineering, the core assets developed in domain engineering

are used generates produce family members rapidly in response to customer requirements.

2.4.3 KOBRA

KobrA [19] has been developed at the Fraunhofer Institute for Experimental Software Engineering (IESE). This systematic method is designed for component-based product line engineering. The key ideia is the integration of the product line and component-based approaches, by unifying in this method the product line paradigm supporting "reuse in large" and the component based paradigm supporting "reuse in small".

KobrA method is divided in framework engineering and application engineering activities. The framework engineering activity is based on the product line approach in order to design and maintain a generic framework describing variabilities and commonalities. A framework is defined as the static representation of a set of KobrA components (*Komponents*) organized in the form of a tree. The results generated by this activity are framework models defined in terms of a mixture of textual and UML-based models. The application engineering activity is based on the component based approach in order to instantiate the generic framework.

Each Komponent is described at the specification level and at the realization level. The former defines the Komponents externally visible properties and behaviors; the latter defines how the Komponent is decomposed in lower level Komponents. The final goal is to develop applications containing specific variants corresponding to particular customers requirements. The results generated by this activity are application models defined in terms of a mixture of textual and UML-based models. Framework engineering comprises four activities:

- **Context Realization**, which elicits the environment properties and defines the framework scope. This phase takes also in charge the variabilities and commonalities analysis. The business process model and the decision models are produced.
- **Komponent Specification** Using the business model and the decision models, the purpose of this process is to describe the externally visible properties of a Komponent. To fulfill this goal, the structural model (UML Class/Object diagrams), the behavioral model (UML Statecharts diagrams), the functional model (operation schemata), and
- **Komponent Realization**, describes the private design of a Komponent realizing specification. To fulfill this goal, the interaction model (UML collaboration diagrams), the structural model (UML Class/Object diagrams), the activity model (UML activity diagrams), and the decision model (textual) are produced for each Komponent.
- **Component Reuse** reuses pre-existing components or integrates directly executable COTS components. The main task is to manage a process of negotiation between the reusing Komponent (desired specification) and the reused component (offered specification) in order to define a mutually acceptable specification.

The application engineering process uses the different models defined in the framework engineering in order to recursively transform all the generic framework models for the particular application. During this phase, the framework built during framework engineering is used to construct specific applications. The application engineering is divided into two activities:

- **Context Realization instantiation**. The purpose of this process is to instantiate the frameworks context realization. The potential users problems are analyzed and compared to the features supported by the framework. To fulfill this goal, context decisions and a concrete realization of the applications context are produced. To finish this process, customer-specific requirements can be added and the realization of the application context must be evaluated concerning its completeness and its correctness.

- **Framework Instantiation**. The purpose of this process is to instantiate recursively the generic Komponent hierarchy of the framework using context decisions and removing unwanted features. the decision model (textual) are produced for each Komponent.

2.4.4 Adoption Strategies

An orthogonal issue to product line development methods is adoption strategy. Independently from the SPL method, there are several approaches for developing SPLs [39]: proactive, reactive, and extractive [81]. In the proactive approach, the organization analyzes, designs, and implements a *fresh* SPL to support the full scope of products needed on the foreseeable horizon. In the reactive approach, the organization incrementally grows *an existing* SPL when the demand arises for new products or new requirements on existing products. In the extractive approach, the organization extracts *existing products* into a single SPL.

Since the proactive approach demands a high upfront investment and offers more risks, it may be unsuitable for some organizations, particularly for small to medium-sized software development companies with projects under tight schedules. In contrast, the other two approaches have reduced scope, require a lower investment, and thus can be more suitable for such organizations. Although the extractive and the reactive approaches are inherently incremental, it should be pointed out that the proactive approach can be incremental as well. In this case, products are simply derived based on whatever assets are in the core asset base at the time. However, there still needs to be a potentially high investment for this first increment and, although we do not need to have all core assets in hand before starting to build products, *all* such assets need to be designed and planned. An interesting possibility is to combine the extractive and the reactive approaches. But, to our knowledge, this alternative has not been addressed systematically at the architectural and at the implementation levels.

In all approaches, variability management must be addressed in the domain: while focusing on exploiting the commonality within the products, adequate support must be available for composing SPL core assets with product-specific artifacts in order to derive a particular SPL instance. The more diverse the domain, the harder it is to accomplish

this composition task, which in some cases may outweigh the cost of developing the SPL core asset themselves.

2.4.5 Scope

Indeed, as mentioned in Section 2.2, variability occurs at different levels, from requirements to implementation and test. However, despite the various existing SPL development methods, there is still lack of detailed guidelines for the implementation level and feature model level ***in the context*** of the SPL extractive and reactive adoption strategies. As mentioned in Section 2.4.4, these are often used in practice to minimize risks and costs. In view of that, the scope of our work is at the feature model level and at the code level in the context of the extractive and reactive SPL adoption strategies. The core of our method is described in Chapters 4 and 5.

Chapter 3

Current Variability Implementation Approaches

In order to enable the implementation of SPL adoption strategies, we consider in this chapter variability implementation approaches, since variability lies at the core of SPLs. Variability management approaches predate software product lines [39, 43]. Indeed, variability within single-software development, despite limited, has been supported by language features in most paradigms, such as structured programming, functional programming, logic programming, and object-oriented programming. However, with the emergence of the product line approach, such language features gained even more importance and were also refined and extended to design principles to meet the reuse goals of this approach.

This chapter reviews and compares essential concepts for handling variability in software product lines. The next chapter presents detailed discussion on a novel technique. The remainder of this chapter is organized as follows. We first consider atomic language underpinnings of variability in Section 3.1. Next, Section 3.2 describes the less fine-grained approach of design patterns; Section 3.3 then considers the more coarse-grained approach of framework technology. Feature-Oriented Programming is reviewed in Section 3.4. Subsequently, Section 3.5 addresses variability at deployment-time and at run-time. Sections 3.6 and 3.7 explain how some program transformation techniques and conditional compilation address SPL variability, respectively. We then review in Section 3.8 some AOP techniques. In Section 3.9, a framework for comparing the approaches is presented and then Section 3.10 describes each approach according to such framework. Finally, in Section 3.11, we compare these approaches.

3.1 Object-Orientation and Polymorphism

The variability mechanisms considered in this work are based on object-orientation [28] and its extensions. Ultimately, variability is implemented in programming languages. In this section, we consider object-oriented language features supporting variability. The next sections build on these features by providing higher level abstraction mechanisms.

Object-oriented languages define classes and manipulate objects, which have state and behavior specified by classes. State is represented by a set of attributes, which are

usually kept private (a capability known as *information hiding*), whereas behavior is represented by a set of methods, which are usually of public access to other objects. Application features in object-oriented programs are implemented by a collaboration of objects exchanging messages, where a message is a method call and frequently changes objects state.

When a message is sent to an object, the particular operation that is performed depends on both the request and the receiving object. Different objects that support identical requests may have different implementations of the operations that fulfill these requests. The run-time association of a request to an object and one of its operations is known as *dynamic binding*. This and subtype polymorphism–described shortly ahead– are the fundamental language mechanisms behind variability in framework technology, as discussed in Section 3.3. Further variability with objects is supported by polymorphism, which we describe next.

The word *polymorphism* is derived from the Greek language and means "the ability to have many forms". In programming languages, it refers to the idea of being able to write code against an abstract interface and plug in different concrete implementations behind the abstract interface. Programming languages support polymorphism with many different mechanisms. We list some of them in the following paragraphs. The original definition of polymorphism by Strachey from 1967 [118] distinguishes between *parametric* and *ad hoc* polymorphism:

> "Parametric polymorphism is obtained when a function works uniformly over a range of types; these types normally exhibit some common structure. *Ad hoc* polymorphism is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type."

Cardelli and Wegner [35] refined this definition as shown in Figure 3.1. Parametric polymorphism is obtained using generic parameters. *Inclusion polymorphism* corresponds to subtype polymorphism in object-oriented languages, that is, variables of a given type can also hold objects of its subtypes. *Coercion* refers to the automatic application of built-in or user-defined type promotions and conversions, for example, when adding an integral and a floating-point number, the integral operand is converted into a floating-point number. *Overloading* means providing different implementations of functions for the same function name, but different operand types. Overloading and coercion may conflict, for example, if both an overloaded function and an appropriate conversion are equally applicable.

For example, all the previously discussed variants of polymorphism are available in C++ [119]. Templates correspond to parametric polymorphism, virtual functions to subtype polymorphism, function overloading to overloading, and built-in or user-defined conversion operators or constructs to coercion. The only kind of polymorphism allowing run-time variability in C++ is subtype polymorphism (via dynamic binding). The remaining forms are completely resolved at compile time. All kinds of polymorphism in C++ rely on a static type system.

As another example, Java [62] also provides all kinds of polymorphism mechanisms.

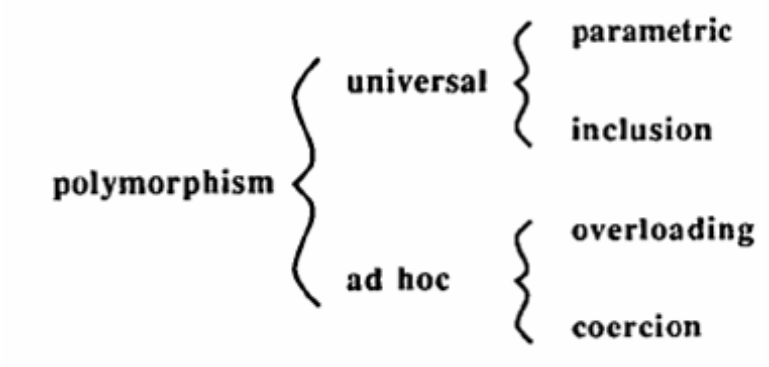


Figure 3.1: Varieties of polymorphism [35].

It supports both subtype polymorphism and parametric polymorphism[1]. Automatic promotions and type conversions are available for built-in types, but not for user-defined types. Methods can be overloaded as in C++, but operators cannot be overloaded by the user. Subtype polymorphism occurs in Java in its pure form, with *interfaces* to represent types; it can also occur with classes and `extends` constructs. Interfaces correspond to C++ abstract classes containing declarations of pure virtual methods, but no method implementations. An interface can be used to declare the type of a variable that can point to any object of any class implementing this interface. The introduction of interfaces relieves classes of their double role of being types and implementation repositories at the same time and lets them concentrate on the latter. Because this is not the case in C++, this language is sometimes said to support *subclass polymorphism.*

3.2 Design Patterns

In the previous section, we briefly characterized object-oriented language mechanisms for supporting variability. As language mechanisms, such features provide atomic support for that goal, but can be organized into coarse-grained structures such as design patterns and frameworks. In this section, we characterize design patterns for handling variability. In the next section, we address this issue with frameworks.

Design patterns involve descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context [55]. A design pattern names, abstracts, and identifies the key aspects of a common design structure that make them useful for creating a reusable object-design. The pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.

In the product line context, design patterns are particularly useful because several design patterns make some part of the design variable. For example, *bridge* allows

[1]Parametric polymorphism only became available in Java in 2004 with the *Generics* feature of J2SE 5.0.

varying the implementation of an object; *state* allows varying behavior depending on the state; *template method* provides a way to vary computation steps while keeping the algorithm structure constant. Table 3.1 summarizes variability support with some design patterns.

Design Pattern	Aspects that can vary
Bridge	implementation of an object
State	behavior
Decorator (wrapper)	responsibilities of an object without subclassing
Adapter	interface to an object
Strategy	an algorithm
Template method	steps of an algorithm

Table 3.1: Examples of variability support with design patterns.

In particular, the design pattern *template method* represents one of the fundamental structures used in object-oriented frameworks, described in Section 3.3. The class diagram in Figure 3.2 explains this structure. The `algorithm()` method calls a number of abstract methods, for example, `action1()` and `action2()`. Concrete implementations of the abstract methods are provided in the subclasses of `Framework`. The usual arrangement is that the base class `Framework` is a part of a framework, and an application defines a concrete subclass. The method `algorithm()` is referred to as *template method* because it defines the overall algorithm structure, but it lets the developer vary some of its steps.

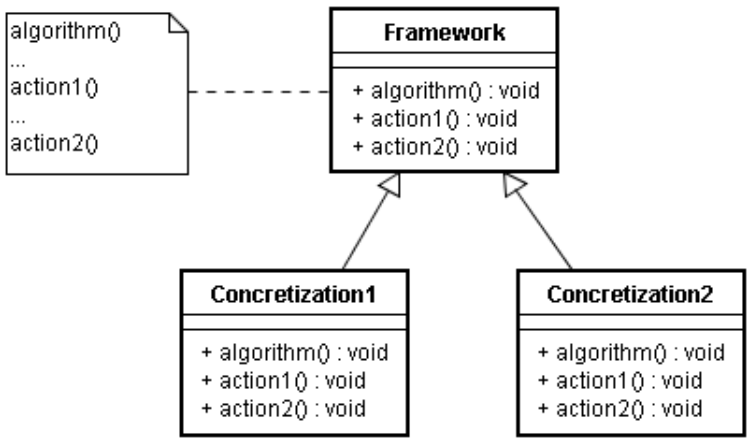


Figure 3.2: Template method design pattern.

Template method handles fine-grained variation and is generally implemented using inheritance to vary part of an algorithm. However, it can also be implemented with parameterized inheritance [45]. The first approach relies on dynamic binding and thus binding time is at run-time; the second approach is static and thus binding time is at compile-time. In contrast, *Strategy* uses delegation to vary the entire algorithm and has run-time binding mode.

3.3 Frameworks

A *framework* is a set of cooperating classes that make up a reusable design for a specific class of software [74]. A framework dictates the architecture of the application. It defines the overall structure, its partitioning into classes and objects, and their key responsibilities, how the classes and objects collaborate, and the thread of control. A framework predefines these design parameters so that the implementer can concentrate on the specifics of the application. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses that can be used directly.

Reuse on this level leads to an inversion of control between the application and the software on which it is based. When a developer uses a conventional library, he or she writes the main body of the application and calls the code to be reused. When the developer uses a framework, he or she reuses the main body and writes the code it calls. The developer will have to write operations with particular names and calling conventions, but that reduces the design decisions to be made.

In the product lines context, frameworks provide a high level of reuse. Indeed, a framework encapsulates domain knowledge and is a semi-complete application in that domain. By instantiating a framework, a product line member is created. The instantiation process consists on defining the behavior for variation points (also known as *hot spots* [107]) in the framework.

Variability in framework is often implemented with design patterns, such as template method, strategy, and state. However, in general, design patterns are also often implemented with dynamic binding, which brings run-time binding mode to frameworks and incurs into some performance degradation. Additionally, since dynamic binding is an intra-application mechanism, using it for inter-application variability often complicates frameworks, which usually tend to grow rapidly in complexity.

3.4 Feature-Oriented Programming

Using classes as the traditional units of organization of object-oriented software does not suffice to implement features modularly [100, 101]. Accordingly, a number of approaches focusing on more appropriate representation of features in the source code have emerged. A class of approaches concentrate on encapsulating features as increments over an existing base program, together with a mechanism for combining different features on demand. Approaches in this class include GenVoca [22], mixin layers [114], and AHEAD [24], and we refer to them as Feature-Oriented Programming (FOP). In this section, we first describe the key principles of FOP and then how it handles variability in the product line context.

Our explanation is driven by an example of a mobile game, whose simplified class structure is shown in Figure 3.3. The central abstraction in Figure 3.3 is the class `MainCanvas`. Once created by `MidletController`, `MainCanvas` also creates an instance of the `Resources` class and of the `GameScreen` class; `MainCanvas` handles user input and regularly updates game status by painting new state on the `GameScreen` object. In the

remainder of this section, we will use the abbreviation MG to refer to the mobile game. We will discuss variability by means of a particular feature of this software, namely `clouds`: as a scenery addition, MG can have clouds scrolling in the background.

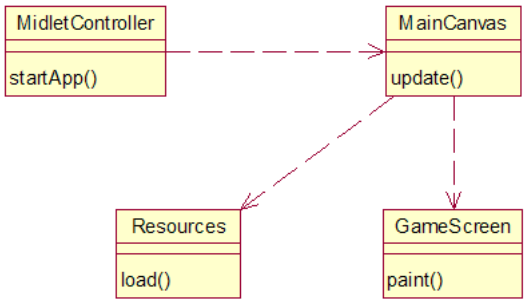


Figure 3.3: Mobile Game (MG).

With FOP, a feature is encoded as a delta over an existing base structure. This delta is usually expressed in a subclass/mixin-like style. Since we assume that we already have a running version of the MG application, using FOP, we basically implement the clouds functionality as a layer on top of MG; accordingly, we consider the MG software as the base element (a `constant` in terms of AHEAD):

```
class MidletController {...}
class MainCanvas{...}
class Resources {...}
class GameScreen{...}
```

and define the clouds feature as a delta on top of it

```
refines class Resources {
 Image clouds;
 Image getClouds() {return clouds;}
}

refines class GameScreen {
 void paint(Resources r) {
  super.paint();
  r.getClouds().paint();
 }
}
```

The delta definition above uses the syntax of AHEAD and contains extensions of `Resources` and `GameScreen` (in AHEAD, the modifier `refine` denotes extension). The first refinement maps cloud abstraction to a base type, while the refinement for `GameScreen` is needed to integrate the cloud feature into the control flow of the base.

Similar to subclasses, class refinements can add fields like in `Resources` or override methods like in `GameScreen`) with the additional flexibility that different layers can be freely combined. For example, we could have another layer that adds an enemy feature to the MG application and combine it with the clouds layer. Every combination of features can be made available in a separate namespace. This means that we would have classes like `base.Resources`, `base.GameScreen`, as well as `clouds.Resources`,...,`enemy.Resources`,...,`cloudsenemy.Resources`. Therefore, it is possible to have multiple different configurations of a product line on top of the same base application.

In terms of variability management, FOP is superior to framework technology, since it introduces a layer module with two distinctive capabilities. First, a layer encapsulates multiple abstractions and deltas of them, which together pertain to the definition of a feature into a single modular unit. A layer localizes the definition of a feature that would otherwise be scattered around the definition of several classes into a single code unit. Second, a layer is a mixin-like module, that is, it abstracts over the concrete variant of the base definition it applies to. Individual abstractions encapsulated within a layer are defined as mixins to their respective base abstractions, and the layer plays a similar role to them as classes play to their method definitions. This is a key to FOP support for variability management: variants of a base behavior can be composed in a plug-and-play fashion.

3.5 Deployment-Time and Run-Time Variability

Extending binding time to deployment-time and run-time is useful in the product line context: most applications are developed to be used by a very large number of customers, expecting different features, or imposing performance or memory constraints. Moreover, most software also has to be adapted in order to be used in specific system architectures. In particular, applications often have to be configured before or after it is delivered to the customer, or even when these are running. Nevertheless, it is desirable that such configuration be made without any extra development effort. Furthermore, some customers require that they configure some aspects of the software by themselves.

A well-known approach to this issue is parameterizing software. In this approach, we separate the source code from the definition of the values that may change from one version of the application to another. We refer to these values as *parameters*. The set of files in which the parameters are defined is referred to as *profile*. Indeed, various programming languages development environments support this technique. In particular, within Java, existing solutions include Properties, ResourceBundle, Jakarta Commons, JConfig, Preferences, and JNDI.

However, recent research [112] shows that such tools do not meet a set of requirements considered essential by developers when handling deployment and run-time variability. Accordingly, such research presents a tool meeting those requirements. In the rest of this section, we briefly consider this tools' features in addressing these requirements.

JPEL

JPEL (**J**ava **P**arameter **E**xpression **L**anguage) [112] is a tool for parametrization of Java applications. It implements a set of features, such as relationship among parameters, hierarchical grouping of parameters, and automatic adjustment of executing processes [112], which are frequently required by developers when addressing deployment and run-time variability. We first explain its basic functionality and then explain such features. The following example shows the `screen.jpel` file, which parameterizes game screen width.

```
module SCREEN {
    WIDTH = 600;
}
```

This simple file has only one parameter which represents the screen width. This file configures, at deployment-time, the game screen to have a 600-pixel width. The following code is an example of a Java class which makes use of the above parametrization file.

```
public class Screen {
    private int width;
    public void initScreen() {
        try {
            StaticConfiguration par;
            par = ConfigurationBuilder.staticConfiguration("screen.jpel");
            this.width = par.getInt("SCREEN.WIDTH");
            if (this.width == 123 ) {...}
        }
        catch (ConfigurationException ce) { ... }
    }
}
```

As shown in the example, variability of behavior can be implemented by having conditionals test the value of the parameter. In the following subsections, we list and explain JPEL's features.

Relationship between parameters

Some of the parameters of a system may have relationships among themselves. These relationships allow us to express some parameters as a function of others. If such relationship is implemented in the source code, the system loses flexibility. On the other hand, if the parametrization tool allows expressing the functional relation between parameters, every time we change a parameter that is at the domain of a function, we do not need to recalculate its image, which is done automatically by JPEL. This is a very desirable feature for a parametrization tool.

Back to the example previously shown, one might wish to parameterize the screen height as being the half of the screen width. This relationship can be expressed in a JPEL parametrization file, `screen.jpel` in this case:

```
module SCREEN {
    WIDTH = 600;
    HEIGHT = WIDTH / 2;
}
```

Pre-defined operators and extensible API

In order to define a relationship between parameters, JPEL provides a wide range of arithmetic and logic operators. Moreover, JPEL allows the developer to define new operators using a functional language. The following example shows the use of a user defined function.

```
module SCREEN {
    WIDTH = 555;
    HEIGHT = integerDivision(SCREEN.WIDTH, 2);
}
integerDivision (x,y) =
    | (x<y)      = 0
    | otherwise  = 1 + integerDivision(x-y, y);
```

This example defines `HEIGHT` as being the result of the integer division between `WIDTH` and 2. To express this relationship, a user defined function (`integerDivision`) is used. Moreover, JPEL allows developers to write relational operators using Java methods. In order to accomplish this, the tool provides a set of abstract classes and interfaces which the developer may implement to define new operators. These new operators can be easily included in a profile.

Parameters hierarchical grouping

In addition to the grouping of parameters in separate files, it is useful to group parameters in a hierarchical structure. In the game screen example, the `WIDTH` and `HEIGHT` parameters are grouped in the `SCREEN` module, and are referenced throughout the profile as `SCREEN.WIDTH` and `SCREEN.HEIGHT` respectively.

Automatic update at run-time

The features above support deployment-time variability. However, JPEL also supports parameters to be set dynamically, that is, run-time variability. JPEL dynamic parameters can be changed at run-time without the need to restart the application.

Back to the game screen example, one could find it necessary to use dynamic parametrization for the `WIDTH` parameter. The following Java code shows this:

```
public class Screen {
    private int width;...
    public void setWidth(int w) {
        this.width = w;
    }
```

```
    public void initScreen() {
        try {
            DynamicConfiguration par;
            par = ConfigurationBuilder.staticConfiguration("screen.jpel");
            par.bind(this, "setWidth", int.class, "SCREEN.WIDTH");
            par.execute();
            PolicyListener listener = new PolicyListenerReload();
            Policy onChange = new PolicyActivateOnChange();
            (( PolicyActivateOnChange) onChange).setPeriod(10000);
            onChange.addPolicyListener(listener);
            onChange.addConfiguration(par);
            onChange.start();
        }
        catch (ConfigurationException ce) { ... }
    }
}
```

This application checks for changes on the `screen.jpel` file every 10 seconds. Whenever this file changes, JPEL updates the system variables with the associated new parameters. In this particular case, the `SCREEN.WIDTH` parameter is bound to the `setWidth` method of the screen object. Whenever there is a change in the parametrization file, the `setWidth` method of `Screen` will be executed receiving the new `SCREEN.WIDTH` value as parameter.

3.6 Program Transformation

In general, program transformation systems should be considered when implementing product line variability because they are able to describe complex variability patterns, some of which can be crosscutting and have different levels of granularity, ranging from a single line of source code to the package or component level. In this section, we investigate how some transformation systems can be used to address product line variability. In particular, we describe a transformation system for Java in Section 3.6.1 and a language independent XML-based transformation system in Section 3.6.2.

3.6.1 Java Transformation System

In this section, we describe the Java Transformation System (JaTS) [98], a system for specifying and executing transformations in Java. First, we briefly explain JaTS main features. Next, we show how this system can be used to manage variability in a mobile device game.

JaTS Features

JaTS transformations are written in a language that extends Java with JaTS constructs. The goal of the constructions is to allow type (class or interface) matching and the

specification of new types that are to be generated. The simplest among these is the JaTS variable, which consists of a Java identifier preceded by the '#' character.

A JaTS transformation consists of two parts: a left-hand side (*matching template*) and a right-hand side (*replacement template*). Both sides consist of one or more type declarations written in JaTS. The left-hand side of a transformation is matched with the source Java type being transformed, which implies that both must have similar syntactic structures. The right-hand side defines the type that will be produced by the transformation.

The application of a JaTS transformation to a Java type is performed in three phases: parsing, transformation, and unparsing. The core phase is the transformation phase. The first phase parses the program to be transformed and the left-hand side template of the transformation and builds their corresponding parse trees. The second phase, transformation, has three sub-steps: matching, replacement, and execution. The first matches the parse tree of the left-hand side of the transformation with the parse tree of the source Java type being transformed. Roughly, a node in the source type matches the one in the left-hand side if they are identical or if the second one corresponds to a JaTS variable. A mapping from variables to the values that they were matched to is produced by the matching. This is called the result map of the matching. The second sub-step consists of replacing occurrences of JaTS variables in the parse tree of the right-hand side by corresponding values in the result map. The last sub-step consists of executing some JaTS structures in the parse tree of the right-hand side of the transformation. Such structures either query or update the parse tree, by optionally using iterative or conditional declarations. Finally, the third phase, unparsing, reads the parse tree and generates the text of the transformed program.

Managing Variability with JaTS

We now consider how JaTS can address variability implementation in the product line context. In particular, we explored this when building a product line of a mobile device game from existing versions for 3 different mobile phones. First, we briefly describe the game and how it was initially implemented without the product line approach. We then explain how JaTS helps with variation management during the product line adoption strategy.

Rain of Fire[2] is a shooting game, where the player is the master guardian of a city and controls ballistas and catapults to defend his/her town from several types of flying dragons with different speeds, power, and attack patterns. It is not necessary to kill every dragon, but to destroy as many as possible in order to prevent the main city buildings from being destroyed.

The game was initially ported in an *ad hoc* way to 3 devices: Nokia Series 40, Nokia Series 60, and Motorola T720. This means that each device-specific version was developed by copying an existing one, and adapting it manually. Clearly, this poses serious maintenance problems, specially in this domain, where the number of versions is frequently large. The goal of the study was to analyze the existing device-specific versions, identify the incurred variation patterns, and propose a technique to manage

[2]In cooperation with Meantime Mobile Creations/CESAR, under FINEP/FLIP, FACEPE/PAPPE and CNPq/Universal research projects.

Template-Variation Type
Extract/Implement Method
Add/Remove attribute
Add/Remove context
Add/Remove beginning block
Add/Remove ending block
Apply change do value
Class Hierarchy change
API Import
Argument in Method Call
Different Expression
Class Constant

Table 3.2: List of templates found.

these variations systematically while exploring the commonality. Although the technique could be applied retroactively, the general benefit would be to use it to either port the game to new platforms or to start porting new games. The approach relies on JaTS.

After analyzing the source code difference patterns of Rain of Fire's implementation for the three platforms, we then used JaTS to handle the variations identified. The process employed was a simplified version of the process described in Chapter 4, focusing on variability identification and extraction. The solution was based on the idea of extracting the code for an abstract platform (the *Core*) from the existing concrete ones, such that this core would contain all game features that were common to all platforms; after that, the code for each platform would be generated by transformations on the *Core*. It is worth mentioning that the code for the *Core* platform is not functional; it cannot be compiled. The main reason for its existence is to delimit the boundaries of the code that can be used in all three platforms without any modification.

In order to achieve this, we first catalogued all variations found. Next, we defined a pair of transformations: one transformation from device-specific code to the *Core* (called T1), and the reverse transformation, which would generate device-specific code from the *Core* (T2). We then identified code patterns that were present in all occurrences of each variation; overall, we were able to identify patterns for 11 variations. Finally, we created JaTS templates to match these patterns in source code and realize both T1 and T2. Table 3.2 shows a list of the templates developed.

Based on these templates for solving the variations, we set for constructing a software product line for the game. Since we had templates to address all the existing variations between the 3 different versions of the game, all we had to do was instantiate these templates for each occurrence of these variations. The first step was to merge all the template sets (left and right hand templates) responsible for implementing T1 (platform-to-core transformation) for every class in the game. This way, each class would have only one set of T1 templates, that included all variations necessary to transform a device-specific code into *Core* code, instead of one set for each variation.

The template set merging process was repeated for the T2 transformation. It was then possible to generate device specific code for any of the three platforms included

in the software product line, by applying the T2 transformations to the *Core* platform. The product line structure after this process is represented in Figure 3.4.

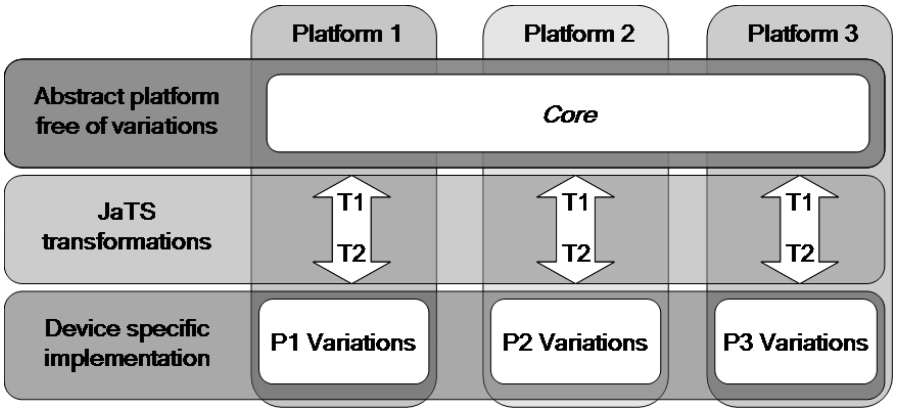


Figure 3.4: Structure of the Software Product Line implemented with JaTS.

For example, one of the problems we came across while analyzing the differences among the three platforms was the *Flip* variable feature. In the game, several images must be drawn in both directions (left to right and vice versa), like catapults, for example (see Figure 3.5); this drawing was implemented differently in Nokia's platforms and Motorola's T720. While in T720 there is a need for image objects facing both directions, Nokia's proprietary API features the flip operation, which can mirror an image upon its drawing on a canvas. In the T720 device, for drawing the two catapults, there were two calls to the drawing method receiving two different images as parameters (to draw the catapults on the left and right, respectively); in contrast, in Nokia's platforms, there was one call to the same drawing method used in T720 (to draw the left catapult), and another one to the proprietary API's method, receiving the same image as a parameter, but indicating that it should be flipped.

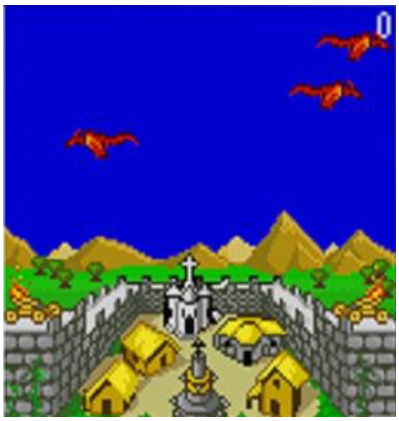


Figure 3.5: Catapults and dragons facing both directions on Rain of Fire.

The approach we chose to solve this variation is as follows: since the first call to the drawing method was common to both platforms, it should be moved to the *Core* with

T1's JaTS templates. The subsequent call, however, would not be there; instead, the *Core* would have a call to a `drawRightCatapult()` method call, whose definition would be implemented differently depending on the platform. The method definition and its composition with the core was accomplished with T2's JaTS templates.

Figure 3.6 shows the templates for implementing the T1 transformation for this variation. The names after the # character are called meta-variables; they are used to represent elements of Java source code such as class names, attributes, constants, code blocks, and so on. These templates are not totally complete; although they are functional, some parts of the code that would make it more generic were omitted for the sake of brevity and legibility.

Upon transformation, the meta-variables in Figure 3.6(a) are matched to the elements of the Java source file; `#ATTRS` (a `FieldDeclarationSet`) is used to store all the atributes of the class, while `#CDS` and `#MTDS` (a `ConstructorDeclarationSet` and `MethodDeclarationSet` respectively) store the constructors and methods of the class. This same template captures a method in the code with the signature `void m()` (hypothetically, the method which draws the catapults), and divides the body of this method in three meta-variables `#B1`, `#B2` and `#B3`. In our example, `#B2` is the code block of the method where the right catapult is drawn, and is specified explicitly by the developer.

The right-hand side template in Figure 3.6(b) makes the transformation generate the Java file with the same structure that was captured with the matching template (it just generates the previously captured attributes, constructors and methods), except for the body of the `m()` method. Notice that instead of a block `#B2` between `#B1` and `#B3` we now have a call to `this.newM()`, where `newM()` represents `drawRightCatapult()` in our case). This transformation is similar to the Extract Method refactoring [53], except for the fact that the extracted method (containing `#B2`'s code) is not implemented anywhere in the resulting class; this code is stored elsewhere, for future use in the T2 transformation (Core to specific platform). This way, it can be implemented differently for each platform in the product line. The resulting code from the transformation is the code of the *Core* abstract platform.

```
class #C {
   FieldDeclarationSet:#ATTRS;
   ConstructorDeclarationSet:#CDS;
   MethodDeclarationSet:#MTDS;

   void m() {
      Block:#B1;
      Block:#B2;
      Block:#B3;
   }
}
```

(a) Matching template

```
class #C {
   FieldDeclarationSet:#ATTRS;
   ConstructorDeclarationSet:#CDS;
   MethodDeclarationSet:#MTDS;

   void m() {
      Block:#B1;
      this.newM();
      Block:#B3 ;
   }
}
```

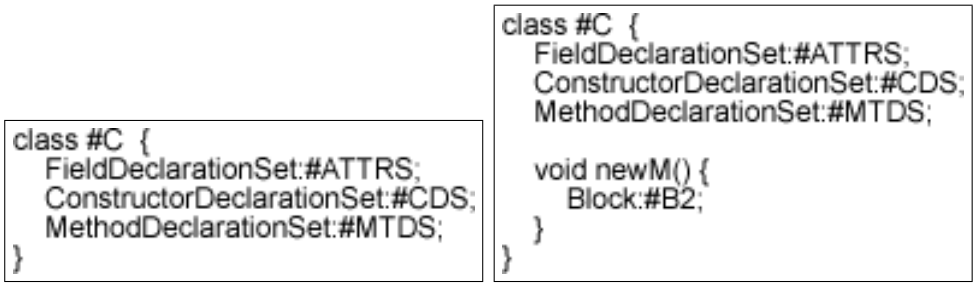
(b) Replacement template

Figure 3.6: T1 templates.

Figure 3.7 represents T2, that is, the reverse transformation of T1 in Figure 3.6. The

template in Figure 3.7(a) just captures the source code as it is, without specifying any matching constraints; Figure 3.7(b) shows the replacement template, which outputs the code and adds the implementation for the `newM()` method specific for the platform, thus generating device specific code. It is the body of this method that differentiates code from one platform to another's with respect to the flip feature: we could here generate code for S60, S40 or T720, depending on the replacement template chosen to apply to the *Core*.

T1 and T2 illustrated here address only one instance of a specific variation; they show how to derive code useful for an abstract platform from a concrete one, and vice-versa. As mentioned before, the complete transformation (the one that addresses all existent variations of a platform) consists of applying several of these template sets (matching and replacement) to many Java source files.



(a) Matching template (b) Replacement template

Figure 3.7: T2 templates.

The solution proved to be effective, and the variations were solved. It was easy working with the code patterns for variations, because JaTS templates work with pattern-matching. Apart from that, since JaTS transformations act directly in the source code, the code for each platform is legible and localized. With the use of T1 and T2 transformations, it is possible to define a porting path from any platform to another one in the product line, via the *Core* platform. However, there are some disadvantages: JaTS pattern matching has limited flexibility, and the templates used for these transformations depend partially on the *Core* code. Therefore, *Core* evolution potentially leads to JaTS templates evolution.

3.6.2 XVCL

XVCL stands for XML-Based Variant Configuration Language [73]. It is a technology designed for facilitating reuse and defining generic solutions on top of existing source code. It can address similarity patterns found on programs, substituting them by generic solutions in the form of XVCL meta-structures, which results in better maintainability and reusability. In other words, the use of XVCL provides programs with the possibility of being able to be easily:

- used as base for similar software (similar systems, from a product line for example, can be developed extending the first one);
- maintained to comply with changes that may arise from evolutionary issues.

The idea behind XVCL is similar to the concept of abstraction over statements in imperative languages (like methods or procedures): whenever a section of code is used several times in several different places, we encapsulate that as an abstraction and parameterize it. Then, when running that section of code, we just call that abstraction using the appropriate parameters. This avoids repeating identical blocks of code throughout the program, and makes it easier to correct or add any functionality to it afterwards, since such tasks become localized.

The corresponding component to abstraction over statements in this case is the `x-frame`, which is the main component of XVCL. Whereas only statements can appear in such abstraction, x-frames can include any kind of code in a program, besides other x-frames. Each XVCL program contains a root x-frame, called SPC (the specification file). A XVCL program consists of a composition of several x-frames. There is a XVCL processor, which analyzes the SPC file and builds the program, composing it with all other x-frames they use. Figure 3.8 illustrates an x-frame hierarchy (an x-framework), and Figure 3.9 shows how the processor traverses on this x-framework to compose the final program. So far we have explained the core behavior of XVCL. Next, we describe XVCL's features more closely.

