“Implementing Software Product Line Adoption Strategies”

Por

Vander Ramos Alves

Tese de doutorado

Recife, Março 2007
Implementing Software Product Line Adoption Strategies

Este trabalho foi apresentado à Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito para a aprovação da tese de doutorado.

ORIENTADORES:
Prof. Paulo Henrique Monteiro Borba
Prof. Geber Lisboa Ramalho
Alves, Vander Ramos
xv, 161 p. : il., fig., tab.


Inclui bibliografia.

1. Engenharia de software. 2. Linhas de produtos de software. 3. Estratégias de implantação. 4. Programação orientada a aspectos. I. Título.

005.1 CDD (22.ed.) MEI2007-091
Aos meus pais.
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 Rain of Fire case study. 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 in the Best Lap case study. 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. Most valuable feedback has also been provided by Rosana Braga, Alessandro García, André Santos, Sérgio Soares, Silvio Meira, and Jacques Robin. I am most grateful to them.

Last, but not least, I most grateful to my family for their support during this period. This work was partially supported by CNPq, FINEP, and 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 levels, development methods lack adequate support for extracting and evolving SPLs. In this context, we present an original method providing concrete guidelines for extracting and evolving SPLs at the implementation and feature model levels, at both of which it supports reuse and safety. The method first bootstraps the SPL and then evolves it with a reactive approach. The method relies on a collection of provided refactorings at both the code level (aspect-oriented refactorings) and at the feature model level. The method was assessed in the highly variant domain of mobile games.

Keywords: Software product lines, adoption strategies, aspect-oriented programming, feature models, mobile games.
Resumo

Linha de Produtos de Software (LPS) é uma abordagem 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 significativa melhoria em produtividade. Um problema-chave associado, no entanto, é o tratamento de estratégias de implantação, em que uma organização decide iniciar uma LPS a partir do zero, fazer bootstrap de produtos existentes em uma LPS, ou evoluir uma LPS. Em particular, no nível de implementação e de modelo de features, métodos de desenvolvimento carecem de apoio adequado para extração e evolução de LPSs. Neste contexto, apresentamos um método original provendo diretrizes concretas para extração e evolução de LPSs no nível de implementação e de modelo de features, nos quais proporciona reuso e segurança. O método primeiro faz o bootstrap da LPS e então a evolui com uma abordagem reativa. O método se baseia em uma coleção de refatoramentos tanto na implementação (refatoramentos orientados a aspectos) como no modelo de features. O método foi avaliado no domínio altamente variável de jogos móveis.

Palavras-chave: Linha de produtos de software, estratégias de implantação, programação orientada a aspectos, modelo de features, jogos móveis.
# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AJDP</td>
<td>AspectJ Development Tools</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect-Oriented Programming</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CC</td>
<td>Conditional Compilation</td>
</tr>
<tr>
<td>FM</td>
<td>Feature Model</td>
</tr>
<tr>
<td>FOP</td>
<td>Feature Oriented Programming</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>J2ME</td>
<td>Java 2 Micro Edition</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>MG</td>
<td>Mobile Game</td>
</tr>
<tr>
<td>MIDP</td>
<td>Mobile Information Device Profile</td>
</tr>
<tr>
<td>RoF</td>
<td>Rain of Fire</td>
</tr>
<tr>
<td>SPL</td>
<td>Software Product Line</td>
</tr>
</tbody>
</table>
## Contents

1 Introduction
   1.1 Summary of Goals .................................................. 3
   1.2 Organization ......................................................... 4

2 Software Variability ...................................................... 5
   2.1 Historical Notes and Terminology .................................. 5
   2.2 Variability in Software Product Lines .............................. 7
   2.3 Domain Example: Mobile Games Product Lines ...................... 8
   2.4 Software Product Line Approaches .................................. 11
      2.4.1 Feature-Oriented Domain Analysis ............................ 11
      2.4.2 FAST ..................................................... 13
      2.4.3 KOBRA .................................................. 14
      2.4.4 Adoption Strategies ....................................... 14
      2.4.5 Scope ................................................... 15

3 Current Variability Implementation Approaches ................................ 18
   3.1 Object-Orientation and Polymorphism ................................ 18
   3.2 Object-Oriented Design Patterns .................................... 20
   3.3 Frameworks .......................................................... 22
   3.4 Feature-Oriented Programming ..................................... 22
   3.5 Deployment-Time and Run-Time Variability ......................... 24
   3.6 Program Transformation ............................................. 27
      3.6.1 Java Transformation System .................................. 27
      3.6.2 XVCL ...................................................... 32
   3.7 Conditional Compilation ............................................. 35
   3.8 Aspect-Oriented Programming ...................................... 38
      3.8.1 AspectJ .................................................... 38
      3.8.2 CaesarJ ................................................... 40
      3.8.3 AspectBox ................................................ 41
   3.9 Comparison Framework .............................................. 43
   3.10 Instantiating the Variability Framework ............................ 45
      3.10.1 Object-Oriented Design Patterns .............................. 45
      3.10.2 Frameworks .................................................. 45
      3.10.3 FOP ....................................................... 46
      3.10.4 JPEL ....................................................... 47
      3.10.5 JaTS ....................................................... 49
# List of Figures

2.1 The Three Essential Activities for Software Product Lines [39]. 6
2.2 Variability funnel with early and delayed variability [66]. The arrow-ended lines under *Possible systems* mean potentially infinite amount of variability when development starts. 8
2.3 Visualizing the effects of porting in the source code. 16
2.4 Feature diagram for Dictionary domain for embedded devices. 17
3.1 Varieties of polymorphism [34]. 20
3.2 Template method design pattern. 21
3.3 Mobile Game (MG). Only method names are shown instead of full method signatures. 23
3.4 Structure of the Software Product Line implemented with JaTS. 30
3.5 Catapults and dragons facing both directions on Rain of Fire. 30
3.6 T1 templates. 31
3.7 T2 templates. 32
3.8 Example of an X-Framework [76]. 33
3.9 Traversal of X-Frames by the XVCL processor [76]. 34
3.10 Refinement of virtual classes in CaesarJ [17]. 41
3.11 Propagating mix-in composition [17]. 42
3.12 Positive and negative variability. 45
4.1 Method for implementing SPL adoption strategies. 58
4.2 Bootstrapping the Product Line. Core assets appear above the dashed line. 59
4.3 Evolving the Product Line. Core assets appear above the dashed line. 60
4.4 Refactoring the Product Line. Core assets appear above the dashed line. 61
4.5 Migrate SPL. 74
4.6 Extract Method to Aspect. 88
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) 89
5.1 Problems in Refactoring SPLs 93
5.2 Feature Diagram Notations 95
5.3 Feature Model Example 96
5.4 Class Diagram depicting Feature Model Components 96
5.5 Feature Model Refactoring Example 98
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Completeness proof of B-refactorings. B-R stands for B-Refactorings.</td>
<td>106</td>
</tr>
<tr>
<td>5.7</td>
<td>Reduction strategy for completeness proof. B-R stands for B-Refactorings. All B-Refactorings are applied from left to right.</td>
<td>107</td>
</tr>
<tr>
<td>5.8</td>
<td>Semantics-based reasoning versus catalog-based reasoning.</td>
<td>107</td>
</tr>
<tr>
<td>5.9</td>
<td>Case Study Program Refactorings</td>
<td>111</td>
</tr>
<tr>
<td>5.10</td>
<td>Case Study Feature Model Refactorings</td>
<td>112</td>
</tr>
<tr>
<td>6.1</td>
<td>Platform variation of Rain of Fire</td>
<td>117</td>
</tr>
<tr>
<td>6.2</td>
<td>Variability within Game Product Line</td>
<td>117</td>
</tr>
<tr>
<td>6.3</td>
<td>Best lap’s main screen</td>
<td>123</td>
</tr>
<tr>
<td>6.4</td>
<td>Variability within Best Lap</td>
<td>138</td>
</tr>
<tr>
<td>6.5</td>
<td>Plug-in for identifying variability in the conditional compilation SPL.</td>
<td>139</td>
</tr>
</tbody>
</table>
## List of Tables

2.1 Effects of porting in the source code, listing the types of variation and corresponding frequency. ........................................... 11

3.1 Examples of variability support with design patterns (excerpt of Table 1.2 in [55]). .................................. 21

3.2 List of templates found. .......................................................... 29

3.3 Framework for describing variability techniques. ................. 44

3.4 Some design patterns according to the variability framework. .. 46

3.5 Framework technology according to the variability framework. ...... 47

3.6 FOP described according to the variability framework. .......... 47

3.7 JPEL described according to the variability framework. .......... 48

3.8 JaTS described according to the variability framework. .......... 49

3.9 XVCL described according to the variability framework. .......... 50

3.10 Conditional compilation described according to the variability framework. 51

3.11 AOP described according to the variability framework. .......... 52

3.12 Comparing implementation mechanisms according to the variability framework. The following abbreviations are used: C (coarse-grained), F (fine-grained), RT (run-time), PT (processing-time), DT (deployment-time), CT (compile-time), empty shape (low/not supported), partially filled shape (medium/partially supported), and filled shape (high/supported). 53

4.1 Summary of Refactorings. .......................................................... 66

4.2 Relating Migration Strategies to Refactorings ......................... 84

4.3 Summary of Refactorings Derivations. Consecutive application of laws is represented by $\rightarrow$. Repeated application of a law is denoted with a superscript *. ................................................................. 90

5.1 Summary of Unidirectional Feature Model Refactorings .......... 100

6.1 Occurrence of each refactoring ............................................. 120

6.2 LOC in original and SPL implementations ............................. 121

6.3 LOC of aspects in the SPL ...................................................... 122

6.4 Application size (Kbytes) in original and SPL implementations ... 122

6.5 Occurrence of the top 10 most frequently used preprocessing tags in Best Lap ................................................................. 124

6.6 Occurrence of preprocessing tags in Best Lap classes ............ 125

6.7 Occurrence of migration strategies ........................................ 126

6.8 Occurrence of programming laws in each refactoring ............. 127
6.9 SPL configuration knowledge .................................. 129
6.10 Reuse of aspects in BestLap SPL .......................... 130
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. ........................................ 140
6.12 Application (jar) Sizes of Best Lap SPL .................... 141
Chapter 1
Introduction

Computational systems are becoming ubiquitous [117]. By using a mobile phone, it is usually possible to access and manipulate information almost everywhere. 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 ever-increasing quality standards, specially availability and usability.

In order to meet 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 such systems. Additionally, the development processes must be productive, and the resulting software must be extensible and reusable [26].

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 programming languages and can benefit from design and architectural patterns [31, 55], the latter not being specific to object orientation. As a result, it offers more effective means to achieve reuse, thereby increasing productivity of future projects and reducing the cost of software maintenance. Object orientation, however, has some shortcomings, such as difficulty in modularizing systemic requirements, like non-functional requirements and complex object protocols [105, 106]. In order to overcome these shortcomings, novel extensions to programming languages–also having a counterpart in object orientation–have been proposed, among which Aspect-Oriented Programming (AOP) [83, 85] is receiving increasing attention [124, 139].

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 reducing development complexity, promoting its predictability and reproducibility. One shortcoming of existing processes in widespread use in the industry, such as the Unified Process [74], is that they do not focus on providing detailed guidelines for the implementation phase of the software life cycle (e.g. implementation workflow in the Unified Process). 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 like the one mentioned and focusing on implementation are already being defined [2, 73, 97, 98, 123].

More recently, in addition to the existing software processes, there has been the
emergence of 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 instantiated for the creation of a particular product. In this process, variability management [89] 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 [66]. Although modeling and design are being intensively discussed in conferences like the Software Product Line Conference, investigation of more suitable implementation techniques cannot be neglected [16].

Moreover, a key orthogonal issue to variability management in SPLs is handling adoption strategies, whereby an organization decides to start the SPL from scratch (proactive), bootstrap existing products into a SPL (extractive), or evolve an existing SPL (reactive). In practice, the last two scenarios are prevalent, since in most organizations there are already existing products that are then to be integrated into a SPL, or there is already a SPL that it evolving [88].

Particularly, the central problem addressed in this thesis is the lack of adequate support for the extractive and reactive SPL adoption strategies at the implementation and at the feature model level, which is the set of possible configurations of the SPL, each configuration being a set of features describing one application in the SPL. In particular, this key problem has three subproblems:

- **Concrete guidelines.** Although previous research addresses elicitation of implementation mechanisms for SPL variability [16] and combination of extractive and adoption strategies [88], there is still a lack of concrete guidelines combining how these implementation mechanisms are to be used in such adoption strategies. Further, at the feature model level, although previous work [129] has proposed a set of transformations relating equivalent feature models, their use does not support the extractive and reactive SPL adoption strategies;

- **Reuse.** The employed guidelines should also support reuse, an important goal in SPL development, so as to maximize return-on-investment [39]; this should happen both at feature model level as well as at the code level.

- **Safety.** There is also a lack of safety discussion regarding the extractive and reactive adoption strategies, thus additional confidence needs to be provided for these strategies. This is particularly important, since, in the SPL context, testing is considerably expensive [112]. In particular, this needs to be addressed further at the program level and at feature model level. At the feature model level, 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 also have three instances. In the case in which the product line reacts (evolves), the feature model should have more than three instances. At the code level, each of the SPL
instances should have the same behavior than that of the existing products before employing the extractive approach.

In this context, we propose a 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 relies on a collection of provided refactorings at both the code level and at the feature model level. Furthermore, the refactorings can be systematically derived from more elementary and simpler programming laws or feature model transformation laws.

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 to alternative implementation techniques. Accordingly, we conduct a comparative analysis of SPL variability implementation techniques, which is useful in this respect.

The hypothesis of this thesis is that the method we propose is adequate for extracting and evolving SPLs, thus addressing the key problem mentioned previously. In particular, by addressing the corresponding subproblems, this hypothesis can be decomposed into the following:

- The method provides concrete guidelines for extracting and evolving SPLs at the implementation and feature model levels;
- the method supports reuse at the implementation and feature model levels;
- the method supports safety at the implementation and feature model levels.

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 on the 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. Regarding drawbacks, the evaluation is conducted only in a specific domain and the mobile games are not large scale applications. However, these applications are non-trivial and this domain is rich in variability, a key property of SPLs. Additionally, such variability can also occur in other domains, thus broadening the potential application of the method.

1.1 Summary of Goals

Our research has the following key goals:
• to provide a method for the extractive and reactive SPL adoption strategies;
• to extend the notion of refactoring to SPLs;
• to evaluate the proposed method in industrial-strength SPLs.

1.2 Organization

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 compares them according to such framework;

• Chapter 4 defines our method for implementing the extractive and reactive SPL adoption strategies. It also 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

The ability of changing or customizing a software system is referred to as variability [66]. Managing variability is at the core of product line development and can be accomplished by a myriad of approaches targeting specific kinds of variabilities and guided by underlying principles such as reusability and flexibility [39, 89]. Some kinds of variabilities can be foreseen, and this knowledge can be used for the development of a system supporting these specific variabilities [66].