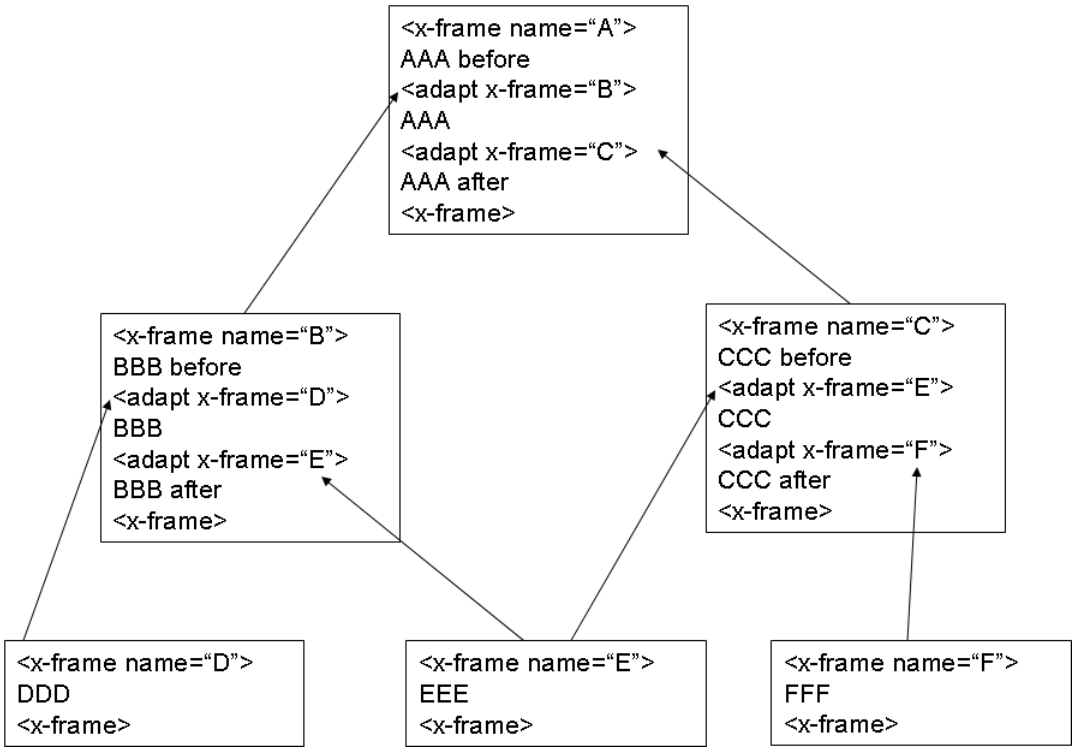


Figure 3.8: Example of an X-Framework.

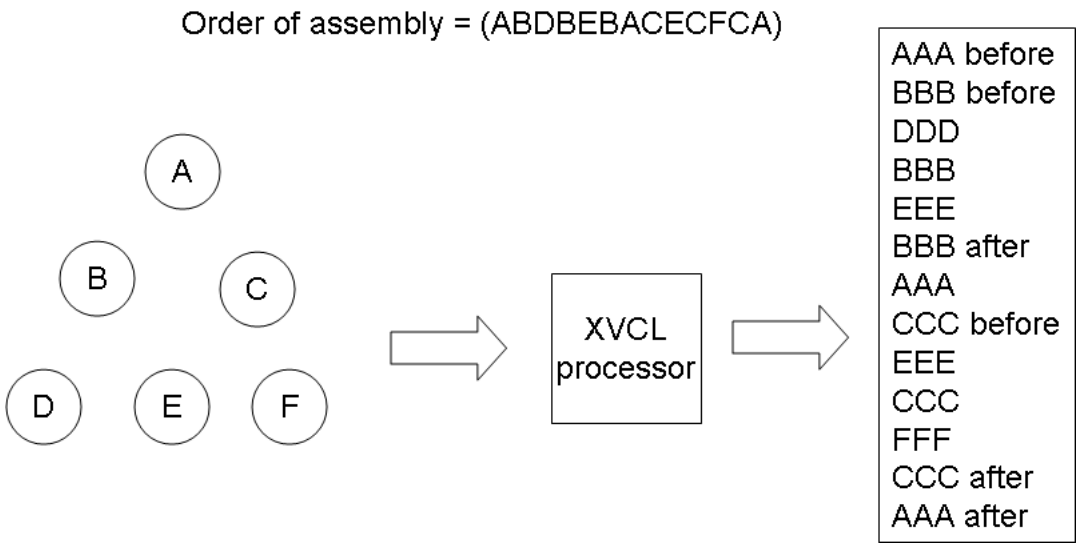


Figure 3.9: Traversal of X-Frames by the XVCL processor.

XVCL Features

This section describes the main commands of XVCL, presenting examples of how to use them for implementing variability.

Adapt

The `<adapt>` command defines an x-frame that the processor should look up to process. The processor finds this x-frame, and after processing it, returns the resulting code on the place where the `<adapt>` command was called. It basically instructs the processor to accomplish the following:

- adapt the x-subframework rooted in the named x-frame by inserting x-frame texts;
- emit/assemble the customized content of the adapted x-subframework into the output;
- resume processing of the current x-frame after processing the x-subframework rooted in the named x-frame.

In the following example, the specification file `MyBuild` adapts another x-frame, `Build.xvcl`.

```
<x-frame name="MyBuild" outfile="Build.java" language="java">
    <adapt x-frame="Build.xvcl"/>
</x-frame>
```

Break

The `<break>` command marks a breakpoint (variation point) at which changes can be made by ancestor x-frames. Additionally, it can define the default code, if any, that

may be replaced by those x-frames. It is similar to an AspectJ join point, which defines execution points in the code that are affected by an advice construct, except that the <break> command explicitly defines sections of code that are to be affected by insert commands (described shortly ahead). In the following example, the <break> construct identifies a variation point in the class where new methods can be added by ancestor x-frames.

```
<x-frame name="MyBuild"> public class Build extends GameItem { ...
    public Build(int x, int y, Image image) {...}
    ... //all the class' methods
    //variation points for adding new methods
    <break name="BUILD_NEWMETHODS"/>
} </x-frame>
```

Insert

<adapt>'s body, mentioned previously, may contain a combination of <insert> constructs. The <insert> command replaces the breakpoint break-name in the adapted x-subframework with <insert>'s body. If we compare it to AspectJ, just like an advice construct indicates code that is to be executed at a given join point, XVCL's <insert> command defines code to be inserted into a given <break> construct. The main difference is that code defined by <insert> is actually copied and then compiled to originate the program (like macro replacing), instead of being woven into bytecode like advice code in AspectJ.

There are two variations of the <insert> constructs. The <insert-before> command inserts the insert-body before the breakpoint break-name in the adapted x-subframework; the <insert-after> command inserts the insert-body after the breakpoint break-name in the adapted x-subframework. insert-body may contain a mixture of textual content and XVCL commands.

For example, suppose we want to add a new method to the MyBuild example shown in previous section. We already have a breakpoint defined to delimit the place where we can add the method. Now the only thing we need to do is to add the appropriate <insert> command in a x-frame that is parent to the frame that contains the <break> command. In order to do this, we will change the x-frame presented on the first example, to add an <insert> command that will add a new method to the class.

```
<x-frame name="MyBuild" outfile="Build.java" language="java">
    <adapt x-frame="Build.xvcl">
        //Breakpoint affected by the insert command
        <insert break="BUILD_NEWMETHODS">
            //Method we want to add
            public void drawDestroyedBuild(int offSetX, Graphics g) {
                ... //Body of the method
                //If desired, we could get this body from another x-frame.
            }
        </insert>
    </adapt>
```

```
</x-frame>
```

3.7 Conditional Compilation

Conditional compilation is a well-known technique for handling variation. It has been used in programming languages like C for decades and is also present in object-oriented languages such as C++ and C#. Basically, preprocessor directives indicate pieces of code that should compile or not based on the value of preprocessor variables. Such decision may be at the level of a single line of code or to a whole file. More recently, conditional compilation has been integrated with build environments such as Ant in order to support novel features, such as the following:

- a more expressive preprocessing language for specifying variants, including preprocessing expression language with logical connectives, for example;
- better user interface support, allowing easy selection of the desired variant.

An interesting tool with both features is the Antenna preprocessor [126]. In this section, we show how variability can be implemented with such preprocessor in the context of the mobile device game domain described in Section 2.3.

Managing Variability with Conditional Compilation

In order to assess the capability of conditional compilation in handling variability, we describe an industrial case study with which we collaborated [5] in porting one game to various mobile phones. First, we briefly describe the game; we then explain the game's specific variability issues; finally, we show how such variabilities were handling during the porting process with the Antenna preprocessor.

My Big Brother[3] is an interactive fantasy J2ME game developed for a TV show called Big Brother, a well-known reality show in the brazilian TV. The game interacts with the TV show, and the players are able to choose one of its characters and take care of them by buying food, hygienic items, gifts, and punishing them whenever they do not behave. Since the game uses a client-server system, the player is able to read news, answer quizzes, vote for characters to be expelled from the show, and update the status of their character.

To reach the target players planned by the customer, the game had to be ported to all major devices in the brazilian GSM market. After carrier's report with the most popular devices, 8 versions of the game were developed to target almost 50 devices (some devices are grouped into families and run the same code).

The most relevant porting issues involved screen size, network connections, key mapping, device known bugs, and J2ME MIDP versions. First, screen size variation implied generating different assets (mostly images) for different platforms, which prompted developers to deal in the code with screen positioning of each image for each platform. Second, the game uses HTTP POST connection to communicate with the game server.

[3]Developed by Meantime Mobile Creations/CESAR, which granted access to such game under FACEPE/PAPPE and CNPq/Universal research projects.

These connections can behave differently in some platforms, for example, by not handling HTTP redirections, or failing to read responses coded with an application/octet-stream content-type. Third, key mapping is a common variation that has to be handled in multi-platforms games: each device has its own key codes for mapping key presses. Fourth, some devices also have known issues in the virtual machine implementation, thereby forcing the developer to rely on work around. Lastly, a device may use a specific MIDP version, which may already provide built in support for a feature, which may not available in other devices; therefore, leveraging functionality across all devices involves the decision of either not using this feature at all or manually implementing it for devices where it is not built in.

The approach to handle these variations in the game was to identify the variability points between the targeted platforms and, using a preprocessor tool, isolate each platform-specific code from the single code base. As mentioned, the tool used to accomplish this task was the Antenna preprocessor [126], a set of Ant tasks suitable for developing wireless Java applications. Antenna provides a simple preprocessor, similar to the ones known from C and other languages. It supports conditional compilation, inclusion of one source file into another, and is helpful when trying to maintain a single source for several devices, each having its own known-issues and add-on APIs, for instance.

The following examples show how some of these variations were implemented using this approach. The first example addresses MIDP implementation variation. In this case, the T610 device uses MIDP version 1.0, which does not provide off-screen buffers. Since the feature is still required for this device, developers had to implement it explicitly. The solution was to implement this device-specific requirement within a preprocessor directive. The resulting structure is as follows:

```
class GameScreen extends Screen {...
  //#ifdef SCREEN_T610
    private Image bufimage
       = Image.createImage(128, 128);
    private Graphics bufgraph
       = bufimage.getGraphics();
  //#endif
  ...
```

where `bufgraph` is the off-screen buffer and is only defined for the T610 device. Next, the `paint` method needs to use this buffer in this platform, whereas for the others such method just uses the `Graphics` object:

```
  void paint(Graphics g){
    //#ifdef SCREEN_T610
      paintBuffer(bufgraph);
      drawProgressBar(bufgraph);
      g.drawImage(bufimage, 0, 0, 20);
    //#else
```

```
      paintBuffer(g);
      drawProgressBar(g);
    //#endif
  }...
}
```

The following code snippet shows how screen size variation was handled. The SCREEN_HEIGHT constant is declared and initialized with different values depending on the platform:

```
class Resources {...
    //#ifdef SCREEN_SIEMENS
    public static final int SCREEN_HEIGHT = 80;
    //#elifdef SCREEN_N40
    public static final int SCREEN_HEIGHT = 128;
    //#elifdef SCREEN_N60
    public static final int SCREEN_HEIGHT = 208;
    ...
    //#endif
    ...
}
```

Finally, the following code snippet shows how the handling of the key mapping variation was accomplished. The BOARD_SOFT_LEFT and BOARD_SOFT_RIGHT static fields are also defined to different values according to the platform.

```
 class GameController {...
  public static int BOARD_SOFT_LEFT = 0;
  public static int BOARD_SOFT_RIGHT = 0; ...
  static{
    //#ifdef KEYS_C650
      BOARD_SOFT_LEFT = -21;
      BOARD_SOFT_RIGHT = -22;
    //#elifdef KEYS_T720
      BOARD_SOFT_LEFT = -6;
      BOARD_SOFT_RIGHT = -7; ...
    //#elifdef KEYS_V300
      BOARD_SOFT_LEFT = -21;
      BOARD_SOFT_RIGHT = -22;
    //#elifdef KEYS_SIEMENS
      BOARD_SOFT_LEFT = -1;
      BOARD_SOFT_RIGHT = -4;
    //#endif
  } ...
}
```

3.8 Aspect-Oriented Programming

Aspect-oriented languages support the modular definition of concerns which are generally spread throughout the system and tangled with core features [79]. Those are called crosscutting concerns and their separation promotes the construction of a modular system, avoiding code tangling and scattering.

3.8.1 AspectJ

AspectJ [78] is an aspect-oriented extension to Java. Programming with AspectJ involves both aspects and classes to separate concerns. Concepts that are well defined with object-oriented constructs are implemented in classes. Crosscutting concerns are usually separated using units called *aspects*, which are integrated with the classes through a process called weaving. Thus, an AspectJ application is composed of both classes and aspects. Therefore, each AspectJ aspect defines a functionality that affects different parts of the system.

Aspects may define *pointcut designators* (or *pointcuts* for short), *advices*, and *inter-type declarations*. Pointcut match join points, which are a set of points during program execution flow, where we may want to execute a piece of code. Code to be executed at join points matched by pointcuts is declared as an advice. Inter-type declarations are structures that allow introducing fields and methods into a class, changing the hierarchy of a type (making a class extend another class or implement an interface), and turning checked exceptions into unchecked exceptions.

For example, aspect `Variation` below uses an inter-type declaration to introduce the `optionalImage` field into class `GameCore`. It also declares an advice which specifying code to be executed after the join point matched by pointcut `p`: the execution of method `loadImages` of class `GameCore`. We use `this(cobj)` to bind the `GameCore` object which is currently executing the `loadImages` method to `cobj`. This advice creates a new image and binds it to the introduced field `optionalImage`, after method `loadImages` has executed.

```
class GameCore {
   private Image mandatoryImage;
   public void loadImages() {
        this.mandatoryImage = Image.createImage(...);
   }
}
aspect Variation {
   private Image GameCore.optionalImage;
   pointcut p(GameCore cobj): execution(public void GameCore.loadImages())
                              && this(cobj);
   after(C cobj): p(cobj)
                  {
                   cobj.optionalImage = Image.createImage(...);
                  }
}
```

In addition to `after` advice, AspectJ also provides `before` and `around` advices definitions. The former allows advice code to execute before the join point; the latter is more expressive and allows conditional execution of advice code at the join point. More details on the language can be found elsewhere [87].

In the example, the image stored in `optionalImage` and its initialization could be part of a variant feature in a SPL. For instance, it could represent an optional image in a mobile device game, such that this image should be present in a version of this game for one device, but not for another device. By coding this variation as an aspect, a more localized and modular representation is achieved, thereby improving variability management for such SPL.

3.8.2 AspectBox

AspectBox [27] extends AOP languages by providing a new construct: an aspectbox, which can contain class and aspect definitions as well as import classes from other aspectboxes. The key idea is that such construct is a namespace mechanism for aspects, in the sense that aspects in the aspectbox affect only classes defined within it or classes imported into it. The base system is not affected outside the scope of an aspectbox. For example, the following piece of code[4] defines an aspect within an aspectbox affecting the imported class `GameCore`:

```
AspectBox Extension1 {
  import GameCore;
  aspect Variation1 {
      pointcut p(): call (* *(..)) ;
      after(): p()
      { ... }
  }
}
```

Accordingly, a pointcut definition contained in an aspect refers only to classes that are defined in the same aspectbox or that are imported: in the example pointcut `p` refers the imported class `GameCore` only. The advice in aspect `Variation1` refines the behavior of such imported class. Since aspectboxes are namespaces, eventual conflicts between aspects are avoided. For instance, there could be another aspect, `Variation2`, in another aspectbox, `Extension2`. By living in different scopes, both aspects `Variation1` and `Variation2` are kept separated, affecting different versions of the same class. Even if such aspects, which are defined in different aspectboxes, had the same join points, there would be no need to define precedence rules for composition ordering. This allows multiple deployment of concurrent modifications in the same base system, avoiding conflicting situations across aspectboxes.

Additionally, classes augmented with the aspect can also be imported from another aspectbox. From the point of view of an importing aspectbox, there is no distinction between classes defined within the aspectbox and those imported. Furthermore, The import relationship is transitive: If aspectbox `Extension2` imports a class `GameCore`

[4] AspectBox's syntax is based on Squeak; the example is based on an equivalent AspectJ-like syntax.

from aspectbox `Extension1`, then a third aspectbox `Extension3` can import `GameCore` from `Extension2`. From the point of view of the importing aspectbox `Extension3`, there is no difference if the class is defined or imported in the provider aspectbox `Extension2`. Nevertheless, because aspects cannot be reused across multiple base systems, aspects cannot be imported.

AspectBox emerges from Classbox [26]. The Classbox module system allows a class to be extended by means of class member additions and redefinitions. These extensions are visible in a locally and well-delimited scope. Several versions of a same class can coexist at the same time in the same system. Each class version corresponds to a particular view of this class. Classboxes and aspectboxes have a common root which is the scoping mechanism for refinement. Whereas classboxes, like FOP, support structural refinement (i.e., class members addition and redefinition), aspectboxes offer a scoping mechanism for behavioral refinement.

3.8.3 CaesarJ

Traditional OO languages have only very limited and lightweight means for organizing sets of collaborating classes, for example packages in Java or name spaces in C++. With these lightweight organizational units, it is not possible to express variants of a collaboration, use collaborations polymorphically, use collaborations as first class values [92]. In other words, all these language mechanisms that proved so useful for single classes are not available for sets of collaborating classes.

In this context, CaesarJ [18] is an aspect-oriented language supporting reusability. It combines the aspect-oriented constructs, pointcut and advice, which can be bound at run-time, with coarser-grained object-oriented modularization mechanisms. From an aspect-oriented point of view, this combination of features is particularly well-suited to make large-scale aspects reusable. From a component-oriented view, on the other hand, CaesarJ is addressing the problem of integrating independent components into an application without modifying the component to be integrated or the application.

The key concept in CaesarJ is that of a *virtual class*, which is an abstraction encapsulated within a module called *family class*. A virtual class, like a virtual method, can have different meanings, which depends on the context of use. Virtual classes are defined as inner classes of an enclosing family class; like methods and fields, they are also members of instances of their enclosing family class. Hence, at any time during the execution their meaning is relative to the dynamic type of the family object. Such abstractions can be overridden and late bound (just like virtual methods). In addition, old relations are inherited but automatically re-directed to the most specific definition of a type reference. Moreover, new classes and relations can be added.

For example, Figure 3.10 shows the family classes `HierarchyDisplay` and `AdjustedHierarchyDisplay`, each of which is a set of collaborating virtual classes and where the latter family class refines the former by adding new members to the `Node` virtual class. Such class in the latter family class refines the same class in the former, and all the collaborating classes refer to the refined node. No cast is necessary, since all references to the type `Node` in the other virtual classes are automatically re-bound to the refined `Node` class, when they are referred to during the execution of an object of type. In particular, in the context of an instance of `AdjustedHierarchyDisplayType`,

a `CompositeNode` is a subclass of the refined `Node` class.

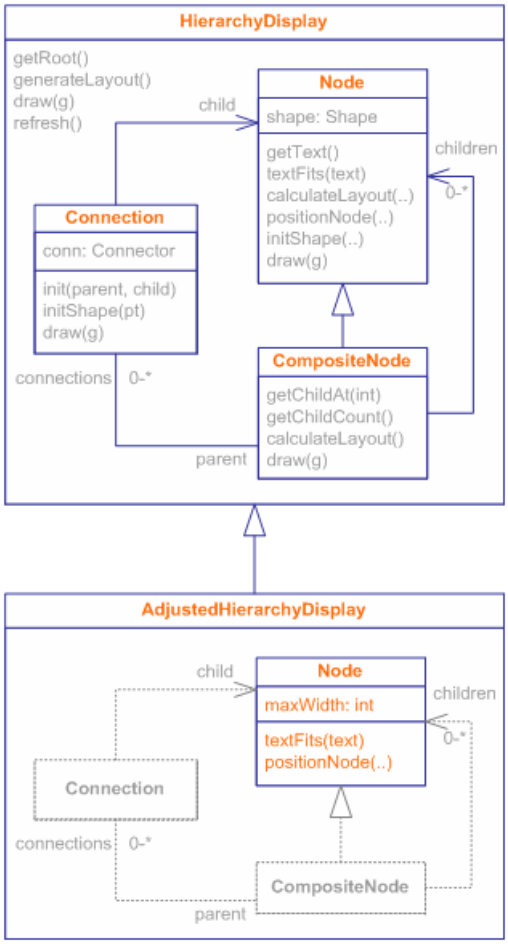


Figure 3.10: Refinement of virtual classes in CaesarJ .

Another feature of CaesarJ providing flexibility is that family class modules can be combined using mixin composition semantics (like FOP) which propagates into virtual classes. For instance, Figure 3.11 shows family classes `AdjustedHierarchyDisplayType` and `AngularHierarchyDisplayType` refining family class `HierarchyDisplay`. These latter are then combined with mixin composition into family class `AdjustedAngularHierarchyDisplayType`. As a result, in the context of this family object, the collaborating classes refers to the refined versions of `Node` and `Connection`. From a design viewpoint, features are implemented as classes and domain objects are modeled by virtual classes. Features can refine classes of other features and features can crosscut other features. Finally, features are combined by propagating mixin composition.

3.9 Comparison Framework

In order to systematically describe techniques for implementing product line variability, we propose a framework following the principle of employing domain analysis in the

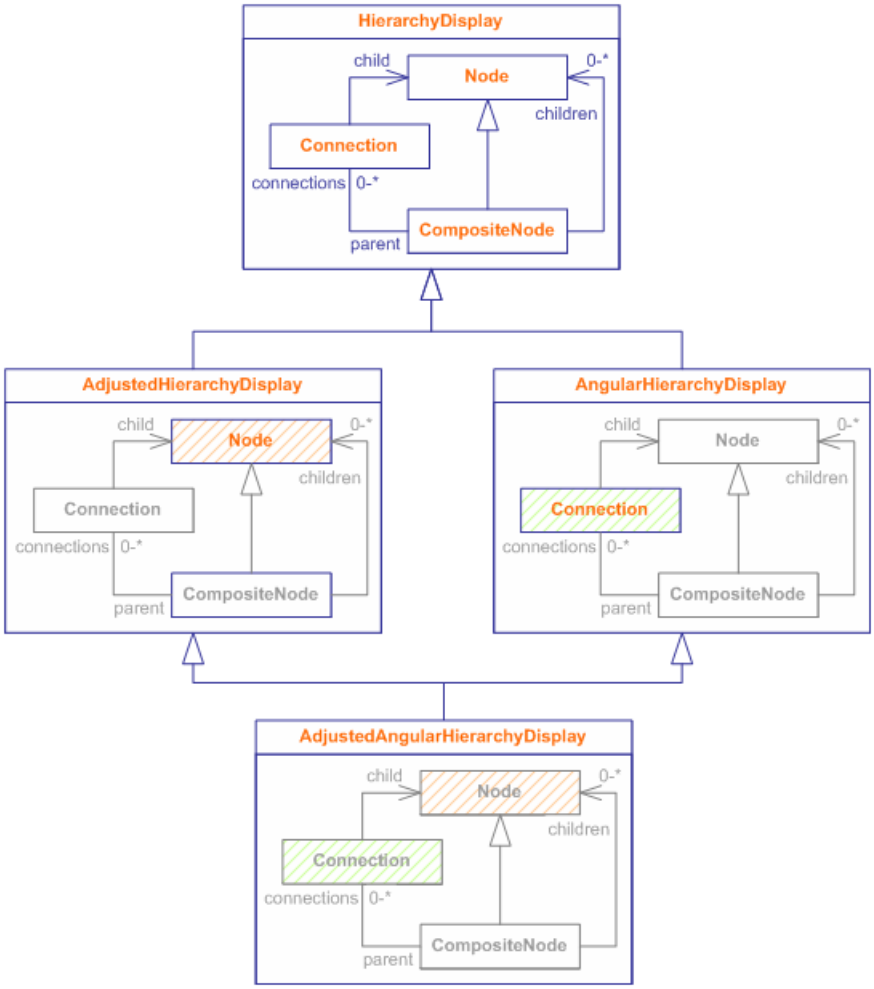


Figure 3.11: Propagating mix-in composition .

solution space, according to Coplien's PhD thesis [43]. Accordingly, the solution space (each technique presented in the previous sections) is described in terms of the kind of variability it supports, the nature of variability, its granularity and binding time.

The framework has also been *extended* to incorporate some elements such as reusability, which is also germane to the product line context [16]. We further extend the framework with other relevant elements, such as performance and application size, which are useful in certain domains such as mobile device games, as described in Section 2.3. Last, but not least, we also consider the support for modular implementation of crosscutting concerns as a parameter in the framework, since, as argued in Chapter 2, the implementation of SPL variability is frequently crosscutting. Following Colyer et al [42], we distinguish between homogeneous and heterogeneous crosscuts: the former refine multiple join points with a single piece of advice, whereas the latter refine multiple join points each with different pieces of advice.

The taxonomic parameters and their possible values are described in Table 3.3. Even though the values of the framework items reusability, performance, and application size are described in a non-quantifiable manner (high, medium, low), we remark that they

are still useful for a qualitative analysis, which is the focus of this chapter.

Framework item	Possible values
Variability type	positive, or negative or both
Variability in structure	supported or not supported
Variability of behavior	supported or not supported
Granularity	fine-grained or coarse-grained
Binding time	preprocessing, compile, deployment, run-time
Reusability	high, medium, or low
Performance	high, medium, or low
Application size	high impact or low impact
Support for modular crosscutting	supported or not supported

Table 3.3: Framework for describing variability techniques.

Variability type indicates addition or removal of behavior or structure when compared to a base feature implementation. There are two ways of how variability can be described: positive variability and negative variability. Positive variability optionally adds structure and/or behavior to a given core. Negative variability optionally removes structure and/or behavior from a given core (Figure 3.12). Not necessarily does negative variability imply reduced number of lines of code, but it usually does. It depends on the variability mechanism: fine-grained mechanisms such as conditional compilation supports negative variability; for other mechanisms such as frameworks and AOP, some refactoring might be necessary, which has to be outweighed against the structure/behavior removed.

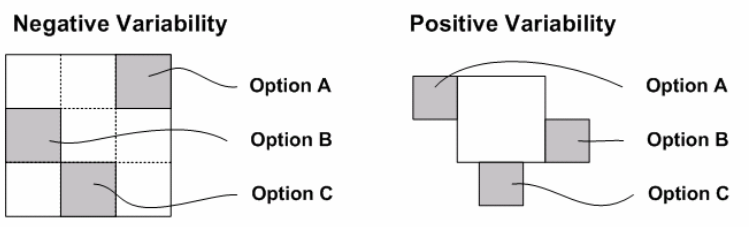


Figure 3.12: Positive and negative variability .

Binding time refers to the point in development time when decisions are bound for the variations, after which the behavior of the final software is fully specified [83]. Fine-grained granularity refers to variation within classes, specially within method bodies, whereas coarse-grained granularity refers to granularity above the class level.

3.10 Instantiating the Variability Framework

This section synthesizes each of the variability techniques described in the previous sections in terms of the variability framework proposed in Section 3.9.

3.10.1 Design Patterns

Table 3.4 shows how *some* design patterns can be classified within the variability framework proposed in Section 3.9. Each framework item can have different values for different patterns. For instance, granularity can be either fine-grained with *Template Method*, which handles intra-method variability, and coarse-grained with *Strategy*, which handles algorithm variability (possibly involving data structures). For variability type, it can be positive with decorator, which addresses behavior variability, or negative with decorator, bridge, and adapter. For the latter, negative variability occurs at a more coarse-grained level, by optionally skipping implementation of selected operations. In terms of binding time, delegation allows *Bridge* and *Strategy* to have run-time binding mode, and dynamic binding allows *Template Method* to have the same binding time. On the other hand, if this pattern is implemented with parameterized inheritance, then its binding time is compile-time. Finally, design patterns do not adequately support modular crosscutting implementation, since even the implementation of most patterns is crosscutting [56, 67].

Framework item	**Possible values for some design patterns**
Variability type	positive (decorator), negative (adapter, bridge, decorator)
Variability in structure	supported (adapter, bridge, decorator)
Variability of behavior	supported (state, strategy, template method)
Granularity	fine-grained (template method), coarse-grained (strategy)
Binding time	compile-time (template method with parameterized inheritance), run-time (strategy, template method with inheritance, bridge)
Reusability	medium
Performance	low (strategy), high (adapter)
Application size	varies
Support for modular crosscutting	not supported

Table 3.4: Some design patterns according to the variability framework.

3.10.2 Frameworks

According to Section 3.3, Table 3.5 shows how framework technology can be classified within the variability framework proposed in Section 3.9. Since frameworks often rely on design patterns to implement variability, frameworks support both variability in structure and in behavior as well as positive and negative variability. Negative variability can often be supported with inheritance with cancelation [115]; otherwise, application size can grow considerably. Further, as a framework is a semi-complete application providing an integrated set of domain-specific structures and functionality [113], granularity and reuse are high. Due to this coarser-grained nature compared to design patterns, frameworks usually have compile-time binding mode, even though the patterns they may rely on may have either binding time. The reason is that, when a framework is instantiated, the variant behavior is often predictable, usually with at most some variation

points frozen during runtime, and thus static optimization can often eliminate dynamic binding. In the cases in which dynamic binding is not eliminated, performance can be affected. Finally, OO frameworks do not support modular crosscutting implementation, since in the clean OO model of frameworks crosscutting behavior is usually expressed by small code fragments scattered throughout several functional components.

Framework item	Possible values for framework technology
Variability type	positive, negative
Variability in structure	supported
Variability of behavior	supported
Granularity	coarse-grained
Binding time	usually compile time
Reusability	high
Performance	varies
Application size	varies
Support for modular crosscutting	not supported

Table 3.5: Framework technology according to the variability framework.

3.10.3 AOP

Based on Section 3.8, we are now able to describe AOP, according to the variability framework. The framework instance for AspectJ is shown in Table 3.6.

Framework item	AspectJ possible values
Variability type	positive, negative
Variability in structure	supported
Variability of behavior	supported
Granularity	fine-grained, coarse-grained
Binding time	compile, run, deployment time
Reusability	medium, high
Performance	high
Application size	varies
Support for modular crosscutting	supported

Table 3.6: AspectJ described according to the variability framework.

According to Table 3.6, AOP supports both positive and negative variability. For instance, the former can be supported with refinement of virtual classes in CaesarJ as well as with inter-type declarations in AspectJ and `before` or `after` advice constructs; the latter is supported with `around` advice constructs where `proceed` is not invoked. Additionally, variation in structure is supported with inter-type declarations in AspectJ or with mixin-based composition and refinement of virtual classes in CaesarJ, while variation in behavior is supported with advice constructs. Further, granularity with

AOP can be either fine-grained (with most constructs) or coarse-grained (with inter-type declarations changing type hierarchy in AspectJ or with virtual classes in CaesarJ).

Besides, reusability can be either medium, because the pointcut language is still mostly syntactically based in AspectJ, which makes aspects too dependent on names, or high with virtual classes in CaesarJ, which allow reuse of collaboration of classes. In terms of binding time, it can at compile time with AspectJ and CaesarJ, at deployment time with AspectBox, and at runtime with CaesarJ. Application size can grow considerably, specially when using generic pointcuts intercepting various join points. This code bloat issue should be considered carefully in domains with constrained resources such as the mobile device domain we described in Section 2.3. According to our empirical experience [14], part of this can be avoided with optimization techniques. Finally, AOP supports modular implementation of crosscutting concerns.

3.10.4 FOP

Table 3.7 categorizes FOP (Section 3.4) according to the variability framework. The technique supports both positive and negative variations, since the delta layers corresponding to features can be freely combined, which also incurs in high reusability. Additionally, variability in both structure and behavior is supported with the addition of new fields or methods, or by overriding existing ones. Further, FOP has a granularity grasp for coarse-grained variations, where the smallest unit of composition is a delta layer of `refine` statements over an existing base.

FOP also causes minimum impact on the performance of programs: all feature code is processed into normal source code of the target language (Java, for instance) that will then be compiled, just as if it were all written directly in that language (all transformations are applied in preprocessing time). However, an alternative implementation approach could directly support FOP at compile-time. Furthermore, since homogeneous crosscutting is not supported in FOP, its impact on application size is not significant. Finally, FOP supports modular implementation of crosscutting concerns, which are localized in the delta layers. However, only heterogenous crosscutting is supported.

Framework item	**FOP possible values**
Variability type	positive, negative
Variability in structure	supported
Variability of behavior	supported
Granularity	coarse-grained
Binding time	preprocessing-time, compile-time
Reusability	high
Performance	high
Application size	low
Support for modular crosscutting	partially supported

Table 3.7: FOP described according to the variability framework.

3.10.5 JPEL

Based on the description of Section 3.5, we are now able to describe deployment-time and run-time variability with JPEL according to the variability framework. The framework instance for JPEL is shown in Table 3.8.

Framework item	JPEL possible values
Variability type	not supported
Variability in structure	not supported
Variability of behavior	supported only for changes in values
Granularity	fine-grained
Binding time	Deployment- and run-time
Reusability	Medium
Performance	potentially low
Application size	potentially high
Support for modular crosscutting	not supported

Table 3.8: JPEL described according to the variability framework.

JPEL has the limitation that it can only implement value variations and thus does not support modular implementation of crosscutting concerns. Using JPEL, it is only possible to improve code reuse by extracting values from the source code to a parametrization profile. Thus, a JPEL profile can not change the program structure. Moreover, the changes in the program behavior are limited to that generated by value changes. Further, positive and negative variability are not supported, because the selection of the desired feature would be on the source code too, implemented using a conditional structure (`if` or `switch`, for example).

JPEL profiles support fine-grained variability. We can use parametrization files to change the values of local variables, or even the value of some constants used to calculate an expression. This also constrains reusability. Additionally, possible binding-times with JPEL are deployment-time and run-time. The former is happens when specifying profile variable values before the application runs; the latter when it is running, which is made possible by JPEL's automatic update at run-time feature discussed previously.

The use of JPEL may generate some impact to the system performance. This impact can be smaller in a deployment context, where the parameters are read from the profile only when the method that fetches the parameter value is called. This usually happens only once, when the application starts. On the other hand, in a run-time context, the system checks for changes in the profile periodically. This can lead to a significant performance degradation, depending on the time between each verification and the number of parameters which have to be updated.

Furthermore, using JPEL implies in increasing the total application size, since in order to extract the definition of some values to a parametrization file, new commands must be added to the source code. Moreover, to run a program which uses JPEL, the JPEL library must be present on the system (on the `classpath`). Finally, when implementing negative features, code which is not called is still in the bytecode. This may be a killer factor for resource-constrained domains, as explained in Section 2.3.

3.10.6 JaTS

Given the description of JaTS in Section 3.6.1, we now describe this technique according to the variability framework. The framework instance for JaTS is shown in Table 3.6. According to Table 3.9, JaTS supports both positive and negative variability as well as variability in structure and behavior. Indeed, such capability is achieved with replacement templates, which can add/remove these issues. Further, granularity with JaTS can be either fine-grained, with templates specifying changes within method bodies, or coarse-grained, with templates applying to a set of classes.

Framework item	JaTS possible values
Variability type	positive, negative
Variability in structure	supported
Variability of behavior	supported
Granularity	fine-grained, coarse-grained
Binding time	preprocessing
Reusability	medium, high
Performance	high
Application size	low
Support for modular crosscutting	not supported

Table 3.9: JaTS described according to the variability framework.

Besides, reusability can be either medium, because the matching templates may depend considerably on code within method bodies, or high, with matching templates applying to whole set of classes. Whereas binding time is at preprocessing and performance is high, application size does not grow considerably, specially because, unlike AspectJ, variation points specified by JaTS constructs are not scattered in various join points. Therefore, the code bloat issue with AspectJ described in Section 3.10.3 does not affect JaTS, which is an advantage for domains with constrained resources such as the mobile device domain we described in Section 2.3. Finally, JaTS does not support modular implementation of crosscutting concerns.

3.10.7 XVCL

Table 3.10 describes XVCL (Section 3.6.2) according to the variability framework. The technique supports both positive and negative variations (by use of the `<break>` construct) as well as variability in both structure and behavior (by use of the `<insert>` construct). XVCL has granularity grasp for both fine- and coarse-grained variations: not only can it perform changes as fine as adding/removing simple lines of code (by use of the `<break>` construct), but it can also compose whole x-frames in order to instantiate a particular variant (by use of the `<adapt>` construct).

Additionally, XVCL provides high reusability: the concept of the x-frames makes it possible for it to be reused several times in various product line instances (by use of the `<adapt>` command). The technique also causes no impact on the performance of programs: all XVCL code is processed into normal source code of the target language

Framework item	XVCL possible values
Variability type	positive, negative
Variability in structure	supported
Variability of behavior	supported
Granularity	fine-grained, coarse-grained
Binding time	preprocessing
Reusability	high
Performance	high
Application size	low
Support for modular crosscutting	not supported

Table 3.10: XVCL described according to the variability framework.

that will then be compiled, just as if it were all written directly on that language (all transformations are applied in preprocessing time). Furthermore, since variability constructs do not specify scattered variation points (which is the case for AspectJ, for instance), the impact on application size is not significant. However, XVCL does not support modular implementation of crosscutting concerns.

3.10.8 Conditional Compilation

Finally, according to the variability framework proposed in Section 3.9, we are now able to describe the conditional compilation. Table 3.11 summarizes such description. The use of preprocessor directives enables this technique to support both positive and negative variations as well as variability in both structure and behavior. Conditional compilation can specify inclusion or not of a whole module (coarse granularity) or inclusion or not of a single line of code (fine granularity). In the latter case, which is more frequently used, reusability is limited and maintenance poor.

Framework item	Conditional compilation possible values
Variability type	positive, negative
Variability in structure	supported
Variability of behavior	supported
Granularity	fine-grained, coarse-grained
Binding time	preprocessing
Reusability	low
Performance	high
Application size	low
Support for modular crosscutting	not supported

Table 3.11: Conditional compilation described according to the variability framework.

On the other hand, the technique causes no impact on the performance of programs, since all preprocessor directives are processed into source code of the target language

that will then be compiled, just as if it were all written directly on that language (all transformations are applied in preprocessing time). Furthermore, since preprocessor directives specify mostly fine-granularity variability, the impact on application size is not significant. Once the preprocessing tags are in place the configurability of the variability is flexible, by involving a combination of tags; nevertheless, conditional compilation does not support modular implementation of crosscutting concerns.

3.11 Comparative Analysis

We now synthesize and contrast the descriptions from the previous subsection. Table 3.12 synthesizes the results. In the table, the following abbreviations are used. **H** (high), **L** (low), **S** (supported), **NS** (not supported), **Ps** (partially supported), **C** (coarse-grained), **F** (fine-grained), **RT** (run-time), **PT**(processing-time), **DT** (deployment-time), **CT** (compile-time).

Since all previously described techniques support variability in behavior and positive/negative variability (except JPEL, which supports neither positive nor negative, since variability is accomplished mostly with configuration constants and variant behavior is pre-implemented in the code), we omit the corresponding rows from Table 3.12, since they would not be relevant for contrasting these techniques. Additionally, for brevity, Table 3.12 does not display design patterns because various patterns have different values according to the comparison framework. Some patterns according to this framework were listed in Table 3.4.

Framework item	**Frame-works**	**FOP**	**JPEL**	**AOP**	**JaTS**	**XVCL**	**Cond. compil.**
Variability in structure	S	S	NS	S	S	S	S
Granularity	C	C	F	C,F	C,F	C,F	C,F
Binding time	CT	PT, CT	DT, RT	CT, DT, RT	PT	PT	PT
Reusability	H	H	M	M,H	M,H	H	L
Performance	H,L	H	L	H	H	H	H
Application size	H,L	L	H	H, L	L	L	L
Support for modular crosscutting	N	Ps	N	S	N	N	N

Table 3.12: Comparing implementation mechanisms according to the variability framework.

Although frameworks provide support for both positive and negative variability, the granularity of such variability is mostly coarse-grained and dependent on underlying design patterns such as *Strategy* and *Bridge*. Regarding structure, JPEL does not support variability in structure, which implies more variability effort at run-time, thereby leading to worse performance.

Fine-grained and coarse-grained granularity can be both supported by XVCL, JaTS, conditional compilation, and AOP, whereas for the other mechanisms either one or the other is supported. XVCL has x-frame hierarchy and `<adapt>` command allowing for both types of variability, respectively. JaTS templates can be applied to individual classes or to sets of classes, and conditional compilation can specify variability at a single line of code or at whole classes. AOP has both a pointcut language for fine-grained variability (not as fine-grained as conditional compilation), and inter-type declarations (AspectJ) and family class (CaesarJ) which allow coarse-grained variability.

In terms of binding time, JPEL supports run-time and deployment-time, due to its automatic-update feature and deployment feature. AOP additionally supports compile-time binding time, which could also be supported by FOP. The other mechanisms support only one binding time, ranging from preprocessing to run-time. Further, we note that poor reuse is achieved with conditional compilation, since its constructs are mostly often applied for fine-grained variability. Reusability improves for JaTS and AOP/AspectJ, but could be limited with templates and pointcuts, respectively. Frameworks and FOP support high level of reuse because they are easily integrated with higher level instantiation mechanisms such as Domain-Specific Languages (DSL) [90] and equational specifications [24]. AOP/CaesarJ also supports high reuse with the concept of family class and mixin-based composition. Reuse with XVCL is also high due to flexible composition of x-frames.

Application size can become a problem with AOP, since too generic pointcuts, despite expressive, lead to pervasive weaving of code into the base bytecode, despite the fact that optimizers can partially alleviate this problem. Frameworks can also become of significant size, since use of inheritance for handling inter-application variability easily leads to an explosion of combination of little classes and interfaces, which does not happen with FOP, since this latter supports plug-and-play combination of features. JPEL can also have significant impact on application size, since it also has run-time binding mode, which means the deployed application must embed the handling of variation.

Although XVCL has attractive features for implementing variability, a problem with this mechanism is poor legibility. Since the approach is language-independent, the developer has to explicitly indicate variation points in the code. AOP (AspectJ), on the other hand, already has an expressive pointcut language which allows expressing variability in a non-invasive way.

Even though conditional compilation does not support the modular implementation of crosscutting concerns, once the preprocessing tags are in place, the configuration activity is relatively simple: selecting or not a tag in the variant will enable or not that crosscutting behavior. This could also be achieved with AOP and FOP. As explained in Section 3.10.8, this is due to the often applied fine granularity capability of the conditional compilation mechanism, but it lacks other desirable properties for SPL variability implementation, such as reusability, locality, adaptability, plugability, and independent development. Section 6 highlights this in terms of case studies.

AOP and FOP offer modular implementation of crosscutting concerns. Although both support heterogeneous crosscutting, only AOP supports homogeneous crosscuts: AOP handles homogenous crosscuts with wildcard pointcuts. Additionally, AOP provides finer control of crosscutting behavior with the `cflow`-like constructs and greater access to runtime information. On the other hand, wildcard pointcuts in AOP should

be used carefully, since they may complicate composition [88]. Considering that cross-cutting is a common phenomenon in SPL variability as mentioned in Chapter 2, AOP then becomes an attractive solution. The method described in the following chapter explores this.

Chapter 4

Implementing Product Lines Adoption Strategies

In the previous chapter, we described techniques addressing variability implementation in software product lines. Although some of these are widespread practices in industry, they often fail to capture variability of crosscutting concerns [100, 101] modularly. As discussed previously in Chapter 2, such concerns occur frequently in the implementation of SPL variability and thus have to be addressed int the context of adoption strategies. This chapter shows how AOP, a paradigm for capturing crosscutting concerns, implements variability in product lines and supports such adoption strategies. The description is based on the SPL adoption strategy context and on the results of our current experience [14, 13, 2, 12, 5, 111, 7, 4, 9, 10, 8].

The remainder of this chapter is organized as follows: in Section 4.1, we define our process for implementing some SPL adoption strategies; we then show, in Section 4.2, how some elements of such process can be understood formally.

4.1 Method

Contrary to the proactive SPL adoption strategy, our method relies on a combination of the extractive and the reactive SPL adoption strategies. Our method first bootstraps the SPL and then evolves it with a reactive approach. Initially, there may be one or more independent products, which are refactored in order to expose variations to bootstrap the SPL. Next, the SPL scope is extended to encompass another product: the SPL reacts to accommodate the new variant. During this step, refactorings are performed to maintain the existing product, and a SPL extension is used to add a new variant. The SPL may react to further extension or refactoring. Alternatively, there may be an existing SPL implemented with a variability mechanism from which we may want to migrate. During such activities, the feature model as well as the configuration knowledge evolve and need to be handled.

More specifically, the state diagram in Figure 4.1 defines the steps of our method. First, **Variability Identification** identifies variation points across existing applications or within an existing SPL. Such variation points are used during either **Migrate SPL**– to change the variability mechanism employed–or **Extract SPL**, to bootstrap the SPL

from the existing applicatinos. Next, **React SPL** can be applied repeatedly either after migration or extraction to accommodate new products into the SPL. **Refactor Feature Model** then updates the incurred changes in SPL configurability at the feature model. Finally, some changes are also necessary at the configuration knowledge, which is performed by **Update Configuration Knowledge** step. The shaded steps in Figure 4.1 are the core of our method and comprise our contribution, whereas the others are templates to our method, that is, we require those steps to be performed but do not specify a specific implementation. Nevertheless, we illustrate how template steps can be performed in case studies (Chapter 6).

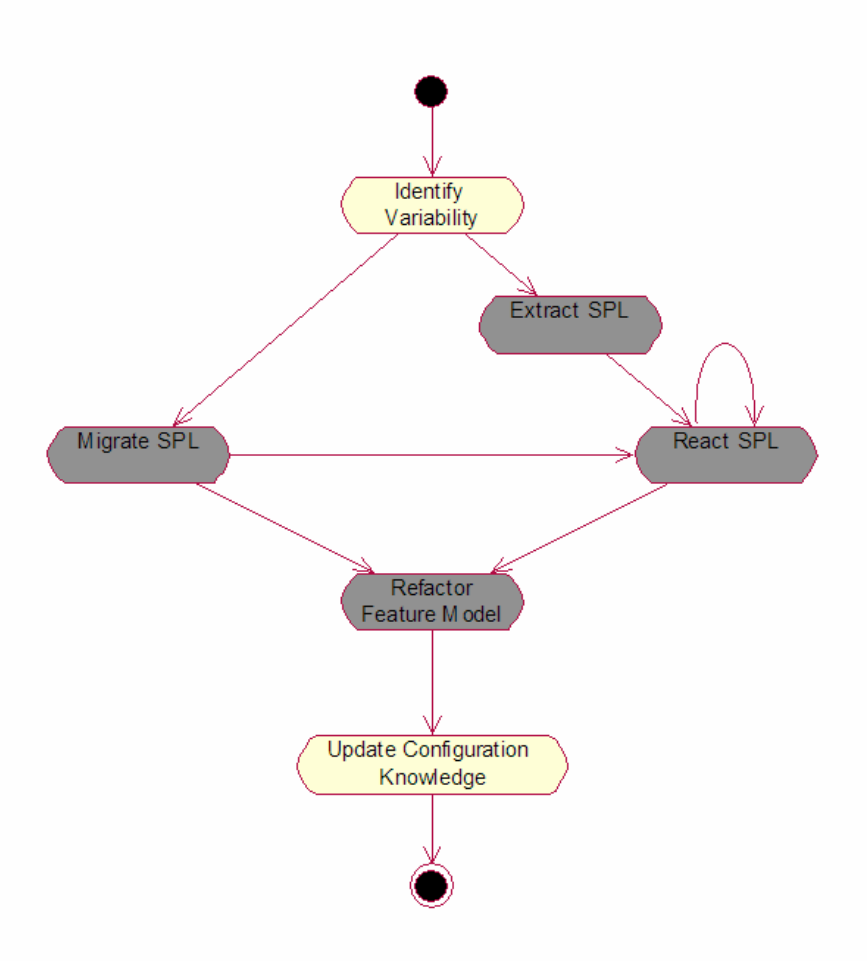


Figure 4.1: Method for implementing SPL adoption strategies.

The method is systematic because it relies on a collection of provided refactorings both at the code level and at the feature model level. Such refactorings are described in terms of templates, which are a concise and declarative way to specify program transformations. In addition, refactoring preconditions (a frequently subtle issue) are more clearly organized and not tangled with the transformation itself. Furthermore, the refactorings can be systematically derived from more elementary and simpler programming laws [40, 69] or feature model transformation laws. These laws are appropriate because

they are considerably simpler than most refactorings, involving only localized program changes, with each one focusing on a specific language construct.

In the following sections, we detail the core steps of our method, explaining the extractive and the reactive steps, and their associated refactorings in Sections 4.1.1, 4.1.2, and 4.1.3. We then present migration step in Section 4.1.4. **Refactor Feature Model** is described in Chapter 5. Finally, in Section 4.2, we explain how extractive and reactive refactorings can be understood in terms of more elementary program transformations.

4.1.1 Extract SPL

After **Variability Identification**, the following step of our method is to extract the SPL: from one or more existing product variants, strategies based on refactorings (detailed in Section 4.1.3) extract core assets and corresponding product-specific adaptation constructs. These constructs correspond to aspects and possibly supporting classes (classes only appearing in one product). Figure 4.2 depicts this approach. In this case, only one core asset is shown, but in general there could be more. Additionally, during evolution of the SPL, a product-specific asset could become a core asset, in which case it be used to derive at least two SPL members.

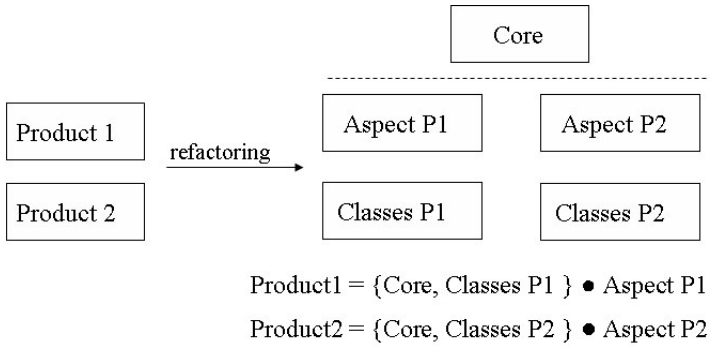


Figure 4.2: Bootstrapping the Product Line. Core assets appear above the dashed line.

Product 1 and *Product 2* are existing applications in the same domain (for example, versions of a J2ME game for two platforms). *Core* represents commonality within these applications; it is usually a partially specialized OO framework, but can also contain aspects, in which case such aspects modularize the implementation of crosscutting concerns shared by at least two SPL instances. The core is composed either with *Aspect P1* and its supporting classes (*Classes P1*), if any, or *Aspect P2* and its supporting classes (*Classes P2*), if any, in order to instantiate the original *specific* products. The • operator represents aspect composition (weaving). These aspects and their supporting classes thus encapsulate product-specific code.

Once the variability is identified, the developer should analyze the variability pattern within that concern. Depending on the pattern, a refactoring may be applied in order to extract it from the core (Section 4.1.3). Indeed, refactorings can be used to create product lines in an extractive approach, by extracting product-specific variations into aspects, which can then customize the common core [14, 13, 2].

Although this step of the method focuses on code assets, other steps describe the interaction of such assets with configurability-level artifacts, such as feature models [45]. Indeed, the method requires feature modeling and a configuration knowledge, which are essential for effectively describing the SPL variability and product derivation. Chapter 5 describes in detail the transformation at the feature model level.

The mapping between features and aspects needs to be specified by a configuration knowledge mechanism [45], which imposes constraints on features and aspect combinations like dependencies, illegal combinations, and default combinations. Constraints involving only feature combinations are also specified in the feature model. Throughout this work, we assume a general configuration knowledge mapping individual features–or sets of features–to aspects and classes: the set of features common to both products map to SPL core assets; the set of product-specific features map to product-specific aspects and supporting classes. However, our method does not bind a particular configuration knowledge scheme. In particular, Sections 6.1.4 and 6.2.4 illustrate schemes with different levels of granularity.

4.1.2 React SPL

Once the product line has been bootstrapped, it can evolve to encompass additional products. In this process, a new aspect is created to adapt the core to the new variant. Moreover, a new feature is added to the feature diagram in order to represent the new product, and the configuration knowledge is updated to map the new feature to the new aspect (Figure 4.3).

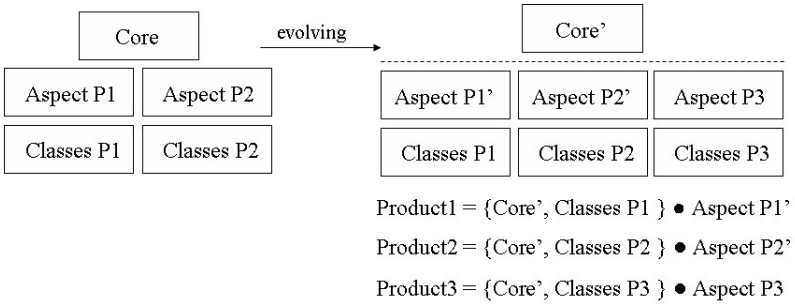


Figure 4.3: Evolving the Product Line. Core assets appear above the dashed line.

The refactorings in Table 4.1 (Section 4.1.3) can also be used for evolution. As Figure 4.3 also indicates, the core itself may evolve because some of the commonality between *Product 1* and *Product 2* might not be shared completely by *Product 3*. That is, *Product 3* has different commonality with *Product 1* and *Product 2* than these latter have with each other; therefore, a slightly different core is necessary. This may trigger further adaptation of the previously existing aspects, too. However, AspectJ tools can identify parts of the core on which these previous aspects depend, and some refactorings could also be aspect-aware according to the definition of Hanenberg et al [66]: evolving the core may change some join points within it, so the aspect-aware refactorings accordingly adjust aspects' pointcuts to refer to the new join points, thereby minimizing the need

to revisit such previous aspects. The end of Section 4.1.3 discusses how our refactorings could be extended to be aspect-aware according to this definition.

Another evolution scenario (Figure 4.4) involves restructuring the product line to explore commonality within aspects. Such commonality (`AspectFlip` aspect) then becomes a core asset, since it is now explicitly shared by at least two SPL instances.

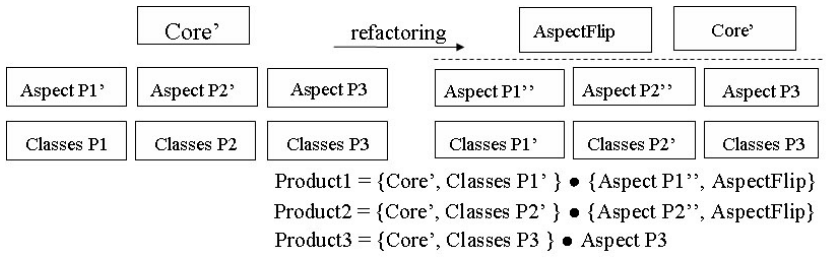


Figure 4.4: Refactoring the Product Line. Core assets appear above the dashed line.

Figure 4.4 can become more complex with the addition of new platforms and identification of reusable aspects. However, constraints in the feature model as well as the configuration knowledge (the mapping of features to aspects) limit aspect combinations, thereby providing support for scalability.

4.1.3 Refactoring Catalog

This subsection defines a refactoring catalog, which is a set of refactorings supporting the extractive and the reactive activities described in the previous subsections. We have developed this catalog empirically by analyzing variability in a number of mobile games [14, 5, 111, 9, 10, 8]. It has allowed us to address most variabilities in this domain, but we have not proved this catalog to be complete. We first specify the AspectJ subset necessary for applying these refactorings. Next, we motivate some refactorings by considering an example of feature extraction. Finally, we list the remaining refactorings. Section 6.1.3 shows some strategies (sequence of applications of refactorings from this catalog) that manage to handle the implementation of variability in the context of a industrial-strength case study.

AspectJ subset

We consider a subset of AspectJ [40]. This simplifies the definition of transformations and does not compromise our results. However, this may limit the number of refactorings we are able to derive with our laws. For example, the use of `this` to access class members is mandatory. Also, the `return` statement can appear at most once inside a method body and has to be the last command. Additionally, we consider only the following pointcut designators: `call`, `execution`, `args`, `this`, `target`, `within` and `withincode`.

Restricting the use of `this` simplifies the preconditions defined for the laws. This can be seen as a global precondition instead of a restriction to the language. Most of the laws dealing with advice require this restriction. This restriction allows an easy mapping

from the executing object referenced from `this` to the executing object exposed inside advice with the pointcut designator `this`.

We only support the mentioned pointcut designators because we think they may represent the core designators of this aspect-oriented language: they have sufficed for us capture join point in 4 different application domains in previous work [40] and in this work. Extending the set of laws to include other AspectJ constructs would be time demanding but not difficult. Besides, it would not affect the already defined laws.

An Example

In the context of the mobile device game domain, we consider the optional figures concern of a game. We examine the code declaring and using the `dragonRight` image. First, we consider class `Resources`.

```
class Resources {...
   Image dragonRight;...
   void loadImages() { ...
      dragonRight = Image.createImage("dragonRight.png");...
  } ...
}
```

where such field is not used anywhere else in the class. The developer may decide that `dragonRight` is an optional feature specific to Platform 1 (P_1) and thus could extract it into an aspect with inter-type declaration and advice constructs. We would thus have

```
class Resources { ...
   void loadImages() {...}
}

Aspect AP1 {
 Image Resources.dragonRight;
 after() returning(): execution(Resources.loadImages()) {
   dragonRight = Image.createImage("dragonRight.png");
 } ...
}
```

where `Resources` now represents a construct in the game core being built and `AP1` denotes an aspect adapting it for a specific platform, namely P_1. The fact that the field is not used anywhere else in the class allowed us to move the attribution towards the method border (end of method in this case), which allows the variation to be described by a single after advice.

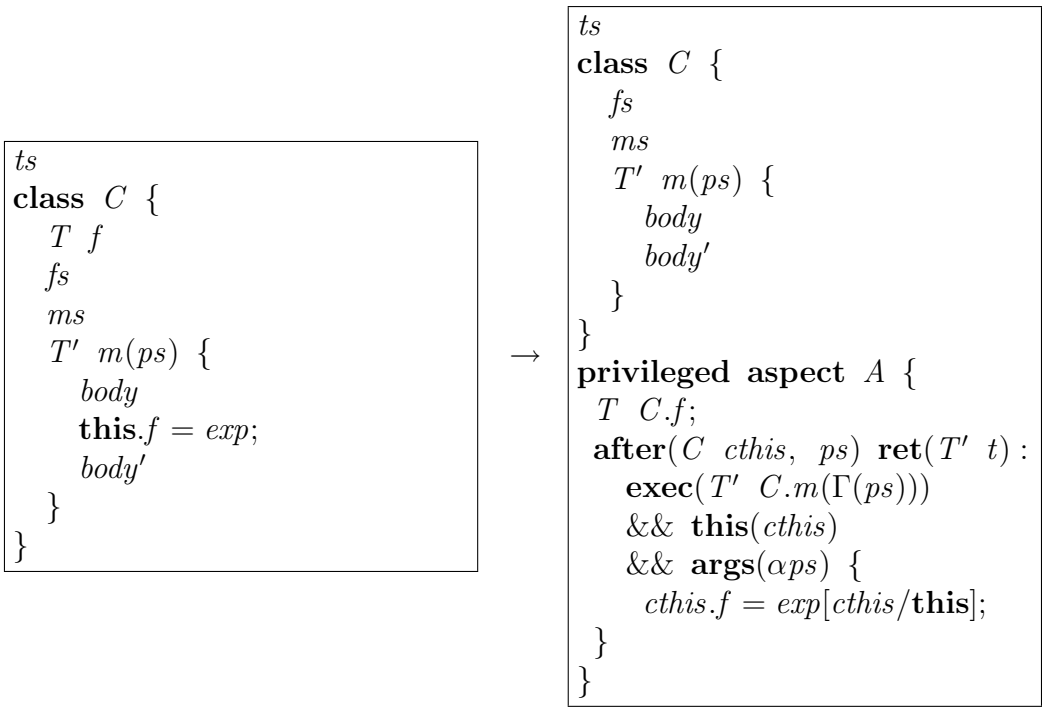
Refactorings like these occur frequently and we thus generalize them using a notation that follows the representation of programming laws [40, 69]. Refactoring *Extract Resource to Aspect - after*, whose transformation template is shown shortly ahead, generalizes this transformation and has the purpose of extracting a single variant field, along with part of its usage, into an aspect.

On the left-hand side of Refactoring 1's transformation template, the `f` field and the `this.f=exp;` command (*exp* is an arbitrary expression) denote the variability pattern to be extracted. On the right-hand side, such variability is extracted into aspect `A`. Aspect `A` uses an inter-type declaration construct to introduce field `f` of type T (in the transformation template, T also encompasses the access modifier) into class `C` and an advice construct to add the extracted command to method `m`.

In the following, we denote the set of type declarations (classes and aspects) by *ts*. Also, *fs* and *ms* denote field declarations and method declarations, respectively. $\sigma(C.m)$ is used to denote the signature of method m of class C, including its return type and the list of formal parameters. $\Gamma(ps)$ denotes the type list from parameters ps, and αps denotes the parameter names from ps. For brevity, we write `exec` and `ret` instead of `execution` and `returning`, respectively.

Each refactoring provides preconditions to ensure that the program is *syntactically valid* (not necessarily syntactically equivalent) and *semantically equivalent* (behavior preserving) after the transformation. The first and second preconditions are necessary to ensure that the code still compiles after applying the transformation, whereas the last three preserve behavior. In particular, although the right-hand side of the refactoring template is not *syntactically* equivalent to the left-hand side, both sides are *semantically* equivalent, since the third refactoring precondition (shown shortly ahead) guarantees that the `this.f=exp` command can be the last one or in the middle of method `m`.

Refactoring 1 ⟨Extract Resource to Aspect - after⟩



provided

- `A` does not appear in *ts*;

- if the field `f` of class `C` is private, such field does not appear in *ts* nor in *ms*;
- `f` does not appear in *body'*; `exp` does not interfere with *body'*;
- `A` has the lowest precedence on the join points involving the signature $\sigma(C.m)$;
- there is no designator `within` or `withincode` capturing join points inside `this.f=exp;`

In the preconditions above, we require that, if the field `f` of class `C` is private, such field does not appear in *ts* nor in *ms* because, when moved to the aspect, the field would be private with respect to the aspect and not with the class, hence a reference to `f` in *ts* or *ms* would not compile (according to AspectJ semantics, visibility modifiers of inter-type declarations are related to the aspect and not to the affected class).

The preconditions on the third bullet are necessary to allow moving the command `this.f=exp;` to the end of method `m`, which is done as an intermediate step during refactoring. Section 4.2.2 and Figure 4.7 explain the refactoring in terms of consecutive applications of elementary fine-grained transformations. The precondition requiring `exp` not to interfere with *body'* is specified at a semantic level, but it can also be specified syntactically if we have further information about the structure of *exp*, which happens frequently, including in our example above and in our case study. In such cases, *exp* is a static method call on third-party API to load image attributes, thus not interfering with *body'*.

Despite its syntactic form, the semantic intent of the lower precedence precondition is the following: the newly created *after advice* has the *lowest* precedence on the join points involving the signature $\sigma(C.m)$. Lowest precedence is required because such advice has to execute immediately after *body'*; if there are other after advice affecting the same joint point, these should execute after the former advice. According to AspectJ's semantics, one after advice executes before another after advice if, and only if, the former has lower precedence the latter, thus the former will be the first to execute among other advice if it has the lowest precedence.

However, the only way AspectJ allows specifying precedence among advice of different aspects is by specifying precedence on *aspects* containing these advice, thus implying that *all* advice of a certain aspect `A` have precedence over *all* advice of another aspect `B`, which is a too coarse-grained way to do so. In fact, we may want some advice of `A` to have precedence over some advice of `B` and some advice of `B` to have precedence over advice of `A`, which would lead to an unsolvable constraint among the precedence of such aspects.

Therefore, applying the same refactoring twice works if the code is extracted into the same aspect (advice precedence within the aspects is addressed as shown shortly ahead); otherwise, it will depend on whether there is already a precedence constraint on the existing aspects. If so, the refactoring might not be applied; otherwise, the refactoring can be applied and the new aspect `A` will have the lowest precedence. This is a limitation of AspectJ's expressiveness. An AspectJ extension could be accomplished to define advice precedence on a finer-grained approach, by using the ABC compiler [20], for example. In this case, the semantic intent of the refactoring could be expressed syntactically.

The fifth precondition means that there are no `within` or `withincode` pointcut designators in any aspect in the SPL that could match join points in the `this.f=exp;` statement. This precondition is necessary because moving such statement may break those pointcuts. Despite declarative, this precondition is verifiable by examining the SPL aspects in the IDE using AJDT's API.

The refactoring described creates aspect `A`. A slight variation of this refactoring assumes `A` already exists. In this case, such aspect would have a particular form after applying the transformation (in the following, *pcs* denotes pointcut declarations):

```
privileged aspect A {
T C.f;
pcs
bars
after(C cthis, ps) ret(T' t) :
  exec(T' C.m(Γ(ps)))
  && this(cthis)
  && args(αps) {
    cthis.f = exp[cthis/this];
}
afs
}
```

Note that, in this case, the advice cannot be considered as a set, since order of declaration dictates precedence of advice. According to the AspectJ's semantics, if two advice declared in the same aspect are `after`, the one declared later has precedence; in every other case, the advice declared first has precedence. Thus, we divide the list of advice in two. The first part (*bars*) contains the list of all `before` and `around` advice, while the second part contains only `after` advice (*afs*). This separation ensures that `after` advice constructs always appear at the end of the aspect. It also allows us to define exactly the point where the new advice should be placed to execute in the same order in both sides of the refactoring: since the new after advice appears before *afs*, it has the lowest priority among after advice constructs and thus will be the first after advice to execute, as intended.

Additionally, for advice declared in different aspects, precedence depends on their hierarchy or their order in a `declare precedence` construct (this is addressed by the fourth precondition of the refactoring). Similar considerations apply to the remaining refactorings. For brevity, we will assume the aspect is created in each case.

Remaining refactorings

Table 4.1 summarizes all refactorings from our catalog.

Some of the refactorings in Table 4.1, such as *Change Class Hierarchy*, are coarse-grained; others, such as *Extract Argument Function*, are fine-grained; some, such as *Extract Method to Aspect*, have medium granularity. Part of their names refers to an AspectJ construct that encapsulates the variation. For example, the *Extract Resource to Aspect - after* we described previously extracts the variant part of a concern, appearing as a field and its uses in the class, into AspectJ's `after` construct. Finally, the

Table 4.1: Summary of Refactorings.

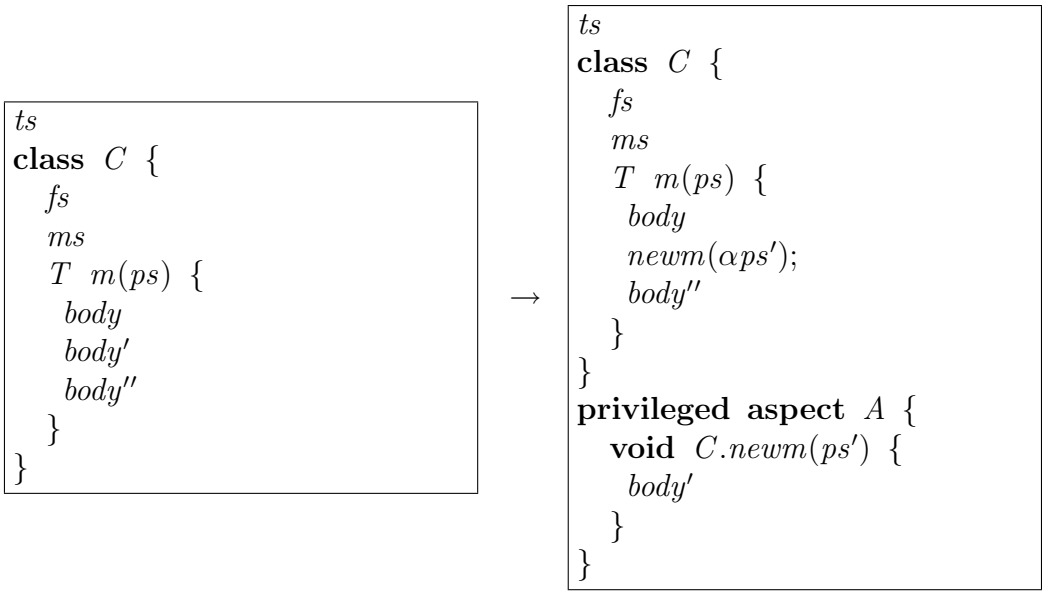
Refactoring	Name
1	Extract Resource to Aspect - after
2	Extract Method to Aspect
3	Extract Context
4	Extract Before Block
5	Extract After Block
6	Extract Argument Function
7	Change Class Hierarchy
8	Extract Aspect Commonality

refactorings we present are not aspect-aware according to the definition of Hannenberg et al [66], but these could be adapted to be so by relaxing some preconditions such as the fifth of the *Extract Resource to Aspect - after* refactoring and accordingly changing the `within` and `withincode` pointcuts involved following the guidelines presented elsewhere [66]. In a broad sense, however, our refactorings are aspect-aware, since they can be used in the presence of aspects and manipulate aspects constructs in transformation templates and preconditions.

We now describe the remaining refactorings. The *Extract Method to Aspect* refactoring is intended to extract the variant part of a concern, appearing in the middle of a method body, into AspectJ's inter-type declaration construct. Such declaration can then be implemented according to the specific variant. The refactoring structure and its preconditions are shown shortly ahead.

Accordingly, method `newm`, used in class `C`, is defined in aspect `A`. Although this could be considered lack of obliviousness [50], that is, in this case, the class *knows* about the aspect, more recent research [120, 64, 84] has shown that obliviousness is not necessary goal. In fact, achieving obliviousness might come at an expense of making aspect constructs complex. In the refactoring, the very existence of method `newm` could be interpreted as a service or contract that has be be provided by the aspect in order for the class to perform its role. Such contract is consistent with the notion of Crosscuting Interfaces (XPIs) [120, 64] and Extension Join Points (EJPs) [84].

Refactoring 2 ⟨Extract Method to Aspect⟩



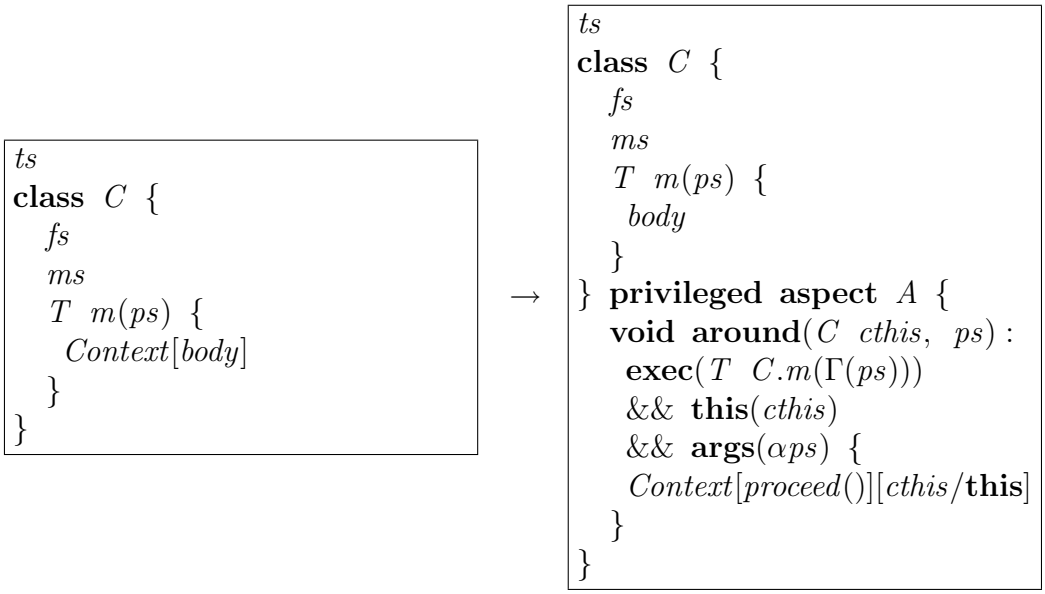
provided

- `newm` is a fresh name;
- `A` is not declared in *ts*;
- *body′* does not change any local variable;
- local variables declared in *body′* are not used in *body″*;
- `return` and `super` do not appear in *body′*;
- $ps' = ps, variables(body)$;
- there is no designator `within` or `withincode` capturing join points inside *body′*;
- there is no aspect in *ts* with an advice intercepting $\sigma(C.newm)$.

where the list of parameters of method `newm`, ps' , is composed by the parameter list of method `m`, *ps*, as well as parameters corresponding to accesses of local variables of *body* used in *body'*, denoted by *variables*(*body*). Despite the number of preconditions for this and the other refactorings presented in this section, we could still apply them in many situations, as described in the case studies conducted in Chapter 6.

The following refactoring extracts the variant part of a concern, appearing as a context over a block of code in a method body, into AspectJ's `around` construct. *Context* is also an arbitrary piece of code encapsulating *body* and, in particular, can have any nesting.