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 underlying ideas of software families and software product lines have been latent in Software Engineering with pioneer works by Dijkstra [48] and Parnas [109, 110]. Indeed, according to Dijkstra, design decisions are used to narrow possible set of systems until family members are individually defined, and this forms the basis of a product family development. Complementarily, with the notion of information hiding, variability management could be addressed by implementing variable aspects of a module behind its stable interface [109]. Additionally, according to Parnas [110], software families are characterized by the first-class status of commonality within its members: such commonality should be focused on first to understand the family; a subsequent activity is to determine how family members differ from one another.

Likewise, according to Withey [133], a product family is defined as a set of applications that are developed from a shared collection of artifacts. In this way, product family defines the scope of its members focusing on their technical commonality rather than on their domain categorization.

The same author [133] defines a product line as a group of products having common features within a given domain. Whereas the definition of product family focuses on technical commonality between member products, this notion of a product line focuses on the domain level. We remark that these definitions might overlap, but may also be
orthogonal: a product line can be based on one or more product families; conversely, a product family can be used in multiple product lines.

Clements and Northrop [39] extended the previous definition in order to incorporate technical similarities between member products. Accordingly, a software product line comprises the systematic development of products with common features in a defined domain and that are built from reusable artifacts, referred to as core assets (a core asset is an artifact employed in the construction of at least two products in a SPL [39]). In the scope of our research and, in particular, during the rest of this document, we assume this definition.

SPL development consists of two fundamental activities [39]: 1) core asset development (also known as Domain Engineering [45]); 2) product development (also known as Application Engineering [45]). In the former, reusable artifacts are built, e.g., a reference architecture and components; in the latter, such reusable artifacts are used in the construction of SPL instances, i.e., products.

We remark that the interaction of these activities is not only from the first to the second (core assets being used in product development), but also in the reverse direction, i.e., core asset development also depends on product development. Indeed, components in existing products are candidates for core assets, depending on how reusable they might be within other existing or envisioned products. Therefore, a mining activity can be performed in order to identify core assets in existing products. In practice, this scenario occurs frequently [88].

Core asset development and product development occur in the context of an organization employing SPL development. Therefore, these activities should be supported by both technical and organizational management. Figure 2.1 depicts these activities [39] and accordingly uses double-ended arrows to emphasize the two-way interactions among these activities.

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

Variability in software assets has become increasingly important in software engineering [66]. Initially, most systems had fewer variations and were frequently static. It was then assumed that variant behavior would entail significant recoding of the software system. In contrast, contemporary software is rich in variability and its binding time (the stage during software development in which the behavior of the system is fully defined) can often occur at different stages of the development cycle.

In contemporary software, SPLs, for instance, embed a significant number of variations, which can be bound at different times (pre-compilation, compilation, deployment, runtime), thus potentially giving rise to a number of instances. Accordingly, SPLs are not developed to match the requirement of only a single application: its development involves construction of a software architecture and set of components, which are then instantiated to meet requirements of the SPL instances. As another example, there are also software systems that can dynamically adapt their behavior at runtime [87, 113]; this can be accomplished by selecting pre-coded alternatives in the software system or by loading new code modules during at runtime, such as plug-and-play functionality or adaptive object models [138], for instance.

As a result of higher demand for variability and shift towards multiple binding times, we have seen the emergence of a myriad of variability mechanisms allowing to postpone decisions regarding the variants to the point in the development cycle that maximizes business goals [66]. For example, in conditional compilation, variability is handled at pre-compilation time; as another example, JPEL [119] allows run-time variability. Differently, different AOP implementations have compile time, deployment, and runtime binding times. These mechanisms and others are surveyed and compared in Chapter 3.

In Figure 2.2, we depict the variability of a software during development cycle [66]. At each stage, the amount of variability is proportional to the length of the intersection between the funnel and the horizontal lines. Therefore, at the beginning of development, variability is highest, which is indicated in Figure 2.2 by arrow-ended lines. As development progresses, design decisions are made, which then constrain variability. Accordingly, the number of potential systems decreases, which eventually leads to only one system at runtime.

In Figure 2.2, the variability funnel on the left depicts a scenario in which variability rapidly decreases throughout development stages. This is most common in single-system software development. In contrast, the right funnel depicts a scenario in which variability decreases smoothly, still preserving a significant amount of variability until later development stages. This latter case is more common in SPLs, in which reusable artifacts inherently embed more variability.

Accordingly, variability management plays the role of the central criterion in distinguishing between traditional software engineering and SPL development. In the former, variability management addresses temporal software variation and is referred to as configuration management. In the latter, variability management addresses variation in both time and space [89]. In such scenario, managing temporal variability refers to configuration management of the SPL as it changes over time, whereas managing variability in space addresses differences in SPL instances in the domain spectrum of a SPL at a given point in time. Managing variation in space itself is challenging. Although the
Figure 2.2: Variability funnel with early and delayed variability [66]. The arrow-ended lines under Possible systems mean potentially infinite amount of variability when development starts.

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 [1], 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 as 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 stronger portability requirements [5, 33]. 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.

[^1]: A service carrier is a telephone company providing local, long distance, or value-added service.
Porting stems from a combination of technical and business constraints. Manufacturers release different devices targeting diverse customer profiles, in ever-shorter 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 providing such service [132].

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 hypotheses that restrict their applicability, and very few have been validated in industry [35, 49, 51, 54, 103].

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 [101] 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 [1, 118], are the following:

- 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;
- Different profiles (MIDP 1.0 and MIDP 2.0)\(^2\);
- Different implementations of a given profile in J2ME [100];
- Device-specific bugs;
- Internationalization.

J2ME technology is evolving with the release of version 2.0 of its specification [115] and the optional libraries specification, which can be present in the devices [100]. 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

\(^2\)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.
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 workarounds from the programmer when he or she has to use the defective libraries. Once again, porting is compromised. Finally, 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 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 (Rain of Fire3) from Motorola’s platform T720 to Nokia’s Series 60, both J2ME-compliant, but the latter relying on proprietary API offering advanced graphics manipulation [118]. Despite the apparent functional game simplicity, the differences between the devices prompted changes in almost all application classes, adding up to 79 modifications. These changes were not due to poor design, since the game was based on architectural and design patterns with the purpose of isolating, as much as possible, game functionality, presentation, and proprietary device APIs. Rather, the changes were due to the inherent widely scoped variability of the platforms (as explained shortly ahead).

The average size of each modification was two lines, which revealed the fine granularity of these changes. Table 2.1 illustrates the types of variations handled in the porting.

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 lefmost 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.

Based on our experience [1, 4, 5, 9, 10, 11, 13, 14, 118], we notice, in general, that platform variations are highly crosscutting, i.e., they affect a large number of classes [1, 5, 33]. This is due not to poor design, but rather to inherent widely scoped variability of platforms. For example, memory constraints of platforms imply choosing to

---

3In cooperation with Meantime Mobile Creations, under FACEPE/PAPPE and CNPq/Universal research projects.
Table 2.1: Effects of porting in the source code, listing the types of variation and corresponding frequency.

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

omit some features in certain devices. Therefore, these features become optional, whose implementation, in turn, is often crosscutting, involving different methods/classes and tangled with the implementation of other features. This has also been observed by a significant body of research conducted elsewhere [16, 22, 92, 96, 99]. As another example, platform variability involves use of device-specific APIs, whose use cannot often be modularized using OO-based techniques, but rather is scattered throughout the code.

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; 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 (Feature-Oriented Domain Analysis) is a domain analysis method focused on the description of variabilities and commonalities by means of features, where a feature is a prominent and user-visible aspect, quality, or characteristic of a software system or systems [79]. 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 [79]:

- 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. 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 [45].

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, meaning that at least one of them should be selected); Server and Memory are alternative features (open arc, meaning that exactly one of them should be selected). 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.

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) [80] is a layered approach to the original FODA and extends the latter to the software design phase, prescribing 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) [64] combines FODA and the Reuse-Driven Software Engineering Business (RSEB) method [75], by including the domain engineering
and feature modeling steps into RSEB, since the 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 [29] 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, for the reason that it focuses on functionality, whereas in other definitions, such as the one proposed by Czarnecki [45], a feature is a distinguishable characteristic of a system that is relevant to some stakeholder of the system. In the scope of this thesis, we do not need the broad definition offered by the later.

- Benavides et al [23] extend feature models with extra-functional features and relations among 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 (Family-Oriented Abstraction, Specification, and Translation) [131] is a SPL development process conceived in the context of telecommunication infrastructure and real-time systems. It relies on a set of principles suggesting that SPL development is worthwhile in a domain. The first principle is that software development is mostly redevelopment, essentially by producing new variants of existing software. Therefore, in general, there are fewer differences than similarities within SPL instances. The second principle relates to the possibility of foreseeing changes that are likely to happen in a software system, thereby inferring future changes from earlier ones. In the last principle, the software and its encompassing organization can be structured to address foreseen changes, which are orthogonal to other kinds of changes. The goal is to isolate changes so that these have effect on only a reduced subset of system modules.

FAST comprises three subprocesses: domain qualification, domain engineering, and application engineering. The first specifies an economic model relating the other two subprocesses, by showing to which extent investment in core asset development pays off in product development. In particular, domain qualification produces an economic model that provides an estimate of the quantity and cost of SPL instances.

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 SPL. Additionally, core assets are designed and implemented. Two important such assets are the definition of the Application Modeling Language (AML) and of the application engineering environment. The former provides support for abstract specifications of applications. The latter provides support for the analysis of AML specifications and for code synthesis from such 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 built in core asset development are used to produce SPL instances rapidly in response to customer requirements.

### 2.4.3 KOBRA

At Fraunhofer IESE, KobrA [18] has been conceived for component-based SPL engineering. The novelty in this method is the synergy of the SPL and the component-based approaches, thus integrating “reuse in the large” from the former approach with “reuse in the small” from the latter approach.

KobrA defines a component (referred to as Komponent) at two levels of abstraction: specification and realization. The former specifies the external behavior and abstract properties of a Komponent; the latter specifies how to express a given Komponent in terms of finer-grained Komponents. In this method, a framework is a hierarchical organization of KobrA components.

There are two central activities in KobrA: framework engineering and application engineering. The former is based on the SPL approach so as to build and maintain a generic framework encompassing commonalities and variabilities. The outputs of framework engineering comprise framework models expressed by a combination of text and UML models. Examples of such models include business process model, decision model, and structural and collaboration UML diagrams of Komponents at both specification and realization levels.

The second activity in the method is application engineering, which relies on the outputs models defined in the first activity so as to instantiate the generic framework models for a specific SPL instance. In particular, the framework developed in the first activity is employed in product development. Customer specific requirements are also taken into account in this phase, whose outputs are SPL instance models expressed by a combination of text and UML models.

### 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 [88]. 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.
Figure 2.3: Visualizing the effects of porting in the source code.
Figure 2.4: Feature diagram for Dictionary domain for embedded devices.
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 [34, 55, 77]. 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 object-oriented 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 [25] 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 allow declaring classes and manipulating 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 an object receives a message, the actual action executed is determined not only by the request, but also by the object receiving the message. In particular, different objects can implement identical messages differently. The determination of which object will execute the request it receives occurs at runtime and is referred to as dynamic binding, which jointly with 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 stems from the Greek language and denotes “the ability to have many forms”. It refers, in the context object-oriented programming languages, to the capability of writing code using an abstract interface and then have variant implementations of such interface. In the seminal definition of this concept, given by Strachey in 1967 [125], there is a distinction 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.”

This initial definition was later extended by Cardelli and Wegner [34], as illustrated in Figure 3.1. Generic parameters enable parametric polymorphism. Inclusion polymorphism refers to subtype polymorphism in OO languages, i.e., variables of a certain type can refer to subtypes objects. Coercion denotes type conversions and promotions that occur automatically and can be either defined by the user or pre-existing in the language. For instance, in the addition of an integer and a floating point number, the type of the former is converted into the type of the latter. Overloading denotes variant implementations of functions with different operand types, but with the same function name.

To illustrate, all such kinds of polymorphism can be found in C++ [126]. Templates play the role of parametric polymorphism, virtual functions play the role of subtype polymorphism, function overloading maps to overloading, and built-in or user-defined conversion operators or constructs correspond to coercion. From these kinds, in C++, only subtype polymorphism provides run-time variability, which is accomplished with dynamic binding. The variability provided by the other mechanisms is bound at compile time.

To further illustrate, Java [62] also provides support for the four types of polymorphism described previously. It supports subtype polymorphism as well as parametric polymorphism (parametric polymorphism only became available in Java in 2004 with the Generics feature of J2SE 5.0). Automatic promotions and type conversions occur only in built-in types. Analogously to C++, methods can also be overloaded; nevertheless,
the user cannot overload operators. Subtype polymorphism manifests itself in Java with in two ways: 1) interfaces corresponding to types; 2) classes and extends constructs. Interfaces play the role of C++ abstract classes declaring pure virtual methods, but declaring no method implementations. In Java, we can declare a variable whose type is an interface and this variable can then refer to any object that is an instance of any class which implements such interface.

3.2 Object-Oriented 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 (hereafter, unless otherwise stated, we use design patterns to refer to object-oriented 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 are abstract collaborations of objects and classes that are instantiated in order to solve a design problem in a particular context [55]. Each pattern has a description clearly stating the problem it addresses, its applicability and consequences, the corresponding participating objects and classes as well as their collaborations. The description generally also includes an example as well as some trade-off analysis of its use.

In the product line context, design patterns are particularly useful because most design patterns allow varying part of the design. For example, bridge allows 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; abstract factory uses an interface to create families of related objects, in order not to specify their concrete classes [55]. Table 3.1 summarizes variability support with some design patterns.

In particular, the design pattern template method plays a key role in OO frameworks, described in Section 3.3. The class diagram in Figure 3.2 illustrates the structure of this design pattern. The algorithm() method invokes some abstract methods, for example,

Figure 3.1: Varieties of polymorphism [34].
Table 3.1: Examples of variability support with design patterns (excerpt of Table 1.2 in [55]).

<table>
<thead>
<tr>
<th>Design Pattern</th>
<th>Aspects that can vary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bridge</td>
<td>implementation of an object</td>
</tr>
<tr>
<td>State</td>
<td>behavior</td>
</tr>
<tr>
<td>Decorator (wrapper)</td>
<td>responsibilities of an object without subclassing</td>
</tr>
<tr>
<td>Adapter</td>
<td>interface to an object</td>
</tr>
<tr>
<td>Strategy</td>
<td>an algorithm</td>
</tr>
<tr>
<td>Template method</td>
<td>steps of an algorithm</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>families of product objects</td>
</tr>
</tbody>
</table>

action1() and action2(). Subclasses of Framework then give concrete implementations of such abstract methods. In general, class Framework belongs to a framework; a specific application then specifies a concrete subclass. We refer to algorithm() as template method, as it specifies the core of the algorithm, while still allowing to change some of its steps.