Refactoring 3 ⟨Extract Context⟩

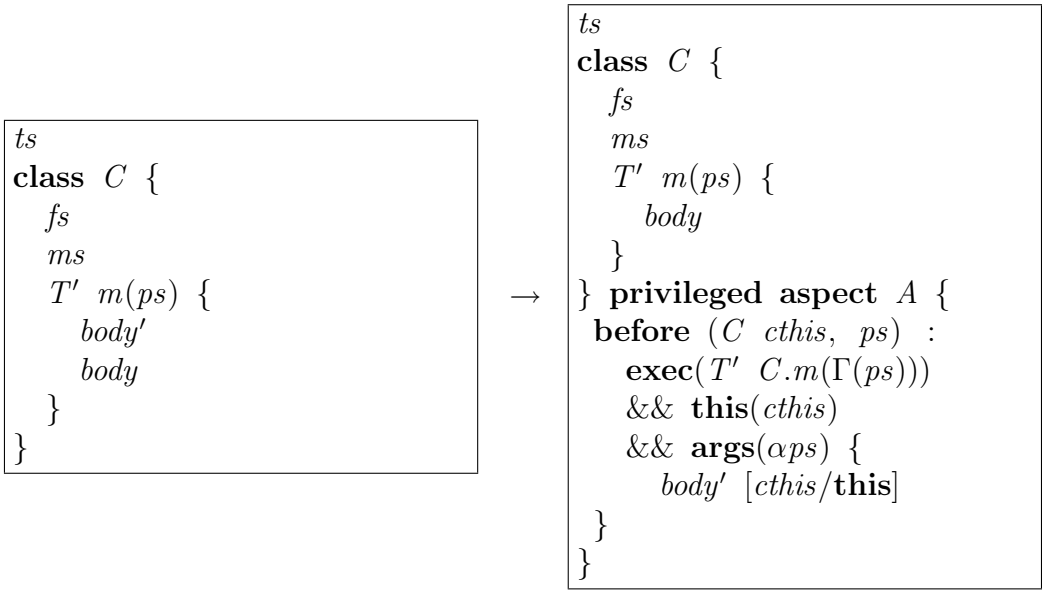


provided

- `super` and `return` do not appear in *Context*;
- *body* does not use local variables declared in *Context*;
- there is no aspect in *ts* affecting the join point $\sigma(C.m)$;
- there is no designator `within` or `withincode` capturing join points inside *Context*.

The next two refactorings extract the variant part of a concern, appearing either at the beginning or at the end of method body, into AspectJ's `before` or `after` constructs, respectively.

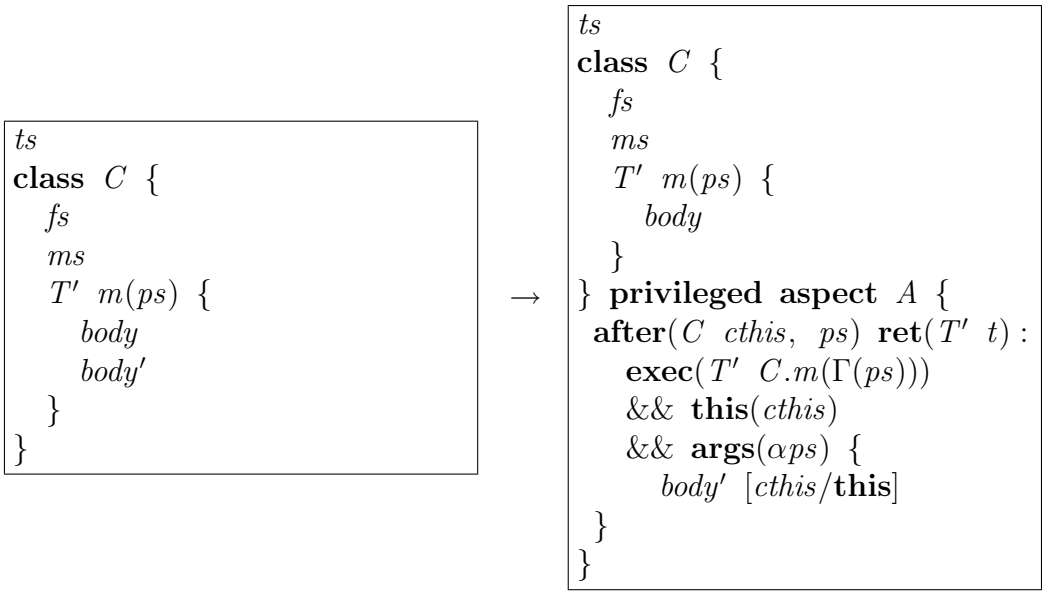
Refactoring 4 ⟨Extract Before Block⟩



provided

- *body* does not use local variables declared in *body′*;
- `super` and `return` do not appear in *body′*;
- `A` has the lowest precedence on the join points involving the signature $\sigma(C.m)$;
- there is no designator `within` or `withincode` capturing join points inside *body′*.

Refactoring 5 ⟨Extract After Block⟩

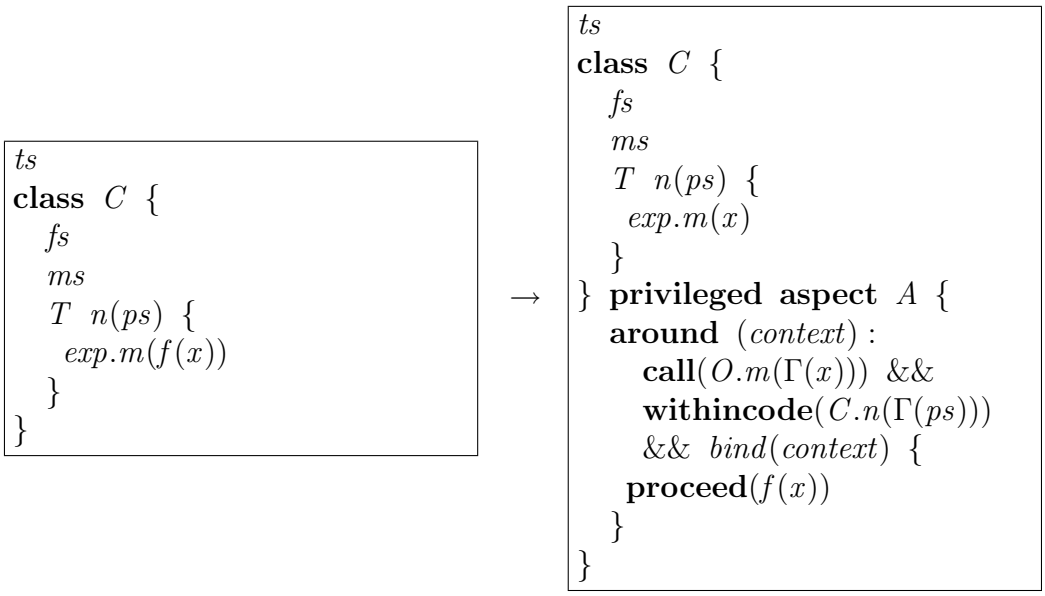


provided

- *body′* does not use local variables declared in *body*;
- `super` does not appear in *body′*;
- `A` has the lowest precedence on the join points involving the signature $\sigma(C.m)$;
- there is no designator `within` or `withincode` capturing join points inside *body′*.

Refactoring 6 extracts the variant part of a concern, appearing as a function over an expression in a method method call, into AspectJ's *around* and *proceed* constructs. The latter construct guarantees that we apply function f over expression x. In particular, if *exp* is *null* and $f(x) = x/0$, the refactoring is still valid: in this case `ArithmeticException` would be raised in both the left-hand and in the right-hand sides.

Refactoring 6 ⟨Extract Argument Function⟩



provided

- local variables and `super` do not appear in $f(x)$;
- there is no aspect in *ts* affecting the join point $\sigma(C.m)$; O is the static type of *exp*;
- there is no designator `within` or `withincode` capturing join points inside $f(x)$.

Refactoring 7 is a coarse-grained refactoring changing the type hierarchy using AspectJ's *declare parents* construct.

Refactoring 7 ⟨Class Hierarchy⟩

ts
class C' **extends** C {...}
class D **extends** C' {...}
	→	*ts*
		class C' **extends** C {...}
		class D **extends** C {...}
		privileged aspect A {
		declare parents :
		D **extends** C'
		}

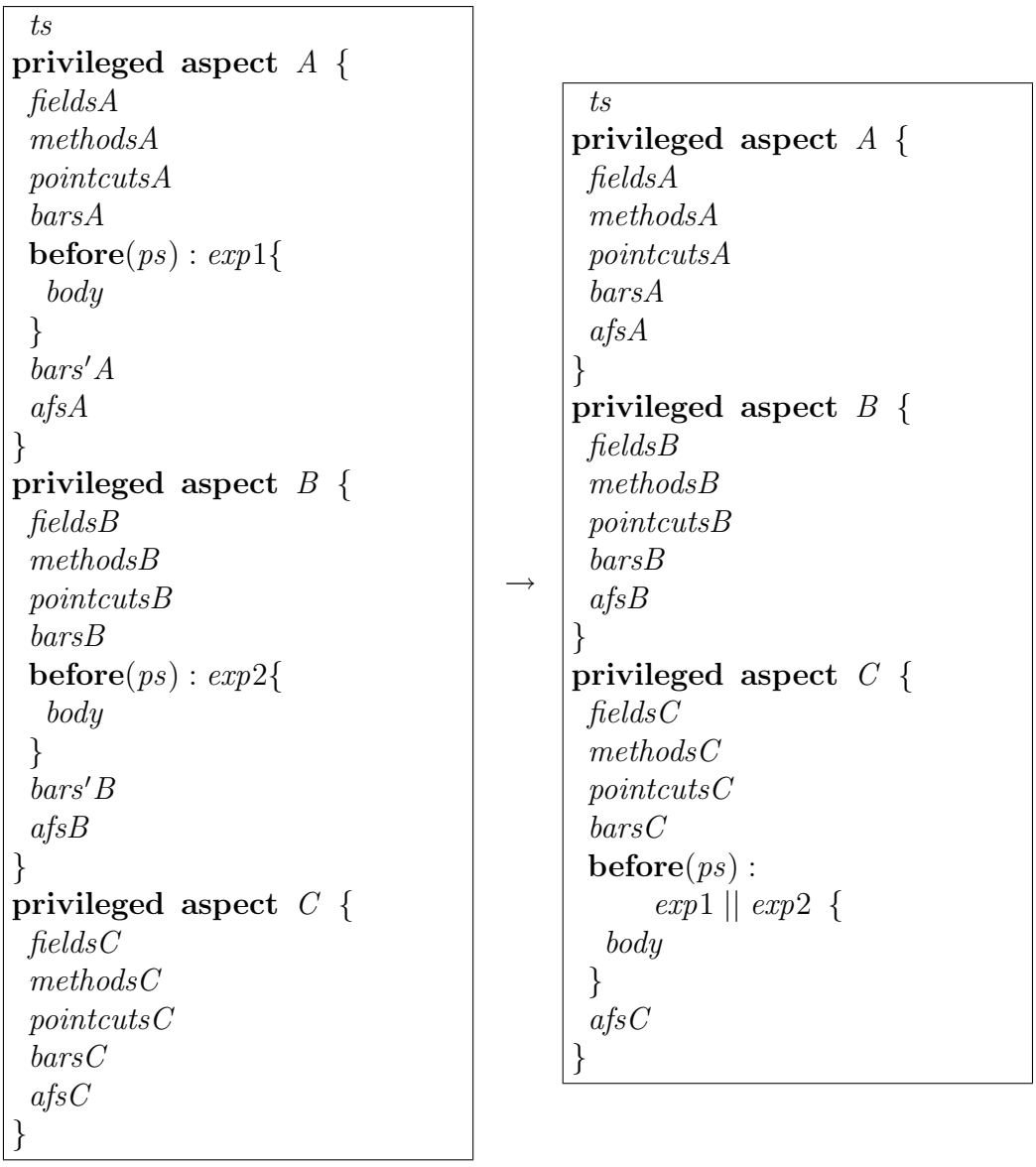
provided

- D and C' have the same interface.

In contrast to the previous refactorings, which can be used both in the extractive and in the reactive scenarios, Refactoring 8 would be used only in the reactive scenario (Figure 4.3). The transformation evolves the SPL, by reusing previously created pieces of advice. There are variations of this refactoring for the other kinds of advice constructs.

On the left-hand side template, *body* occurs in aspects `A` and `B`. The idea is to factor out such commonality into another aspect, `C`, in which the new advice, on the right-hand side template, is executed whenever the ones in aspects `A` and `B` execute. The second and third preconditions ensure that the transformation is syntactically valid; the first and fourth preconditions ensure that behavior is preserved. For example, for the first precondition, if such sets were not disjoint, then, on the left-hand side, *body* would be executed twice (for the joinpoints matched by both *exp1* and *exp2*), whereas on the right-hand side *body* would execute only once. The fourth precondition has a similar role to that of the fifth precondition of Refactoring 1.

Refactoring 8 ⟨Extract Aspect Commonality⟩



provided

- The set of join points captured by *exp*1 and *exp*2 are disjoint;
- *exp*1 and *exp*2 do not rely on locally defined pointcuts;
- *body* does not use attributes and methods of the aspects;
- there is no designator `within` or `withincode` capturing join points inside *body*.

4.1.4 Migrate SPL

Apart from the extractive and reactive adoption strategies, there may be a case when there is an existing SPL already implemented using some variability mechanisms and we would like to implement it using another variability mechanism. We refer to the process of accomplishing this as *migration strategy*, and reasons for accomplishing it include moving to a mechanism that better supports understandability, traceability, and further evolution of the SPL in the reactive scenario.

In this section, based on our experience in the mobile games domain [14, 13, 2, 12, 5, 111, 7, 4, 9, 10, 8], we present some migration strategies from one SPL implemented with conditional compilation to one using AOP. The strategies present a variability pattern handled by the first mechanism and show how it can be translated into a pattern using AOP constructs. We first present such patterns within an example and later show them in template form so that they can be reused in contexts other than the mobile games domain. Figure 4.5 illustrates the process:

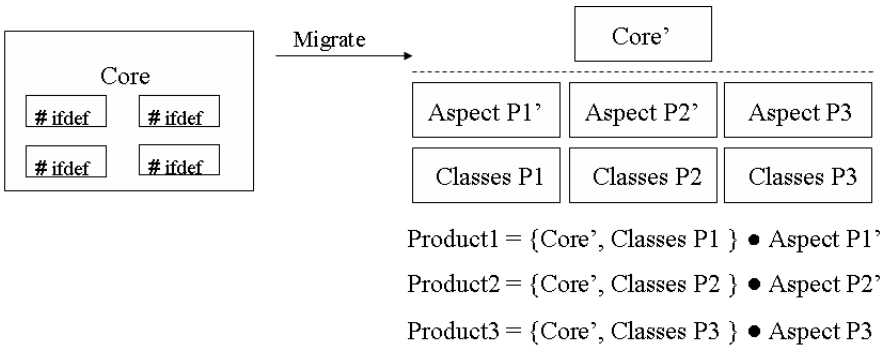


Figure 4.5: Migrate SPL.

Super Class Variation

There can be variations in the super class of some classes. These variations occur, for example, when defining a `Canvas` class that is used to draw shapes and images on the screen. For Nokia devices, it is required that these classes extend the Nokia API class `com.nokia.mid.ui.FullCanvas` instead of MIDP [108] class `javax.microedition.lcdui.Canvas`. If the device supports MIDP 2.0 or is a Siemens mobile device, `Canvas` super classes are also different, called respectively `javax.microedition2.lcdui.game.GameCanvas` and `com.siemens.mp.color_game.GameCanvas`. As a consequence, dealing with this variation requires changing an `import` declaration and the corresponding super class name in the `extends` clause. Using conditional compilation tags, it is possible to define a different import and extends declaration for each of those variations. The following piece of code shows how this variability mechanism is employed to address such variations for configurations corresponding to Nokia and MIDP devices.

```
//#ifdef nokia_device
//#  import com.nokia.mid.ui.FullCanvas;
//#else
//#  import javax.microedition.lcdui.Canvas;
//#endif
...
//#ifdef nokia_device
//#  public class MainCanvas extends FullCanvas {
//#else
//#  public class MainCanvas extends Canvas {
//#endif
...
```

Using AspectJ, such variability can be addressed by declaring an aspect for each possible super class alternative, corresponding to a different configuration. A `declare parents` clause with the required class name is defined in the aspect. Additionally, the corresponding import declaration is transferred to the aspect. The piece of code below shows the result of applying this strategy to the example just mentioned.

```
//core
public class MainCanvas {...}

//Nokia configuration
import com.nokia.mid.ui.FullCanvas;
public aspect NokiaCanvasAspect {
  declare parents: MainCanvas extends FullCanvas;
  ...
}

//MIDP configuration
import javax.microedition.lcdui.Canvas;
public aspect MIDPCanvasAspect {
   declare parents: MainCanvas extends Canvas;
   ...
}
```

The approach presented above only works because `FullCanvas` is a subclass of `Canvas`, which is a precondition of declare parents. The classes `GameCanvas` (MIDP 2.0 and Siemens) also respect this rule.

This strategy can be generalized by a pair of source and target templates specifying a transformation on code assets of the SPL. The source template is as follows:

```
//#ifdef TAG
//#  ts'
//#else
//#  ts''
//#endif
```

```
//#ifdef TAG
//#  public class C extends C' {
//#else
//#  public class C extends C'' {
//#endif
       fs
       ms
     }
```

Where `TAG` is a conditional compilation tag, whose selection in the SPL configuration binds the superclass of `C` to `C'`, including the corresponding import. When not selected in the SPL configuration, the superclass of `C` is bound to `C''`, also including its corresponding import. We denote the set of type declarations by `ts'` and `ts''`. Also, `fs` and `ms` denote field declarations and method declarations, respectively.

Code assets matching the source template are transformed according to the following target template, where aspect `A` binds the superclass of `C` to `C'`. The import required by `C'` is in `ts'` and is moved aspect `A`.

```
//core
public class C {
  fs
  ms
}

//configuration 1
ts'
public aspect A {
  declare parents: C extends C';
}

//configuration 2
ts''
public aspect B {
  declare parents: C extends C'';
}
```

Interface Implementation Variation

Another kind of variation in hierarchy that can arise is to make a class implement a different interface. It usually happens due to the use of different APIs requiring the implementation of specific interfaces. This variability issue is similar to the one presented in the previous subsection and can be handled similarly in the migration strategy. The main difference is that it uses `declare implements` instead of `declare parents`.

If Condition Variation

A common variation in mobile devices is the number and type of keys in the keypad. Additionally, the values that represent key pressing events differ between mobile devices families. This latter variability is usually implemented through blocks of constant

definitions with different values subject to conditional compilation. Other possible implementations include macro and configuration files.

When migrating to AspectJ, it is possible to introduce constants via inter-type declarations with the appropriated values. Additionally, there are variations in `if` conditions responsible for checking whether a specific key has been pressed and launch the code that handles the event. These variations usually required to add more `or-conditions` to treat the additional keys. The following code shows an example of this situation.

```
public class MainCanvas extends Canvas {
  protected void keyPressed(int keyCode) {...
    if (keyCode == LEFT_SOFT_KEY
//#ifdef device_keys_motorola
//#     ||  keyCode == -softKey
//#endif
    ) {
      // handle key event
    }
    ...
  }
}
```

The previous example shows that an additional `or-condition` can activate the code inside `if` command for Motorola mobile devices. With conditional compilation, using one or more `ifdef`'s addresses this variability issue.

We defined a migration strategy that involved: 1) the extraction of `if-condition` to a new method defined in the class containing the base condition; 2) the use of an around advice in an aspect to enhance the base condition. The result is as follows:

```
public class MainCanvas extends Canvas {
  protected void keyPressed(int keyCode) {...
    if(compareEquals(keyCode, softkey)) {
       // handle key event
    }
  }
  private boolean compareEquals(int keyCode,
                                int softKey) {
    return keyCode == softKey;
  }...
}

//Motorola device configuration
public privileged aspect DeviceKeysMotorola {
  boolean around(int keyCode, int softKey) :
    execution(private boolean MainCanvas.compareEquals(..))
    && args(keyCode, softKey)
  {
    return keyCode == softKey || keyCode == -softKey;
  }
```

```
  ...
}
```

The source template of the migration strategy is shown next:

```
ts
public class C {
 fs
 ms
 T m(ps) {
   body
   if (cond
//#ifdef TAG
//#    op cond'
//#endif
    ) {
      body'
    }
    body''
  }
}
```

where `cond` represents the base condition and the variation is an additional expression `op cond'`. The expression `op` represents binary operators and `cond'`, any boolean expression. Also, `body`, `body'`, and `body''` denote blocks of statements in a method. The target template of this strategy is presented next:

```
ts
public class C
 fs
 ms
 T m(ps) {
   body
   if (getCond(ps')) {
     body'
   }
   body''
 }
 boolean getCond(ps') {
   return cond;
 }
}

//SPL configuration handling variability issue
public aspect A {...
  boolean around(ps') :
    execution(boolean C.getCond(..))
    && args(ps')
  {
```

```
    return cond op cond';
  }
}
```

It is important to notice that using an `around-advice` allows substituting or complementing the original condition specified in the `if` statement, by executing or not a `proceed` statement.

Feature Dependency

This section presents the strategy employed to migrate a feature depending on others features. For example, there can be a feature called Arena, that allows posting game results to a public server for ranking purposes. This feature also presents results on the device screen. Since screen size is variable across devices, it would be necessary to develop an Arena feature to each appropriated screen size. Using conditional compilation, this feature implementation is spread in many classes and tangled with other functionalities.

In the following code, if the tag `feature_arena_enabled` is enabled during SPL instantiation, some common constants to paint the scroll bar are defined, but the constants `ARENA_SCROLL_HEIGHT` and `ARENA_SCROLL_POS_Y` have different values depending on the device's screen size.

```
public class MainScreen {
//#if feature_arena_enabled
   /** Constants to paint the scroll bar */
    //#if device_screen_128x128
    //#  public static final int ARENA_SCROLL_HEIGHT = 92;
    //#  public static final int ARENA_SCROLL_POS_Y = 17;
    //#elif device_screen_128x117
    //#  public static final int ARENA_SCROLL_HEIGHT = 81;
    //#  public static final int ARENA_SCROLL_POS_Y = 16;
    //#endif
//#endif
...
}
```

The strategy adopted to implement this feature dependency was to define an aspect called `ArenaAspect` to handle the core of the feature and, for each screen size variation inside Arena, define others aspects, `ArenaScreen128x128` and `ArenaScreen128x117`. Additionally, there is the following constraint on the SPL configuration knowledge: when the optional feature Arena is enabled, one of the aspects `ArenaScreenWxH` is automatically selected depending on the screen size of the device. The piece of code below shows the result of applying this strategy to the class `MainScreen` mentioned previously.

```
public class MainScreen {... }

public aspect ArenaAspect {
```

```
   /** Constants to paint the scroll bar */
}

public aspect ArenaScreen128x128 {
  public static final int
      MainScreen.ARENA_SCROLL_HEIGHT = 92;
  public static final int
      MainScreen.ARENA_SCROLL_POS_Y = 17;
}

public aspect ArenaScreen128x117
  public static final int
      MainScreen.ARENA_SCROLL_HEIGHT = 81;
  public static final int
      MainScreen.ARENA_SCROLL_POS_Y = 16;
}
```

The template generalizing this migration strategy is presented next. It is important to notice that `TAG_A` represents an optional feature and tags `TAG_B1` and `TAG_B2` represent features depending on `TAG_A`.

```
public class C {
  fs
  ms
 ...
//#if TAG_A
//#  fs'
//#  ms'
   //#if TAG_B1
   //#  fs''
   //#  ms''
   //#elif TAG_B2
   //#  fs'''
   //#  ms'''
   //#endif
//#endif
}
```

The target template of this strategy is presented next, where `C.fs'`, `C.fs''` and `C.fs'''` are the sets of fields introduced via inter-type declaration into class `C` by the aspects composed with `C`. The same pattern is used for methods, but they are named `C.ms'`, `C.ms''` and `C.ms'''` instead. Aspect `A` is included in the SPL instance if, and only if, feature `A` is selected; aspects `AB1` and `AB2` are present in the SPL instance if, and only if, their corresponding features are present and feature `A` is also selected.

```
public class C {
  fs
  ms
}
```

```
public aspect A {
  C.fs'
  C.ms'
}
public aspect AB1 {
  C.fs''
  C.ms''
}
public aspect AB2 {
  C.fs'''
  C.ms'''
}
```

Variability in Constant Declaration

A considerably frequent variability is the declaration of class constants referring to screen elements. For example, the following code snippet shows the declaration of different values for the same constant depending on the device:

```
/** Width used to show loading message */
//# if device_screen_128x117
   //# public static final int LOADING_MESSAGE_AREA = 118;
//# elifdef device_screen_176x205
   //# public static final int LOADING_MESSAGE_AREA = 154;
//# endif
```

To handle this, we declare the constants in a interface and use a `declare parents` construct to state that such interface should be implemented by the class. In this way, behavior is preserved, since a reference to a constant such as `LOADING_MESSAGE_AREA` will be available through the interface:

```
public aspect Screen128X117 {
   declare parents : GameScreen implements GameScreen128x117;
   public interface GameScreen128x117 {
   /** Width used to show loading message */
    public static final int LOADING_MESSAGE_AREA = 118;
   }
}
public aspect Screen176X205 {
   declare parents : GameScreen implements GameScreen176x205;
   public interface GameScreen1176x205 {
   /** Width used to show loading message */
    public static final int LOADING_MESSAGE_AREA = 154;
   }
}
```

Variability in Method Body

Another variability pattern that can occur is finding out the language used in the game in order to set a default or a customized one:

```
public static void initDefaultLanguage() throws IOException {
//# ifdef general_multi-language
//#   [code X]
//# else
//#   [code Y]
//# endif
}
```

A related example is variant behavior when playing sound. There are three variabilities subsuming the whole method playing the sound, as shown in the following code snippet:

```
private void playSound(int soundIndex) {
//# if device_sound_api_nokia
   [code X]
//# elif device_sound_api_samsung
//#   [code Y]
//# elif device_sound_api_mmapi || device_sound_api_siemens
//#   [code Z]
//# endif
}
```

This can be handled by implementing each variant behavior as a method introduced by an inter-type declaration:

```
public privileged aspect SoundPlayerNokia {
    public void SoundEffects.playSound () {
       [code X]
    }
}
public privileged aspect SoundPlayerSamsung {
    public void SoundEffects.playSound () {
       [code Y]
    }
}
public privileged aspect SoundPlayerMMAPI {
    public void SoundEffects.playSound () {
       [code Z]
    }
}
```

Variability in Method Call

This pattern refers to variant behavior at the beginning, middle, or end of a method. For example, the following variability pattern occurs at the end of a method and refers to calling network facility for posting results after calculating player's score:

```
 void gc_computeLevelScore() {
 ...
//#if feature_arena_enabled
//# NetworkFacade.setScore(this.scr_levelTotalScore);
//# NetworkFacade.setLevel(this.gc_getCurrentLevel());
//#endif
}
```

This can be by implementing the posting of players score in an `after` advice , which then interacts with the network service.

```
public aspect ArenaAspect {
...
   after(GameScreen cthis) :
     execution(void GameScreen.gc_computeLevelScore()) && target(cthis) {
     NetworkFacade.setScore(cthis.scr_levelTotalScore);
     NetworkFacade.setLevel(cthis.gc_getCurrentLevel());
   }
}
```

Similarly, for variability occurring at the beginning of the method, we rely on `before` advice and appropriate pointcut. For variability in the middle of the method, we identify special anchor points for which to write a pointcut and write according specific behavior in an `after` advice.

Discussion

Some of the strategies presented previously could benefit from general OO techniques (e.g. using abstract methods and subclassing, patterns and so forth), but this would imply having a subclass for each possible device, thus leading to complex class hierarchies. Additionally, many more classes would be involved, thus incurring into a penalty in terms of bytecode size, a critical issue in the mobile application domain.

The strategies replace the scattered `ifdef`s by a number of aspects, which have to be managed. This can addressed by a configuration knowledge, relating device configurations to configurations involving sets of aspects and core classes. Section 6.2.4 shows this in the context of a case study. The AO advantage lies in the fact that the extracted variability can be used elsewhere without replicating code, whereas the ifdef variability can only be used in that context.

Although some variabilities addressed are very fine-grained, they are crosscutting, because they can be logically grouped together with other fine-grained variability affecting other join points, such that this unit,the aspect,implements a feature. More generally, we could further cluster crosscutting variability so that it can be more broad in a module-classes and aspects–implementing a given feature. Finally Table 4.2 relates the migration strategies and the refactorings from the refactoring catalog of Section 4.1.3.

Table 4.2: Relating Migration Strategies to Refactorings

Migration Strategy	Refactoring
Super class variation	Change class hierarchy
Interface implementation variation	Change class hierarchy
If condition variation	Extract context
Feature dependency	Extract resource to aspect
Variability in constant declaration	Extract resource to aspect
Variability in method body	Extract method to Aspect
Variability in method call	Extract Before/After Block

4.2 Formal Reasoning for AspectJ Refactorings

This section analyzes how the extractive and reactive refactorings from Section 4.1.3 can be decomposed into or derived from existing elementary programming laws [40], which are simpler and easier to reason about than the refactorings, thereby increasing correctness confidence in such extractive transformations.

Although the derivation presented in this section is not a novel technical achievement by itself, since previous work by Cole and Borba [40] proposed a catalog of programming laws and showed derivations of other refactorings, the derivation we present here, in the context of SPL, is relevant. First, such previous work did not explore the derivation of SPL refactorings. Second, the derivation presented here is important for establishing soundness of the SPL refactorings we presented previously. Not only is this relevant for tool developers–who indeed have to carry out the derivations–but also for SPL developers–who use the tool. Indeed, even though SPL developers do not need to understand the derivation of the refactorings, they should be interested in the very fact that they do exist, since that can be interpreted as a quality attribute (or certification stamp) of the tool they use: the more the tool uses derivation, the more reliable the tool is and the less effort SPL developers spend in testing, which is extremely expensive in the SPL scenario [106].

Section 4.2.1 reviews some existing fine-grained aspect-oriented programming laws [40]. Then, in Section 4.2.2, we relate such refactorings and laws by showing how the former can be described in terms in of the latter.

4.2.1 Programming Laws

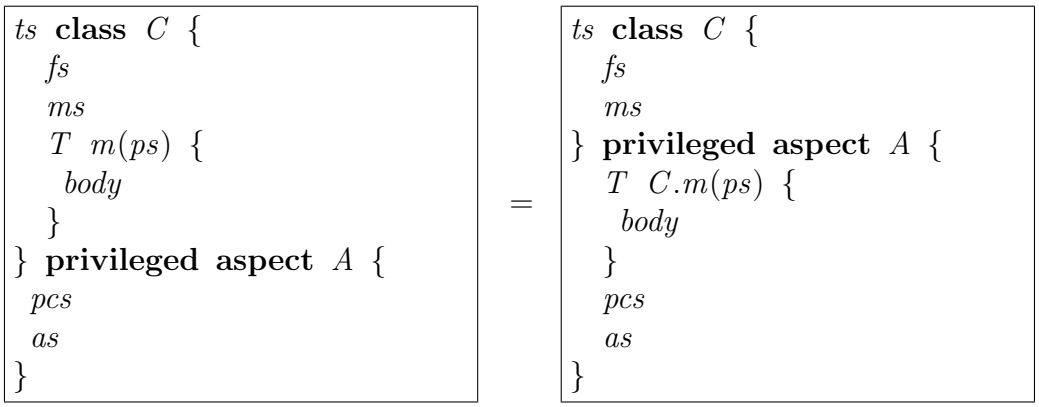
Programming laws [69, 40], like refactorings, are transformation structures which preserve program consistence and behavior. In contrast, they are much simpler than most refactorings: they involve only localized program changes, and each one focuses on a specific language construct.

Differently from refactorings, laws can be applied not only from the left to right side, but also in the opposite direction. Therefore, there are different preconditions depending on the direction the law is used. This is represented by arrows, where the symbol ($\leftarrow$) indicates that a precondition must hold when applying the law from right to left. Similarly, the symbol ($\rightarrow$) indicates that a precondition must hold when applying the law from left to right. Finally, the symbol ($\leftrightarrow$) indicates that a precondition must

hold in both directions.

For example, the Law 1 has the purpose of moving the implementation of a single method into an aspect using an inter-type declaration. This simple transformation is easier to understand and reason about. According to AspectJ semantics, visibility modifiers of inter-type declarations are related to the aspect and not to the affected class. Hence, it is possible to declare a private field as a class member and as an inter-type declaration at the same time and using the same name. As a consequence, transforming a member method that uses this field into an inter-type declaration implies that the method now uses the aspect inter-typed field. This leads to a change in behavior. A precondition is necessary to avoid this problem.

Law 1 ⟨Move Method to Aspect⟩

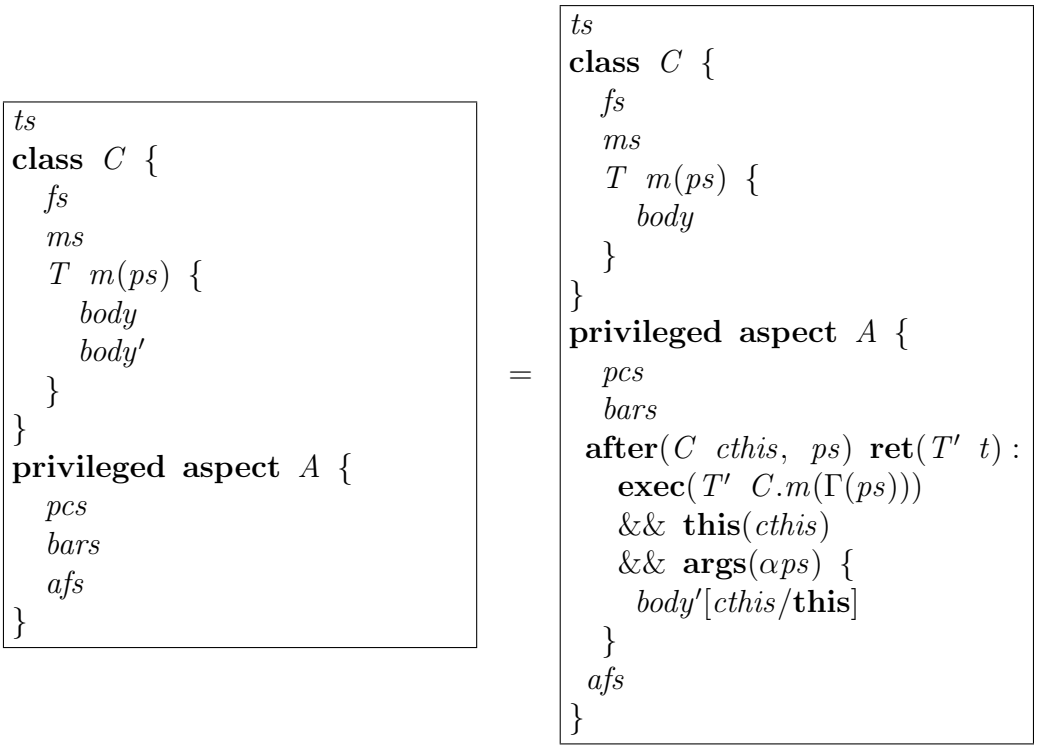


provided

(↔) `A` does not introduce any field to C with the same name of a C field used in *body*.

For example, the Law 2 has the purpose of adding an after advice. On the left-hand side of the law, *body*′ is the last block of code to execute in method `m`. Thus, we can extract it to an after advice. On the right-hand side, *body*′ is not present in method `m`, although it is executed after the execution of method `m` by an after advice declared in aspect `A`. In this aspect, the symbols used in the advice construct have the same meaning as in Refactoring 1.

Law 2 ⟨Add After-Execution Returning Successfully⟩



provided

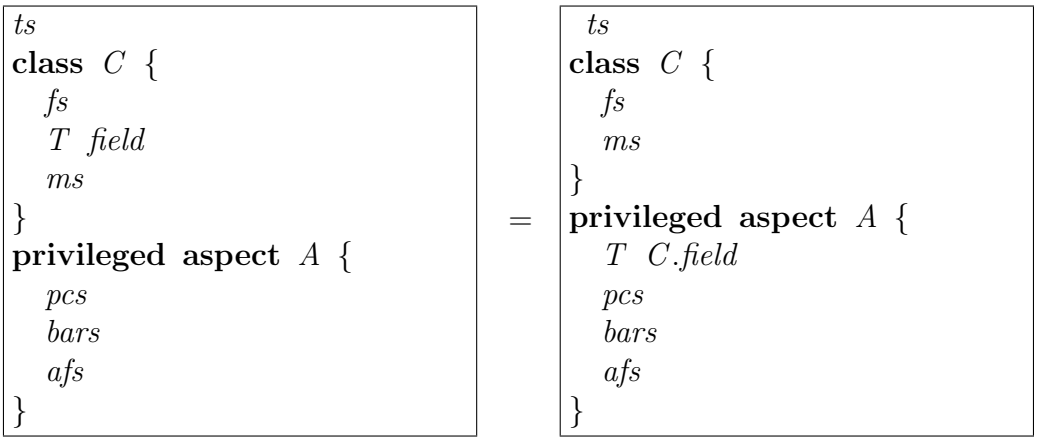
- (→) *body′* does not use local variables declared in *body*; *body′* does not call `super`;
- (↔) `A` has the lowest precedence on the join points involving the signature $\sigma(C.m)$;
- (↔) there is no designator `within` or `withincode` capturing join points inside *body′*;

The lowest precedence precondition of this law is analogous to the lowest precedence precondition of Refactoring 1, which was discussed in Section 4.1.3. Likewise, the last precondition of this law corresponds to the fifth precondition of Refactoring 1. Therefore, the constraint refers to any aspect.

Examining the left-hand side of this refactoring, we see that *body′* executes before all after advice possibly declared for this join point. This means that the new advice on the right-hand side of the law should be the first one to execute, preserving the order in which the code is executed in both sides of the law. Thus, the after advice should be placed at the beginning of the after list (*afs*). Moreover, in order to ensure that the new advice created with this law is the first one to execute, we have a precondition stating that aspect `A` has the lowest precedence over other aspects defined in *ts*. This precondition must hold in both directions.

Law 3 represents the language construct which introduces a field into a class. Analyzing this transformation from the left to the right, we can see that `field` declaration is removed from class `C`. However, we introduce `field` in this class by using an inter-type declaration construct declared in aspect `A`.

Law 3 ⟨Move Field to Aspect⟩



provided

(→) The field *field* of class C does not appear in *ts* and *ms*.

This precondition is necessary for the same reason that the second precondition of Refactoring *Extract Resource to Aspect - after* is necessary, which was explained in Section 4.1.3.

4.2.2 Deriving Refactorings

In this section we use aspect-oriented programming laws [40] to show that the refactorings previously discussed in Section 4.1.3 are behavior preserving transformations. Although we do not conduct a strictly formal proof, the derivation is still useful for understanding refactorings in terms of simpler transformations. Additionally, representing the refactorings as a composition of programming laws helps to better define the preconditions under which those refactorings are valid. For their simplicity, programming laws [69] are suitable for this. A complete formal proof requires establishing the validity of the laws with respect to a formal semantics, which is still on going work [41].

The laws we use (defined elsewhere [40]) consider the entire context, and therefore apply to *closed* programs. Nevertheless, their associated side conditions are purely syntactic. Furthermore, although the context is captured for each particular law application, this is by no means a requirement that the context be fixed for successive transformations. If, eventually, a modified context no longer satisfies the conditions of a law previously applied, this does not invalidate the effected transformation; it just means that in the current context the application of the law would not be valid. Accordingly, the laws compose in the sense that their consecutive application is equivalent to a coarse-grained transformation (refactoring). Indeed, such composition is not as flexible as in Hoare's laws [69]–which can be applied to *open* programs–, but has sufficed to derive the refactorings.

We can derive the *Extract Method to Aspect* refactoring (Section 4.1.3) from the *Move Method to Aspect* law (Section 4.2.1). This law is the only aspect-oriented transformation necessary to accomplish this refactoring. Nevertheless, we first need to apply

an object-oriented refactoring: *Extract Method* [53]. This refactoring creates a new method in class `C` called `newm` with proper parameters and return type, which executes the piece of code labelled as *body'*. *Extract Method* can only be applied if the extracted code does not change more than one local variable, otherwise the extracted method would need multiple return values. The object-oriented refactorings can be proven to be sound using object-oriented programming laws [31].

Note that the scenario after the method extraction matches the left side of *Move Method to Aspect* law, with `newm` corresponding to `m` in the law. If the target aspect already exists, we can apply this law to end the transformation. Otherwise, it would be necessary to use *Add empty aspect* and *Make aspect privileged* laws [40] to create a new aspect and make it privileged. At this point, we complete the derivation of *Extract Method to Aspect* refactoring. The sequence of steps necessary to accomplish this refactoring is shown in Figure 4.6.

Extract Method → Move Method to Aspect

Figure 4.6: Extract Method to Aspect.

As another example, Refactoring 7 (*Extract Resource to Aspect - after*), presented in Section 4.1.3, can be represented as a sequence of object-oriented transformations and aspect-oriented programming laws (Figure 4.7). In this case, starting from the left-hand side template of this refactoring, we first need to rearrange the source code manipulating field `f` because AspectJ does not provide any mechanism to introduce crosscutting behavior in the middle of a method. In order to move the crosscutting code to an aspect, we first need to move such code to the beginning or end of the method; this allows the creation of a `before` or `after` advice, respectively. In this refactoring, the crosscutting code was moved to end of the method (we name such transformation by OO law in Figure 4.7). The OO law holds if the code to be moved is independent of the remaining method code, which is guaranteed by the third precondition of the Refactoring 7 (Section 4.1.3). Once the crosscutting code is at the end of the method, we can use Law 2 (*Add after-execution returning successfully*), mentioned in Section 4.2.1, to create a new advice that is triggered after the method's successful execution. At this point, Law 3 (*Move Field to Aspect*) can be applied to extract field `f` into the aspect. The summary of transformations necessary to accomplish this refactoring is shown in Figure 4.7.

The remaining refactorings can be similarly derived from programming laws. In Table 4.3, each row summarizes the *derivation* of a refactoring whose name is on the first column (this matches the refactorings from Table 4.1) in terms of the consecutive applications of aspect-oriented laws (defined elsewhere [40]) in the second column. In the table, consecutive application of laws is represented by →, and repeated application of the same law is denoted with a superscript *.

We notice that refactorings can have different levels of complexity when compared to laws. Some refactorings, like *Extract Aspect Commonality*, can be considerably coarse-grained, representing a combination of some laws. On the other hand, some refactorings, like *Extract Before Block*, can be mapped directly into a single law.

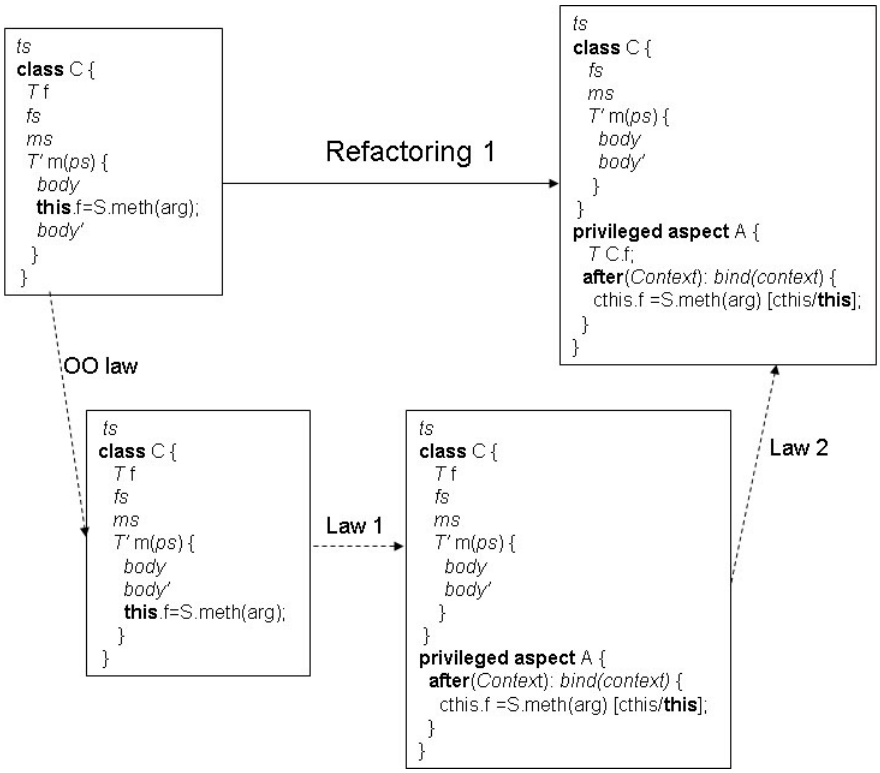


Figure 4.7: Derivation of Refactoring *Extract Resource to Aspect - after*. The dashed lines denote application of programming laws (fine-grained transformations); the continuous line denote the application of the refactoring (coarse-grained transformation)

Table 4.3: Summary of Refactorings Derivations. Consecutive application of laws is represented by →. Repeated application of a law is denoted with a superscript *.

Refactoring	**Derivation of refactoring in terms of laws**
Extract Method to Aspect	Extract Method → Move Method to Aspect
Extract Resource to Aspect - after	OO Refactoring → Add After-Execution Returning Successfully → Move Field to Aspect
Extract Context	Add Around-Execution
Extract Before Block	Add Before-Execution
Extract After Block	Add After-Execution Returning Successfully
Extract Argument Function	Add Around-Call
Class Hierarchy	Extend From Super Type
Extract Aspect Commonality	Change Advice Order → Move Advice* → Merge Advice

Chapter 5

Refactoring Feature Models

As discussed in Chapter 2, adoption strategies for Software Product Lines (SPL) frequently involve bootstrapping existing products into a SPL and extending an existing SPL to encompass another product. One way to do that is to use program refactorings. However, the traditional notion of refactoring does not handle appropriately feature models (FM), nor transformations involving multiple instances of the same SPL. For instance, it is not desirable to apply a refactoring into a SPL and reduce its configurability. At the same time, the method described in Chapter 4 relies on an activity to refactor the feature model. In this chapter, based in our previous work [11], we extend the traditional notion of refactoring to an SPL context. Besides refactoring programs, FMs must also be refactored. We present a set of sound refactorings for FMs. We evaluate this extended refactoring definition for SPL in a real case study in the mobile games domain.

The remainder of this chapter is organized as follows. Section 5.1 further motivates the need for an extended notion of refactoring, where feature models are also considered. Next, Section 5.2 extends the notion of refactoring to the SPL context. Section 5.3 then formalizes feature models, which is necessary for subsequent description of feature model refactoring in Section 5.4. Finally, Section 5.5 illustrates an strategy for employing these concepts in the context of a case study in the mobile games domain.

5.1 Motivation

The method for implementing SPL adoption strategies described in Chapeter 4 includes a combination of bootstrapping existing products into a SPL (extractive approach) and extending an existing SPL to encompass another product (reactive approach). Additionally, it requires a feature modeling refactoring step. Although the extractive and reactive approaches can be enacted by the application of *program refactorings*, the traditional definition of program refactoring [99, 53] does not take into account intrinsic characteristics of SPL: feature models and configuration knowledge [45] mapping instances of the feature model (FM) to classes and aspects in the solution space. For instance, refactoring of a SPL may have the undesirable effect of reducing its configurability. Another problem is that the traditional notion of refactoring applies only to a single product rather than to a SPL, thereby not taking into account transformations

involving more than one product. Therefore, the standard definition of refactoring needs to be extended for SPLs, taking into account their specific characteristics.

In this chapter, we extend the traditional notion of program refactorings for SPLs in such a way that, in addition to regular program refactoring, FMs are also refactored, thus completing the definition of the method described in Chapter 4. In order to achieve this goal, we propose a set of sound feature model refactorings. A FM transformation is a refactoring when the resulting FM improves (maintains or increases) the set of all possible configurations (products) of the initial FM. So, a SPL refactoring not only improves code structure, but also the quality of the FM by maintaining or increasing the SPL configurability in extractive or reactive scenarios, respectively. The main contributions of this chapter are the following:

- A new extractive program refactoring for software product lines (Section 5.2);
- A refactoring notion relating multiple feature models (Section 5.4);
- A catalog of sound feature model refactorings (Section 5.4);
- An approach for verifying soundness of software product lines refactorings (Section 5.5).

We evaluate this extended refactoring definition in extracting a SPL from legacy code in the mobile games domain. In this way, developers not only employ traditional refactorings (accordingly compiling and testing to verify whether type safety and behavior are preserved), but also use the sound catalog of FM refactorings we propose so as to guarantee configurability improvement.

5.2 Refactoring Product Lines

In this section, we explain issues that need to be addressed when considering refactoring in the SPL context (Section 5.2.1). We then propose an extended definition of refactoring for such context (Section 5.2.2). Finally, we present a new extractive program refactoring involving multiple programs.

5.2.1 Issues in Product Line Refactoring

The term refactoring was coined by Opdyke in his thesis [99]. He proposed refactorings as behavior-preserving program transformations in order to support the iterative design of object-oriented application frameworks [99]. The cornerstone of his definition is that refactorings must maintain correct compilation and observable behavior. In practice, behavior preservation is guaranteed by successive compilation and tests. Opdyke's work and many of the later refactorings apply to frameworks (a technology heavily used today in SPL development) and often introduce variation points. Nevertheless, as *program transformations*, they do not handle configurability-level issues (better addressed at the FM level), nor do they define extractive transformations from two or more existing applications into a SPL.

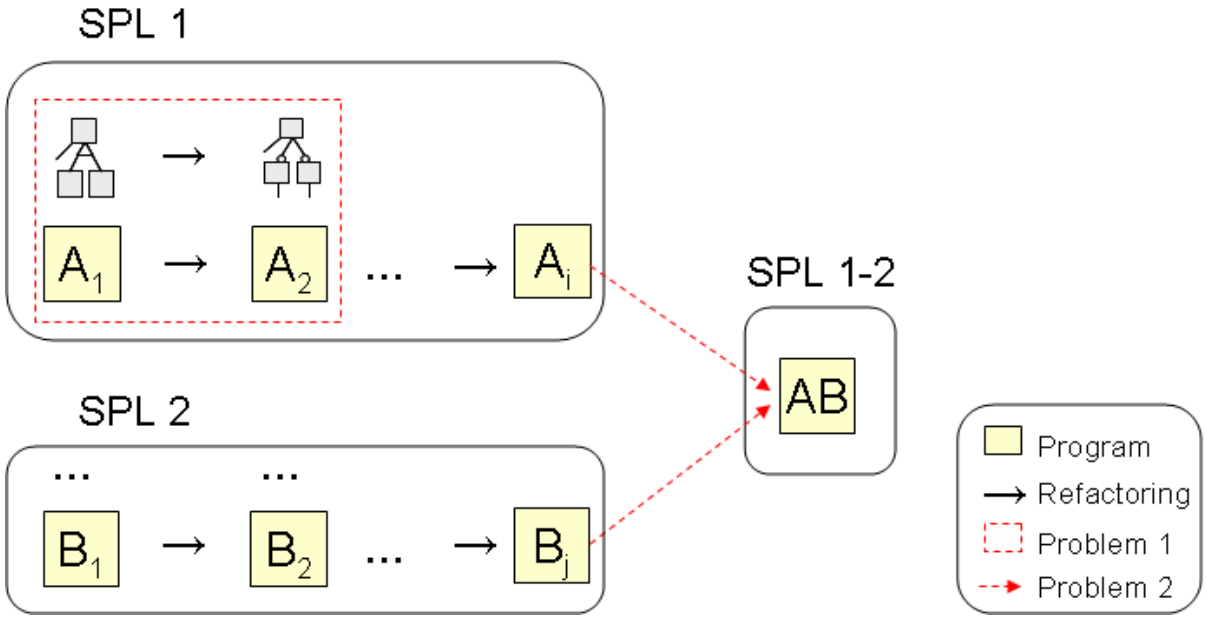


Figure 5.1: Problems in Refactoring SPLs

Figure 5.1 describes a scenario with two SPLs that are merged into a SPL, where A and B are programs from $SPL1$ and $SPL2$, respectively. In order to accomplish this, refactorings are employed. The $\rightarrow$ arrow represents a refactoring. Figure 5.1 shows that $SPL1$ and $SPL2$ are refactored to add or expose a set of optional features, as can be seen in their respective FMs (we deliberately omit the FMs of $SPL2$ and $SPL1$-2). Finally, both SPLs are extracted into $SPL1$-2, which addresses all the products configurability. Relying on the standard definition of refactoring, we notice two main issues:

- The definition of refactoring needs to be extended for SPL's context, encompassing configurability improvement by dealing with FMs (Problem 1);
- We need more program refactorings merging multiple programs into one program (Problem 2).

As aforementioned, in practice, a decrement in SPL configurability while refactoring is undesired. However, the traditional notion of refactoring does not take configurability into account, as we can see in Figure 5.1. If program (source code) A_1 is correctly refactored into A_2, following traditional refactoring steps, still it is not guaranteed that configurability is improved. We must certify that FM_2 (corresponding to program A_2) improves the possible configurations of FM_1. Since the configurability is described by a FM, such model should also be considered during SPL refactoring. So, we need to extend the traditional definition in order to apply FM refactorings (Problem 1).

In order to check configurability improvement, we may rely directly on the semantics of FM to analyze whether the final FM encompasses all the configurations of the initial one. Nevertheless, this may be time-consuming, error-prone, and costly, since analyzing semantics of models may become exponentially hard for large FM models potentially annotated with logical constraints. In order to solve this problem, we propose a catalog

of sound FM refactorings that improve configurability and would thus help the developer to evolve FMs (Section 5.4).

Furthermore, evolving a SPL often involves adapting two or more applications and unifying them, as for extracting products into a SPL. However, this requires program refactorings merging multiple programs into a product line (Problem 2). Traditional refactorings [53] usually transform one program into another. For instance, the traditional refactoring notion is not straightforward in considering a refactoring that merges programs A_i and B_j into the AB program in Figure 5.1, improving configurability of both SPLs into the new one. In this case, specific program refactorings for SPL are required.

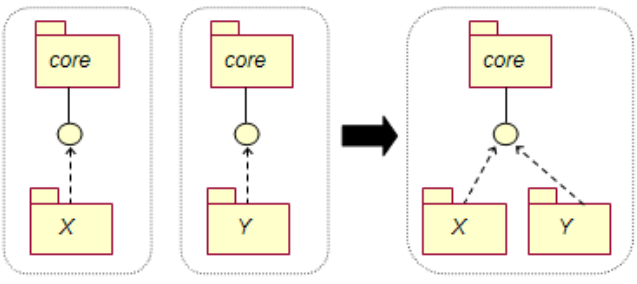
5.2.2 Definition of Product Line Refactoring

In order to deal with Problem 1, we first extend the definition of refactoring for SPLs (in addition to the refactoring catalog for FMs shown in Section 5.4) :

Definition 1 *SPL refactoring is a change made to the structure of a SPL in order to improve (maintain or increase) its configurability, make it easier to understand, and cheaper to modify without changing the observable behavior of its original products*

In order to deal with Problem 2, we propose a refactoring dealing with several programs. For instance, Refactoring SPL 1 shows an extractive refactoring, in which two existing applications are extracted into a SPL. The refactoring exposes reusable code (*Core*) among the existing applications, thereby removing code duplication. Other code artifacts (X and Y) are kept the same. Each application is now instantiated by reusing asset *Core*. Configurability is not guaranteed to be maintained; for that, feature models must be considered (using the FM refactorings shown in Section 5.4).

Refactoring SPL 1 ⟨*merge programs*⟩



The SPL refactoring definition consists of three templates: 1) two templates match the code of the existing applications (left side of the arrow); 2) the third template defines how the code of these two applications is extracted into the SPL code. In our approach illustrated in Section 5.5, Refactoring SPL 1 is used together with traditional program refactorings. In order to guarantee behavior-preservation, compilation and tests are used.

5.3 Formalizing Feature Models

In order to define feature model refactoring, we first need to formalize its semantics. This section presents a formalization latter employed in Section 5.4. As presented previously in Section 2.4.1, a FM represents the common and the variable features of concept instances and the dependencies between the variable features [45]. Each feature model describes, in a tree, a set of features and how they are related.

Relationships between a parent feature and its child features (or subfeatures) are categorized as: *Optional* (features that are optional), *Mandatory* (features that are required), *Or* (one or more must be selected - represented by a filled triangle), and *Alternative* (exactly one subfeature must be selected - represented by a unfilled triangle). Figure 5.2 depicts these relationships graphically.

Figure 5.2: Feature Diagram Notations

In order to formalize feature models, besides these relationships, we also allow feature models to include propositional logic formulas about features. For instance, the formula $B \Rightarrow \neg\, C$ states that if feature B is selected, feature C cannot be selected.

Example. Figure 5.3 depicts a FM. It has four features (A, B, C and D), one formula ($B \Rightarrow \neg\, C$) and two relationships: an option relationship between A and B, and an or relationship between A, C and D.

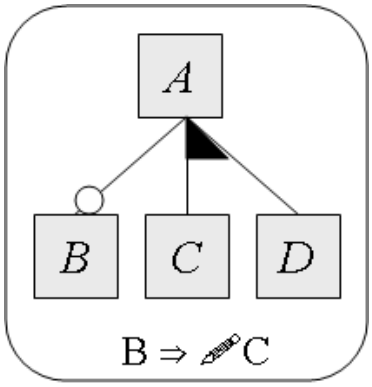


Figure 5.3: Feature Model Example

The *semantics* of a FM is the set of its possible (valid) configurations. A configuration contains a set of feature names; if *valid*, it satisfies all constraints of the model. For example, the configurations { A, B, D } and { A, C } are valid for the model in Figure 5.3. However, the configuration { A, B } is invalid because the or relationship between A, C and D states that whenever A is selected, C or D must be selected.

In order to support the definition of FM refactorings (Section 5.4), we specified a formal semantics for FMs. Next we show an UML class diagram [30] graphically describing the abstract syntax of this formalization. A feature model contains a set of feature names and a set of formulas. A configuration contains a set of names selected, as specified in Figure 5.4.

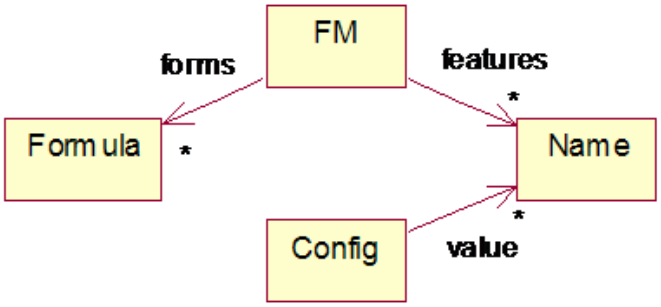


Figure 5.4: Class Diagram depicting Feature Model Components

We express all FM relationships in formulas. For example, a mandatory relationship between features A and B is represented by the formula $A \Leftrightarrow B$.

Next, we formalize the semantics of a FM, which is given by a set of configurations that satisfy all modeled constraints. The expression `values(c)` yields all selected features in the configuration `c`.

```
semantics(fm:FM): set[Config] =
  {c:Config |
    values(c) ⊆ features(fm) ∧
    ∀ f:forms(fm) | satFormula(f,c)
  }
```

The relation `satFormula` checks whether a configuration satisfies a propositional formula. For instance, considering the model in Figure 5.3, configuration c satisfies the formula $B \vee C$ if c contains B or C. As another example, a configuration satisfies the formula $B \Rightarrow \neg\, C$ if c contains B but not C.

5.4 Feature Model Refactoring

According to Section 5.2.2, SPL refactoring involves not only program refactoring, but also FM refactoring. In this section, based on the definition of SPL refactoring, we initially propose a corresponding definition of FM refactoring; next, we define a catalog of such refactorings in Sections 5.4.3 and 5.4.4. Finally, we discuss additional aspects of such refactorings (Section 5.4.5).

We define FM refactorings as follows:

Definition 2 *A feature model refactoring is a transformation that improves the quality of a feature model by improving (maintaining or increasing) its configurability.*

Let `m1` and `m2` be two FMs. So, according to Definition 2, `m2` refactors `m1` if and only if all valid configurations of `m1` are valid configurations of `m2`, as formalized next.

```
refactoring(m1,m2: FM): boolean =
  semantics(m1) ⊆ semantics(m2)
```

5.4.1 Motivation

Figure 5.5 depicts two small FMs. It describes the colors of a car. In the left-hand side (LHS) FM, a car can be black or white. Suppose that we would like to refactor the LHS model to the right-hand side (RHS) model by adding a new alternative. So we can have an additional blue car in the resulting model, while still maintaining the previous configurations.

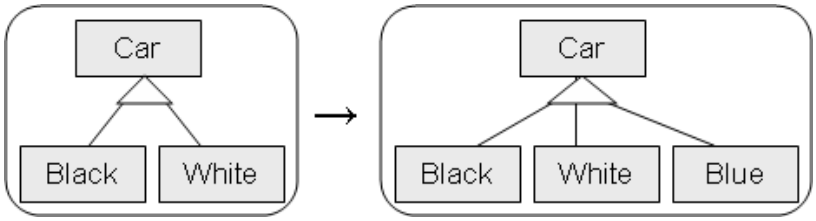


Figure 5.5: Feature Model Refactoring Example

For ensuring correctness of the refactoring depicted in Figure 5.5, we have to show that the resulting FM improves the configurability of the initial FM (Definition 2). The LHS FM has two valid configurations: { *Car*, *Black* } and { *Car*, *White* }. The RHS FM has the same configurations of the LHS FM plus the configuration { *Car*, *Blue* }. Since the RHS model contains all valid configurations of the LHS FM, it is a valid FM refactoring.

Following a similar approach to prove FM refactorings containing considerably more features, relations and formulas may be difficult, time-consuming and error-prone. In order to avoid that, we propose a catalog of sound FM refactorings (Sections 5.4.3 and 5.4.4). As discussed in Section 5.4.5, the catalog provides a more abstract alternative to ensuring correctness than directly relying on semantics. Next we give an overview of the notation used to state the refactorings.

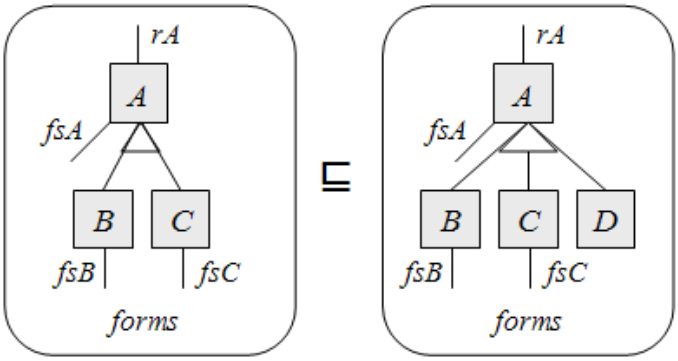
5.4.2 Refactoring Notation

Each refactoring consists of two templates (patterns) of FMs, on the left-hand (LHS) and right-hand (RHS) sides. We can apply a refactoring whenever the left template is matched by a given FM. A matching is an assignment of all variables occurring in LHS/RHS models to concrete values. Any element not mentioned in both FM templates remains unchanged, so the refactoring templates only show the differences between the FMs. Moreover, a dashed line on top of a feature indicates that this feature may have a parent feature. A dashed line below a feature indicates that this feature may have additional subfeatures.

5.4.3 Unidirectional Refactorings

Next we propose some FM refactorings in order to solve Problem 1 (Section 5.2.1). Refactoring 5 allows us to add a new node D and increase the alternative between B, C and D. This refactoring is the general version of the specific refactoring shown in Figure 5.5. We can apply Refactoring 5 to the specific models depicted in Figure 5.5 by matching the variables A, B, C and D with the specific features *Car*, *Black*, *White* and *Blue*, respectively.

Refactoring 5 *add new alternative*

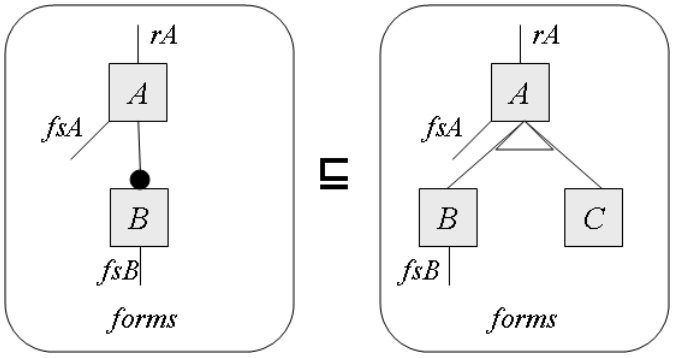


Note that there is no dashed line below D in the RHS of this refactoring because it only introduces a new feature without subfeatures. The other dashed lines of the RHS are necessary to preserve the features matched by dashed lines in the LHS.

Refactoring 5 is sound because the resulting model contains all configurations from the original one, also allowing a configuration containing A and D in the absence of B and C. Therefore, this transformation improves a model by increasing its configurability.

A slight variation of Refactoring 5 considers the case when the left-hand side template has only one direct subfeature of A that is not already matched by fsA. The right-hand side template then adds another subfeature as an alternative to this one, thus enlarging the configurability and preserving soundness:

Refactoring 5 *add new alternative (variant)*



Another general refactoring, Refactoring 2, collapses an optional feature and an or

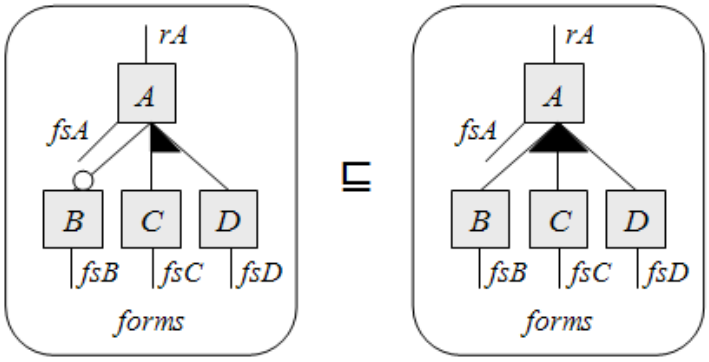
Table 5.1: Summary of Unidirectional Feature Model Refactorings

Refactoring	Name
1	Convert Alternative to Or
2	Collapse Optional and Or
3	Collapse Optional and Alternative to Or
4	Add Or Between Mandatory
5	Add New Alternative
6	Convert Or to Optional
7	Convert Mandatory to Optional
8	Convert Alternative to Optional
9	Pull Up Node
10	Push Down Node
11	Remove Formula
12	Add Optional Node

relation into a general or relation encompassing all features. We can propose a similar refactoring for more than two child feature nodes.

Note that Refactoring 2 cannot be applied from right to left because the RHS model can select features A and B (not selecting C and D), which is not possible on the LHS model. Therefore, the resulting model does not contain all valid configurations of the original FM, hence it is not a refactoring. This counter-example illustrates that there are non-trivial configurability-improvement issues in the SPL context, thus further motivating the need for FM refactorings.

Refactoring 2 *collapse optional and or*

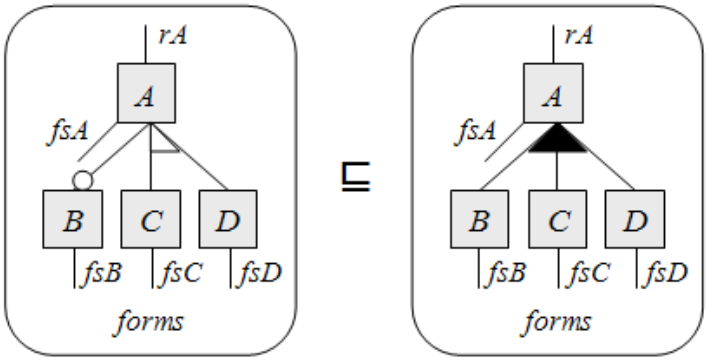


Our catalog of refactorings is summarized in Table 5.1 and explicitly listed in Section 5.6. For instance, we have refactorings for pulling up (Refactoring 9) or pushing down (Refactoring 10) feature nodes. Another example, removing a formula (Refactoring 11), is a refactoring since the resulting model is less constrained, hence increasing configurability. Refactoring 12 allows us to introduce an optional feature. In fact, there are additional refactorings, since most of them can be applied similarly in contexts with more than two features, such as Refactorings 2 and 5.

By composing the refactorings, we can derive other valuable refactorings. For in-

stance, by composing Refactorings 1 and 2, we derive Refactoring 3. This is possible because starting from the LHS of Refactoring 3, we can first apply Refactoring 1, thus turning the alternative relationship between C and D into an or-relationship; at this point B is still optional, but we can now apply Refactoring 2 to group B together with C and D into an or-relationship.

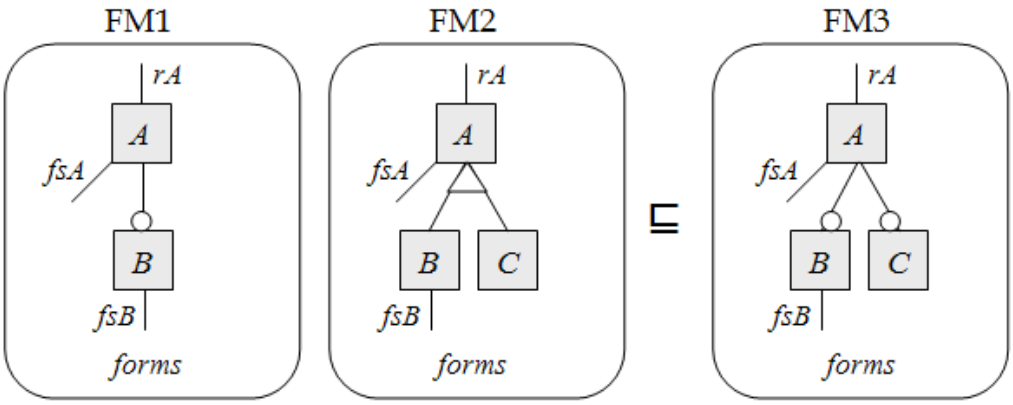
Refactoring 3 *collapse optional and alternative to or*



So far we focused on *refactoring single FMs*. However, as we described in Section 5.2, we may deal with previously existing products or SPLs, each one having its own FM. Accordingly, during extractive SPL adoption strategy, we may want, for instance, to merge these products and SPLs into a single new SPL. In this case, we give support to FMs in the merging refactoring notion presented in Section 5.2, by defining refactoring between more than two FMs.

For example, the subsequent refactoring (which we call *Extractive*) allows us to merge optional and alternative relations. The resulting FM refactors the initial ones if and only if the resulting FM refactors each FM on the left side of the arrow. We can actually model an extractive refactoring as a sequence of single-FM refactorings applied to both original FMs separately.

Extractive 1 ⟨merge optional and alternative⟩



Suppose that $fm1$, $fm2$ represent the two LHS FMs, and $fm3$ the RHS FM, re-

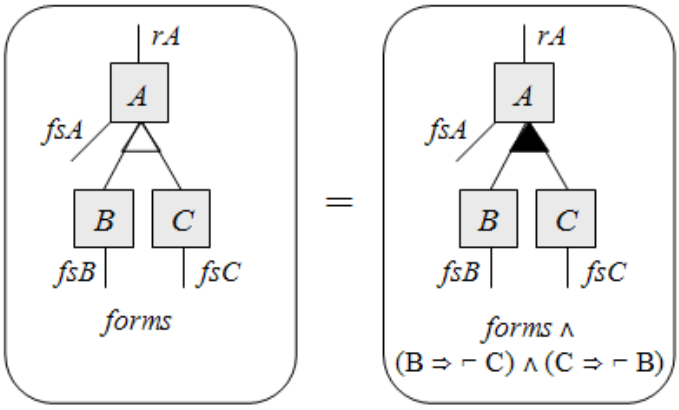
spectively, of the Extractive 1 refactoring. $fm3$ improves the configurations of $fm1$ by applying Refactoring 12 in order to introduce the optional feature node C. Moreover, $fm3$ improves the configurations of $fm2$ by applying Refactoring 8 in order to convert an alternative to option. Therefore, $fm3$ refactors both $fm1$ and $fm2$. Using the refactorings from Table 5.1, we can derive other refactorings between more than two FMs.

5.4.4 Bidirectional Refactorings

A bidirectional refactoring is a *special case* of FM refactoring that *maintains* the configurability of a model. In this section, we propose a set of bidirectional refactorings (B-Refactorings) for FMs. In other words, if two FMs have the same configurability (semantics), we can *always* relate them by applying B-Refactorings.

B-Refactorings also define two FM templates, although being applicable in both directions. B-Refactoring 1 relates the alternative and or relations. Applying B-Refactoring 1 from left to right allows us to convert an alternative to an or relation along with two formulas establishing the same constraints. Similarly, by applying the transformation from right to left, we can convert an or to an alternative relation.

B-Refactoring 1 ⟨*replace alternative*⟩



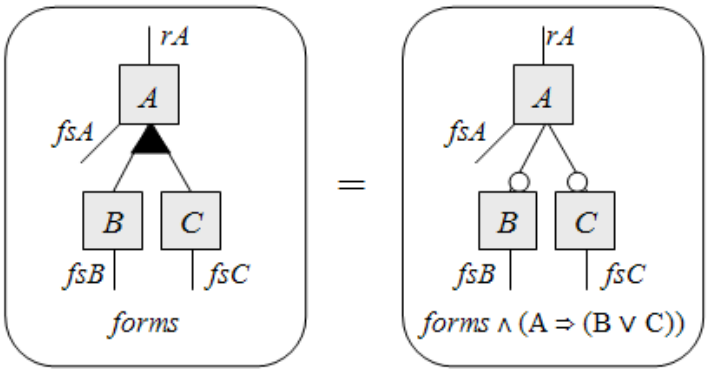
B-Refactoring 2 relates an or relation and optional nodes. Moreover, B-Refactorings 1 and 2 can be applied when there are more than two child features.

Next, B-Refactoring 3 relates a mandatory feature with an optional feature with a formula stating the same fact, whereas B-Refactoring 4 removes an optional feature and states the same fact in a formula.

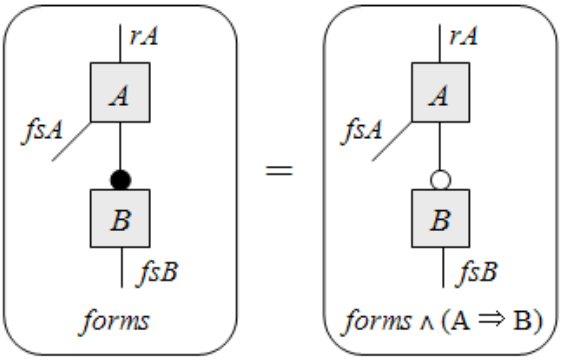
The root of a FM always appears in all valid configurations. B-refactoring 5 removes a root and includes a formula stating that the root is always present. B-refactoring 6 removes a feature that can never be selected. Similarly, this one allows us to add a set of nodes if we add a formula stating that the nodes cannot be selected. Finally, B-refactoring 7 allows us to add or remove formulas deducible from the model. Since it is a deducible formula, the configurability is maintained.

Properties of B-Refactorings. Some of the previous transformations may not be useful in practice since they convert a valid FM to another that is not a tree, such as B-Refactoring 4. However, they are important for theoretical reasoning, as we discuss

B-Refactoring 2 ⟨*replace or*⟩



B-Refactoring 3 ⟨*replace mandatory*⟩



in the following. In practice, developers should only be aware of the FM refactoring catalog.