Figure 3.2: Template method design pattern.

Template method addresses fine-grained variation and is generally implemented using inheritance to allow changing part of an algorithm. Nevertheless, 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. Such binding mode is also shared by Abstract Factory, but this later allows changing the families of objects created instead of varying an algorithm.
3.3 Frameworks

A framework is a set of collaborating classes constituting a reusable design for a specific family of software systems [77]. A framework is a reusable artifact defining an application’s architecture by specifying its main control flow and structure, specifying main classes and objects as well as their main responsibilities and collaborations.

The overall control flow between a framework and an application based on it is that the latter specifies code that will be called by the former. This is in contrast to use of conventional libraries, where the developer of the application calls library code instead. With this inversion of control, the application developer is reusing design decisions embedded in the framework.

In the product lines context, frameworks provide a high level of reuse. Indeed, a framework embeds domain knowledge and is a semi-complete application in that domain. Accordingly, frameworks are important assets in core asset development. During product development, by instantiating a framework, a product line member is created. In this process, the behavior for variation points (also known as hot spots [114]) is defined in the framework.

Design patterns, such as template method, strategy, and state are frequently used to concretize variability in frameworks. However, in general, dynamic binding is also often employed in the implementation of design patterns, which in turn brings run-time binding mode to frameworks and thus incurs into some performance penalty. Additionally, since dynamic binding is an intra-application variability mechanism, using it for inter-application variability often complicates frameworks, which usually tend to grow rapidly in complexity.

3.4 Feature-Oriented Programming

In object-oriented software, classes cannot fully modularize features [105, 106]. Accordingly, research has been carried out to define more adequate representation of features at the implementation level. Some approaches define features as increments over an existing base program and provide support for on demand composition of features [99]. Examples of such approaches encompass AHEAD [22], mixin layers [121], and GenVoca [21], 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 given by an example of a mobile game. Figure 3.3 depicts the simplified class structure of such application. The key concept in Figure 3.3 is 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.

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;
Figure 3.3: Mobile Game (MG). Only method names are shown instead of full method signatures.

Accordingly, we consider the MG software as the base element (a constant in terms of AHEAD):

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

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

```java
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 employs AHEAD’s syntax and contains extensions of Resources and GameScreen (in AHEAD, the modifier refines denotes extension). The first refinement incorporates the cloud abstraction into a base type; the refinement for GameScreen weaves the cloud feature into the control flow of the base.

Analogously to subclasses, class refinements can incorporate new fields as done in Resources or override methods like in GameScreen (Figure 3.3 shows only method names instead of full method signatures). In contrast, class refinements have the advantage of allowing to combine different layers. For instance, another layer adding an enemy feature to the MG application could be combined with the clouds layer. In general, a separate namespace encompasses a different combination of features. In this way, there could be classes such as base.Resources, base.GameScreen, as well
as clouds.Resources,...,enemy.Resources,...,cloudenemy.Resources. Therefore, the base application can be employed in different product line configurations.

When compared to frameworks, FOP is superior due to its layer module, which provides a localized definition (into a single implementation unit) of a feature that would otherwise be dispersed across different classes. Moreover, since a layer is similar to a mixin module, the former can be applied to different base programs and combined freely with other layers, an essential capability for variability management [99].

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 [119] shows that such tools do not meet a set of requirements considered essential by developers when handling deployment and run-time variability, such as relationship among parameters, hierarchical grouping of parameters, and automatic adjustment of executing processes. Accordingly, such research presents a tool—named JPEL—meeting those requirements. In the rest of this section, we briefly consider this tool’s features in addressing these requirements.

**JPEL**

JPEL (Java Parameter Expression Language) [119] is a tool for parametrization of Java applications. It implements a set of features mentioned previously, which are frequently required by developers when handling 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.

```plaintext
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(WIDTH, 2);
}

integerDivision (x,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:

```java
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

Program transformation systems manipulate source programs, changing one program into another [30, 111, 130]. Such systems either transform a program in the source language into a new program in a different language (e.g. in compilation) or transform a program into a new program in the same language (e.g. in refactoring). In the scope of this thesis, we are interested in the latter type of transformation. In particular, 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. The variability patterns for SPL extraction and evolution (presented in detail in Chapter 4) are expressed declaratively as refactorings, which are then enabled by the program transformation systems. 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) [36], 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 three 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 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 three devices: Nokia Series 40, Nokia Series 60, and Motorola T720. This means that each device-specific version was developed by verbatim reuse, i.e., 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 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 incrementally 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.

---

1In cooperation with Meantime Mobile Creations/CESAR, under FINEP/FLIP, FACEPE/PAPPE and CNPq/Universal research projects.
Table 3.2: List of templates found.

<table>
<thead>
<tr>
<th>Template-Variation Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract/Implement Method</td>
</tr>
<tr>
<td>Add/Remove attribute</td>
</tr>
<tr>
<td>Add/Remove context</td>
</tr>
<tr>
<td>Add/Remove beginning block</td>
</tr>
<tr>
<td>Add/Remove ending block</td>
</tr>
<tr>
<td>Apply change to value</td>
</tr>
<tr>
<td>Class Hierarchy change</td>
</tr>
<tr>
<td>API Import</td>
</tr>
<tr>
<td>Argument in Method Call</td>
</tr>
<tr>
<td>Different Expression</td>
</tr>
<tr>
<td>Class Constant</td>
</tr>
</tbody>
</table>

...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. Figures 3.6 and 3.7, explained shortly ahead, illustrate the first template in this table: the former figure shows method extraction, whereas the latter shows method implementation. In Table 3.2, context is an arbitrary Java compound statement.

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.

One problem 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, 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 attributes 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; } }](image)

(a) Matching template

![class #C { FieldDeclarationSet:#ATTRS; ConstructorDeclarationSet:#CDS; MethodDeclarationSet:#MTDS; void m() { Block:#B1; this.newM(); Block:#B3; } }](image)

(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.

![Figure 3.7: T2 templates.](image)

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 [76]. It aims at improving reuse and at providing generic solutions over existing source code. It can address similarity patterns in programs, replacing them by generic solutions in the form of XVCL meta-structures, resulting in improved maintainability and reusability. Accordingly, XVCL offers applications the following benefits:

- to be used as the core for similar software (e.g., a product line);
- maintenance in accordance with changes stemming from evolutionary issues.
The key idea of XVCL is similar to the concept of abstraction over statements in imperative languages (like procedures or methods): whenever a piece of code is employed several times in several different sections of the program, it is possible to encapsulate that as an abstraction and parameterize it. Therefore, when running that piece of code, we can just invoke such abstraction using suitable parameters. As a consequence, identical blocks of code throughout the program are avoided, thus making it easier to fix bugs or add any functionality to the code afterwards, as such tasks become localized [76].

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 consists of a root x-frame, referred to as the specification file (SPC). 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 [76]. 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.

![X-Frame Hierarchy](image_url)

Figure 3.8: Example of an X-Framework [76].

**XVCL Features**

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

The `<adapt>` command specifies an x-frame the processor should process. The processor locates this x-frame, and after processing it, returns the resulting code on the place where the `<adapt>` command was called. 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 is used to denote a variation point where changes can be made by x-frames higher up in the hierarchy. Additionally, it may be employed to specify default code that might be overridden by such x-frames. Such command is similar to an AspectJ join point (Section 3.8.1), 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 binds a given breakpoint with a piece of code in the underlying x-subframework. If we compare this command 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.

Also similarly to AspectJ, where advice code can be executed before or after a given join point, there are two variations of the <insert> construct, whereby the piece of code is inserted either before or after the variation point, with the <insert-before> or <insert-after> constructs, respectively.

For example, suppose we want to add a new method to the MyBuild example shown in previous section. We already have a variation point 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 enclosing the <break> command. In order to accomplish this, we will change the x-frame presented in the first example, by adding 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 [134]. In this section, we show how variability can be implemented with such a 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 handled during the porting process with the Antenna preprocessor.

My Big Brother 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. 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 [134], a set of Ant tasks suitable for developing wireless Java applications. Antenna provides a simple preprocessor, similar

\footnotetext{\[2\]Developed by Meantime Mobile Creations/CESAR, which granted access to the game under FACEPE/PAPPE and CNPq/Universal research projects.}
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
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 = -25;
      BOARD_SOFT_RIGHT = -28;
    //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 [85]. Those are called crosscutting concerns and their separation promotes the construction of a modular system, avoiding code tangling and scattering. In this section, we briefly describe three AOP languages: AspectJ [84], CaesarJ [17], and AspectBox [24]. These languages are described because previous research has suggested their potential use in addressing variability in SPLs [15, 24, 99].

3.8.1 AspectJ

AspectJ [84] 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. Pointcuts 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 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.

```java
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 [94].

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 CaesarJ

Packages in Java and name spaces in C++ are ways of organizing classes in traditional object-oriented languages. However, these mechanisms provide insufficient support for relating sets of classes that collaborate and, in particular, expressing variants of a collaboration, using collaborations in polymorphic way and as first class values [99].

In this context, CaesarJ [17] is a language combining AOP constructs, pointcut and advice, with object-oriented mechanisms for achieving coarser granularity in modularization. On one side, this synergy contributes to improving reusability of aspects; on the other, it contributes by making it possible to integrate components developed independently into an application without affecting either. Further, such integration can be bound at run-time.

The key concept in CaesarJ is that of a virtual class, which is an abstraction encapsulated within a module called family class. Depending on the runtime context of usage, a virtual class, similarly to a virtual method, can have different behavior. Virtual classes are defined within a family class as inner classes and, similarly to fields and methods, virtual classes are members of instances of their encompassing family class. Accordingly, their meaning depends on the type of the family object at runtime. Further, analogously to virtual methods, these CaesarJ abstractions can be overridden and late bound. Additionally, previous relations are inherited, but are cast automatically to the most specific definition of a type reference. Further, new classes and relations can be added [17].

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, in these other collaborating virtual classes, there is an automatic re-binding of references from Node to the refined Node class, when these references point to an object of type AdjustedHierarchyDisplay. In particular, CompositeNode is subtype of the refined Node class, when considered in the context of an instance of the virtual class AdjustedHierarchyDisplay [17].

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 AdjustedHierarchyDisplay and AngularHierarchyDisplay refining family class HierarchyDisplay. These latter are then combined with mixin composition into family class AdjustedAngularHierarchy Display. Accordingly, 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.
3.8.3 AspectBox

AspectBox [24] extends AOP languages by providing a new construct: an aspectbox, which can encompass aspect and class declarations 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 have effect on only classes declared 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 defines an aspect within an aspectbox affecting the imported class GameCore (AspectBox’s syntax is based on Squeak; the example is based on an equivalent AspectJ-like syntax):

```java
AspectBox Extension1 {
    import GameCore;
    aspect Variation1 {
        pointcut p(): call (* *(..)) ;
        after(): p() { ... }
    }
}
```
Accordingly, pointcuts declared in an aspect specify join points only in classes from 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. Being in different scopes, aspects Variation1 and Variation2 are kept separated, affecting different versions of the same class. Even if such aspects, which are defined in different aspectboxes, shared join points, it would not be necessary to specify rules of precedence for composition ordering. This makes it possible to deploy various simultaneous changes in the same core system and thus to prevent conflicts in aspectboxes from happening.

Additionally, an aspectbox can import classes, which then become in the same scope as classes declared in this aspectbox. Furthermore, the import relationship is transitive: if class GameCore from aspectbox Extension1 is imported by aspectbox Extension2, then another aspectbox, Extension3, can also import GameCore from Extension2. For Extension3, it is not important whether GameCore is declared or imported in
Nevertheless, since it is not possible to reuse aspects in different base systems, aspects cannot be imported [24].

3.9 Comparison Framework

In this Section, we propose a comparison framework whose purpose is to allow selection of variability implementation mechanisms: this selection is important because, as we have surveyed, there is a myriad of such implementation mechanisms; on the other hand, as we will show shortly, such mechanisms perform differently across a set of criteria relevant for SPLs. The relevance weight of such criteria may also shift from domain to domain and from SPL to SPL. Therefore, it is not feasible to assume that a single technique will maximize all such criteria and be used uniformly across all projects. Rather, it is realistic to assume that in a single project different such techniques may be employed depending on how each satisfies the criteria for each variability, hence the utility of this comparison framework.

The framework is in accordance with the rest of the thesis, since, in the following chapter, Section 4.1 describes the core of our method for SPL extraction and reaction, which is independent of a particular variability mechanism (later in Chapter 4, the method is detailed with AOP for reasons explained afterwards), and thus can benefit from an informed decision based on comparative analysis based on the framework proposed in this chapter.

The criteria used in the framework is an extended version of criteria previously proposed by some researchers in the SPL domain whose goal was also to offer selection of variability implementation mechanism [16, 43]. Accordingly, the taxonomic parameters and their possible values are described in Table 3.3.

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.

Variability in structure means diversity in statically arranging program elements, whereas variability in behavior involves different execution semantics. Fine-grained granularity refers to variation within classes, specially within method bodies, whereas coarse-grained granularity refers to granularity above the class level. 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 [90].

The framework has also been extended to incorporate some elements such as reusability, which is also germane to the product line context [15]: in order to manage variability, it is important to reuse assets as much as possible, so as to achieve better return on investment on the time spent developing those assets. We further extend the frame-
work with other relevant elements, such as application size (size of executable of SPL instances) and performance. It is important to assess the impact on application size of variability mechanisms in certain domains such as mobile device games due to constraint on footprint size. For instance, a certain mechanism may promote reuse, but may cause unacceptable code bloat. Performance should also be taken into account because one mechanism may, for example, promote variability in behavior at the cost of unacceptably poor performance.

We also consider the support for modular implementation of crosscutting concerns as a criterion in the framework. By crosscutting concerns, we mean those having scattered and tangled implementation. We consider such criterion because the implementation of SPL variability may be frequently crosscutting, as pointed out by a significant body of contemporary research [16, 22, 92, 96, 99]. Therefore, their modularization is important to promote adequate extraction and reaction of SPLs. 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.

Last, but not least, the comparison framework additionally includes legibility. In the SPL context, the degree to which a variability mechanism is legible becomes specially important due to the possibly high number of core assets and product-specific assets, which have to be composed varied ways during both extractive and reactive SPL scenarios.