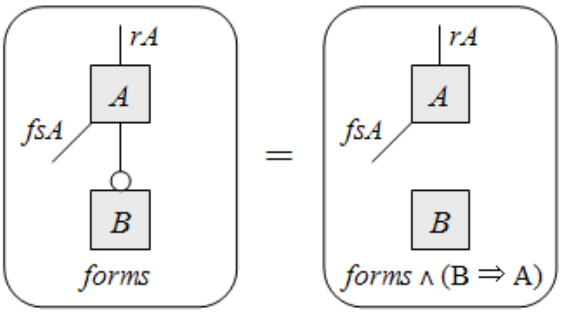
The set of seven B-Refactorings is *sound*, *minimal* and *complete*. Since each B-Refactoring defines two simple localized transformations, we can verify that they are sound. These transformations are minimal since each one deals with one different construct each time. Therefore, one transformation cannot be derived from another.

With respect to completeness, if two FMs are equivalent (have the same configurations), we can always reduce one model to another by applying B-Refactorings. Suppose that two FMs `fm1` and `fm2` have the same configurations (`semantics(fm1)=semantics(fm2)`). Next we show how we can relate them by applying our B-Refactorings.

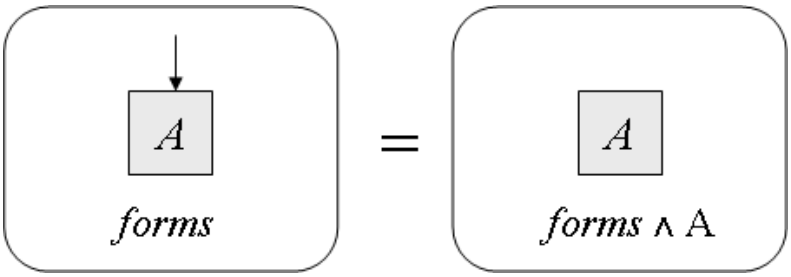
1. Remove all features from `fm1` and `fm2` that cannot be selected (applied in the presence of a formula negating the corresponding features) by applying B-Refactoring 6 from left to right;

2. Replace all graphical relations by equivalent formulas expressed in propositional logic by applying B-Refactorings 1-5 from left to right. These B-Refactorings present no conditions for applications. As a consequence, `fm1` and `fm2` are reduced to models containing only features and formulas (without relations).

B-Refactoring 4 ⟨*replace optional*⟩



B-Refactoring 5 ⟨*remove root*⟩



3. Since `fm1` and `fm2` have the same semantics, they present the same features (`features(fm1)=features(fm2)`). However, $fm1$ and $fm2$ may have syntactically-different formulas, although equivalent. Since the propositional logic calculus is complete, we can always prove that `forms(fm1)=forms(fm2)` by applying B-Refactoring 7 for introducing deduced formulas.

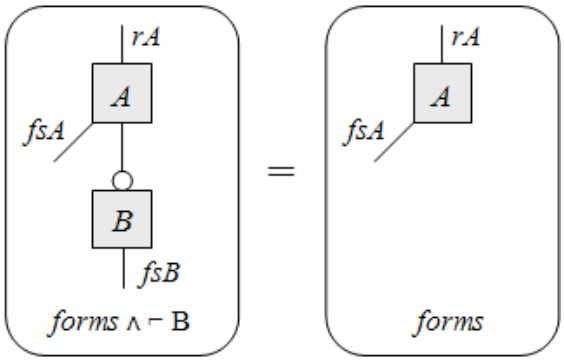
Therefore, we have shown that we can relate $fm1$ and $fm2$ using our B-Refactorings, whenever they have the same semantics. Figure 5.6 summarizes the completeness proof, and Figure 5.7 illustrates its reduction strategy.

The completeness result is very important for this kind of work. It shows that our catalog of B-Refactorings is representative enough to derive any kind of refactoring that maintains configurability. In practice, the developer will not need any other transformation when the initial and final FMs have the same configurability. Contrasting, another work [53] proposes a comprehensive set of program refactorings, but it does not show that this set is complete. As a consequence, we may have some situations where we would like to apply a program refactoring but the catalog does not have it.

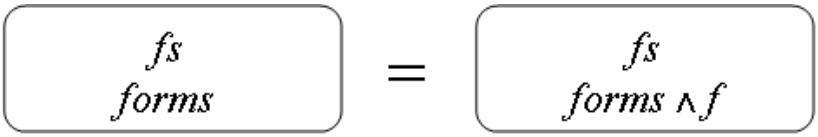
5.4.5 Discussion

In practice, the developer may choose between *semantics based reasoning* and *reasoning with our catalog of sound refactorings* in order to apply a FM refactoring. Our catalog of sound refactorings can be seen as a high level API, which is much easier to use (based on template matching), whereas semantics based reasoning is similar to using no API at all, as illustrated in Figure 5.8. Additionally, as shown in Section 5.4.3, the refactorings can

B-Refactoring 6 ⟨*remove node*⟩



B-Refactoring 7 ⟨*add formula*⟩



$(\leftrightarrow) f$ can be deduced from *forms* and *fs*.

be composed to achieve even coarser-grained transformations and they are also extended to handle more than one feature model in the left-hand side template.

Furthermore, we note that, once a refactoring is used and variability of a feature model increases, there is no incurred constraint on the relationship among the instances of the improved feature model. Such constraints are established during the definition of the configuration knowledge. The precise definition of such constraints are outside the scope of this work. Nevertheless, we illustrate these in Chapter 6.

We emphasize the difference between FM refactoring, which we introduce here, and FM specialization, formalized by Czarnecki [46]: FM refactoring is a transformation that either *maintains or increases* the set of all FM configurations, whereas FM specialization is a transformation that *decreases* the set of all FM configurations.

5.5 Case Study

In this section, we evaluate the extended refactoring notion for SPLs. First, we describe the context of the case study (Section 5.5.1); next, in Section 5.5.2, we describe our approach for verifying the correctness of the case study refactorings in terms of FMs discussed in Sections 5.4.3 and 5.4.4.

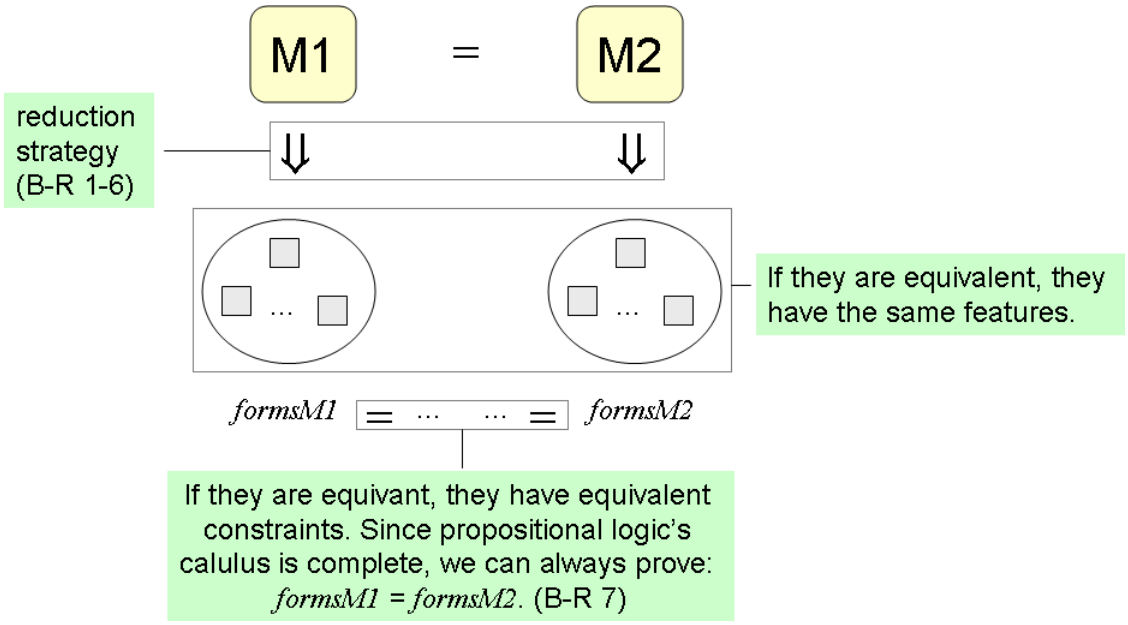


Figure 5.6: Completeness proof of B-refactorings. **B-R** stands for B-Refactorings.

5.5.1 Context

This case study focuses on FM refactoring and is based on a simplified version of the case study presented in Section 6.1. It combines the extractive and the reactive SPL adoption strategies [81] in the mobile games domain. As explained in Section 2.3, J2ME games are mainstream mobile applications of considerable complexity in comparison with other mobile applications [13, 2]. The major variability issues within these products are as follows: optional images, alternative image loading policies, proprietary API, application size limit, screen dimensions, and additional keys [13, 2]. It is essential to note that these features are not independent. Indeed, application size constrains other features, such as optional images and additional keys.

Figure 5.9 depicts our case study, focusing on the source code. We started from a scenario in which the same game ran in two devices, thus having two initial applications, *Product*1 and *Product*2. Both applications have the same core functionality, but differ in some features, since Device 1 is not a resource-constrained device and, for instance, can afford enough heap and application size for *Product*1 to have the *croma* feature of clouds scrolling in the background and the simple image loading policy of loading all images at game startup. On the other hand, Device 2 is resource-constrained and thus *Product*2 does not have the *croma* feature implemented. Instead, *Product*2 has an optimized image loading policy of loading images on demand during changing screen events. From this initial scenario, we have two goals:

- bootstrap the existing products (*Product*1 and *Product*2) into a new SPL (named *SPL*1-2);
- during this process, react the emerging SPL *SPL2* to encompass another product, which should be the game with a partially (hybrid) optimized image loading policy.

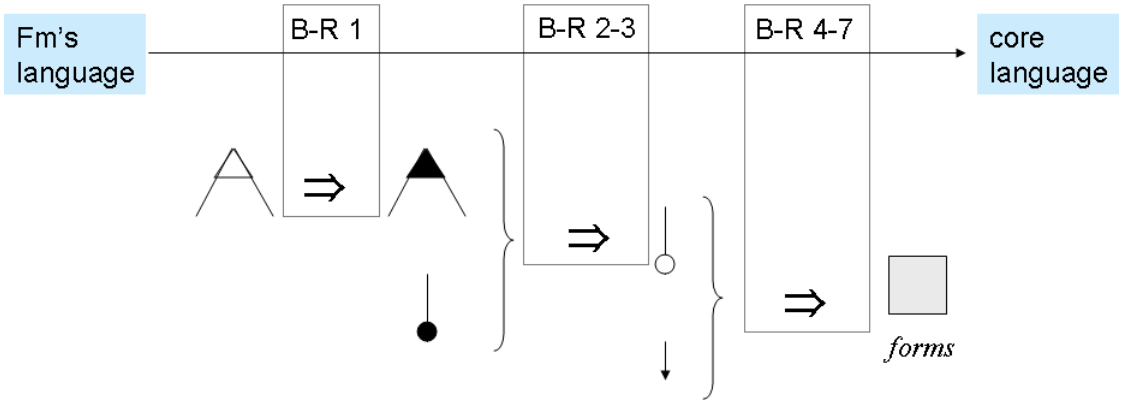


Figure 5.7: Reduction strategy for completeness proof. **B-R** stands for B-Refactorings. All B-Refactorings are applied from left to right.

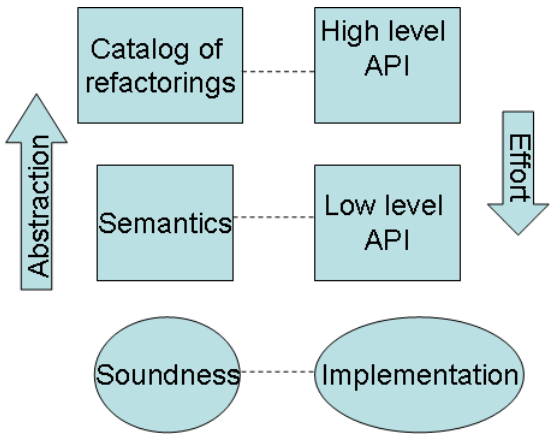


Figure 5.8: Semantics-based reasoning versus catalog-based reasoning.

5.5.2 SPL Refactoring

We apply program and FM refactorings for achieving those goals using our extended definition for SPLs presented in Section 5.2.2. Although confidence in program refactorings can be increased with a mechanics added by compilation and tests [53], variability improvement is hard to ensure, and tests may not easily uncover such inconsistencies. These problems can usually be detected on the problem space, with FMs. We use the refactorings presented in Section 5.4.3 for ensuring correctness of FM refactorings by analyzing their application on corresponding FMs after program refactoring steps. We assume that the FM associated with each product is determined from the product documentation or by code examination.

Program Refactoring

In this step, we apply program refactorings in order to ensure the behavior preservation. In order to accomplish our first goal from Section 5.5.1, we first started applying a

sequence of program refactorings to *Product*1 with the aim of modularizing the *croma* and the image loading policy features. In Figure 5.9, the + symbol in *Product*1 indicates that the implementation of such features is scattered and tangled with the application core. Accordingly, we apply a sequence of refactorings from Section 4.1.3 in order to extract such features into aspects *Clouds* and *Startup*, respectively (since the focus of this chapter is on feature modeling refactoring, details of applying such program refactorings are found in Section 6.1.3). The result is SPL *SPL*1.

Similarly, we apply those refactorings in order to modularize the *OnDemand* loading image feature of *Product*2. After that, we accomplish our second goal, evolving the product into a new SPL (named *SPL*2) by adding a new kind of image loading policy (*Hybrid*).

Finally, in order to avoid code duplication, our aim is to integrate *SPL*1 and *SPL*2, due to their similar core. As *SPL*1 and *SPL*2 must have exactly the same core for merging both SPLs, we apply adjustment refactorings (such as renaming) to the core of *SPL*1. We can now apply our merge program Refactoring SPL 1, which was presented in Section 5.2.2, in order to merge *SPL*1 and *SPL*2 into *SPL*1-2.

Feature Model Refactoring

Besides dealing with programs, we must ensure that the resulting transformations improve the configurability of the SPL. For that, some configuration knowledge [45] must be used for defining the correspondence between features and components (for instance, classes and aspects). In the example, we adopt a convention in which features may be tangled with core functionality or implemented as separate aspects; their optionality can be implemented by configuration scripts used for building SPL instances. Other configuration knowledge choices may be used likewise.

Our approach consists in generating FMs for the initial and resulting product or SPL, investigating the use of the proposed FM refactorings for verifying the configurability improvement between both models. If a sequence of refactorings can be applied, additional confidence on the safety of the SPL refactorings is provided. The original and resulting FMs corresponding to the refactoring applied in *Product*1 are shown in Figure 5.10. For making *Clouds* optional, we can apply Refactoring 7 resulting into SPL *SPL*1. In a later step, as we applied program refactorings to start preparing *SPL*1 for an extractive refactoring, it demands no changes on the FM (a reflexive step).

In the source code for *Product*2, the image loading policy feature was isolated into *OnDemand* aspect. As *OnDemand* feature is maintained mandatory, no changes are needed in the FM. At this point in the program, we use the reactive approach for creating *SPL*2, adding the alternative *Hybrid* aspect. The variant of Refactoring 5, presented in Section 5.4.3, adds the *Hybrid* feature, which verifies this step on the FM.

The final step in the source code was an extractive refactoring merging *SPL*1 and *SPL*2. The resulting SPL (*SPL*1-2) thus encompasses *SPL*1 and *SPL*2. At the FM level, we can generate *SPL*1-2 (Figure 5.10), which includes the three alternative features for image loading (*Startup*, *OnDemand* and *Hybrid*) and an optional feature (*Clouds*). With the two intermediate FMs for *SPL*1 and *SPL*2, we can now generate a single FM, based on the definition of extractive refactoring given in Section 5.4.3. We ensure correctness by applying refactorings to both FMs, as shown in following statement:

$SPL1 \rightarrow SPL1\text{-}2$ [*by applying* $2x$ *Refactoring* 5]
$SPL2 \rightarrow SPL1\text{-}2$ [*by applying Refactoring* 5 *and* 12]

In the first branch, $SPL1 \rightarrow SPL1-2$, the variant of Refactoring 5 was used, then its general form was used. In the second branch, only the general form of such refactoring was used. Our approach shows the application of FM refactorings in a real scenario. Besides ensuring confidence on correct transformations, this approach may also help identifying incorrect steps in a refactoring application, usually not easily detected when inspecting or testing source code. The impossibility of applying certain refactorings may be a consequence of such errors, more easily uncovered at the FM level. It must be stressed that, although some synchronism between FMs and programs with code is important, it is not the focus of this chapter. Rather than *monitoring* source code refactoring, the chapter proposes a *complementary* transformation level: FM refactorings, for aiding refactoring soundness for SPLs. More ambitious accomplishments, such as, a completely model-driven approach to SPL refactoring, require a formal notion of conformance between FMs and programs, which is regarded as future work.

5.6 Unidirectional Refactorings Catalog

In this section, we present the unidirectional refactorings proposed that were listed in Table 5.1 but not explicitly defined in Section 5.4.3. In each case, we argue that the transformation is sound by showing how configurability increases. The f and *forms* variables used in Refactoring 11 denote a formula and a set of formulas, respectively.

In Refactoring 1, an alternative feature relation is subsumed by an or-feature relation. Configurability then increases, since features B and C can both appear in a configuration of the resulting feature model. Refactoring 4 replaces mandatory features with or-features, which increases the configuration. New possible configurations now include either feature B or feature C. In Refactoring 6, an or-relation is changed to optional features. The configurability increases, since now a configuration of the resulting feature model can lack both features B and C. Refactoring 7 turns a mandatory feature B into a optional feature. The resulting feature model can have, in addition to the initial configurations, a new configuration lacking feature B.

In Refactoring 8, an alternative relation is changed to optional features. The configurability increases, since now a configuration of the resulting feature model can lack both features B and C. Another possible configuration has both features. Refactorings 9 and 10 are actually equivalences; we just list them separated to ease referring to them. The configurability is maintained because B is mandatory. Refactoring 11 removes a formula from the feature model, thereby decreasing logical constraints on it, which in turn increases its configurability. Finally, in Refactoring 12, an optional feature B is added. As a result, the configurability of the resulting feature model increases, by allowing a configuration having such feature.

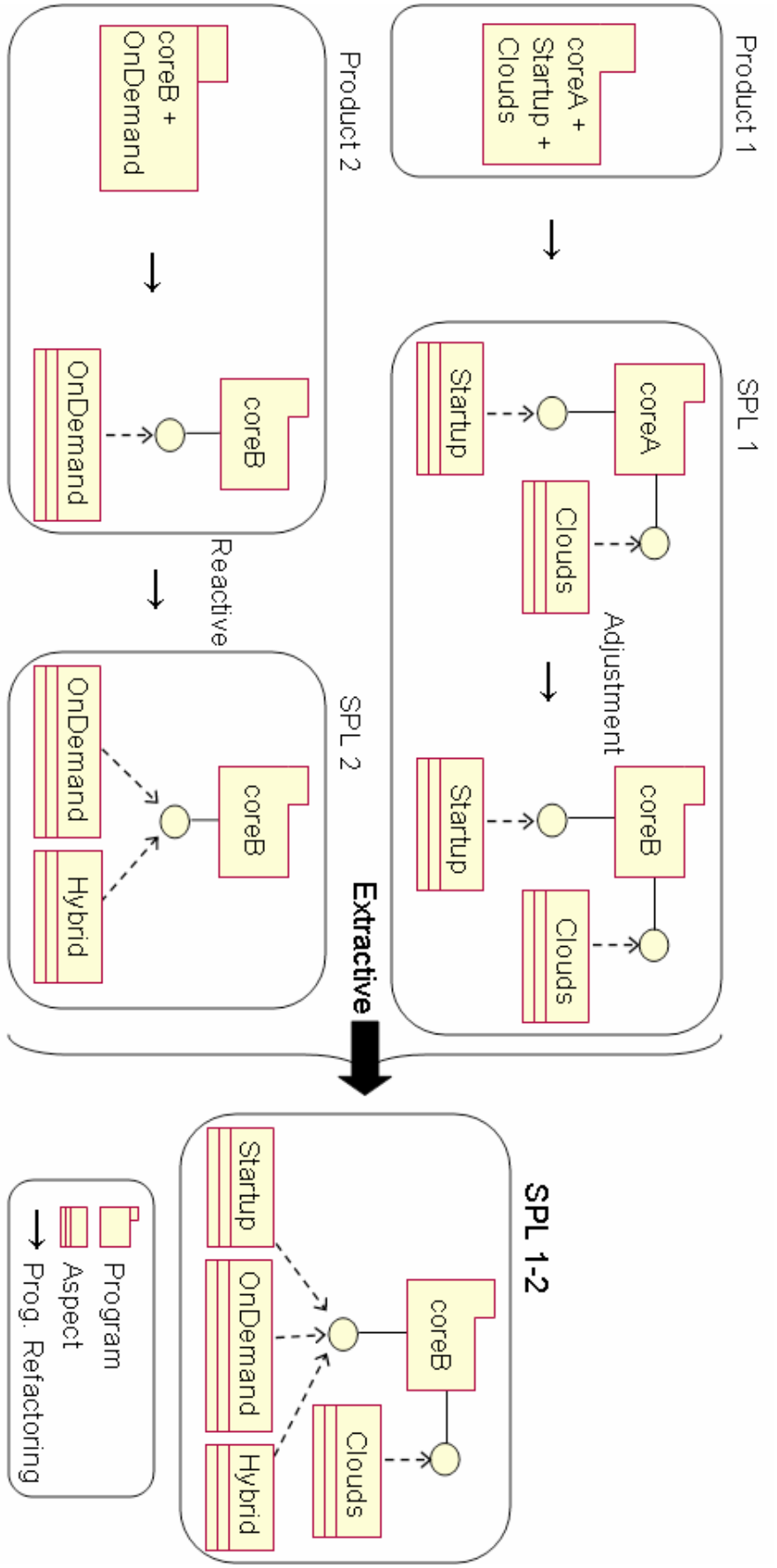


Figure 5.9: Case Study Program Refactorings

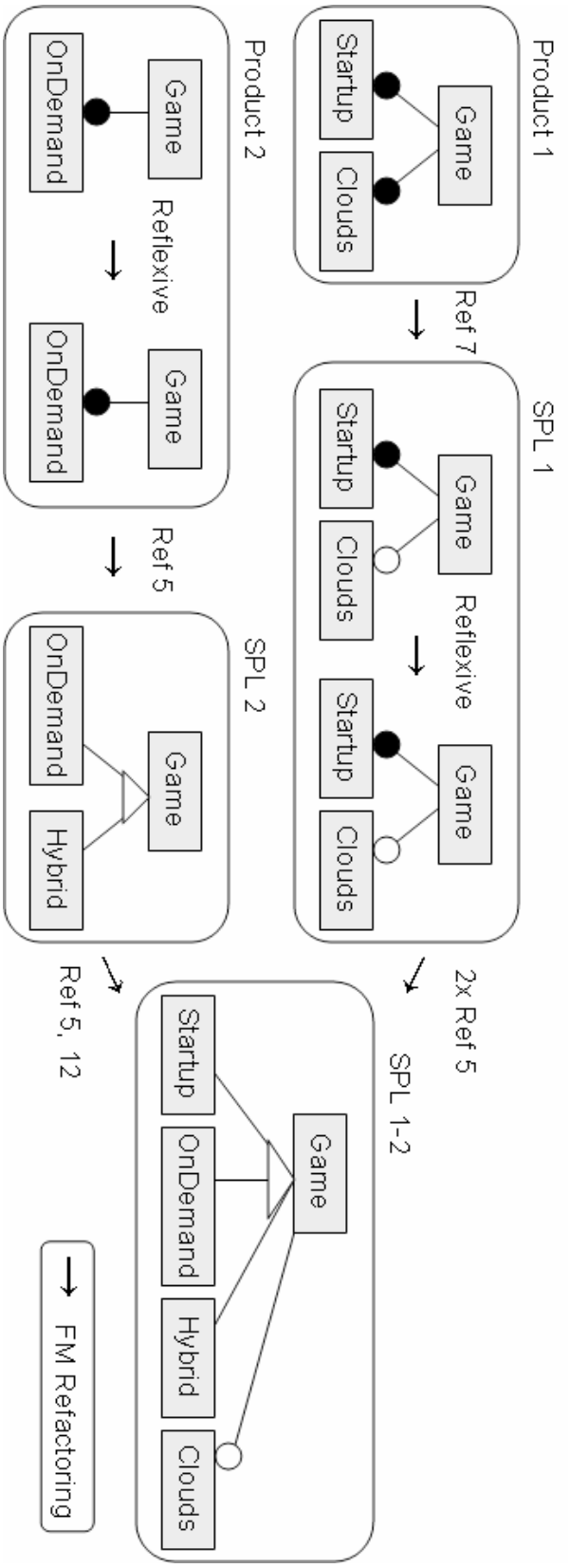
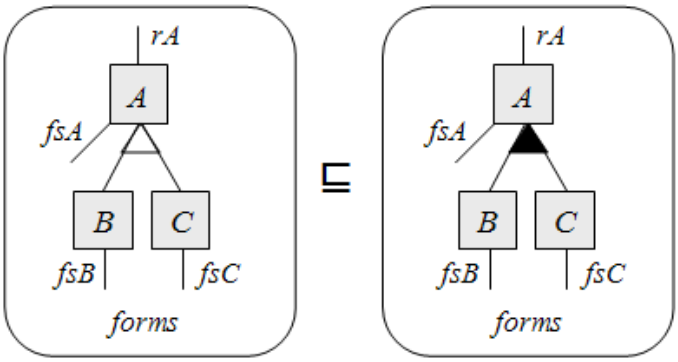
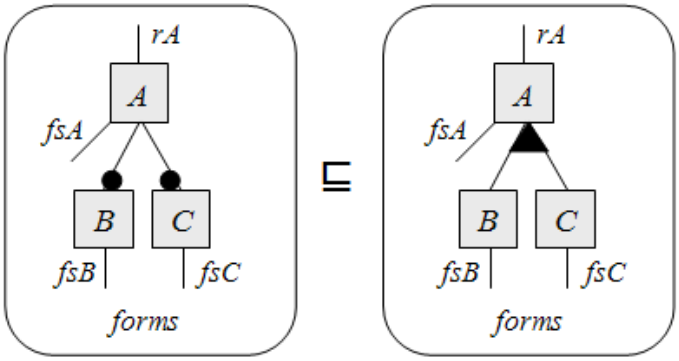


Figure 5.10: Case Study Feature Model Refactorings

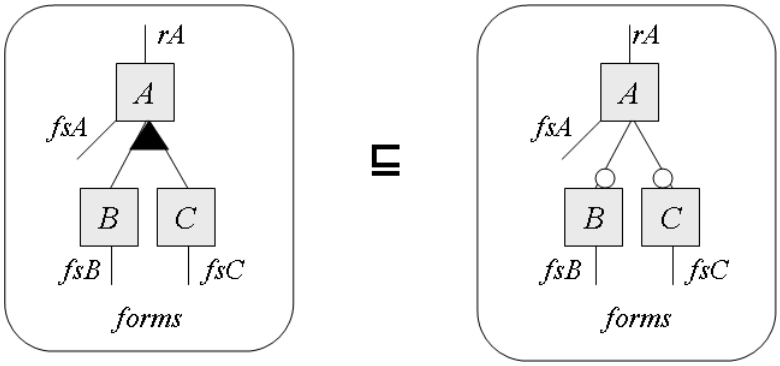
Refactoring 1 ⟨*convert alternative to or*⟩



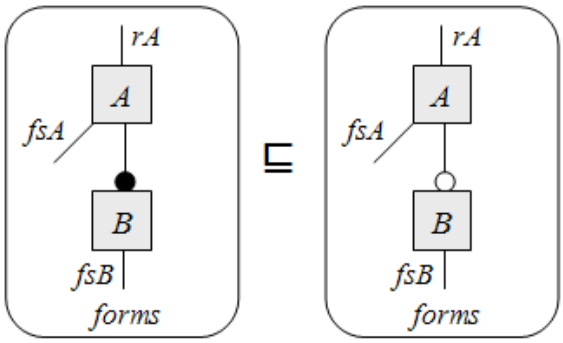
Refactoring 4 ⟨*add or between mandatory*⟩



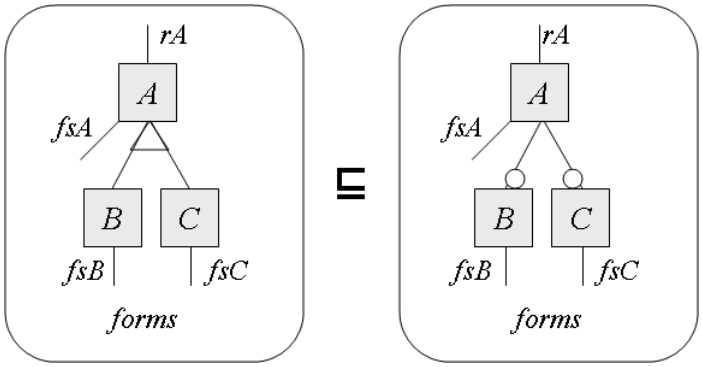
Refactoring 6 ⟨*convert or to optional*⟩



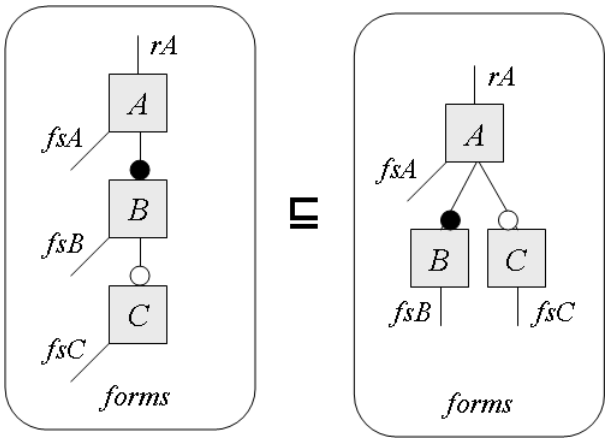
Refactoring 7 ⟨*convert mandatory to optional*⟩



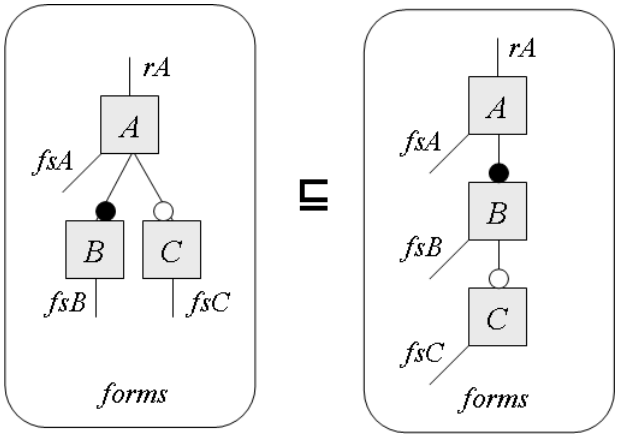
Refactoring 8 ⟨*convert alternative to optional*⟩



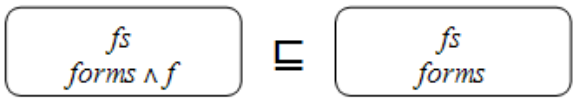
Refactoring 9 ⟨*pull up node*⟩



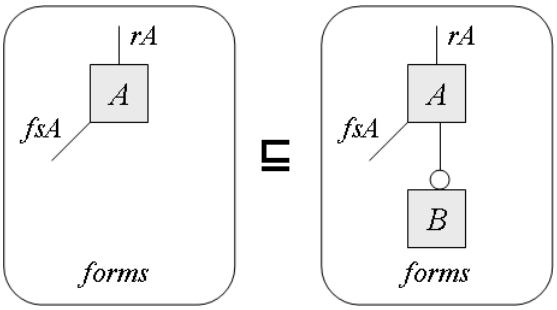
Refactoring 10 ⟨*push down node*⟩



Refactoring 11 ⟨*remove formula*⟩



Refactoring 12 ⟨*add optional node*⟩



Chapter 6

Case Studies

This chapter evaluates the method described in Chapter 4 in the context of industrial-strength mobile game SPLs. As explained in Section 2.3, mobile games are mainstream mobile applications of considerable complexity in comparison with other mobile applications and represent a highly variant domain due to portability requirements. By describing some case studies, we identify the variabilities addressed in these SPLs, the refacotings employed to manage them, and the resulting configurability. In particular, the goals of the case studies are the following:

- describe the method in industrial-strength applications;
- evaluate its application, using analytical and quantitative data;
- identify possible enhancements to the method.

The remainder of this chapter is organized as follows. Sections 6.1 and 6.2 each describe a case study, evaluating the proposed method in Chapter 4. Next, Section 6.3 presents and addresses some open issues of the method. Finally, Section 6.4 compares the results of both case studies.

6.1 Rain of Fire

Rain of Fire (RoF) is a classic arcade-style game where the player protects a village from different kinds of dragons with catapults. Figure 6.1 illustrates its main screen. The game is a *commercial product* currently offered by service carriers in South America and Asia. Although it is less than 5K LOC, LOC is neither a necessary nor sufficient condition for complexity. In fact, complexity in the mobile game domain arises mostly due to variability. In general, the mobile game domain is highly variant due to a strong portability constraint: applications have to run in numerous platforms, giving rise to many variant products [34, 5], which are under a tight development cycle, where proactive planning is often unfeasible to achieve.

RoF was developed for 12 different devices. Although the game itself is based on a game engine framework, the process of developing a new version of the game was based on copying it from a previously developed version.

Figure 6.1: Platform variation of Rain of Fire.

6.1.1 Study Setting

In this case study, we evaluated the extractive and reactive steps of our method (Figure 4.1). Although the SPL that actually exists in our industrial partner encompasses 12 members, in this case study we investigated how RoF was adapted to run in 3 platforms (P_1, P_2, and P_3), which encompass most variability issues in this SPL. P_1 relies solely on MIDP 1.0, whereas P_2 and P_3 rely on MIDP 1.0 and a proprietary API. Some of the variability issues within these products are as follows: optional images, proprietary API for flipping images, screen sizes, and image loading policy. After applying our approach (details shortly ahead), the resulting SPL has the feature model of Figure 6.2, and the following instances, as described by the selection of features.

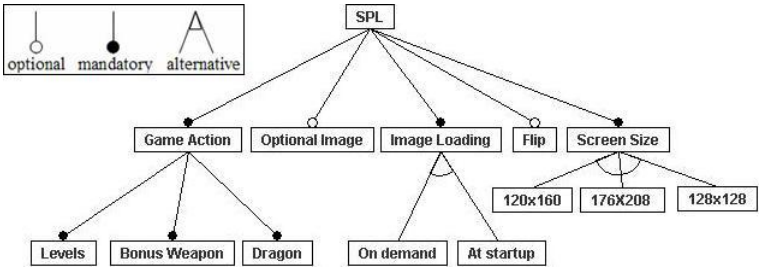


Figure 6.2: Variability within Game Product Line

```
P1 = {Dragon, Bonus Weapon, Levels, Optional Image, At startup, Flip, 176x208};
P2 = {Dragon, Bonus Weapon, Levels, On demand, Flip, 120x160};
P3 = {Dragon, Bonus Weapon, Levels, Optional Image, At startup, 128x128};
```

Although this case study has focused only on 3 instances, the feature model shows that other configurations are also possible: the feature model has a total of 24 configurations.

In order to evaluate our approach, we created a SPL implementation of the three products and then compared the SPL version with the original implementation of these products. To create and evolve the SPL, we first identified the variabilities (such as optional images). Those variabilities were then extracted. We also describe the configuration knowledge impact. The changes in the feature model were described according to the FM refactorings of Chapter 5, which also illustrates them in a simplified version of the model (Figure 5.10).

6.1.2 Variability Identification

In order to better identify and understand some variations, we could, for instance, use concern graphs [109]. Concern graphs localize an abstracted representation of the program elements contributing to the implementation of a concern, making the dependencies between the contributing elements explicit. Such graphs are created iteratively by querying a model of the program, and by determining which elements (class, methods, and fields) and relationships returned as part of the queries contribute to the implementation of the concern. The querying process starts with a *seed* [109], usually a class found with a lexical tool. From this class, the remaining elements are added with tool support. For example, the concern graph C for the optional images concern (oi) in P_1 would be as follows:

$$C_{p1,oi} = (V_{p1,oi}, V^*_{p1,oi}, E_{p1,oi}),\ V^*_{p1,oi} = \varnothing$$

$$V_{p1,oi} = \left\{ \begin{array}{l} Resources, GameScreen, Resources.dragonRight, \\ Resources.loadImages(), GameScreen.wakeEnemy() \end{array} \right\},$$

$$E_{p1,oi} = \left\{ \begin{array}{l} (reads, GameScreen.wakeEnemy(), Resources.dragonRight), \\ (writes, Resources.loadImages(), Resources.dragonRight), \\ (declares, Resources, Resources.dragonRight), \\ (declares, Resources, loadImages()), \\ (declares, GameScreen, wakeEnemy()) \end{array} \right\},$$

The set $V_{p1,oi}$ describes the vertices (classes, methods, attributes) partially implementing the concern. Set $V^*_{p1,oi}$ consists of vertices (classes, methods) solely dedicated to the concern implementation. Finally, set $E_{p1,oi}$ groups edges relating elements from the previous sets.

6.1.3 Extraction

After identifying variabilities, we then moved their definition to aspects using the *Extract Resource to Aspect* refactoring. In another step, we addressed method body variability within the platforms. Accordingly, we made extensive use of the *Extract Method to Aspect* refactoring. The *Extract After Block* and *Extract Before Block* refactorings were used when the variant code appeared at the end or beginning of the method body. On the other hand, the *Extract Context* refactoring was used when the variation surrounded common code, representing a context to it. The *Extract Argument Function* refactoring was used when variation appeared as an argument for a method call. Finally, we used the *Change Class Hierarchy* refactoring to deal with class hierarchy variability.

During the evolution of the SPL to include P_3, we had to deal with the *load images on demand* concern. This concern was specific to this platform, as it had constrained memory and processing power. To implement this concern, we had to define a method for each screen that could be loaded. Before a screen was loaded, the corresponding method was called. In contrast, in P_1 and P_2 implementations, the images were loaded only once, during game start-up. In this case, there was only one method that loaded all the images into memory. This situation illustrates the scenario in Figure 4.3.

We addressed this by applying a sequence of *Extract Method* refactorings in the core to break the single method loading all images into finer-grained methods loading images for each screen; the call of this single method was then moved from the core to P_1's and P_2's aspects, and the calls to such smaller methods were moved to P_3's aspect by the *Extract Before Block* refactoring.

Another evolution scenario took place when we realized that some commonality existed between P_1 and P_2 with respect to the *Flip* feature (proprietary graphic API allowing an image object to be drawn in the reverse direction, without the need for an additional image): these two platforms are from the same vendor and share this feature, which is not shared by P_3, from another vendor. Therefore, the *Flip* feature is isolated in the corresponding aspects of P_1 and P_2, but it would be useful to extract this commonality into a single module. In fact, we were able to factor this out into a single generic aspect (`AspectFlip`) with the *Extract Aspect Commonality* refactoring, thus illustrating the scenario in Figure 4.4.

Table 6.1 reports the occurrence of each refactoring for achieving the resulting SPL. According to this table, *Extract Method to Aspect* was the most frequently employed, since variability within method body was common for extracting most features. As the SPL evolves, we expect to employ *Extract Aspect Commonality* more frequently.

For the resulting the SPL, we also employed the *Move Field to Aspect* programming law from Section 4.2.1. This law was used 28 times. This is consistent with the results of Table 6.1, since we do not claim these to be complete (the argument in Section 4.2 is on soundness). Additionally, if we had only used the programming laws themselves instead of the refactorings (composition of the programming laws), we would have to apply approximately twice as many programming laws. In general, the method can combine the refactorings and the programming laws themselves. As the set of refactorings evolve closer to completeness, the direct use of the fine-grained programming laws is expected to decrease and the proportional use of the coarse-grained refactorings is expected to increase.

Table 6.1: Occurrence of each refactoring

Refactoring	**Name**	**Occurrence**
1	Extract Resource to Aspect - after	5
2	Extract Method to Aspect	41
3	Extract Context	1
4	Extract Before Block	2
5	Extract After Block	10
6	Extract Argument Function	1
7	Change Class Hierarchy	1
8	Extract Aspect Commonality	1

6.1.4 Configuration Knowledge

The resulting configuration knowledge maps sets of features to implementation artifacts:

```
{Levels, Dragon, Bonus Weapon} -> CoreClasses
{Flip} -> AspectFlip
{176x208, Optional Image, At startup} -> AspectP1
{120x160, On demand} -> AspectP2
{128x128, Optional Image, At startup} -> AspectP3
```

where *CoreClasses* is a set of core assets thatt comprises classes common to all products; `AspectFlip` is a core asset aspect dealing with the *Flip* feature; `AspectP1`, `AspectP2`, and `AspectP3` deal partially with specific products features of products P1, P2, and P3, respectively. The arrow notation means that the set of features to its left, which are from the feature model represented in Figure 6.2, map to the aspects or classes to its right. According to this configuration knowledge and to the configuration of each product presented previously, the SPL instances are synthesized by

```
P1 = CoreClasses • {AspectP1, AspectFlip};
P2 = CoreClasses • {AspectP2, AspectFlip};
P3 = CoreClasses • {AspectP3};
```

where • denotes composition. According to this derivation of the SPL members, the SPL core assets consist of the following: 1) eighteen classes in *CoreClasses*; 2) one core aspect (`AspectFlip`). P1 and P2 each comprise *CoreClasses* and two product-specific aspects; P3 consists of *CoreClasses* and one product-specific aspect. Indeed, the configuration knowledge is coarse-grained: there are few reusable aspects across different SPL instances. In fact, `AspectFlip` is the only reusable aspect. The case study in Section 6.2 identifies more reusable aspects and configurations.

6.1.5 Analysis

After creation and evolution of the SPL, we analyzed code metrics. Table 6.2 shows the number of Lines of Code (LOC) for each product in the original implementation, in contrast with the SPL implementation. We calculate the LOC of a SPL instance as the sum of the core's LOC and the LOC of all aspects necessary to instantiate this specific product.

According to Table 6.2, LOC is slightly higher when comparing each SPL instance with the corresponding product in the original implementation. This is caused by the extraction of methods and aspects, which increase code size due to new declarations. On the other hand, there is a 48% reduction in the total LOC of the SPL implementation, when compared to the sum of LOCs of the single original versions. This was possible because of the core assets, which represent 57% of the SPL LOC. Although there is considerable commonality between the three original products source code, it is worth to consider it as different code, because it is repeated for each product and tightly coupled with it. This code repetition increases the effort of program reasoning

Table 6.2: LOC in original and SPL implementations

Original Implementation				**SPL Implementation**					
P_1	P_2	P_3	Total	**Core assets**		P_1	P_2	P_3	Total
				Core classes	Core aspects				
2965	2968	3143	9076	2477	72	3042	3047	3210	4405

Table 6.3: LOC of aspects in the SPL.

Aspect	**LOC**
AspectP1	421
AspectP2	426
AspectP3	661
AspectFlip	72

and maintenance. A reduction due to the avoidance of code repetition could also be obtained using a different product line approach or some modularization techniques, like componentization. Another factor that contributes to the reduction in SPL LOC is the existence of reusable aspects.

Table 6.3 shows the sizes of the SPL aspects. The only reusable aspect is considerably smaller than the product-specific ones. The small size of this aspect is convenient for it to be reusable across different SPL instances. With a more fine-grained configuration knowledge, we expect that there would be a higher number of reusable aspects and the relative size of the product-specific ones would decrease. Eventually, it could happen that, for some SPL instances, no product-specific aspect would be necessary, in which case such instance would be derived solely by reusing different combinations of core aspects.

Analyzing Table 6.3 in conjunction with the configuration knowledge presented previously, we can infer that the relative size of aspect code in the SPL members ranges from 16% for P1 and P2 to 20% for P3.

Another analyzed metric was the packaged application (jar files) sizes of the original and of SPL implementations (Table 6.4). Jar files, that are released to final users, include not only the bytecode files, but also every resource necessary to execute the application, such as images and sound files In the case study products, additional resources represents, on average, 45% of the total jar file size. To measure the impact of our approach on bytecode size, we are considering, in Table 6.2, the jar files containing only the class files, excluding other resources. The jar file size is a very important factor in games for mobile devices, due to memory constraints.

We can notice a jar size increase from original versions to SPL instances. The reason for this is the overhead generated by the AspectJ weaver on the bytecode files. We also noticed that very general pointcuts intercepting many join points can lead to greater increases in bytecode file sizes. This considerably influenced us in the definition and use

Table 6.4: Jar size (kbytes) in original and SPL implementations

	Original Implementation		**SPL Implementation**	
	size	reduced size	size	reduced size
P_1	32,4	29,0	67,5	38,4
P_2	33,2	28,8	69,1	33,3
P_3	56,1	52,4	93,5	56,7
Total	98,1	86,6	206,6	104,8

of the refactorings. Moreover, we can gain a significant reduction in the jar size when using a bytecode optimization tool [128]. The reduced size of each original version and SPL instance are shown in Table 6.4.

Although in this case study the SPL implementation offers to the user of our approach the same functionality but with a higher application size, our approach is useful mostly because of the benefits that the SPL approach brings to the development process: reuse and maintenance are improved, code replication is minimized, and derivation of new products is faster and less costly. Further, the increase on bytecode size can be minimized by further advances in optimization tools. Our initial results show that, in cases where pointcuts matches few join points, by inlining the body of the advice in the base code, we can already reduce bycode size.

6.2 Best Lap

Best Lap is a casual race game where the player tries to achieve the best time in one lap and qualify for the pole position. In order to do that, the player has to succeed at several time trial mini games. The better the performance at the games, the better the time for the lap. Player score is dependent on lap time and bonuses acquired during the games. The best scores are saved in high scores tables. Optionally, user socres are posted in a server and ranked against each other. Figure 6.3 illustrates its main screen. The game is a *commercial product* developed by Meantime Mobile Creations/CESAR (which granted access to such game under collaborative research projects) and currently offered by service carriers in South America, Europe, and Asia.

The game is deployed in 65 devices. These devices are logically grouped into families, where each family represents a set of compatible devices running the same game code. The game is developed as a SPL with a number of 16 instances, where each instance corresponds to such a family of devices. Best Lap has approximately 15 KLOC and its variability is implemented using condition compilation in J2ME with the Antenna preprocessor. The decision model is implemented by Ant scripts reading property files specifying values for the conditional compilation tags and then using those values as arguments for calling the Antenna preprocessor to generate the instances. These tags are related to variability issues such as screen sizes, and correspond to leaf features of the game feature model, whose diagram is shown in Figure 6.4. One example of instance of this SPL are configured as follows:

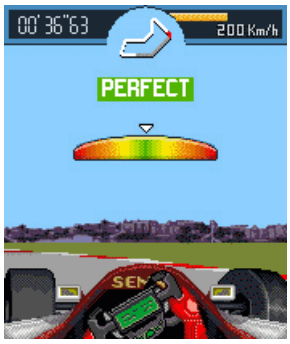


Figure 6.3: Best lap's main screen

```
MOT1 = {device_screen_128x117, device_keys_motorola, device_graphics_canvas_midp2,
device_graphics_transform_midp2, game_sprite_api_midp2, device_sound_play_thread,
device_sound_api_mmapi, device_sound_ctype_midi, device_sound_policy_preallocate,
general_debug_mode, feature_arena_enabled, known_issue_sound_prefetch,
known_issue_set_media_time, sku_id_mot1}
```

In addition to the diagram, there is a also a constraint in the feature model: feature Arena affects feature Screen. The former provides support for posting user results after the game ends and also for ranking users accordingly. Therefore, the selection of such feature affects the latter by providing coordinates and sizes of fields in the specific device screen for handling such posting and ranking information. This constraint in the feature model also reflects at the configuration knowledge level, as discussed in Section 6.2.4.

6.2.1 Study Setting

In this case study, we evaluate the migration step of our method (Figure 4.1). Accordingly, we migrate the SPL implementation of Best Lap from conditional compilation to aspects and then compare both versions. In this study, since the we only migrated the SPL implementation, the FM remained the same. Therefore, we the **Refactor FM** step of our method is not performed here. The remaining steps were carried out and are described and analyzed in the following sections.

6.2.2 Variability Identification

In order to migrate the implementation technique, we first identified the variabilities. Since we are in the migration context, these variabilities were already partially exposed in the original variability implementation mechanism, in particular by pre-compilation tags from the Antenna preprocessor. In order to help locating these tags in the implementation, we used a tool provided by our industrial partner. Such tool is an Eclipse plug-in extending the Eclipse search view feature to locate preprocessing tags in the code. Figure 6.5 illustrates its user interface:

The results of using this tool are summarized in Tables 6.5 and 6.6. Table 6.5 lists the occurrences of each of the 10 most frequently employed preprocessing tags (out of a total of 54 different tags). The number of classes in which each tag occurs illustrates the scattering of that variation (Best lap has 14 classes in its implementation). For example,

`feature_arena_enabled` (representing Arena feature) occurs in 7 classes, with a total of 26 occurrences.

Table 6.5: Occurrence of the top 10 most frequently used preprocessing tags in Best Lap

Tag	Total Occurrence	Classes involved
device_screen_128x128	33	7
device_screen_128x149	32	6
device_screen_128x117	32	7
device_screen_128x160	32	7
device_screen_132x176	32	7
feature_arena_enabled	26	7
device_sound_api_nokia	12	2
device_graphics_canvas_midp2	8	2
device_graphics_canvas_siemens	7	2
known_issue_no_softkeys	6	2

Additionally, by examining a specific class, we notice that it tangles different kind of variations, as Table 6.6 shows. This is similar to metrics like concern diffusion over LOC [56], but now in the SPL context for variability implemented with conditional compilation.

Table 6.6: Occurrence of preprocessing tags in Best Lap classes

Class	Number of tags in class
GameScreen	29
MainScreen	3
GameMenu	4
LevelManager	1
MainCanvas	4
MidletController	1
NetworkFacade	3
Resources	9
SoundEffects	8
Screen	2

Locating these tags then drove the migration process, as described in the following subsection.

6.2.3 Migration

Once the variabilities were identified, we applied the migration patterns from Section 4.1.4 in order to implement such variabilities as aspect constructs. From the variability identification step performed previously, we note that the Arena feature is considerably crosscutting. In order to extract it, we performed suitable strategies from Section 4.1.4.

For example, since feature Arena uses network service to post players scores and this variation point is present in the conditional compilation version at the end of a method for computing game score, the strategy used was *Variability in method call*. Also,

before creating the game screen, some initialization settings in the network component is necessary, using also a variant of the same strategy, handling variation at the beginning. The aspect then interacts with the network service.

When loading resources according to the screen mode, feature arena implies the existence of variation points within methods for starting the loading process of different images for the menu, thus *Variability in method call* was also used. Additionally, we applied the *Variability in method body* strategy to move methods using those constants/fields to `ArenaAspect`. Then we applied the *Variability in constant declaration* to move Arena related constants from classes such as `Resources` to the aspect implementing the Arena feature (`ArenaAspect`).

Lastly, we also note that in the conditional compilation version, variation points relating to feature Arena are frequently nested with variation points for screen sizes. This means that Arena interferes with the screens features, both at the implementation level and at the configuration knowledge level, which is described in Section 6.2.4. Accordingly, we applied strategy *Feature dependency* from Section 4.1.4.

Table 6.7 reports the occurrence of the migration strategies applied. The strategies *Variability in constant declaration* and *Field extraction* together account for the high number of occurrences, since the variation points within the conditional compilation version were mostly static. The *Feature dependency* strategy was also extensively applied, mainly in the context where the Arena feature was the outer feature of an internal dependent feature, which could be either screen features, as describe previously, or product-specific features. The *Super class variation* strategy was used in the context of coarser-grained variability: the game had to adhere to a device-specific behavior by having its canvas extend vendor-specific API or by declaring that a class implements a vendor specific-API interface (for example to signal, the sound of the game should be played in a different thread). The remaining strategies were used for handling variability within methods.

Table 6.7: Occurrence of migration strategies

Migration Strategy	**Name**	**Occurrence**
1	Variability in constant declaration	9
2	Field extraction	3
3	Super class variation	4
4	Variability in method call	4
5	If condition variation	4
6	Variability in method body	5
7	Feature dependency	6

As discussed in Chapter 4, the migration strategies are based on the refactorings and thus ultimately consist of sequential application of possibly different programming laws. Table 6.8 reports the overall usage of programming laws after complete migration of Best Lap. According to this table, law *Move Field to Aspect* was the most frequently used, since the variation points within the conditional compilation version were not only static, but also considerably fine-grained. In fact, the occurrence of this law was due to the use of the *Variability in constant declaration* and *Field extraction* strategies. The high number of the *Add new aspect* law was also due to the fine granularity of the variations. Section 6.2.4 discusses the impact of such granularity. In general, the

SPL developer performing the migration should rely on the strategies rather than on the programming laws directly, since the strategies are high-level guide for applying the chunks of low-level programming laws. We also note that some variabilities could not be migrated with the strategies and programming laws. These are discussed in Section 6.3.

Table 6.8: Occurrence of programming laws in each refactoring

Law	Name	Occurrence
1	Make aspect privileged	1
2	Add before execution	2
3	Add before call	1
4	Add after-execution	6
7	Add around-execution	1
18	Move Field to Aspect	193
19	Move method to aspect	18
22	Add new aspect	68
23	After-call	2
22	Around call	1
22	Move implements declaration	16
-	Non-defined strategy/refactoring/law	10

6.2.4 Configuration Knowledge

With the migration process, the variability implementation mechanism was changed. Although the feature model itself did not change, we had to update the resulting configuration knowledge, so that features are then related to aspects instead of compilation tags. Table 6.9 shows the resulting configuration knowledge:

In Table 6.9, the left column shows features and the right column shows aspects. Features on the left column are grouped into clusters, where each cluster denotes a set of more related features, by preceding each feature name with its path from the root of feature model. The aspects are also shown with their fully qualified names.

Some mappings are 1–1. For example, all features in clusters *device/keys*, *device/sound/policy*, *device/sound/content*, *device/sound/api*, *device/sound/thread* are each mapped to only one aspect and those aspects are mapped to only those features. This reflects the low granularity of these features.

Differently, mappings in clusters *device/graphics/canvas* and *device/screen* are 1–n, thus reflecting the coarse-grained nature of corresponding features. Additionally, for the latter cluster, aspects such as *LowEndScreen* and *HighEndScreen* are reused across mappings of multiple features. This occurs because in the original implementation with conditional compilation some variation points were related not to only one preprocessing tag, but rather to a disjunction for tags. Such disjunction actually reflected a feature cluster and thus the variation point should occur for any feature in the cluster. Accordingly, during the migration process, such variability was extracted to more general aspects, being reused across cluster features.

For features in the *knownIssues* cluster and for *general/multilanguage*, features are mapped to a single aspect; however, if the feature is not selected in the SPL instance, then another aspect should be included in the instance. This shows that mapping

Table 6.9: SPL configuration knowledge

Feature	**Aspect**
device/graphics/canvas/Midp2Canvas	device.graphics.canvas.Midp2CanvasAspect.aj
	device.graphics.canvas.PaintCanvasGraphics.aj
	device.graphics.canvas.Midp2_SiemensCanvasAspect
device/graphics/canvas/NokiaCanvas	device.graphics.canvas.NokiaCanvasAspect.aj
	device.graphics.canvas.RepaintCanvasAspect.aj
device/graphics/canvas/SiemensCanvas	device.graphics.canvas.SiemensCanvasAspect.aj
	device.graphics.canvas.PaintCanvasGraphics.aj
device/keys/KeysNokiaSonyEricsson	device.keys.DeviceKeysNokiaSonyEricsson.aj
device/keys/KeysMotorola	device.keys.DeviceKeysMotorola.aj
device/keys/KeysSiemens	device.keys.DeviceKeysSiemens.aj
device/screen/Screen128x117	device.screen.Screen128x117.aj
	Screen128x117_128x160_128x149_128x128
	device.screen.LowEndScreen
device/screen/Screen128x128	device.screen.Screen128x128.aj
	Screen128x117_128x160_128x149_128x128
	device.screen.LowEndScreen
device/screen/Screen128x149	device.screen.Screen128x149.aj
	Screen128x117_128x160_128x149_128x128
	device.screen.LowEndScreen
device/screen/Screen128x160	device.screen.Screen128x160.aj
	Screen128x117_128x160_128x149_128x128
	device.screen.LowEndScreen
device/screen/Screen132x176	device.screen.Screen132x176.aj
	Screen128x117_128x160_128x149_128x128
	device.screen.LowEndScreen
device/screen/Screen176x205	device.screen.Screen176x205.aj
	Screen176x205_176x208
	device.screen.HighEndScreen
device/screen/Screen176x208	device.screen.Screen176x208.aj
	Screen176x205_176x208
	device.screen.HighEndScreen
device/screen/Screen176x220	device.screen.Screen176x220.aj
	device.screen.HighEndScreen
device/sound/policy/SoundOnDemand	device.sound.policy.SoundOnDemand.aj
device/sound/policy/SoundPreallocation	device.sound.policy.SoundPreallocation.aj
device/sound/content/TypeMID	device.sound.content.TypeMID.aj
device/sound/content/TypeMIDI	device.sound.content.TypeMIDI.aj
device/sound/content/TypeXMID	device.sound.content.TypeXMID.aj
device/sound/content/TypeXMIDI	device.sound.content.TypeXMIDI.aj

device/sound/api/PlayerMMAPI	device.sound.api.SoundPlayerMMAPI.aj
device/sound/api/PlayerNokia	device.sound.api.SoundPlayerNokia.aj
device/sound/api/PlayerSiemens	device.sound.api.SoundPlayerSiemens.aj
device/sound/api/PlayerSamsung	device.sound.api.SoundPlayerSamsung.aj
device/sound/thread/ThreadedSound	device.sound.thread.ThreadedSoundAspect.aj
device/sound/thread/BlockingSound	device.sound.thread.BlockingSoundAspect.aj
Multiplayer/arena	feature.arena.ArenaAspect.aj
	ArenaScreenXY
knownIssues/Garbage collector	knownIssues.MemoryHasGC.aj
[if not selected]	knownIssues.MemoryNoGC.aj
knownIssues/Softkey	knownIssues.HasSoftkeyAspect.aj
[if not selected]	knownIssues.NoSoftkeyAspect.aj
general/multilanguage	general.multilanguage.MultiLanguageEnabled.aj
[if not selected]	general.multilanguage.MultiLanguageDisabled.aj
general/debug	general.debug.DebugMode.aj

in this cluster is not compositional, which arises due to the intimate dependency between the game core and aspects implementing extension behavior. A compositional mapping would be one in which, for example, if feature *knownIssues/softkey* were not selected, then aspect `HasSoftKeyAspect` would not be selected and *no other* aspect –like `NoSoftkeyAspect`– would be included in the instance for that feature. Principles such as XPIs [120, 64] and EJPs [84] can control lack of compositionality, but in general this might constrain functional approaches such for feature composition, such as stepwise refinement [23].

Feature Arena is mapped to a pair of aspects. The second aspect depends on the specific feature selected under the *device/screen* cluster. We noted this by *ArenaScreenXY* in the configuration knowledge, since for each screen dimension XY, there is one such aspect, according to migration strategy Feature dependency. This reflects feature dependency/constraint between these features, which was already present in the feature model, as mentioned in the beginning of Section 6.2. Tools like pure::variants [121] can express dependencies both at the feature model level and at the configuration knowledge.

Each of the 16 Best Lap SPL instances are synthesized according to this configuration knowledge and to the feature configuration of each product presented previously in Section 6.2. For example, the `MOT1` instance, whose feature configuration was presented previously, can be synthesized as follows:

```
MOT1 = CoreClassesCC •

{SoundPreallocation, ThreadedSoundAspect, DeviceKeysMotorola, LoopVariablesCommonInitialization,
SoundPlayerMMAPI, StopPrefetchedSound,  ContentTypeMID, Midp2CanvasAspect,
Midp2_SiemensCanvasAspect, Screen128X117, LowEndScreen, Screen128x117_128x160_128x149_128x128_notS40,
ArenaAspect, ArenaScreen128x117, BuildIdMOT1, HasSoftkeyAspect, DebugMode, MultiLanguageDisabled,
NoLowHeap, MemoryHasGC}
```

Table 6.10: Reuse of aspects in BestLap SPL

	# aspects	# instances	% of SPL code
Product specific aspects	29	1	11
Core aspects	17	[10,16]	6
	20	[2,9]	15
Total	66	[1,16]	32

where • denotes composition. *CoreClassesCC* denotes the core of the SPL consisting of 14 classes and still some variability with conditional compilation. *CoreClassesCC* was not mentioned in Table 6.9, since it was used for all mandatory features. Also, such core still contains some conditional compilation tags as remarked in Section 6.3 , since some variability could not be extracted with the the migration strategies. The other elements of `MOT1` are aspects necessary for instantiating this particular instance.

Each Best Lap SPL instance consists of *CoreClassesCC* and 18 aspects on average. The instances having fewer aspects are `SAM2` and `S40`, each with 13 aspects and lacking aspects for features such as Arena and Multi-language support. In contrast, the instances having higher number of aspects are `SIEM4` and `MOT3`, each with 21 aspects, including those supporting those features.

Additionally, by analyzing the configuration equations for each of the 16 SPL instances, we can assess the reuse degree of aspects across the SPL. Table 6.10 breaks up the number of aspects according to intervals representing the number of instances in which such aspects are used. Accordingly, 26% of the aspects are used in 10 or more instances, and 30% are used in 2 to 9 instances. However, 44% of them are used in specific instances. Nevertheless, these device-specific aspects represent only 11% of the SPL LOC. Further, the SPL core contains not only *CoreClassesCC*– as already mentioned previously–, but also reusable aspects, i.e., aspects that are used in at least 2 instances (core aspects). Therefore, according to Table 6.10, the core aspects consists of 56% of the total aspects and 21% of the SPL LOC. This shows that the fine granularity of the mapping enabled tangible reuse of aspects across SPL instances.

6.2.5 Analysis

After the migration process, we also analyzed code metrics. Table 6.11 shows the LOC for each SPL in the internal columns and the LOC for each corresponding build in the outer columns. As we did for RoF, we calculate the LOC of a SPL instance as the sum of the core's LOC and the LOC of all aspects necessary to instantiate this specific product.

According to Table 6.11, LOC is slightly higher in the AO version than in the CC (Conditional Compilation) version. The same holds for each corresponding SPL instance. This is caused by the extraction of methods and aspects for each variation point, thereby increasing code size due to new declarations. Nevertheless, the AO version outperforms the CC version in terms of higher *reusability*, *locality*, *adaptability*, *plugability*, and *independent development*.

Indeed, in the CC version, variability is implemented by unnamed context-dependent code snippets, whereas in the AO version, variability is implemented by product-specific aspects as well as core aspects. Some core aspects are reused in some instances and not in

others. On the other hand, with CC, even if a code snippet is used in another instance, it has to be repeated in verbatim, which complicates adaptability and maintenance. Even product-specific aspects could eventually become core aspects during SPL evolution. Further, each aspect handles locally a specific concern, which as previously scattered and tangled in the CC version, as described in Section 6.2.2. As a consequence, adaptability is also improved. Finally, the fine granularity of the configuration knowledge involving the aspects allows enhanced plugability in instantiating the SPL.

Table 6.11: Sizes of Best Lap SPL. Exterior columns show sizes for instances; interior columns show sizes of the SPL for both the conditional compilation version and the AO version.

	LOC			
instance	OO	SPL CC	SPL OA	AO
mot1	10102			10227
mot2	10102			10279
mot3	10104			10232
s40m2	9974			10182
S40m2v3	10000			10189
s40	9894			10103
s60m1	9965			10254
s60m2	9974			10263
sam1	8781	14516	14848	8930
sam2	8700			8827
se02	9949			10174
se03	10009			10147
se04	10001			10186
siem02	9937			10169
siem3	9924			10229
siem4	9925			10235

Table 6.12 shows the application sizes collected for both SPL implementations. We can notice a jar size increase from the CC version to the AO version. This occurs both in the optimized versions (14% on average) and in the non-optimized versions (64% on average). As in the RoF case, the reason for this is the overhead generated by the AspectJ weaver on the bytecode files. Moreover, we can gain a significant reduction in the jar size when using a bytecode optimization tool [128].

Table 6.12: Application (jar) Sizes of Best Lap SPL

	CC SPL		AO SPL	
	optimized size	size	optimized size	size
MOT1	71.5	80.7	82.6	130
MOT2	83.6	92.8	94.7	142
MOT3	71.5	80.7	82.6	130
S40	64.1	73.5	74.7	121
S40M2	74.6	80.8	82.1	129
S40M2V3	71.1	80.3	82.5	130
S60M1	83.9	93.3	95.1	143
S60M2	82.8	91.9	94	141
SAM1	65.3	73.4	72.7	110
SAM2	78.9	87	84.5	120
SE02	82.7	91.8	92.7	138
SE03	71.2	80.4	81.7	128
SE04	71.1	80.3	82.2	129
SIEM02	80.3	80.3	83	129
SIEM03	71.8	81	83.1	130
SIEM04	71.1	80.2	83.1	131

6.3 Open Issues and Possible Extensions

In the Best Lap case study, there were some variations for which we could not define a migration strategy using AspectJ. In this section, we address those by showing how AspectJ's current implementation does not support them. In some cases, we provide alternative solution using other approaches; in others, we present candidate extensions to the AspectJ language.

6.3.1 Import Variation

In the performed case study, there are variations between device families that use different APIs. These APIs define types with the same name and the same interfaces to facilitate the porting task. However, those types are defined in different name spaces, since each API has its own package name. For instance, the following piece of code depicts an example of such variation. The code originally written with conditional compilation tags imports a `Sprite` type from `javax.microedition.lcdui.game` package or from `com.meantime.j2me.util.game` depending on the MIDP version it uses. The latter is used when generating a release to device families that use MIDP 2.0, and the former otherwise.

```
//#ifdef game_sprite_api_midp2
//#   import javax.microedition.lcdui.game.Sprite;
//#elif
//#   import com.meantime.j2me.util.game.Sprite;
//#endif
...
```

Since the AspectJ language in its current version (1.5) does not handle variability at the import clauses granularity, there is not a solution to migrate this conditional

compilation code to AspectJ code. One alternative for such kind of variations would be extending AspectJ with inter-type declarations that insert an import clause in a type. Another possibility would be using a transformation system [98] that uses generative techniques allowing to control such kind of elements in the source code.

This concrete example can be generalized to variations that demand different imports clauses, regardless of the types' name. The form of such problem is presented in the following piece of code.

```
...
//#if TAG_1
//#   import I_1;
//#elif TAG2
//#   import I_2;
...
//#elif TAG_n
      import I_n;
//#endif
...
```

where `TAG_1`, `TAG_2`, and `TAG_n` are conditional compilation tags that define variation code and `I_1`, `I_2`, and `I_n` are the imports expressions.

6.3.2 Superclass Constructor Call

Another example of conditional compilation code that could not be migrated to AspectJ is a call to a superclass constructor. In this example, two variants demand calling the superclass constructor with the parameter `false` if the device uses MIDP 2.0 or if it is a Siemens device; otherwise, no explicit super call is needed, thus implying an implicit to the empty superclass constructor.

```
...
  public MainCanvas() {
//#if device_graphics_canvas_midp2 ||
//#   device_graphics_canvas_siemens
//#      super(false);
//#endif
         ...
  }
...
```

AspectJ does not support such migration since an advice cannot call a constructor using neither `super` nor `this`. In fact, it is possible to write code that prevents the superclass constructor to execute, but not a code that executes one constructor instead of another.

A possible solution would be extending AspectJ to allow writing an advice that executes first in a constructor call and can call the superclass constructor or another constructor in the same class, or using the transformation system mentioned before to add such constructor call.

This issue can be generalized to any variation that demands a different superclass constructor call:

```
...
  CONSTRUCTOR(PARS) {
//#if TAG
//#   super(ARGS);
//#endif
      ...
  }
```

or a change in the inline calls of class constructors.

```
...
  CONSTRUCTOR(PARS) {
//#if TAG
//#   this(ARGS);
//#endif
      ...
  }
```

where `PARS` is the constructor parameter list, which can be empty, and `ARGS` is the argument list, possible empty, of the class or superclass constructor call.

6.3.3 Adding an else-if Block

Another migration issue occurs when a variation demands the insertion of new `else-if` blocks in a conditional statement. This case is common with feature variations that add new screens to the game. The code that paints the current screen must check the type of the current screen in a long `if-else-if` structure; therefore, new screen type checks are added as `else-if`'s to the end of this structure.

```
     ...
     if (this.screenMode ==
           Resources.MAIN_SCREEN_MODE_SPLASH) {
       //code
     } else if (this.screenMode ==
           Resources.MAIN_SCREEN_MODE_LOGO) {
       //code
     }
//#ifdef feature_arena_enabled
//#  else if (this.screenMode ==
//#        Resources.MAIN_SCREEN_MODE_ARENA_WELCOME) {
//#    //code
//#  } else if (this.screenMode ==
//#        Resources.MAIN_SCREEN_MODE_ARENA_LOGIN) {
//#    //code
//#  }
//#endif
```

There is no construction in AspectJ that deals with conditional statements or any similar that would address this issue. The alternative would be again using the transformation system to generate the code to be added. An AspectJ extension that intercepts conditional statements does not seam very useful, since the conditional statements are not named, which leads to ambiguity when a method has more than one conditional statement.

This issue can be generalized by the following form:

```
    ...
    if(EXP_1) {
      // code
    } else if (EXP_2) {
      // code
    }
    ...
//#ifdef TAG
    else if(EXP_n) {
      // variation
    }
//#endif
```

where `EXP_1`, `EXP_2`, and `EXP_n` are boolean expressions.

6.4 Case Studies Synthesis

After performing both case studies, we compare their results. The RoF case study assessed the extractive/reactive approach of our method, whereas the Best Lap case study assessed the migration strategies of our approach. The latter game was approximately three times bigger in LOC than the first game and also encompassed significantly more variability. Lastly, 3 instances were handled in the first case, whereas 16 were handled in the latter case.

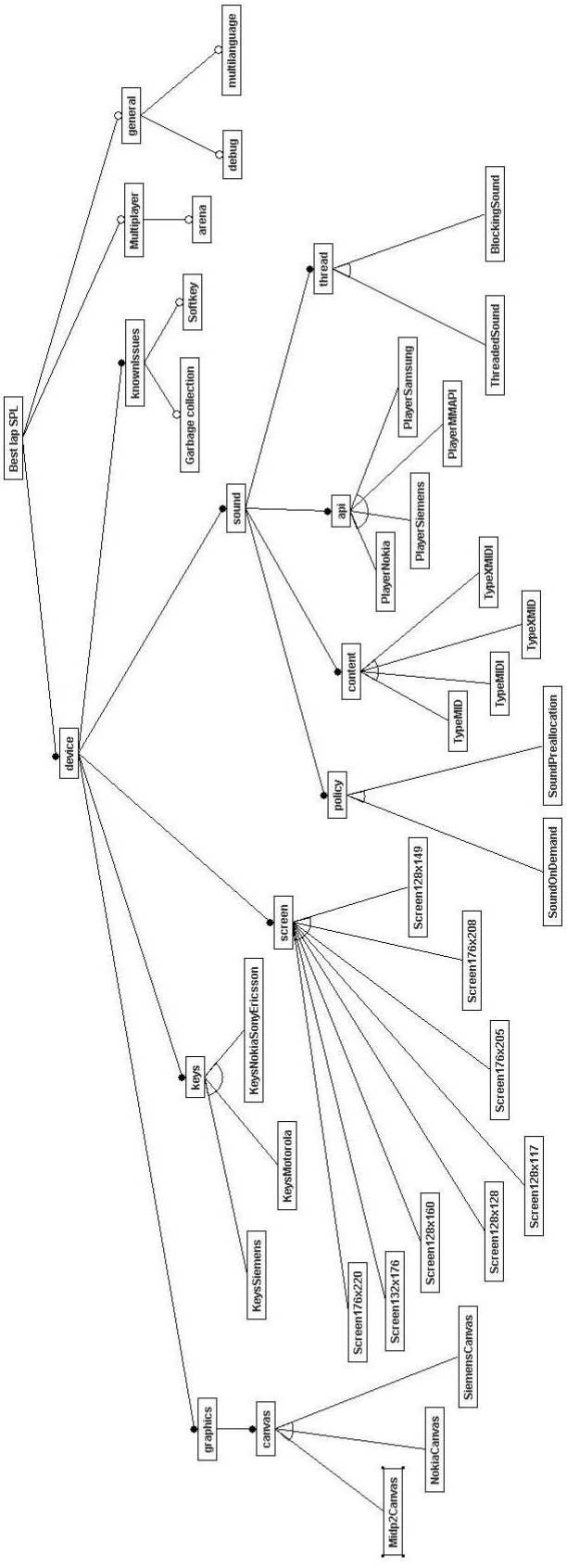
Variability in RoF was identified using concern graphs, whereas in Best Lap we relied on identifying existing preprocessing tags. In both case, we relied on tool support, which partially automated the task. Manual intervention was necessary in the first case for setting the seed and evaluating proposed edges in the graph, since the tool actually provides an approximation of the actual graphs (due to dynamic binding). In the latter case, the Eclipse-plugin used allowed fast identification of the preprocessing tags.

During extraction in the RoF case, refactoring and programming laws were employed to extract/react variability from the code guided by the identified concern graphs. In Best Lap, migration strategies were used to migrate variability from identified occurrences of preprocessing tags. Refactorings and migration strategies play a similar role, as already mentioned in Section 4.1.4: both provide a higher-level guide for either extraction or migration. Nevertheless, direct use of the programming laws may be necessary. In Best Lap, some variabilities could not be migrated, as discussed in Section 6.3; nevertheless, some extensions to AspectJ other mechanisms have been proposed to handle them.

The configuration knowledge in RoF is coarse-grained and relates sets of features to classes and aspects. There is only one reusable aspect and this represents a reduce fraction of the SPL LOC. In contrast, The configuration knowledge in Best Lap is fine-grained, relating each aspect to one aspect and to a set of aspects. This fine granularity is associated with a significant number of reusable aspects, and these compose a tangible amount of SPL LOC. Additionally, the configuration knowledge in Best Lap has constraints and some non-compositional mappings.

Regarding LOC sizes, in RoF, considerable reuse was achieved, which as expected since a SPL was extracted from existing applications. In Best Lap, the LOC of each SPL implementation is approximately the same, but the AO implementation has improved *reusability*, *locality*, *adaptability*, *plugability*, and *independent development*. Additionally, in both cases, increase on Jar size was noticed, but that has not prevented installation of the games in the phones and their proper functioning. We are currently studying how optimization techniques can further minimize this potential issue.