According to the scope of this thesis, the framework is not intended to be exhaustive. Indeed, additional criteria could be considered in a refined version of the framework, such as usage convenience, language complexity, support for incremental generation of code, and validation support. Further, in terms of techniques covered we do not cover DSLs [72] and Software Factories [63] because we are focusing on implementation level techniques and these may be orthogonally used with the other techniques presented here at a higher level of abstraction.
Even though the values of the framework items reusability, performance, application size, and legibility are described and explained 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.

![Figure 3.12: Positive and negative variability.](image)

### 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 Object-Oriented 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. Further, OO design patterns do not adequately support modular crosscutting implementation, since even the implementation of most patterns is crosscutting [56, 68]. Finally, OO design patterns generally have low legibility because their implementation is difficult to separate from the application classes, which also limits reusability.

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

<table>
<thead>
<tr>
<th>Framework item</th>
<th>Possible values for some design patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability type</td>
<td>positive (decorator), negative (adapter, bridge, decorator)</td>
</tr>
<tr>
<td>Variability in structure</td>
<td>supported (adapter, bridge, decorator)</td>
</tr>
<tr>
<td>Variability of behavior</td>
<td>supported (decorator, state, strategy, template method)</td>
</tr>
<tr>
<td>Granularity</td>
<td>fine-grained (template method), coarse-grained (strategy)</td>
</tr>
<tr>
<td>Binding time</td>
<td>compile-time (template method with parameterized inheritance), run-time (strategy, template method with inheritance, bridge)</td>
</tr>
<tr>
<td>Reusability</td>
<td>medium</td>
</tr>
<tr>
<td>Performance</td>
<td>low (strategy), high (adapter)</td>
</tr>
<tr>
<td>Application size</td>
<td>varies</td>
</tr>
<tr>
<td>Support for modular implementation of crosscutting concerns</td>
<td>not supported</td>
</tr>
<tr>
<td>Legibility</td>
<td>generally low</td>
</tr>
</tbody>
</table>

often be supported with inheritance with cancelation [122]; 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 [120], 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. Moreover, 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 [91], which also implies poor legibility.

3.10.3 FOP

Table 3.6 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 contributes to reusability. However, reusability is not high, due to the potentially high fragmentation of code in the refine statements. 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
Table 3.5: Framework technology according to the variability framework.

<table>
<thead>
<tr>
<th>Framework item</th>
<th>Framework technology values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability type</td>
<td>positive, negative</td>
</tr>
<tr>
<td>Variability in structure</td>
<td>supported</td>
</tr>
<tr>
<td>Variability of behavior</td>
<td>supported</td>
</tr>
<tr>
<td>Granularity</td>
<td>coarse-grained</td>
</tr>
<tr>
<td>Binding time</td>
<td>usually compile time</td>
</tr>
<tr>
<td>Reusability</td>
<td>high</td>
</tr>
<tr>
<td>Performance</td>
<td>varies</td>
</tr>
<tr>
<td>Application size</td>
<td>varies</td>
</tr>
<tr>
<td>Support for modular implementation of crosscutting concerns</td>
<td>not supported</td>
</tr>
<tr>
<td>Legibility</td>
<td>generally low</td>
</tr>
</tbody>
</table>

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. Additionally, FOP supports modular implementation of crosscutting concerns, which are localized in the delta layers. However, only heterogenous crosscutting is supported. Finally, FOP has good legibility due to modularization of features into increments.

Table 3.6: FOP described according to the variability framework.

<table>
<thead>
<tr>
<th>Framework item</th>
<th>FOP possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability type</td>
<td>positive, negative</td>
</tr>
<tr>
<td>Variability in structure</td>
<td>supported</td>
</tr>
<tr>
<td>Variability of behavior</td>
<td>supported</td>
</tr>
<tr>
<td>Granularity</td>
<td>coarse-grained</td>
</tr>
<tr>
<td>Binding time</td>
<td>preprocessing-time, compile-time</td>
</tr>
<tr>
<td>Reusability</td>
<td>medium</td>
</tr>
<tr>
<td>Performance</td>
<td>high</td>
</tr>
<tr>
<td>Application size</td>
<td>low</td>
</tr>
<tr>
<td>Support for modular implementation of crosscutting concerns</td>
<td>partially supported</td>
</tr>
<tr>
<td>Legibility</td>
<td>high</td>
</tr>
</tbody>
</table>

3.10.4 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 3.7.

Table 3.7: JPEL described according to the variability framework.

<table>
<thead>
<tr>
<th>Framework item</th>
<th>JPEL possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability type</td>
<td>not supported</td>
</tr>
<tr>
<td>Variability in structure</td>
<td>not supported</td>
</tr>
<tr>
<td>Variability of behavior</td>
<td>supported only for changes in values</td>
</tr>
<tr>
<td>Granularity</td>
<td>fine-grained</td>
</tr>
<tr>
<td>Binding time</td>
<td>Deployment- and run-time</td>
</tr>
<tr>
<td>Reusability</td>
<td>Medium</td>
</tr>
<tr>
<td>Performance</td>
<td>potentially low</td>
</tr>
<tr>
<td>Application size</td>
<td>potentially high</td>
</tr>
<tr>
<td>Support for modular implementation of crosscutting concerns</td>
<td>not supported</td>
</tr>
<tr>
<td>Legibility</td>
<td>high</td>
</tr>
</tbody>
</table>

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 cannot 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 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). 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. Lastly, JPEL has
high legibility, with a declarative way of specifying relationship among parameters and a simple interface for automatic adjustment of executing processes.

3.10.5 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 3.11. According to Table 3.8, 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.

Table 3.8: JaTS described according to the variability framework.

<table>
<thead>
<tr>
<th>Framework item</th>
<th>JaTS possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability type</td>
<td>positive, negative</td>
</tr>
<tr>
<td>Variability in structure</td>
<td>supported</td>
</tr>
<tr>
<td>Variability of behavior</td>
<td>supported</td>
</tr>
<tr>
<td>Granularity</td>
<td>fine-grained, coarse-grained</td>
</tr>
<tr>
<td>Binding time</td>
<td>preprocessing</td>
</tr>
<tr>
<td>Reusability</td>
<td>low, medium</td>
</tr>
<tr>
<td>Performance</td>
<td>high</td>
</tr>
<tr>
<td>Application size</td>
<td>low</td>
</tr>
<tr>
<td>Support for modular implementation</td>
<td>not supported</td>
</tr>
<tr>
<td>of crosscutting concerns</td>
<td></td>
</tr>
<tr>
<td>Legibility</td>
<td>medium</td>
</tr>
</tbody>
</table>

Besides, reusability can be either low, because the matching templates may depend considerably on code within method bodies, or medium, with matching templates applying to 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 3.10.8 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. Further, JaTS does not support modular implementation of crosscutting concerns. Finally, JaTS has medium legibility because, although the template language is declarative and an extension of Java, variability is often expressed invasively.

3.10.6 XVCL

Table 3.9 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).

Table 3.9: XVCL described according to the variability framework.

<table>
<thead>
<tr>
<th>Framework item</th>
<th>XVCL possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability type</td>
<td>positive, negative</td>
</tr>
<tr>
<td>Variability in structure</td>
<td>supported</td>
</tr>
<tr>
<td>Variability of behavior</td>
<td>supported</td>
</tr>
<tr>
<td>Granularity</td>
<td>fine-grained, coarse-grained</td>
</tr>
<tr>
<td>Binding time</td>
<td>preprocessing</td>
</tr>
<tr>
<td>Reusability</td>
<td>high</td>
</tr>
<tr>
<td>Performance</td>
<td>high</td>
</tr>
<tr>
<td>Application size</td>
<td>low</td>
</tr>
<tr>
<td>Support for modular implementation</td>
<td>not supported</td>
</tr>
<tr>
<td>of crosscutting concerns</td>
<td></td>
</tr>
<tr>
<td>Legibility</td>
<td>low</td>
</tr>
</tbody>
</table>

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 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. Finally, XVCL has low legibility, since this approach is language-independent and the developer has to explicitly indicate variation points in the code.

3.10.7 Conditional Compilation

Finally, according to the variability framework proposed in Section 3.9, we are now able to describe the conditional compilation. Table 3.10 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.

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
Table 3.10: Conditional compilation described according to the variability framework.

<table>
<thead>
<tr>
<th>Framework item</th>
<th>Conditional compilation possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability type</td>
<td>positive, negative</td>
</tr>
<tr>
<td>Variability in structure</td>
<td>supported</td>
</tr>
<tr>
<td>Variability of behavior</td>
<td>supported</td>
</tr>
<tr>
<td>Granularity</td>
<td>fine-grained, coarse-grained</td>
</tr>
<tr>
<td>Binding time</td>
<td>preprocessing</td>
</tr>
<tr>
<td>Reusability</td>
<td>low</td>
</tr>
<tr>
<td>Performance</td>
<td>high</td>
</tr>
<tr>
<td>Application size</td>
<td>low</td>
</tr>
<tr>
<td>Support for modular implementation of crosscutting concerns</td>
<td>not supported</td>
</tr>
<tr>
<td>Legibility</td>
<td>low</td>
</tr>
</tbody>
</table>

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. Finally, like XVCL, conditional compilation has low legibility, since this approach is also language-independent and the developer has to explicitly indicate variation points in the code by means of preprocessing tags.

### 3.10.8 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 3.11.

According to Table 3.11, 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, AOP reusability can be low, since most pointcut languages are still syntactically based [81], which makes aspects too dependent on names. On the other hand, appropriate reuse can increase with EJPs, which allows reusing aspects and SPL core [91], and with virtual classes and propagating mixin composition in CaesarJ [17]. In terms of binding time, it can be at compile time with AspectJ and CaesarJ, at deployment time with AspectBox, and at runtime with CaesarJ. These respective binding times imply
Table 3.11: AOP described according to the variability framework.

<table>
<thead>
<tr>
<th>Framework item</th>
<th>AOP possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability type</td>
<td>positive, negative</td>
</tr>
<tr>
<td>Variability in structure</td>
<td>supported</td>
</tr>
<tr>
<td>Variability of behavior</td>
<td>supported</td>
</tr>
<tr>
<td>Granularity</td>
<td>fine-grained, coarse-grained</td>
</tr>
<tr>
<td>Binding time</td>
<td>compile, run-time, deployment time</td>
</tr>
<tr>
<td>Reusability</td>
<td>low, medium</td>
</tr>
<tr>
<td>Performance</td>
<td>high</td>
</tr>
<tr>
<td>Application size</td>
<td>varies</td>
</tr>
<tr>
<td>Support for modular implementation of crosscutting concerns</td>
<td>supported</td>
</tr>
<tr>
<td>Legibility</td>
<td>high</td>
</tr>
</tbody>
</table>

high, medium, or low performance. 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 [11], part of this can be avoided with optimization techniques. Further, AOP supports modular implementation of crosscutting concerns. Lastly, AOP languages often have an expressive pointcut language which allows expressing variability in a non-invasive way.

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: C (coarse-grained), F (fine-grained), RT (run-time), PT (processing-time), DT (deployment-time), CT (compile-time), empty shape (low/not supported), partially filled shape (medium/partially supported), and filled shape (high/supported).

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.
Table 3.12: Comparing implementation mechanisms according to the variability framework. The following abbreviations are used: C (coarse-grained), F (fine-grained), RT (run-time), PT (processing-time), DT (deployment-time), CT (compile-time), empty shape (low/not supported), partially filled shape (medium/partially supported), and filled shape (high/supported).

<table>
<thead>
<tr>
<th>Framework item</th>
<th>Frameworks</th>
<th>FOP</th>
<th>JPEL</th>
<th>AOP</th>
<th>JaTS</th>
<th>XVCL</th>
<th>Conditional Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability in structure</td>
<td>C</td>
<td>C</td>
<td>F</td>
<td>C,F</td>
<td>C,F</td>
<td>C,F</td>
<td>C,F</td>
</tr>
<tr>
<td>Granularity</td>
<td>Binding time</td>
<td>CT</td>
<td>PT, CT</td>
<td>DT, RT</td>
<td>CT, DT, RT</td>
<td>PT</td>
<td>PT</td>
</tr>
<tr>
<td>Reusability</td>
<td>Performance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application size</td>
<td>Support for modular implementation of crosscutting concerns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Legibility</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 in AOP could be limited with syntactic pointcuts, but could also be improved with EJs [91], which allows reusing aspects and SPL core, and with virtual classes and propagating mixin composition in CaesarJ [17]. Likewise, in FOP, features can also be freely combined with mixin composition, but reusability is not high due to the potentially high fragmentation of code in the refine statements. On the other hand, due to their coarse granularity as a semi-complete application providing an integrated set functionality [120], frameworks support high level of reuse. Similarly, reuse with XVCL is also high due to flexible composition of x-frames. Differently, reusability could be limited in JaTS with templates depending considerably on code within method bodies, or medium, with matching templates applying to set of classes.

Regarding performance, we note that, from the techniques presented, JPEL is the only having a definite low value, since it focuses on providing run-time variability. Frameworks can also be potentially low due to frequent use of dynamic binding to implement variability, which may be pervasive across the framework. AOP can also have low or medium values depending on whether dynamic or deployment binding times are used, respectively. Other techniques have high performance due to preprocessing-time or compile-time binding times.

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.

Even though conditional compilation does not support the modular implementation of crosscutting concerns, once the preprocessing tags are in place, the configuration ac-
tivity 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 3.10.7, 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, most notably reusability. 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 \texttt{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 [95].

As mentioned previously, a significant body of contemporary research [16, 22, 92, 96, 99] has pointed out that the implementation of SPL may be frequently crosscutting. Since AOP aims at providing modular implementation of crosscutting concerns, it is an important candidate for variability management. Indeed, as observed by the comparative analysis, AOP does not maximize all the criteria in the table; however, neither does the other techniques. Even though AOP may have limited reusability and some negative impact on application size (yet to be precisely assessed and possibly tamed as revealed later), the remainder of this thesis will focus on this technique for the reason that the precise exploration for its use in extraction and evolution of SPLs still remains largely unexplored [10, 11]. By doing this, we are not tying our method for implementation of product line adoption strategies to AOP. Indeed, although AOP is used to detail the method, the description of the core of our method (Section 4.1) does not bind a specific technique. In fact, this allows combining different techniques in a project. Accordingly, as pointed out in the case studies (Section 6.3), some variabilities not solved with AOP could be solved with other techniques surveyed and compared in this chapter, such as JaTS, conditional compilation, and OO design patterns.