Figure 6.4: Variability within Best Lap



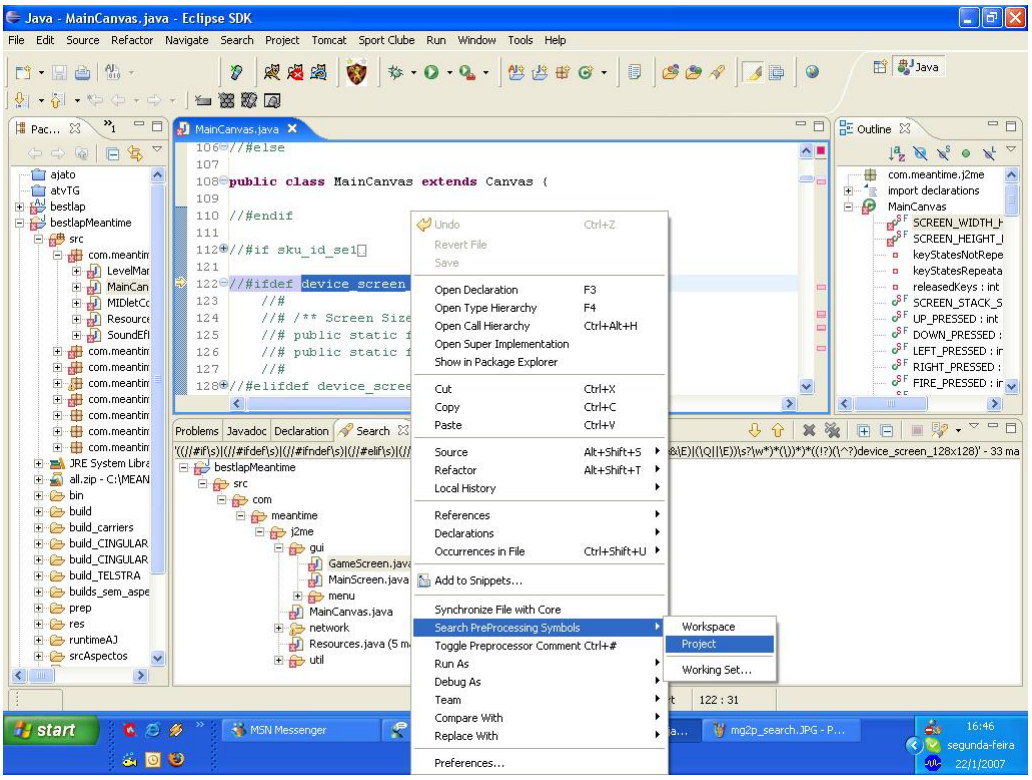


Figure 6.5: Plug-in for identifying variability in the conditional compilation SPL.

Chapter 7

Conclusion

Software Product Line is a promising process framework for developing a set of products scoped within a market segment and based on common artifacts. Potential benefits are large scale reuse and significant boost in productivity. An incurred key challenge, however, is handling adoption strategies, whereby an organization decides to start the SPL from scratch, bootstrap existing products into a SPL, or evolve an existing SPL. Our research brings contributions in this context.

First, we reviewed existing SPL processes and adoption strategies and identified that the extractive and reactive adoption strategies, which are very useful in practice, lack suitable support at the implementation and at the feature model level. In either case, variability management is a central issue. Accordingly, we presented an original state-of-the-art on the implementation of software product line variability. We set a comparison framework based on domain analysis of the solution space and then evaluated a number of techniques for implementing product line variability according to this framework. We also compared these against each other, assessing their relative advantages and drawbacks. Some of these techniques are already in widespread use in industry, whereas other are still more academic.

Next, we presented an original method for creating and evolving product lines. Contrary to the proactive adoption strategy, our method relies on a combination of the extractive and the reactive approaches. Our method first bootstraps the SPL and then evolves it with a reactive approach. Initially, there may be one or more independent products, which are refactored in order to expose variations to bootstrap the SPL. Next, the SPL scope is extended to encompass another product: the SPL reacts to accommodate the new variant. During this step, refactorings are performed to maintain the existing product, and a SPL extension is used to add a new variant. The SPL may react to further extension or refactoring. Alternatively, there may be an existing SPL implemented with a variability mechanism from which we may want to migrate. During such activities, the feature model as well as the configuration knowledge evolve and need to be handled.

The method is systematic because it relies on a collection of provided refactorings at both the code level and at the feature model level. Such refactorings are described in terms of templates, which are a concise and declarative way to specify program transformations. In addition, refactoring preconditions (a frequently subtle issue) are more clearly organized and not tangled with the transformation itself. Furthermore, the refac-

torings can be systematically derived from more elementary and simpler programming laws or feature model transformation laws. These laws are appropriate because they are considerably simpler than most refactorings, involving only localized changes, with each one focusing on a specific language construct. Therefore, they are easier to reason about than the refactorings, increasing correctness confidence in such extractive transformations. This is specially relevant because it reduces the burden on testing, which is extremely expensive in the SPL scenario.

Our program refactorings rely on AOP to modularize crosscutting concerns, which often occur in SPLs. Furthermore, in order to perform extractive and reactive tasks at the feature model, we extended the traditional concept of refactoring to encompass not only code, but also feature model transformations. Such transformations should either preserve or increase variability, which can be checked with the catalogs of corresponding transformations we provided.

We evaluated our method in existing industrial-strength mobile games, which are highly variant, a key issue in SPLs. The program refactorings were evaluated in two case studies, whereas the feature model transformations were evaluated in one case study. The evaluation shows that, in the extractive adoption strategy, we can benefit from extensive code reuse and that, in the reactive scenario, we can evolve the SPL to encompass other products while still maintaining code reliability. The method also provided useful in migrating a SPL implemented with one variability mechanism to another variability mechanism, which resulted in improvement of reusability, locality, adaptability, plugability, and independent development. It also shows that the sequence of applied refactorings must be strategically chosen. Although increase on binary application size was noticed, that has not prevented installation of the games in the phones and their proper functioning. Further, optimization techniques can reduce this potential issue.

Although the evaluation is in the mobile game domain, we argue that the method and the issues addressed here are valid for mobile applications in general, of which mobile games are representative. We also suspect that other highly variant domains could benefit from our method.

One limitation of our method is that, since it relies on AOP, it does not handle all variability issues. For example, it does not handle very fine-grained variability, which is better addressed by conditional compilation. Nevertheless, the crosscutting variability addressed by AOP is quite representative from the SPL approach. Additionally, in the cases in which variability issues cannot be addressed by AOP, we could rely on alternative techniques surveyed in our comparative analysis of variability implementation mechanisms. Furthermore, we also considered how to enhance AOP to handle additional variability issues.

Another limitation is that AOP languages still lack more semantic pointcut languages. Indeed, these are too syntactic and thus bring some complexity to our refactorings: as they are used to evolve the SPL, previously existing aspects may have to be adapted. Although this is an undesirable side effect, there is tool support from current IDEs that alleviates this problem, by showing the aspects affecting the SPL core.

Lastly, in order to apply our approach in constrained-resources domains, like the mobile game domain, we need to provide more optimized implementations for AOP weaving, since current implementations have a code bloat issue for some expressive

pointcuts.

In addition to the benefits of the method outlined above, which are the core contributions of this work, we have also designed and implemented a prototype of a tool for supporting variability management in the SPL context [14]. Currently, the tool aims at extracting variations from existing products, by isolating such variations into aspects, which in turn customizes the incrementally emerging SPL core. The tool is an Eclipse plug-in and currently implements a subset of the refactorings discussed in Chapter 4. The prototype is being extended in the context of research projects with which we collaborate.

7.1 Future Work

Our method relies on the existence of a configuration knowledge mechanism. Although such mechanism is not the focus of our work, we still illustrated how it can be used in the case studies, where we identified some important issues that need to be considered further in such mechanism. Issues include proper handling of feature interaction and lack of compositionality. Additionally, such mechanism, as a tracing mechanism, should also support explicit traceability among variations points of feature models and variation points of implementation artifacts or variation points of whatever artifacts whose level of abstraction is in between those two. This is important for enabling model-driven development of SPLs and is, in fact, one of the goals of ongoing research [1].

In particular, establishing the traceability of variability between tests and other SPL artifacts is of paramount importance. Indeed, managing variability in test artifacts in the SPL context is very challenging and resource consuming [106]. For example, in the Federated Database Domain [6], where one needs to develop wrappers to connect one data source with another (invariably addressing complex issues such as transaction control), different instantiations of the (coding) framework requires different test scripts. Although there is great similarity among such scripts, these are still mostly ported/adapted manually. Yet, little in-depth research has been conducted in this area.

Variability management should also encompass domain specific artifacts. In the mobile games domain, for example, handling variability within art artifacts, such as game soundtrack and images, is time-consuming and requires specific techniques [15]. Some of these can be drawn on analogy with architectural design capabilities such as composition. However, considerably more research needs to be done in this area.

Although we argue that the method and the issues addressed here are valid for SPLs in general, it is important to verify this with case studies and experiments in other domains that might benefit from a SPL approach.

As a future work, we aim at applying our approach in more case studies, in order to assess the benefits and limitations of this work in additional real projects. Moreover, from these case studies we can propose more FM refactorings, and make considerations on their usefulness.

Future work should also address the definition of metrics for SPLs. Although they do exist for AOP [56], their goal focuses on traditional and relevant software engineering principles such as cohesion and coupling. Nevertheless, in the SPL context, these concepts should be refined as well as complemented by others such as configurability

and plugability. Such refinement and extension could lead to the definition of a metric suite for SPLs at different level of abstractions, involving architecture, code, and tests, for example. To the best of our knowledge, the SPL community still lacks a clear vision of this issue.

Moreover, another point for further research is extending the prototype supporting variability management which we have developed. In order to support fully support our method, the prototype should also be integrated with a feature model tool. Our initial investigation has shown that this is feasible, for example with pure::variants [121]. In a typical scenario, the developer using the tool during the extractive talks would select variability points in the code, execute a corresponding refactoring provided by the too and then evaluate the effect at the feature model level, in a feature model view.

7.2 Related Work

Our research is in the convergence of a number of areas involving SPLs, AOP, refactoring, programming laws, model refactoring, and portability of mobile games. In the next sections, we compare our work to research in recurring combinations of these areas.

7.2.1 AOP and SPLs, and Refactoring

Prior research also evaluated the use of AOP for building J2ME product lines [16]. We complement this work by considering the implementation of more features in an industrial-strength application, explicitly specifying the refactorings to build and evolve the SPL, and raising issues in AspectJ that need to be addressed in order to foster widespread application in this domain. Additionally, we rely on concern graphs [109] to identify variant features. Concern graphs provide a more concise and abstract description of concerns than source code. Once the concern is identified, we extract it into an aspect and may further revisit it during SPL evolution.

AOP refactorings have also been described elsewhere [95, 68]. The former proposes a catalog for object-to-aspect and aspect-to-aspect refactorings, whereas the latter provides an abstract representation of object-to-aspect refactorings as roles. However, their use in the SPL setting is not explored, and the refactorings format follows the imperative style [53]; in contrast, our approach is template-oriented, abstract, concise, and thus does not bind a specific implementation, which could be done, for instance, with a transformation system receiving as input refactoring templates.

In another approach, a language-independent way to represent variability is provided, and it is shown how it can be used to build J2ME game SPLs product lines [130]. Our approach differs from such work because, although ours relies on language-specific constructs, it has the advantage of not having to specify join points in the base. Moreover, their approach, despite language-independent, considerably complicates understanding the source code due to the tags introduced to represent variability.

A recent work [129] reports the AOP refactoring of a middleware system to modularize features such as client-side invocation, portable interceptors, and dynamic types. Nevertheless, such work does not describe the refactorings abstractly and does not attempt to express them in terms of simpler programming laws as a way to guarantee

behave preservation, as we do.

Colyer et al [42] present principles for creating flexible, configurable, and aspect-oriented systems. These principles are illustrated in a SPL in the middleware domain. In particular, that work assumes that features added to a base system should be orthogonal to the base system. However, unlike our work, those principles do not provide explicit guidelines at the implementation level and at the feature model level for extracting and reacting a SPL. Further, we do not assume that features added to a base system should be orthogonal, since the variability may dictate some contract between the base and extensions. Also, the non-compositionality identified in the configuration knowledge of the case study in Section 6.2.4 suggest that feature interactions may violate their assumption. In this respect, the work by Kulesza et al [84] build on the notion of crosscutting interfaces [120, 64] to define contracts such that both aspects, representing SPL extensions, and the SPL core evolve independently.

7.2.2 Programming Laws and Model Refactoring

Previous work [40] presented 30 aspect-oriented programming laws and showed how these could derive some aspect-oriented refactorings. In our work, we have explored the usefulness of such approach in validating extractive and reactive refacotorings for building product lines in the mobile game domain. Additionally, this task prompted not only an extension of the number of laws initially proposed, but also a more careful description of some subtle issues of these laws, such as handling AspectJ's precedence semantics, which were skipped in the original work. Finally, the experience in using the laws during derivation suggested that these be organized in a more concise notation, which could lead to the implementation of a generative library.

The process of defining programming laws and showing how these can be used to derive refactorings has also been addressed for for object-oriented languages [31]. Such research additionally formally proves not only the completeness of such set of laws, but also the correctness of each law, by relying on a weakest precondition semantics [37]. Our work, despite not formally proving the laws, still benefits from understanding coarse-grained transformations in terms of simpler ones.

In a related work, Gheyi et al. encoded a semantics for FMs in the Prototype Verification System (PVS) [103], which is a formal specification language. Using the PVS theorem prover, they proved all refactorings proposed with respect to a formal semantics [57]. This experience in PVS was very important for proposing other FM refactorings. Proving them increases the knowledge about other transformations that do not improve configurability. The PVS prover gives insights of what need to be considered. The formalization and proofs are important in order to increase the reliability when refactoring SPLs, as we present in Section 5.5.

In Chapter 5, we proposed the extension of the concept or refactoring to SPL so that it also takes into account a transformation in the feature model level and argued that such transformation preserves or increases an important property (configurability). Similar work has been done for refactoring object models [58]. In such work, model transformations have been defined for a formal object-oriented modelling language and expressed in terms of elementary laws. An equivalence notion [59] was established and such laws have been proved [60, 61] sound with respect to it using PVS [102], which

encompasses a formal specification language and a theorem prover.

7.2.3 Refactoring Product Lines

A related approach proposes [85] Feature Oriented Refactoring (FOR), which is the process of decomposing a program, usually legacy, into features. Such work focuses on configuration knowledge, specifying the relationships between features and their implementing modules, backed by a solid theory. Also, the authors present a semi-automatic refactoring methodology to enable the decomposition of a program into features. However, FOR focuses so far on bootstrapping a SPL from an *existing* application, rather than two or more existing products, as we explore in our work. Further, our approach of verifying SPL improvement addresses this requirement by allowing us to evaluate the impact of SPL refactorings based on our theory for reasoning on *feature models*. Nevertheless, we believe that their theory for relating features and implementation modules may be complementary to ours for more ambitious applications of FM refactorings: given a systematic way of mapping FM constructs to software components, we can infer SPL refactorings on programs from analogous refactorings on corresponding feature models. Therefore, a model-driven approach to SPL refactoring would thus become feasible.

Another work [44] explores the application of refactoring to SPL Architectures. They present metrics for diagnosing structural problems in a SPL Architecture, and introduce a set of architectural refactorings that can be used to resolve those problems. These metrics can be useful for detecting bad smells. In contrast to our work, they apply the traditional notion of refactoring. Also, they do not propose a set of refactorings for FMs as in our work. A similar work [80] shows a case study in refactoring legacy applications into SPL implementations. They defined a systematic process for refactoring products, in order to define SPLs. However, configurability of the resulting SPLs are only checked by testing; we believe that our approach for verifying configurability with FM refactorings can be useful in this process, especially when extracting a SPL from more than one product or SPL, improving reliability in the process. Also, they do not deal with refactoring in reactive contexts.

Batory [21] presents a semantics for FMs, connecting it to grammars and propositional formulas. The connection between FMs and propositional formulas enables the use of SAT solvers to perform a finite number of analysis. We specified a similar semantics for FM considering the same propositional formulas. We additionally argued for the correctness of a number of refactorings. As mentioned by such author, FMs do not have unique representations as feature diagrams. By using our bidirectional refactorings, we show a reduction strategy (completeness result) that relates any two FM semantically equivalent. He is concerned with building a tool for FMs for checking specific properties; In our case, we can specify and prove not only specific but any kind of general property that holds for FMs.

Extensions to cardinality-based FMs can be found elsewhere [46], including a formal semantics to FMs with these features, translating FMs into context-free grammars. Our semantics could also be extended similarly, and new FM refactorings can be proposed for dealing with such cardinality. We also note that their formal treatment of FM specialization could be seen as the opposite of our notion of FM refactoring, except for the fact that our refactoring notion also relates multiple FMs.

Another work [122] proposes a textual language for describing features. Their language is similar to ours, but do not consider propositional formulas. They propose a notion of FM semantics that is equivalent to ours. Also, a set of fifteen rules relating equivalent FMs are proposed, which are very similar to our bidirectional refactorings. All proposed rules can be derived using our bidirectional refactorings following the reduction strategy (completeness result). These rules are not proven to be complete, as in our work. Moreover, we show that our set of bidirectional refactorings is minimal. So, one B-Refactoring cannot be derived using another B-Refactoring. If they had proposed a more general rule, similar to a refactoring for introducing or removing formulas, their Rules 1-4, for instance, would be derived from this more general rule. So, our set of B-Refactorings is more concise (minimal) and contain less transformations. Moreover, they do not use their FM refactorings to apply refactorings in SPL, as in our work. Their work also presents an option for configuration knowledge, mapping features to classes; our approach for verifying configurability improvement in Section 5.5 can be used in the presence of such option.

Another work [25] extends FMs in order to include constraints. They can automatically analyze five properties in this language, such as number of instances of a FM. However, they do not propose a set of refactorings for FM and use them to refactor SPL.

If high-level algebraic specification of products were available, as described in [86], an efficient optimization algorithm could be applied in order to extract the product line core from these specifications with the Shared Class Extractor operator [86]. However, the hypothesis of having this high-level specification may not be met in practice, in such a way that the domain engineer would need to address handling legacy software directly at the design or at the implementation level.

Chen et al [38] propose a purely extractive approach to constructing feature models based on requirements clustering, which automates the activities of feature identification, organization, and variability modeling to a great extent. The underlying idea of this approach is to analyze the relationships between individual requirements and cluster tight-related requirements into features. Algorithms are presented for constructing a feature diagram from requirements and for merging application feature diagrams into a SPL feature diagram. However, their FM meta-model does not encompass alternative feature like ours; further, our approach also handles implementation assets.

7.2.4 Portability of Mobile Games

Current approaches to porting can be classified in the following categories: preprocessing tools, general guidelines, specific guidelines, semi-automatic services, and formal approaches.

Tools like Antenna [126] and J2ME Polish [127] provide a preprocessing feature by which guidelines define a conditional compilation of the source code (written to comprise several platforms) according to the device in question. Besides that, J2ME Polish contains a device database (described with their peculiarities), which is used in the process of instantiating a specific variation. However, the use of compilation directives may compromise source code legibility, as we described in Section 3.7. Also, it solves variability at pre-compilation time, whereas our approach with AOP has compilation

binding time. Accordingly, another significant disadvantage with preprocessing is that refactoring tools may not be applied with it.

Some approaches are specific to source and target devices, and consist of a descriptive document of their characteristics [97]. They specify the direction (source/target devices) of portability, but are more descriptive in terms of device features than prescriptive in terms of actually carrying out the porting. Other approaches offer broader guidelines [49], involving a research of the target device, an architecture reorganization and source code transformation, but underestimate the effort necessary for this last task. On the other hand, our approach could be applied to porting games across different devices; additionally, it provides concrete guidelines for the porting process by relying on a refactoring catalog.

A more recent approach [124] consists of specifying reference devices and specific guidelines to programming for these devices, and then generating the code for the target device with tool support. This approach is described as automatic, but demands that the game be coded according to the guidelines, which may itself be a resource demanding task. Similar guidelines may also be required in our case, before we apply our refactorings. These guidelines could be useful, for instance, for minimizing very-fine grained variability, which otherwise would have to be handled with conditional compilation.

Some formal approaches [54, 36, 70] propose an abstract specification of the elements of Graphical User Interface (GUI), devices characteristics, and user interface usage scenarios. Based on these, they generate code for different types of GUI. Unfortunately, such approaches depend on hypotheses which restrain the GUI's organization, have a considerable specification effort and address only GUI, not taking into consideration issues like heap memory and maximum application size constraints, which we use to evaluate our approach.

In previous work, a language-independent way to represent porting-related variability is provided, and it is shown how it can be used to port J2SE applications to a J2ME product line [131]. This is similar to the program transformation approach we describe, but differs in that ours relies on language-specific constructs and variation points are identified in the program transformation language, whereas the latter is language independent, but requires the developer to explicitly specify the variation points in the base code.

Bibliography

[1] *AMPLE Project*. http://www.ample-project.net/, March 2007.

[2] Vander Alves. Identifying variations in mobile devices. *Journal of Object Technology*, 4(3):47–52, April 2005.

[3] Vander Alves and Paulo Borba. An Implementation Method for Distributed Object-Oriented Applications. In *XV Brazilian Symposium on Software Engineering*, pages 161–176, Rio de Janeiro, Brazil, October 2001.

[4] Vander Alves and Paulo Borba. Aspects and software product lines. In *Brazilian Workshop on Component-Based Development - Tutorial (in Portuguese)*, Joao Pessoa, Brazil, September 2004.

[5] Vander Alves, Ivan Cardim, Heitor Vital, Pedro Sampaio, Alexandre Damasceno, Paulo Borba, and Geber Ramalho. Comparative analysis of porting strategies in J2ME games. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 123–132, Budapest, Hungary, September 2005. IEEE Computer Society.

[6] Vander Alves, Curt Cotner, Mary Roth, Morgan Tong, and Daniel Wolfson. *Systems, methods, and computer program products to integrate user-defined operations into a database transaction*. http://www.uspto.gov/, 2005. US Patent Application number 20050108255.

[7] Vander Alves, Ayla Dantas, and Paulo Borba. AOP-driven variability in product lines of pervasive computing applications. In *GPCE'03 - Poster Session*, Erfurt, Germany, September 2003.

[8] Vander Alves et al. *Tool and Process for Porting Mobile Games (in Portuguese)*. FACEPE-funded project.

[9] Vander Alves et al. *Tools for Developing Mobile Game Software Product Lines (in Portuguese)*. FINEP-funded project.

[10] Vander Alves et al. *Investigating the Development of Mobile Application Software Product Lines (in Portuguese)*, 2005. CNPq-funded project.

[11] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *Proceedings of the 5th ACM International Conference on Generative Programming and Component Engineering (GPCE'06)*. ACM Press, Oct 2006.

[12] Vander Alves, Pedro Matos Jr., and Paulo Borba. An incremental aspect-oriented product line method for J2ME game development. In *Workshop on Managing Variability Consistently in Design and Code at OOPSLA'04*, Vancouver, Canada, October 2004.

[13] Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and evolving mobile games product lines. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *Lecture Notes in Computer Science*, pages 70–81, Rennes, France, Sep 2005. Springer-Verlag.

[14] Vander Alves, Pedro Matos Jr, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. *Transactions on Aspect-Oriented Software Development (TAOSD): Special Issue on Software Evolution*, 2007. Accepted for publication, to appear.

[15] Vander Alves, Gustavo Santos, Fernando Calheiros, Vilmar Nepomuceno, Davi Pires, Alberto Costa Neto, and Paulo Borba. Beyond code: Handling variability in art artifacts in mobile game product lines. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms Workshop, in conjunction with the 10th International Software Product Line Conference (SPLC 2006)*, Baltimore, USA, Aug 2006.

[16] M. Anastasopoulos and D. Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *Proceedings of the International Conference on Software Reuse (ICSR)*, 2004.

[17] Michalis Anastasopoulos and Cristina Gacek. Implementing product line variabilities. In *Symposium on Software Reusability*. ACM Press, May 2001.

[18] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Overview of caesarj. *Transactions on AOSD I, LNCS*, 3880:135 – 173, 2006.

[19] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jurgen Wst, and Jorg Zettel. *Component-based Product Line Engineering with UML*. Addison Wesley, 2002.

[20] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotk, Ondej Lhotk, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[21] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference of Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[22] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, 1992.

[23] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[24] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[25] David Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE)*, 3520:491–503, 2005.

[26] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189. ACM Press, 2005.

[27] Alexandre Bergel, Robert Hirschfeld, Siobhàn Clarke, and Pascal Costanza. Aspectboxes – controlling the visibility of aspects. In Markus Helfert Joaquim Filipe, Boris Shiskov, editor, *In Proceedings of the International Conference on Software and Data Technologies (ICSOFT 2006)*, pages 29–38, September 2006.

[28] Grady Booch. Object-oriented development. *IEEE Trans. Software Eng.*, 12(2):211–221, 1986.

[29] Grady Booch. The limits of software, 2002. Software Development Forum, PARC,Palo Alto,CA.

[30] Grady Booch, Ivar Jacobson, and James Rumbaugh. *Unified Modeling Language — User's Guide.* Addison-Wesley, 1999.

[31] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52:53–100, oct 2004.

[32] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[33] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

[34] Tarcisio Camara, Rodrigo Lima, Rangner Guimaraes, Alexandre Damasceno, Vander Alves, Pedro Macedo, and Geber Ramalho. Massive mobile games porting: Meantime study case. In *Brazilian Symposium on Computer Games and Digital Entertainment - Computing track*, Recife, Brazil, 2006.

[35] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[36] Richard Cardone, Adam Brown, Sean McDirmid, and Calvin Lin. Using mixins to build flexible widgets. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 76–85. ACM Press, 2002.

[37] Ana Cavalcanti and David Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Transactions on Software Engineering*, 26(8):713–728, August 2000.

[38] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. An approach to constructing feature models based on requirements clustering. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 31–40, Paris, France, September 2005. IEEE Computer Society.

[39] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[40] Leonardo Cole and Paulo Borba. Deriving refactorings for AspectJ. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented software development*, pages 123–134. ACM Press, 2005.

[41] Leonardo Cole, Paulo Borba, and Alexandre Mota. Proving aspect-oriented programming laws. In *Foundations of Aspect-Oriented Languages Workshop at the 4th International Conference on Aspect-oriented software development*, pages 1–9. Iowa State University Technical Report, 2005.

[42] Adrian Colyer, Awais Rashid, and Gordon Blair. On the separation of concerns in program families. Technical Report COMP-001-2004, Lancaster University, 2004.

[43] James O. Coplien. *Multi-Paradigm Design*. PhD thesis, Vrije Universiteit Brussel, 2000.

[44] Matt Critchlow, Kevin Dodd, Jessica Chou, and André van der Hoek. Refactoring product line architectures. In *IWR: Achievements, Challenges, and Effects*, pages 23–26, 2003.

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

[46] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[47] Ayla Dantas, Paulo Borba, and Vander Alves. Using aspects to structure small devices applications. In *First Workshop on Reuse in Constrained Environments (RICE'03) at the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, Anaheim, CA, USA, October 2003.

[48] Edsger Dijkstra. Notes on structured programming. In *Structured Programming*. Academic Press, 1968.

[49] X. Facon. *Porting Your MIDlets to New Devices*. World Wide Web, http://www.microjava.com/articles/techtalk/, 2004.

[50] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Mehmet Akşit, Siobhan Clarke, Tzilla Elrad, and Robert E. Filman, editors, *Aspect-Oriented Software Development*. Addison-Wesley, Reading, MA, 2004.

[51] Nokia Forum. *Developing Java Games for Platform Portability. Case Study: Miki's World*. World Wide Web, 2004.

[52] Nokia Forum. *Develop/Optimize Case Study: Macrospace's Dragon Island*. World Wide Web, http://www.forum.nokia.com/main/1,,040,00.html?fsrParam=2-3-/main.html&fileID=5412, 2004.

[53] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[54] Krzysztof Gajos and Daniel S. Weld. Supple: automatically generating user interfaces. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interface*, pages 93–100. ACM Press, 2004.

[55] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[56] Alessandro Garcia, Cláudio SantAnna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: A quantitative study. *Transactions on Aspect-Oriented Software Development (TAOSD)*, pages 36–74, 2006.

[57] Rohit Gheyi, Vander Alves, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Theory and proofs for feature model refactorings. Technical Report TR-UFPE-CIN-200608027, Federal University of Pernambuco, 2006.

[58] Rohit Gheyi, Tiago Massoni, and Paulo Borba. Basic laws of object modeling. In *Third Specification and Verification of Component-Based Systems (SAVCBS), affiliated with ACM SIGSOFT 2004/FSE-12*, pages 18–25, Newport Beach, United States, oct 2004.

[59] Rohit Gheyi, Tiago Massoni, and Paulo Borba. An abstract equivalence notion for object models. *Elsevier's Electronic Notes in Theoretical Computer Science, Proceedings of Brazilian Symposium on Formal Methods*, 130:3–21, may 2005.

[60] Rohit Gheyi, Tiago Massoni, and Paulo Borba. A rigorous approach for proving model refactorings. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages xx–yy, Long Beach, United States, nov 2005.

[61] Rohit Gheyi, Tiago Massoni, and Paulo Borba. Type-safe refactorings for alloy. In *Eight Brazilian Symposium on Formal Methods (SBMF)*, Porto Alegre, Brazil, nov 2005.

[62] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[63] Martin Griss, John Favaro, and Massimo d'Alessandro. Integrating feature modeling with the rseb, 1998.

[64] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Softw.*, 23(1):51–60, 2006.

[65] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WISCA'01)*, pages 45–54, Amsterdam, The Netherlands, August 2001.

[66] Oberschulte C. Hanenberg S. and Unland R. Refactoring of aspect-oriented software. In *Net.ObjectDays*, Erfurt, Germany, September 2003.

[67] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press.

[68] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, New York, NY, USA, 2005. ACM Press.

[69] Charles Antony Richard Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.

[70] Hao hua Chu, Henry Song, Candy Wong, Shoji Kurakake, and Masaji Katagiri. Roam, a seamless application framework. *Journal of Systems and Software*, 69(3):209–226, 2004.

[71] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[72] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[73] Stan Jarzabek and Li Shubiao. Eliminating redundancies with a "composition with adaptation" meta-programming technique. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 237–246, New York, NY, USA, 2003. ACM Press.

[74] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 63–76. ACM Press, 1992.

[75] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[76] 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. *Ann. Softw. Eng.*, 5:143–168, 1998.

[77] Gregor Kiczales. Aspect–oriented programming. *ACM Computing*, 28(154), December 1996.

[78] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[79] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, Finland, June 1997. Springer-Verlag.

[80] Ronny Kolb, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 369–378. IEEE Computer Society, 2005.

[81] Charles Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering.*, pages 282–293, Bilbao, Spain, October 2001.

[82] Charles Krueger. Variation management for software production lines. In *Proceedings of the 2nd International Software Product Line Conference (SPLC'02)*, pages 37–48, San Diego, California, August 2002. Lecture Notes in Computer Science (LNCS).

[83] Charles Krueger. Towards a taxonomy for software product lines. In *Proceedings of the 5th International Workshop on Software Product Family Engineering*, Siena, Italy, November 2003.

[84] Uirá Kulesza, Vander Alves, Alessandro Garcia, Carlos J. P. de Lucena, and Paulo Borba. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In *Proceedings of the 9th International Conference on Software Reuse (ICSR-9)*, Lecture Notes in Computer Science, pages 231–245. Springer-Verlag, Jun 2006.

[85] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th International Conference on Software Engineering*, pages 112–121. ACM Press, 2006.

[86] Jia Liu and Don S. Batory. Automatic remodularization and optimized synthesis of product-families. In *GPCE*, pages 379–395, 2004.

[87] Cristina Lopes and Gregor Kiczales. Recent developments in AspectJ. *Workshop on Aspect–Oriented Programming at ECOOP'98*, July 1998.

[88] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA, 2006. ACM Press.

[89] Roberto E. Lopez-Herrejon, Don Batory, and William R. Cook. Evaluating support for features in advanced modularization technologies. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*.

[90] L. Moura M. F. Fontoura, C. Braga and C. J. Lucena. Using domain specific languages to instantiate object-oriented frameworks. *IEE Proceedings - Software*, 147(4):109–11, 2000.

[91] Tiago Massoni. A Software Process with Progressive Implementation Support (in portuguese). Master's thesis, Informatics Center (CIn) — Federal University of Pernambuco (UFPE) — Brazil, February 2001.

[92] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. *SIGSOFT Softw. Eng. Notes*, 29(6):127–136, 2004.

[93] Sun MicroSystems. *J2ME Optional Packages*. World Wide Web, http://developers.sun.com/techtopics/mobility/midp/-articles/optional/, 2004.

[94] Sun MicroSystems. *Java 2 Platform, Micro Edition (J2ME)*. World Wide Web, http://java.sun.com/j2me/, 2004.

[95] Miguel P. Monteiro and Joao M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 111–122, New York, NY, USA, 2005. ACM Press.

[96] Motorola. *Porting guide: Motorola i95cl to T720*. World Wide Web, http://www.microjava.com/articles/MJN, 2004.

[97] Motorola. *Porting guide: Motorola i95cl to T720*. World Wide Web, http://www.microjava.com/articles/-MJN_Porting.Guide_i95cl-T720.pdf, 2004.

[98] Federal University of Pernambuco. *JaTS - Java Transformation System*. World Wide Web, http://www.cin.ufpe.br/~jats/, 2001.

[99] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[100] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying suject–oriented composition. *TAPOS*, 2(3):179–202, 1996. Special Issue on Subjectivity in OO Systems.

[101] Harold Ossher and Peri Tarr. Using subject–oriented programming to overcome common problems in object–oriented software development/evolution. In *International Conference on Software Engineering, ICSE'99*, pages 698–688. ACM, 1999.

[102] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. The PVS language reference version 2.3. In *SRI International*, Technical Report, 1999.

[103] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Germany, 1998. Springer-Verlag.

[104] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications ACM*, 15(12):1053–1058, 1972.

[105] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[106] Klaus Pohl and Andreas Metzger. Software product line testing. *Commun. ACM*, 49(12):78–81, 2006.

[107] Wolfgang Pree. *Design patterns for object-oriented software development*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[108] Java Community Process. *Mobile Information Device Profile 2.0*. World Wide Web, http://jcp.org/aboutJava/communityprocess/-final/jsr118/index.html, 2004.

[109] Martin P. Robillard and Gail C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *In Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, May 2002.

[110] Manuel Roman, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: a middleware platform for active spaces. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):65–67, 2002.

[111] Pedro Sampaio, Alexandre Damasceno, Igor Sampaio, Vander Alves, Geber Ramalho, and Paulo Borba. Porting games in J2ME: Challenges, case study, and guidelines (in portuguese). *Scientia*, 16(1):66–72, January/June 2005.

[112] Thiago Santos and André Santos. Parameterizing java software (in portuguese). In Martin A. Musicante and Ricardo Massa F. Lima, editors, *Proceedings SBLP'05 IX Brazilian Symposium on Programming Languages*, pages 257–270, 2005.

[113] Douglas C. Schmidt and Frank Buschmann. Patterns, frameworks, and middleware: their synergistic relationships. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 694–704, Washington, DC, USA, 2003. IEEE Computer Society.

[114] Yannis Smaragdakis and Don S. Batory. Implementing layered designs with mixin layers. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 550–570, London, UK, 1998. Springer-Verlag.

[115] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, New York, NY, USA, 1986. ACM Press.

[116] Sérgio Soares. Progressive Develpoment of Object Oriented Concurrent Programs (in portuguese). Master's thesis, Informatics Center (CIn) — Federal University of Pernambuco (UFPE) — Brazil, February 2001.

[117] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, OOPSLA'02*, pages 174–190. ACM Press, November 2002. ACM SIGPLAN Notices 37(11).

[118] Christopher Strachey. Fundamental concepts in programming languages. *Lecture notes for the International Summer School in Computer Programming*, 1967.

[119] Bjarne Stroustrup. *The C++ Programming Language (Special Edition)*. Addison–Wesley, 2000.

[120] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175, New York, NY, USA, 2005. ACM Press.

[121] Pure Systems. *Pure Variants*. World Wide Web, http://www.pure-systems.com/Variant_Management.49.0.html. Last accessed in February 2007.

[122] Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[123] David Weiss and Chi Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison Wesley, 1999.

[124] Tira Wireless. *TiraJump*. World Wide Web, http://www.tirawireless.com/jump/, 2004.

[125] J. Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI-99-TR-029, Software Engineering Institute, 1996.

[126] World Wide Web, http://antenna.sourceforge.net. *Antenna Preprocessor*, 2004.

[127] World Wide Web, http://www.j2mepolish.org. *J2ME Polish*, 2004.

[128] World Wide Web, http://proguard.sourceforge.net/. *ProGuard*, 2005.

[129] Charles Zhang and Hans-Arno Jacobsen. Resolving feature convolution in middleware systems. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 188–205, New York, NY, USA, 2004. ACM Press.

[130] Weishan Zhang and Stan Jarzabek. Reuse without compromising performance: Industrial experience from RPG software product line for mobile devices. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, pages 57–69, 2005.

[131] Weishan Zhang, Stan Jarzabek, Neil Loughran, and Awais Rashid. Reengineering a pc-based system into the mobile device product line. In *Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, 2003.