Finally, 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. Likewise, conditional compilation has poor legibility, since the developer has to pinpoint the variation points with preprocessing tags. AOP (AspectJ), on the other hand, already has an expressive pointcut language which allows expressing variability in a non-invasive way. FOP also offers high legibility due to modularization of features into increments. Differently, JaTS has medium legibility because, although the template language is declarative and an extension of Java, variability is often expressed invasively. Similarly, frameworks and design patterns have limited legibility, because their use and implementation, respectively, is usually difficult to separate from the application classes. Lastly, JPEL differently offers high legibility with a declarative way of specifying relationship among parameters and a simple interface for automatic adjustment of executing processes.
Chapter 4

Implementing Product Lines
Adoption Strategies

In the previous chapter, we described techniques addressing variability implementation in software product lines. The use of such techniques and SPL adoption strategies are orthogonal [16, 88]. This chapter then presents a method for implementing SPL adoption strategies combining the extractive and reactive SPL adoption strategies and extending the notion of refactoring to SPLs. The description is based on the results of our current experience [1, 3, 4, 5, 7, 9, 10, 11, 13, 14, 118]. The 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

The goals of our method are the following: 1) to combine systematically the extractive and reactive SPL adoption strategies; 2) to extend the notion of refactoring to SPLs. 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. 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).
First, **Variability Identification** identifies variation points across existing applications or within an existing SPL. Such variation points are used in each of these activities: a) **Migrate SPL**, to change the variability mechanism employed; b) **Extract SPL**, to bootstrap the SPL from the existing applications; c) **React SPL**, to evolve the SPL to add another instance.

In **Migrate SPL**, since the variation points are only used to perform changes in the variability mechanism, the configurability of the SPL is not changed and thus there is no refactoring of the FM; control then proceeds to **Update Configuration Knowledge**. Alternatively, once the migration is completed, control can proceed to **React SPL**. In **Extract SPL**, the bootstrapping of the SPL from existing applications can proceed either directly at the FM level (transition from **Extract SPL** to **Refactor Feature Model**) and then back to the implementation level (transition from **Refactor Feature Model** to **Extract SPL**) or vice-versa. Control then goes to **React SPL**. Similarly, in **React SPL**, the evolution of the SPL to accommodate another instance can proceed either directly at the FM level and then back to the implementation level or vice-versa. From **Refactor Feature Model**, control can iterate back to **Identify Variability** or proceed to **Update Configuration Knowledge**.

The method supports typical SPL adoption strategies [88], for example by allowing SPL extraction and then reaction. Additionally, the method can be used once for SPL extraction and executed again for SPL migration, for example.

The method 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, 70] 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.

Finally, the method is generic, not being bound to any specific variation mechanism. Complementarily, the comparative analysis in Section 3.11 shows that, depending on the criteria, one technique may be more suitable than another for handling variability, in which case the result of such comparative analysis offers alternative techniques. However, for the reasons given towards the end of Section 3.11, this chapter hereafter details the method in the context of aspects.

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** is performed in existing applications, 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 cor-
responding product-specific adaptation constructs. As explained towards the end of Section 4.1, such constructs could be from the techniques compared in Chapter 3, but, for the purposes of this chapter, the constructs are aspects and possibly supporting classes (classes only appearing in one product). Figure 4.2 depicts this approach (industrial case studies assessing the method are presented in Chapter 6). 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 would be used to derive at least two SPL members.

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 this latter case, these aspects modularize the implementation of crosscutting concerns shared by at least two SPL instances, and the frontier between the core and variants in Figure 4.2 is not an aspect, but instead directly contains the supporting classes of the aspect in the core. The core is composed either with Aspect P1 and its supporting classes (Classes P1), if any, or Aspect P2 and its supporting
classes, if any, in order to instantiate the original specific products. The • operator represents aspect 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 [1, 10, 11]. However, our method is not a “silver bullet”. Indeed, if the previously existing products (Product 1 and Product 2 in Figure 4.2) have design and implementation flaws, the extractive approach in our method could lead to an emerging SPL less robust and less resilient to changes. Nevertheless, as mentioned previously, our method relies on a variability identification step, without which reusable parts of the core may not be discovered. The case studies carried out in Chapter 6 show that considerable commonality is shared across existing products.

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, 78]. 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 corresponding product-specific adaptation construct is created to adapt the core to the new variant. As explained towards the end of Section 4.1, such construct could be one from the techniques compared in Chapter 3, but, for the purposes of this chapter, it is an aspect.

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 and supporting classes, too (class changes are not represented in Figure 4.3).

The impact on the core depends on the degree of commonality between the new product and the existing product line: if such commonality is low, the impact on the core is higher, since a smaller fraction of it will be used to encompass the new product; conversely, if such commonality is high, the impact is lower. Therefore, as in the extractive step (Section 4.1.1), variability identification is essential in the reactive step. Additionally, the decision of whether it is worth or not to encompass the new product depends on using some impact analysis method/tool. Our method assumes that such a decision has been made for the reactive step. Regarding the impact on existing SPL products, AspectJ tools, for example, can identify parts of the core on which these previous aspects depend, and some refactorings could also be aspect-aware [67]: 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.

Another evolution scenario (Figure 4.4) involves restructuring the product line to explore commonality within aspects, thereby aspects and classes depending on aspects can change. Such commonality (AspectFlip aspect) then becomes a core asset, since
it is now explicitly shared by at least two SPL instances (management of assets used in at least two instances of the SPL is done by the configuration knowledge, as illustrated in Section 6.2.4).

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

The scenario illustrated in Figure 4.4 can continue with further reaction of the SPL and restructuring to explore additional reusable aspects. The latent complexity in interaction of the elements of the evolving SPL can be tamed by constraints in the feature model as well as by the configuration knowledge (the mapping of features to aspects), which 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. In the context of R&D projects [4, 13, 14], we have developed this catalog empirically by analyzing variability in a number of mobile games [5, 11, 118]. We asked a team of four software developers from our research projects to use the catalog, which has allowed the team to address most variabilities in this domain. Tool support for the catalog is currently being implemented [11]. However, we have not proved this catalog to be complete, neither do we claim the catalog to be the best option in handling all variabilities, since, as we remarked earlier in Section 4.1, other variability handling mechanisms could be used and the choice of which to use could be guided by the comparative analysis carried out in Section 3.11.

We first specify the language used 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

For specifying the catalog, we consider a subset of AspectJ [40]. The choice of AspectJ is due to the fact that our research is carried out in the context of R&D projects where team members are mostly familiar with this language, which is also the mostly widely used AOP language and the one having most tool support in IDE.
Selecting a subset of AspectJ simplifies the definition of transformations and does not compromise our results. 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`. Finally, we also consider the `declare precedence` construct.

Restricting the use of `this` simplifies the preconditions defined for the refactorings. This can be seen as a global precondition instead of a restriction to the language. Most of the refactorings 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 to capture join points in four different application domains in previous work [40] and in this work.

**An Example of the Refactoring Catalog**

In the context of the mobile device game domain, we consider the optional figures feature of a game. As mentioned in Section, variability identification is required by our method but not part of its core; nevertheless, Chapter 6 illustrates how it can be performed. For the purpose of this example, we examine the corresponding code of such feature declaring and using the `dragonRight` image. First, we consider class `Resources`.

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

where `dragonRight` 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

```java
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. If the attribution could not be moved, we could employ other refactorings, e.g. Refactoring 2, shortly described ahead.

Refactorings like these occur frequently and we thus generalize them using a notation that follows the representation of programming laws [40, 70]. 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 $\textit{this.f}=\textit{exp};$ command ($\textit{exp}$ 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 $\alpha ps$ denotes the parameter names from $ps$. For brevity, we write $\textit{exec}$ and $\textit{ret}$ instead of execution and returning, respectively. In the expression $\textit{exp}[c\textit{this}/\textit{this}]$, the brackets are used to indicate that $\textit{this}$ should be replaced by $c\textit{this}$ everywhere in $\textit{exp}$.

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 $\textit{this.f}=\textit{exp}$ 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

```java
class C {
    T f
    fs
    ms
    T' m(ps) {
        body
        this.f = exp;
        body'
    }
}
```

```java
privileged aspect A {
    T C.f;
    after(C cthis, ps) ret(T' t) : exec(T' C.m(Γ(ps)))
    && this(cthis)
    && args(αps) {
        cthis.f = exp[cthis/this];
    }
}
```
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. Although one could consider dividing A into two aspects to solve the precedence problem, this would mean violating the semantic intention of handling one variability issue within a single aspect.

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 [19], 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):

```java
privileged aspect A {
T C.f;

pcs
bars

after(C cthis, ps) ret(T' t) :
exec(T' C.m(\Gamma(ps)))
&& this(cthis)
&& args(\alpha ps) {
  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.

Table 4.1: Summary of Refactorings.

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Extract Resource to Aspect - after</td>
</tr>
<tr>
<td>2</td>
<td>Extract Method to Aspect</td>
</tr>
<tr>
<td>3</td>
<td>Extract Context</td>
</tr>
<tr>
<td>4</td>
<td>Extract Before Block</td>
</tr>
<tr>
<td>5</td>
<td>Extract After Block</td>
</tr>
<tr>
<td>6</td>
<td>Extract Argument Function</td>
</tr>
<tr>
<td>7</td>
<td>Change Class Hierarchy</td>
</tr>
<tr>
<td>8</td>
<td>Extract Aspect Commonality</td>
</tr>
</tbody>
</table>

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 refactorings we present are not aspect-aware [67]: evolving the core may change some join points within it, so the SPL instance aspects may need to have their pointcuts adjusted to refer to the new join points. On the other hand, our refactorings could be adapted to be aspect-aware 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 [67]. 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. In related work (Subsection 7.2.1), we compare our refactorings to the ones proposed by Hanenberg et al. [67] and by Monteiro and Fernandes [102]. In the following, we describe the remaining of our refactorings.

**Refactoring 2** 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.

**Refactoring 2** *(Extract Method to Aspect)*

```
    ts
    class C {  
        fs
        ms
        T m(ps) { 
            body
            body'
            body''
        }
    }

    privileged aspect A {  
        void C.newm(ps') {  
            body'
        }
    }
```

**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 **σ(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.

This refactoring could be used, for example, when moving the attribution towards the end of the method in Refactoring 1 is not possible. It is also an alternative for
cases when Refactoring 4 and 5 do not apply, since these latter handle variability at beginning or end of method. Concrete examples of using the refactorings are given in Section 6.1.3.

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 [65, 91, 127] has shown that obliviousness is not a 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 to be provided by the aspect in order for the class to perform its role. Such contract is consistent with the notion of Crosscutting Interfaces (XPIs) [65, 127] and Extension Join Points (EJPs) [91].

Refactoring 3 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 an arbitrary Java compound statement encapsulating body and, in particular, can have any nesting.

Refactoring 3 (Extract Context)

```
\begin{verbatim}
s
        • 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 σ(C.m);
        • there is no designator within or withincode capturing join points inside Context.
        
    We can exemplify the use of this refactoring as follows. Considering the context of the SPL mobile games domain, this refactoring could be used to extract the code handling the flipping of an image (Context) into aspect A in devices having the flip feature, whereas the code for basic image manipulation would stay in the core (body in class C), thereby also being shared by devices not having the flip feature.
\end{verbatim}
```
**Refactoring 4** extracts the variant part of a concern, appearing at the beginning of a method body, into AspectJ’s **before** construct.

**Refactoring 4** (Extract Before Block)

\[
\begin{align*}
\text{class } C & \{ \\
& \text{fs} \\
& ms \\
& T' \ m(ps) \{ \\
& \quad \text{body}' \\
& \quad \text{body} \\
& \} \\
\} \rightarrow \\
\} \text{privileged aspect } A & \{ \\
& \text{before } (C \ cthis, \ ps) : \\
& \quad \text{exec}(T' \ C.m(\Gamma(ps))) \\
& \quad \&\& \text{this}(cthis) \\
& \quad \&\& \text{args}(\alpha ps) \{ \\
& \quad \quad \text{body}' [cthis/this] \\
& \} \\
\}
\end{align*}
\]

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’**.

For example, in the context of the SPL mobile games domain, where method \( m \) could be implementing screen changing, this refactoring could be used to extract the code handling on demand image loading, which appears at the beginning of the method (**body’**), into aspect **A** in devices having the on demand feature for image loading, whereas the code for basic screen change would stay in the core (**body** in class **C**), thus also being shared by devices not having such feature.
Unlike the previous refactoring, **Refactoring 5** extracts the variant part of a concern, appearing at the end of a method body, into AspectJ’s **after** construct.

**Refactoring 5** (Extract After Block)

```java
class C {
  fs
  ms
  T' m(ps) {
    body
    body'
  }
}
```

→

```java
privileged aspect A {
  after(C cthis, ps) ret(T' t):
    exec(T' C.m(Γ(ps)))
    & & this(cthis)
    & & args(αps) {
      body' [cthis/this]
    }
}
```

**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'`.

An example of usage of this refactoring would be the following. Considering the context of the SPL mobile games domain, where method \( m \) could be computing player’s score and posting it on a network board to compare across different players (Arena feature), this refactoring could be used to extract the code related to this feature, which appears at the end of the method (\( body' \)), into aspect \( A \) in devices having the Arena feature, whereas the code for basic score computation would stay in the core (\( body \) in class \( C \)), thus also being shared by devices not having the Arena feature.
Refactoring 6 extracts the variant part of a concern, appearing as a function over an expression in a 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)

```
-ts
class C {
  fs
  ms
  $T_n(ps)$ {
    exp.$m(f(x))$
  }
}

→
} privileged aspect A {
  around (context):
    call($O.m(\Gamma(x)))$ &&
    withincode($C.n(\Gamma(ps)))$
    && bind(context) {
      proceed($f(x))$
    }
}
```

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 $f(x)$.

An illustrative use of Refactoring 6 would be as follows. In the context of the SPL mobile games domain, where $f$ could be performing arithmetic calculation regarding offset/scale of player elements in device-specific screens, this refactoring could be used to extract the code referring to such device-specific calculation into aspect A so that the code for basic player element display would remain in the core ($exp.m(x)$ in class C), thus also being customized by aspects for devices with different screen sizes.
While the previous refactorings are more fine-grained, **Refactoring 7** is coarse-grained, since it changes the type hierarchy using AspectJ’s *declare parents* construct: the fact that type D is not only a subtype of C but also a subtype of C’ is expressed by an *declare parents* construct in aspect A.

**Refactoring 7** (Class Hierarchy)

```markdown
<table>
<thead>
<tr>
<th>ts</th>
<th>class C' extends C {...}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>class D extends C' {...}</td>
</tr>
</tbody>
</table>

→

```markdown
<table>
<thead>
<tr>
<th>ts</th>
<th>class C' extends C {...}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>class D extends C' {...}</td>
</tr>
<tr>
<td></td>
<td>privileged aspect A {</td>
</tr>
<tr>
<td></td>
<td>declare parents</td>
</tr>
<tr>
<td></td>
<td>D extends C'</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
```

provided

- D and C' have the same interface.

To illustrate, Refactoring 7 could be used to extract the variability resulting from the requirement of certain APIs to make application classes conform to some specific type hierarchies. For example, in the SPL mobile games domain, C could be the Canvas superclass of a standard J2ME profile, and C’ could be a device-specific super class part of an API to be used by the game developer to instantiate the SPL for that device. Game class D, part of the core of the SPL, should then become a subclass of C’, a fact which is expressed in aspect A by a *declare parents* construct.

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.

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. For example, in the SPL mobile games domain, *body* could be code related to flipping an image by using device-specific API, which would then be factored out in aspect C. This aspect is then omitted for games for devices not supporting such API.

The second and third preconditions of Refactoring 8 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)

```
privileged aspect A {
  fieldsA
  methodsA
  pointcutsA
  barsA
  before(ps) : exp1{
    body
  }
  bars'A
  afsA
}
privileged aspect B {
  fieldsB
  methodsB
  pointcutsB
  barsB
  before(ps) : exp2{
    body
  }
  bars'B
  afsB
}
privileged aspect C {
  fieldsC
  methodsC
  pointcutsC
  barsC
  before(ps) : exp1 || exp2 {
    body
  }
  afsC
}
```

provided

- The set of join points captured by \( exp1 \) and \( exp2 \) are disjoint;
- \( exp1 \) and \( exp2 \) 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`. 
There are variations of this refactoring for the other kinds of advice constructs. In another variation, there is no aspect $C$ on the left-hand side template, in which case this aspect is then created on the right-hand side template. Likewise, other variations do not have the privileged modifier. Such variations can be related to the original version of the refactorings.

There may be cases where the code to be extracted could be matched by the left-hand side template of different refactorings. For example, code matched by $\text{body}'$ and $\text{body}''$ in the left-hand side template of Refactoring 2 (Extract Method to Aspect) could also be matched by $\text{body}'$ in the left-hand side template of Refactoring 5 (Extract After Block). In such case, the disambiguation is done by using the refactorings intent, which is given by its name. In the example, we are either interested in extracting such code into a method (Refactoring 2) or as an after advice (Refactoring 5).

### 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 [1, 3, 4, 5, 7, 9, 10, 11, 13, 14, 118], 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](image)

**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 [115] 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 dif-
ferent 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.

```java
#ifndef nokia_device
// import com.nokia.mid.ui.FullCanvas;
#else
// import javax.microedition.lcdui.Canvas;
#endif
...
#ifndef 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.

```java
//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:

```java
//ifdef TAG
//  ts'
//endif

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

//ifdef TAG
// public class C extends C'' {
//endif

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.

```java
//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.

```java
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:

```java
public class MainCanvas extends Canvas {
    protected void keyPressed(int keyCode) {...
        if(compareEquals(keyCode, LEFT_SOFT_KEY)) {
            // 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'
        }
    }
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 other 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.

```java
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;
    // ElseIf device_screen_128x117
    // public static final int ARENA_SCROLL_HEIGHT = 81;
    // public static final int ARENA_SCROLL_POS_Y = 16;
    // 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.

```java
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`.

```java
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.

```java
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:

```java
/** Width used to show loading message */
#ifdef device_screen_128x117
    public static final int LOADING_MESSAGE_AREA = 118;
#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:

```java
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;
    }
}
```
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:

```java
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:

```java
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:

```java
public privileged aspect SoundPlayerNokia {
    public void SoundEffects.playSound () {
        [code X]
    }
}
```

```java
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 done 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 unnecessarily complex class hierarchies. Additionally, many more classes would be involved unnecessarily, thus also incurring into a penalty in terms of bytecode size, a critical issue some domains, such
Table 4.2: Relating Migration Strategies to Refactorings

<table>
<thead>
<tr>
<th>Migration Strategy</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super class variation</td>
<td>Change class hierarchy</td>
</tr>
<tr>
<td>Interface implementation variation</td>
<td>Change class hierarchy</td>
</tr>
<tr>
<td>If condition variation</td>
<td>Extract context</td>
</tr>
<tr>
<td>Feature dependency</td>
<td>Extract resource to aspect</td>
</tr>
<tr>
<td>Variability in constant declaration</td>
<td>Extract resource to aspect</td>
</tr>
<tr>
<td>Variability in method body</td>
<td>Extract method to Aspect</td>
</tr>
<tr>
<td>Variability in method call</td>
<td>Extract Before/After Block</td>
</tr>
</tbody>
</table>

as the mobile application domain. However, the strategies presented are not intended to be specific to such domain.

The strategies replace the scattered `ifdef` statements by a number of aspects, which have to be managed. This can be 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. This benefit is independent of the domain of application.

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.

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 [112].

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 of the latter.

### 4.2.1 Programming Laws

Programming laws [40, 70], 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, 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)**

\[
\text{ts class } C \{ \\
\quad fs \\
\quad ms \\
\quad T \ m(ps) \{ \\
\quad \quad \text{body} \\
\quad \} \\
\} \ \text{privileged aspect } A \{ \\
\quad pcs \\
\quad \quad as \\
\} = \text{ts class } C \{ \\
\quad fs \\
\quad ms \\
\quad T \ C.m(ps) \{ \\
\quad \quad \text{body} \\
\quad \} \\
\quad pcs \\
\quad \quad as \\
\} \]

provided

$\leftrightarrow$ A does not introduce any field to $C$ with the same name of a $C$ field used in $\text{body}$. 
A variant of this law lacks aspect A on the left-hand side template. This variant and the original version of the law can be related by the Add empty aspect law \[40\]: in the left-hand side template of the variant, we apply the Add empty aspect law to obtain the original version of Law 1. Likewise, another variant of this law does not have the privileged modifier on the templates. In this case, the variant and the original version of the law can be related by the Make aspect privileged law \[40\]: in the left-hand side template of the variant, we apply the Make aspect privileged law to obtain the original version of Law 1.

For example, 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⟩

\[
\text{class } C \{ \\
\quad \text{fs} \\
\quad \text{ms} \\
\quad T \ m(ps) \{ \\
\quad \quad \text{body} \\
\quad \qquad body' \\
\quad \} \\
\} \\
\text{privileged aspect } A \{ \\
\quad \text{pcs} \\
\quad \text{bars} \\
\text{after}(C\ cthis,\ ps)\ \text{ret}(T'\ t): \\
\quad \text{exec}(T'\ C.m(\Gamma(ps))) \\
\quad \&\&\ \text{this}(cthis) \\
\quad \&\&\ \text{args}(\alpha ps) \{ \\
\quad \quad body'[\text{cthis/this}] \\
\quad \} \\
\} \\
\text{afs}
\]

provided

(\(\rightarrow\)) body’ does not use local variables declared in body; body’ does not call super;

(\(\leftrightarrow\)) A has the lowest precedence on the join points involving the signature \(\sigma(C.m)\);

(\(\leftrightarrow\)) 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 \textit{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 (\textit{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 \textit{A} has the lowest precedence over other aspects defined in \textit{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 \texttt{field} declaration is removed from class \texttt{C}. However, we introduce \texttt{field} in this class by using an inter-type declaration construct declared in aspect \texttt{A}.

**Law 3** (Move Field to Aspect)

\[
\begin{array}{l}
\text{ts} \\
\text{class } C \{ \\
\quad \textit{fs} \\
\quad T \texttt{ field} \\
\quad \textit{ms} \\
\} \\
\text{privileged aspect } A \{ \\
\quad \textit{pcs} \\
\quad \textit{bars} \\
\quad \textit{afs} \\
\} \\
\end{array}
= \\
\begin{array}{l}
\text{ts} \\
\text{class } C \{ \\
\quad \textit{fs} \\
\quad \textit{ms} \\
\} \\
\text{privileged aspect } A \{ \\
\quad T \texttt{ C.field} \\
\quad \textit{pcs} \\
\quad \textit{bars} \\
\quad \textit{afs} \\
\} \\
\end{array}
\]

provided

\[\rightarrow\) The field \textit{field} of class \textit{C} does not appear in \textit{ts} and \textit{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 pre-
conditions under which those refactorings are valid. For their simplicity, programming
laws [70] 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 applica-
tion, 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. Accord-
ingly, 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 [70]—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 trans-
formation 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 \texttt{newm} with proper parameters and return type, which executes
the piece of code labelled as \texttt{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 [28].

Note that the scenario after the method extraction matches the left side of Move
Method to Aspect law, with \texttt{newm} corresponding to \texttt{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.

\begin{figure}[h]
\centering
\begin{tabular}{c}
\textbf{Extract Method $\rightarrow$ Move Method to Aspect} \\
\end{tabular}
\caption{Extract Method to Aspect.}
\end{figure}

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 \texttt{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.

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).

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 \(\rightarrow\), 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.
Table 4.3: Summary of Refactorings Derivations. Consecutive application of laws is represented by $\rightarrow$. Repeated application of a law is denoted with a superscript $^*$. 

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Derivation of refactoring in terms of laws</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract Method to Aspect</td>
<td>Extract Method $\rightarrow$ Move Method to Aspect</td>
</tr>
<tr>
<td>Extract Resource to Aspect - after OO Refactoring</td>
<td>Add After-Execution Returning Successfully $\rightarrow$ Move Field to Aspect</td>
</tr>
<tr>
<td>Extract Context</td>
<td>Add Around-Execution</td>
</tr>
<tr>
<td>Extract Before Block</td>
<td>Add Before-Execution</td>
</tr>
<tr>
<td>Extract After Block</td>
<td>Add After-Execution Returning Successfully</td>
</tr>
<tr>
<td>Extract Argument Function</td>
<td>Add Around-Call</td>
</tr>
<tr>
<td>Class Hierarchy</td>
<td>Extend From Super Type</td>
</tr>
<tr>
<td>Extract Aspect Commonality</td>
<td>Change Advice Order $\rightarrow$ Move Advice $^*$</td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$ Merge Advice</td>
</tr>
</tbody>
</table>
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, in the context of the extractive and reactive adoption strategies, in which existing products are bootstrapped into a SPL or a SPL evolves, respectively, the resulting FM has increased number of instances, i.e., higher configurability. Therefore, 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 [8], 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 Chapter 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 in order to consistently describe variability at a higher level of abstraction in SPL development. In particular, such step guarantees that SPL configurability (number of SPL instances) does not decrease: in the extractive approach, configurability remains the same (SPL bootstrapping), whereas, in the reactive approach, such configurability increases (SPL evolving). Although the extractive and reactive approaches can be enacted by the application of program refactor-
ings, the traditional definition of program refactoring [53, 104] 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 [104]. He proposed refactorings as behavior-preserving program transformations in order to support the iterative design of object-oriented application frameworks [104]. The cornerstone of his definition is that refactorings must maintain correct compilation and observable behavior. However, in practice [53], 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 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, i.e., the resulting FM could have fewer instances, which is not desirable in the extractive and reactive SPL strategies. We must certify that $FM_2$ (corresponding to program $A_2$) improves the possible configurations of $FM_1$, i.e., such FM has the same number or a higher number of configurations. 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). Shortly ahead, in Section 5.2.2, we present and explain an extended definition of refactoring encompassing FMs and relying on the notion of configurability improvement.

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.

We remark that the notion of configurability improvement is well-aligned with the usual intended meaning of refactoring. Indeed, refactoring means that 1) behavior is preserved (what worked before still works after the refactoring); 2) a quality attribute increases. By configurability improvement in the definition above, we mean that a) previously existing configurations still exist after the refactoring, thus achieving 1); b) there may be new configurations; in particular, this does not affect the existing configurations in a), so 1) is still valid; additionally, the possibly new configurations imply that the overall FM may have a higher number of instances; considering the extractive and reactive SPL context, this represents a desirable quality property, which is then increased, thereby achieving 2). Our definition of refactoring is only presented in this chapter and not before in Chapter 4 because it is an extension of the definition based only on code, which was used in Chapter 4.

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 preexisting reusable code (Core) among these 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 \(\langle\text{merge programs}\rangle\)

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, which are used to guarantee that the programs to be merged have a common core.

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.

![Feature Diagram Notations](image)

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 \( (A, B, C \text{ and } D) \), one formula \( (B \Rightarrow \neg C) \) and two relationships: an option relationship between \( A \) and \( B \), and an or relationship between \( A, C \) and \( D \).

![Feature Model Example](image)

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 [27] 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.

![Class Diagram depicting Feature Model Components](image)

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 \( \text{values}(c) \) 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 \( B \lor C \) if \( c \) contains \( B \) or \( C \). As another example, a configuration satisfies the formula \( B \Rightarrow \neg C \) if \( c \) contains \( B \) but not \( C \).

We remark that this formalization is orthogonal to other elements of the FM, such as feature definitions and rationale for features. Indeed, our method makes no use of feature rationale. This could be explored in future work in the context of testing strategies for SPLs. In terms of feature definitions, detailed feature descriptions are useful for any method involving transformations of FM. The binding time in feature descriptions could be used for constraining the search for an appropriate variability mechanism in Section 3.11.

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.

According to the explanation presented directly after Definition 1, in Section 5.2.2, we emphasize that Definition 2 is in accordance with the intuitive definition of refactoring. In Definition 2, the fact that new features are possible (e.g., alternative and optional features) only occurs in the new configurations; previously existing configurations remain the same after the refactoring, thus behavior is preserved. The sole purpose of the new configurations is to show a FM with new instances, an essential property in extracting and evolving SPLs.

Let \( m_1 \) and \( m_2 \) be two FMs. So, according to Definition 2, \( m_2 \) refactors \( m_1 \) if and only if all valid configurations of \( m_1 \) are valid configurations of \( m_2 \), as formalized next:

\[
\text{refactoring}(m_1,m_2: FM): boolean = 
\text{semantics}(m_1) \subseteq \text{semantics}(m_2)
\]

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.

![Diagram of feature model refactoring example](image)

**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 line on top of a feature indicates that this feature may have a parent feature, whereas a line below a feature indicates that this feature may have additional subfeatures.

### 5.4.3 Unidirectional Refactorings

Next we propose FM refactorings in order to solve Problem 1 (Section 5.2.1). We refer to these as unidirectional refactorings because they increase FM configurability and thus are only applied from LHS to RHS. Later, in Section 5.4.4, we present bidirectional refactorings, which maintain FM configurability, thereby being applied in either direction. The unidirectional refactorings are listed in Table 5.1. In particular, 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 line below $D$ in the RHS of this refactoring because it only introduces a new feature without subfeatures. The other lines of the RHS are necessary to preserve the features matched by 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 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, because in this latter only $C$ and $D$ are or-features and thus at least one of
Table 5.1: Summary of Unidirectional Feature Model Refactorings

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Convert Alternative to Or</td>
</tr>
<tr>
<td>2</td>
<td>Collapse Optional and Or</td>
</tr>
<tr>
<td>3</td>
<td>Collapse Optional and Alternative to Or</td>
</tr>
<tr>
<td>4</td>
<td>Add Or Between Mandatory</td>
</tr>
<tr>
<td>5</td>
<td>Add New Alternative</td>
</tr>
<tr>
<td>6</td>
<td>Convert Or to Optional</td>
</tr>
<tr>
<td>7</td>
<td>Convert Mandatory to Optional</td>
</tr>
<tr>
<td>8</td>
<td>Convert Alternative to Optional</td>
</tr>
<tr>
<td>9</td>
<td>Pull Up Node</td>
</tr>
<tr>
<td>10</td>
<td>Push Down Node</td>
</tr>
<tr>
<td>11</td>
<td>Remove Formula</td>
</tr>
<tr>
<td>12</td>
<td>Add Optional Node</td>
</tr>
</tbody>
</table>

these should be selected. Therefore, the configuration \(\{A, B\}\) is on RHS model but not on the LHS model. Therefore, the LHS model is not a refactoring of the RHS model. 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 instance, 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

\[
\begin{array}{c}
\text{Refactoring 3:} \quad \text{collapse optional and alternative to or} \\
\end{array}
\]

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)

\[
\begin{array}{c}
\text{Extractive 1:} \quad \text{merge optional and alternative} \\
\end{array}
\]

Suppose that \( fm_1, fm_2 \) represent the two LHS FMs, and \( fm_3 \) the RHS FM, respectively, of the Extractive 1 refactoring. \( fm_3 \) improves the configurations of \( fm_1 \) by
applying Refactoring 12 in order to introduce the optional feature node $C$. Moreover, $fm_3$ improves the configurations of $fm_2$ by applying Refactoring 8 in order to convert an alternative to option. Therefore, $fm_3$ refactors both $fm_1$ and $fm_2$. 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)**

![Diagram of B-Refactoring 1]

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 \(\langle replace \ or \rangle\)

\[
\begin{array}{c}
\text{forms} \\ \text{A} \\ f\hat{A} \\
\text{B} \\ f\hat{B} \\
\text{C} \\ f\hat{C} \\
\end{array}
\]

\[
\begin{array}{c}
\text{forms} \land (A \Rightarrow (B \lor C)) \\
\end{array}
\]

B-Refactoring 3 \(\langle replace \ mandatory \rangle\)

\[
\begin{array}{c}
\text{forms} \\ \text{A} \\ f\hat{A} \\
\text{B} \\ f\hat{B} \\
\end{array}
\]

\[
\begin{array}{c}
\text{forms} \land (A \Rightarrow B) \\
\end{array}
\]

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

The set of seven B-Refactorings is \textit{sound, minimal} and \textit{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 \((\text{semantics}(fm1)=\text{semantics}(fm2))\).

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).
3. Since $fm_1$ and $fm_2$ have the same semantics, they present the same features ($\text{features}(fm_1)=\text{features}(fm_2)$). However, $fm_1$ and $fm_2$ may have syntactically-different formulas, although equivalent. Since the propositional logic calculus is complete, we can always prove that $\text{forms}(fm_1)=\text{forms}(fm_2)$ by applying B-Refactoring 7 for introducing deduced formulas.

Therefore, we have shown that we can relate $fm_1$ and $fm_2$ 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
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. FM refactoring is used in the context of extractive and reactive SPL adoption strategies, which define the scope of this thesis. On the other hand, FM specialization is not used in the context of SPL adoption strategies, but rather in the context of optimization based on partial evaluation, and hierarchical enforcement of policy standards, and software supply chains [46].

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

discussed in Sections 5.4.3 and 5.4.4.

### 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 [88] 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 [1, 10]. 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 [1, 10]. 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);
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.

- 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.

### 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 Product1 with the aim of modularizing the croma and the image loading policy features. In Figure 5.9, the $+$ symbol in Product1 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 SPL1.

Similarly, we apply those refactorings in order to modularize the OnDemand loading image feature of Product2. After that, we accomplish our second goal, evolving the product into a new SPL (named SPL2) by adding a new kind of image loading policy (Hybrid).

Finally, in order to avoid code duplication, our aim is to integrate SPL1 and SPL2, due to their similar core. As SPL1 and SPL2 must have exactly the same core for merging both SPLs, we apply adjustment refactorings (such as renaming) to the core of SPL1. We can now apply our merge program Refactoring SPL 1, which was presented in Section 5.2.2, in order to merge SPL1 and SPL2 into SPL1-2.

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 Product1 are shown in Figure 5.10. For making Clouds optional, we can apply Refactoring 7 resulting into SPL SPL1. In a later step, as we applied program refactorings to start preparing SPL1 for an extractive refactoring, it demands no changes on the FM (a reflexive step).

In the source code for Product2, 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 SPL2, 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 SPL1 and SPL2. The resulting SPL (SPL1-2) thus encompasses SPL1 and SPL2. At the FM level, we can generate SPL1-2 (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 SPL1 and SPL2, 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 the following statement:

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

In the first branch, \( \text{SPL1} \rightarrow \text{SPL1-2} \), 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.

Assessing the hypotheses underlying our method, in terms of FMs, we remark that our FM refactoring catalog is a useful guide for extracting and evolving SPLs. Indeed, the catalog defines FM transformations that either maintain or increase configurability. These refactorings are also specified in a declarative way in terms of templates, which increases their applicability and legibility. Moreover, each transformation focuses on few FM constructs and has a specific goal, thus enabling its composition in consecutive use. Further, we have illustrated the application of FM refactorings in a real scenario.

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.

Regarding reuse, since each transformation in our FM refactoring catalog either maintains or increases configurability, the resulting FM does not have worse configurability than at the beginning of the extractive/reactive process. Therefore, the resulting FM represents a domain model that is more reusable than the pre-existing configurations/FMs.

In terms of safety, our FM refactoring catalog defines sound transformations as argued in Sections 5.4.3 and 5.6, thus supporting safety. Besides ensuring confidence on correct transformations, our 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.

### 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 \( \text{forms} \) 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 an 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 \(\langle\text{convert alternative to or}\rangle\)

Refactoring 4 \(\langle\text{add or between mandatory}\rangle\)

Refactoring 6 \(\langle\text{convert or to optional}\rangle\)
Refactoring 7 \( (\text{convert mandatory to optional}) \)

Refactoring 8 \( (\text{convert alternative to optional}) \)

Refactoring 9 \( (\text{pull up node}) \)
Refactoring 10 \(\langle\text{push down node}\rangle\)

Refactoring 11 \(\langle\text{remove formula}\rangle\)

Refactoring 12 \(\langle\text{add optional node}\rangle\)
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 refactorings 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 [5, 33], 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.
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.

\[ P_1 = \{\text{Dragon, Bonus Weapon, Levels, Optional Image, At startup, Flip, 176x208}\}; \]
\[ P_2 = \{\text{Dragon, Bonus Weapon, Levels, On demand, Flip, 120x160}\}; \]
\[ P_3 = \{\text{Dragon, Bonus Weapon, Levels, Optional Image, At startup, 128x128}\}; \]

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. These steps were executed by the second author of paper [11], under guidance of the author of this thesis. 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 and supporting tool [116]. 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 [116], 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:

$$C_{p_1,oi} = (V_{p_1,oi}, V_{p_1,oi}^*, E_{p_1,oi}), V_{p_1,oi}^* = \emptyset$$

$$V_{p_1,oi} = \{\text{Resources, GameScreen, Resources.dragonRight, Resources.loadImages(), GameScreen.wakeEnemy()}\}$$

$$E_{p_1,oi} = \{(\text{reads, GameScreen.wakeEnemy()}, \text{Resources.dragonRight}), (\text{writes, Resources.loadImages()}, \text{Resources.dragonRight}), (\text{declares, Resources, Resources.dragonRight}), (\text{declares, Resources, loadImages()}), (\text{declares, GameScreen, wakeEnemy()})\}$$

The set $V_{p_1,oi}$ describes the vertices (classes, methods, attributes) partially implementing the concern. Set $V_{p_1,oi}^*$ consists of vertices (classes, methods) solely dedicated to the concern implementation. Set $E_{p_1,oi}$ groups edges relating elements from the previous sets. $(\text{reads, } m, f)$ and $(\text{writes, } m, f)$ denote that field $f$ is read and written, respectively, in method $m$. $(\text{declares, } c, mb)$ means that member $mb$ (method or field) is declared in class $c$.

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. In the extractive and in the evolution scenarios, if we had first considered $P_1$ and $P_2$, before handling $P_3$, then (*AspectFlip*) would probably have been identified earlier.

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 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.

### 6.1.4 Configuration Knowledge

The resulting configuration knowledge maps sets of features to implementation artifacts as in the following:
Table 6.1: Occurrence of each refactoring

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Name</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Extract Resource to Aspect - after</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Extract Method to Aspect</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>Extract Context</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Extract Before Block</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Extract After Block</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>Extract Argument Function</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Change Class Hierarchy</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Extract Aspect Commonality</td>
<td>1</td>
</tr>
</tbody>
</table>

{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 that 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 aspect weaving. 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. Although a feature like Optional Image is implemented in more than one aspect, its implementation actually changes according to the platform, since each such aspect is targeting a specific platform. Therefore, duplication of features in the configuration knowledge does not imply code replication. In fact, we could apply our refactorings to further explore eventual code reuse. However, since this product line has a small number of instances, this has not been explored. We intentionally explore more fine-grained code reuse in the case study in Section 6.2, which identifies more reusable aspects and configurations.
Table 6.2: LOC in original and SPL implementations

<table>
<thead>
<tr>
<th>Original Implementation</th>
<th>SPL Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Core assets</td>
</tr>
<tr>
<td></td>
<td>Core classes</td>
</tr>
<tr>
<td></td>
<td>Core aspects</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2,965</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2,968</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3,143</td>
</tr>
<tr>
<td>Total</td>
<td>9,076</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2,477</td>
</tr>
<tr>
<td>$P_2$</td>
<td>72</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3,042</td>
</tr>
<tr>
<td>Total</td>
<td>4,405</td>
</tr>
</tbody>
</table>

6.1.5 Analysis

After creation and evolution of the SPL, we analyzed code metrics. Table 6.2 shows the 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 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 Tables 6.2 and 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 $P_1$ and $P_2$ to 20% for $P_3$: the configuration knowledge shows which aspects are part of each product; LOC sizes of these aspects are then obtained from Table 6.3; finally, the overall size of aspect code for each product is divided by the size of each product, the latter obtained from Table 6.2.

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
Table 6.3: LOC of aspects in the SPL.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectP1</td>
<td>421</td>
</tr>
<tr>
<td>AspectP2</td>
<td>426</td>
</tr>
<tr>
<td>AspectP3</td>
<td>661</td>
</tr>
<tr>
<td>AspectFlip</td>
<td>72</td>
</tr>
</tbody>
</table>

Table 6.4: Application size (Kbytes) in original and SPL implementations

<table>
<thead>
<tr>
<th></th>
<th>Original Implementation</th>
<th>SPL Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>reduced size</td>
</tr>
<tr>
<td>$P_1$</td>
<td>32.4</td>
<td>29.0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>33.2</td>
<td>28.8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>56.1</td>
<td>52.4</td>
</tr>
<tr>
<td>Total</td>
<td>98.1</td>
<td>86.6</td>
</tr>
</tbody>
</table>

application, such as images and sound files. In the case study products, additional resources represent, 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 of the refactorings. Moreover, we can gain a significant reduction in the jar size when using a bytecode optimization tool [136]. 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 reduced, 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 match few join points, by inlining the body of the advice in the base code, we can already reduce bytecode 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 scores 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.

![Figure 6.3: Best lap’s main screen](image)

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 conditional 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 is configured as follows:

```plaintext
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_mmaapi, 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 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 we only migrated the SPL implementation, the FM remained the same. Therefore, 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. The execution of steps in Sections 6.2.2, 6.2.3, and 6.2.4 was done by four developers (acknowledged at the beginning of this thesis) in a R&D project [14], under guidance of the author of this thesis.

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

<table>
<thead>
<tr>
<th>Tag</th>
<th>Total Occurrence</th>
<th>Classes involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>device_screen_128x128</td>
<td>33</td>
<td>7</td>
</tr>
<tr>
<td>device_screen_128x149</td>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td>device_screen_128x117</td>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>device_screen_128x160</td>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>device_screen_132x176</td>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>feature_arena_enabled</td>
<td>26</td>
<td>7</td>
</tr>
<tr>
<td>device_sound_api_nokia</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>device_graphics_canvas_midp2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>device_graphics_canvas_siemens</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>known_issue_no_softkeys</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

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.
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

<table>
<thead>
<tr>
<th>Class</th>
<th>Number of tags in class</th>
</tr>
</thead>
<tbody>
<tr>
<td>GameScreen</td>
<td>29</td>
</tr>
<tr>
<td>MainScreen</td>
<td>3</td>
</tr>
<tr>
<td>GameMenu</td>
<td>4</td>
</tr>
<tr>
<td>LevelManager</td>
<td>1</td>
</tr>
<tr>
<td>MainCanvas</td>
<td>4</td>
</tr>
<tr>
<td>MidletController</td>
<td>1</td>
</tr>
<tr>
<td>NetworkFacade</td>
<td>3</td>
</tr>
<tr>
<td>Resources</td>
<td>9</td>
</tr>
<tr>
<td>SoundEffects</td>
<td>8</td>
</tr>
<tr>
<td>Screen</td>
<td>2</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Migration Strategy</th>
<th>Name</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Variability in constant declaration</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>Field extraction</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Super class variation</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Variability in method call</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>If condition variation</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>Variability in method body</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>Feature dependency</td>
<td>6</td>
</tr>
</tbody>
</table>

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.

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
Table 6.8: Occurrence of programming laws in each refactoring

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

\[
\text{MOT1} = \text{CoreClassesCC} \bullet
\]

\{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\}

where \( \bullet \) denotes composition. \text{CoreClassesCC} denotes the core of the SPL consisting of 14 classes and still some variability with conditional compilation. \text{CoreClassesCC} was not mentioned in Table 6.9, since it was used for all mandatory features. Additionally, such core still contains some conditional compilation tags as remarked in Section 6.3, since some variability could not be extracted with the migration strategies, as also described and explained in Section 6.3. The other elements of MOT1 are aspects necessary for instantiating this particular instance.

Each Best Lap SPL instance consists of \text{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 \text{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. The first percentage is obtained by adding 17 and 20–corresponding to the number of core aspects–under column \# \text{aspects} of Table 6.10, then dividing by 66, the total number of aspects. The second percentage is obtained by adding 6 and 15, corresponding to percentage amounts of core aspects under column \% of SPL code of Table 6.10. This shows that the fine granularity of the mapping enabled tangible reuse of aspects across SPL instances.

\subsection*{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
Table 6.9: SPL configuration knowledge

<table>
<thead>
<tr>
<th>Feature</th>
<th>Aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>device/graphics/canvas/Midp2Canvas</td>
<td>device.graphics.canvas.Midp2CanvasAspect.aj</td>
</tr>
<tr>
<td></td>
<td>device.graphics.canvas.PaintCanvasGraphics.aj</td>
</tr>
<tr>
<td>device/graphics/canvas/NokiaCanvas</td>
<td>device.graphics.canvas.NokiaCanvasAspect.aj</td>
</tr>
<tr>
<td>device/graphics/canvas/SiemensCanvas</td>
<td>device.graphics.canvas.SiemensCanvasAspect.aj</td>
</tr>
<tr>
<td>device/keys/KeysNokiaSonyEricsson</td>
<td>device.keys.DeviceKeysNokiaSonyEricsson.aj</td>
</tr>
<tr>
<td>device/keys/KeysMotorola</td>
<td>device.keys.DeviceKeysMotorola.aj</td>
</tr>
<tr>
<td>device/keys/KeysSiemens</td>
<td>device.keys.DeviceKeysSiemens.aj</td>
</tr>
<tr>
<td>device/screen/Screen128x117</td>
<td>device.screen.Screen128x117.aj</td>
</tr>
<tr>
<td></td>
<td>Screen128x117_128x160_128x149_128x128</td>
</tr>
<tr>
<td></td>
<td>device.screen.LowEndScreen</td>
</tr>
<tr>
<td>device/screen/Screen128x128</td>
<td>device.screen.Screen128x128.aj</td>
</tr>
<tr>
<td></td>
<td>Screen128x117_128x160_128x149_128x128</td>
</tr>
<tr>
<td></td>
<td>device.screen.LowEndScreen</td>
</tr>
<tr>
<td>device/screen/Screen128x149</td>
<td>device.screen.Screen128x149.aj</td>
</tr>
<tr>
<td></td>
<td>Screen128x117_128x160_128x149_128x128</td>
</tr>
<tr>
<td></td>
<td>device.screen.LowEndScreen</td>
</tr>
<tr>
<td>device/screen/Screen128x160</td>
<td>device.screen.Screen128x160.aj</td>
</tr>
<tr>
<td></td>
<td>Screen128x117_128x160_128x149_128x128</td>
</tr>
<tr>
<td></td>
<td>device.screen.LowEndScreen</td>
</tr>
<tr>
<td>device/screen/Screen132x176</td>
<td>device.screen.Screen132x176.aj</td>
</tr>
<tr>
<td></td>
<td>Screen128x117_128x160_128x149_128x128</td>
</tr>
<tr>
<td></td>
<td>device.screen.LowEndScreen</td>
</tr>
<tr>
<td>device/screen/Screen176x205</td>
<td>device.screen.Screen176x205.aj</td>
</tr>
<tr>
<td></td>
<td>Screen176x205_176x208</td>
</tr>
<tr>
<td>device/screen/Screen176x208</td>
<td>device.screen.Screen176x208.aj</td>
</tr>
<tr>
<td></td>
<td>Screen176x205_176x208</td>
</tr>
<tr>
<td>device/screen/Screen176x220</td>
<td>device.screen.Screen176x220.aj</td>
</tr>
<tr>
<td></td>
<td>device.screen.HighEndScreen</td>
</tr>
<tr>
<td>device/sound/policy/SoundOnDemand</td>
<td>device.sound.policy.SoundOnDemand.aj</td>
</tr>
<tr>
<td>device/sound/policy/SoundPreallocation</td>
<td>device.sound.policy.SoundPreallocation.aj</td>
</tr>
<tr>
<td>device/sound/content/TypeMID</td>
<td>device.sound.content.TypeMID.aj</td>
</tr>
<tr>
<td>device/sound/content/TypeMIDI</td>
<td>device.sound.content.TypeMIDI.aj</td>
</tr>
<tr>
<td>device/sound/content/TypeXMID</td>
<td>device.sound.content.TypeXMID.aj</td>
</tr>
<tr>
<td>device/sound/content/TypeXMIDI</td>
<td>device.sound.content.TypeXMIDI.aj</td>
</tr>
<tr>
<td>Feature/aspect</td>
<td>File Name</td>
</tr>
<tr>
<td>----------------</td>
<td>----------</td>
</tr>
<tr>
<td>device/sound/api/PlayerMMAPI</td>
<td>device.sound.api.SoundPlayerMMAPI.aj</td>
</tr>
<tr>
<td>device/sound/api/PlayerNokia</td>
<td>device.sound.api.SoundPlayerNokia.aj</td>
</tr>
<tr>
<td>device/sound/api/PlayerSiemens</td>
<td>device.sound.api.SoundPlayerSiemens.aj</td>
</tr>
<tr>
<td>device/sound/api/PlayerSamsung</td>
<td>device.sound.api.SoundPlayerSamsung.aj</td>
</tr>
<tr>
<td>device/sound/thread/ThreadedSound</td>
<td>device.sound.thread.ThreadedSoundAspect.aj</td>
</tr>
<tr>
<td>device/sound/thread/BlockingSound</td>
<td>device.sound.thread.BlockingSoundAspect.aj</td>
</tr>
<tr>
<td>Multiplayer/arena</td>
<td>feature.arena.ArenaAspect.aj</td>
</tr>
<tr>
<td>knownIssues/Garbage collector</td>
<td>knownIssues.MemoryHasGC.aj</td>
</tr>
<tr>
<td>knownIssues/Softkey</td>
<td>knownIssues.HasSoftkeyAspect.aj</td>
</tr>
<tr>
<td>knownIssues/NoSoftkey</td>
<td>knownIssues.NoSoftkeyAspect.aj</td>
</tr>
<tr>
<td>general/multilanguage</td>
<td>general.multilanguage.MultiLanguageEnabled.aj</td>
</tr>
<tr>
<td>general/debug</td>
<td>general.debug.DebugMode.aj</td>
</tr>
</tbody>
</table>

Table 6.10: Reuse of aspects in BestLap SPL

<table>
<thead>
<tr>
<th>aspect category</th>
<th># aspects</th>
<th># instances</th>
<th>% of SPL code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product specific aspects</td>
<td>29</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>Core aspects</td>
<td>17</td>
<td>[10,16]</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>[2,9]</td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td>66</td>
<td>[1,16]</td>
<td>32</td>
</tr>
</tbody>
</table>
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 6.12 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 [136].

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 explain that such shortcomings are inherent to the AO paradigm. In either case, we provide alternative solution using other approaches, such as those surveyed and compared in Chapter 3; 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.
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 (such as JaTS [36]) that uses generative techniques allowing to control such kind of elements in the source code. Yet another alternative would be to use the Abstract Factory design pattern; nevertheless, it provide runtime variability instead of preprocessing time.

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.

```java
//#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 call 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. Alternatively, it would not be desirable to use an inter-type declaration construct to introduce the specialized constructor, since this would lead to code replication, as the only difference between the constructors would be regarding the superclass constructor call.

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.

```java
... 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
    }  //ifdef feature_arena_enabled
```

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–or any AO language–extension that intercepts conditional statements does not seem very useful, since the conditional statements are not named, which leads to ambiguity when a method has more than one conditional statement. Yet another alternative would be to use the State design pattern, which, nevertheless, provides runtime variability instead of preprocessing time; additionally, such pattern is usually preferred for coarser-grained variability, since it requires a new class defining the variant algorithm, which can be an issue in resource constrained domains such as mobile games.

This issue can be generalized by the following form:

```java
... 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, assessing the hypotheses of our method and discussing limitations of the case studies. This section focuses on the implementation level, since Section 5.5.2 assessed the method in terms of FM. 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. Accordingly, we handled fewer instances in RoF: three instances, which are thus a subset of the instances existing at the industrial partner; on the other hand, in Best Lap we addressed more instances, i.e., sixteen instances, which correspond to the actual number of instances of the existing SPL in our industrial partner.

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.

In terms of providing concrete guidelines, during extraction in the RoF case (Section 6.1.3), refactorings and programming laws were employed to extract/react variability from the code guided by the identified concern graphs. In Best Lap (Section 6.2.3), 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.

Regarding reuse, although in RoF there is only one reusable aspect and this represents a reduced fraction of the SPL LOC, considerable reuse was achieved (48% reduction in the total LOC of the SPL implementation), which as expected since a SPL was extracted from existing applications. Additionally, the configuration knowledge in RoF is coarse-grained and relates sets of features to classes and aspects. In Best Lap, the LOC of each SPL implementation is approximately the same as in the CC implementation, but the AO implementation has improved reusability, locality, adaptability, plugability, and independent development. In contrast to the configuration knowledge in RoF, 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 (56% of the total aspects), and these compose a tangible amount of SPL LOC (21%). Moreover, the configuration knowledge in Best Lap has constraints and some non-compositional mappings.

Our method supports safety because its 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 transformations (Section 4.2). In fact, according to the developers involved in the case studies, the bugs found during such case studies
related only to how the bytecode optimization tool [136] worked together with AspectJ, an orthogonal issue to our method.

Despite being evaluated only in one domain, the method is still adequate for extracting and reacting SPLs. As explained in Section 2.3, the domain used for evaluation—the mobile games domain—is representative of mobile applications in general and is also rich in variability, which is a key concern in SPLs. Moreover, despite the fact that mobile applications are not large scale applications, the former are non-trivial and offer considerable complexity because they must adhere to stronger portability requirements than desktop applications (Section 2.3).

Additionally, although we have not proved the completeness of our refactorings, these have been used to extract most variabilities in the mobile games domain. In cases in which variability could not be extracted in this domain (Section 6.3), we have suggested extensions in the AOP language used (AspectJ) or the use of alternative variability implementation mechanisms, the choice of which relies on the mechanisms surveyed and compared in Chapter 3. Regarding other domains, we believe that our method is still useful for extracting and reacting SPLs, but possibly need to be extended with more refactorings to address further variability. However, more empirical studies are needed in order to assess completeness in other domains.

In terms of usability, we remark that the method has been effectively used in industrial-strength mobile games in order to extract and react SPLs. In both case studies, the refactoring catalog (Section 4.1.3) has provided guidelines on the use for extracting/reacting SPLs. Similarly to the FM refactorings from Chapter 5, the refactorings in the implementation-level catalog are also specified in a declarative way in terms of templates, which promotes their legibility and potential applicability in other domains. Moreover, each transformation focuses on few language constructs and has a specific goal, thus enabling its composition in consecutive use. Sections 6.1.3 and 6.2.3 show, in the case studies, recurring strategies that can be used with the refactorings. Further, although an increase on application size was noticed, that has not prevented installation of all of the games in the phones and their proper functioning. We are currently studying how optimization techniques can further minimize this potential issue.

The choice of the mobile games domain prevented us from using certain mechanisms of the language used in the refactorings (AspectJ), such as cflow, because its runtime implementation has considerable size for this domain and would thus limit the resulting application size, since that runtime implementation has also to be included in the application executable for the devices, which do not support AspectJ currently. Despite this, the AspectJ subset we use is expressive, since it has sufficed for capturing join points and modularizing concerns in four different application domains in previous work [40]. The extensions of AspectJ that we suggested (Section 6.3) are not prompted because omission of AspectJ mechanisms, but rather by inherent limitations of the language in handling certain variabilities. The impact of extending the language to include such mechanisms would involve defining new refactorings and underlying programming laws in our method.

In RoF, before applying our extractive method, the existing products had been previously developed by our industrial partner in a scenario by copying and adapting the previously existing versions. Although it could be argued that this scenario helps the extractive step, we note that, based on our experience [1, 4, 5, 13, 14, 118], this
scenario is common in practice. Further, our extractive step assumes the existence of a variability identification step, the quality of which is essential for the applicability of the method. Therefore, had the RoF versions been developed initially in a different scenario without copying and adapting, the variability identification step could have been more elaborate, and possibly require complementary techniques in addition to concern graphs such as those surveyed elsewhere [82].
Figure 6.4: Variability within Best Lap
Figure 6.5: Plug-in for identifying variability in the conditional compilation 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.

<table>
<thead>
<tr>
<th>instance</th>
<th>LOC</th>
<th>SPL CC</th>
<th>SPL OA</th>
<th>AO</th>
</tr>
</thead>
<tbody>
<tr>
<td>mot1</td>
<td>10,102</td>
<td></td>
<td></td>
<td>10,227</td>
</tr>
<tr>
<td>mot2</td>
<td>10,102</td>
<td></td>
<td></td>
<td>10,279</td>
</tr>
<tr>
<td>mot3</td>
<td>10,104</td>
<td></td>
<td></td>
<td>10,232</td>
</tr>
<tr>
<td>s40m2</td>
<td>9,974</td>
<td></td>
<td></td>
<td>10,182</td>
</tr>
<tr>
<td>s40m2v3</td>
<td>10,000</td>
<td></td>
<td></td>
<td>10,189</td>
</tr>
<tr>
<td>s40</td>
<td>9,894</td>
<td></td>
<td></td>
<td>10,103</td>
</tr>
<tr>
<td>s60m1</td>
<td>9,965</td>
<td></td>
<td></td>
<td>10,254</td>
</tr>
<tr>
<td>s60m2</td>
<td>9,974</td>
<td></td>
<td></td>
<td>10,263</td>
</tr>
<tr>
<td>sam1</td>
<td>8,781</td>
<td>14,516</td>
<td>14,848</td>
<td>8,930</td>
</tr>
<tr>
<td>sam2</td>
<td>8,700</td>
<td></td>
<td></td>
<td>8,827</td>
</tr>
<tr>
<td>se02</td>
<td>9,949</td>
<td></td>
<td></td>
<td>10,174</td>
</tr>
<tr>
<td>se03</td>
<td>10,009</td>
<td></td>
<td></td>
<td>10,147</td>
</tr>
<tr>
<td>se04</td>
<td>10,001</td>
<td></td>
<td></td>
<td>10,186</td>
</tr>
<tr>
<td>siem02</td>
<td>9,937</td>
<td></td>
<td></td>
<td>10,169</td>
</tr>
<tr>
<td>siem3</td>
<td>9,924</td>
<td></td>
<td></td>
<td>10,229</td>
</tr>
<tr>
<td>siem4</td>
<td>9,925</td>
<td></td>
<td></td>
<td>10,235</td>
</tr>
</tbody>
</table>
Table 6.12: Application (jar) Sizes of Best Lap SPL

<table>
<thead>
<tr>
<th></th>
<th>CC SPL</th>
<th>AO SPL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>optimized size</td>
<td>size</td>
</tr>
<tr>
<td>MOT1</td>
<td>71.5</td>
<td>80.7</td>
</tr>
<tr>
<td>MOT2</td>
<td>83.6</td>
<td>92.8</td>
</tr>
<tr>
<td>MOT3</td>
<td>71.5</td>
<td>80.7</td>
</tr>
<tr>
<td>S40</td>
<td>64.1</td>
<td>73.5</td>
</tr>
<tr>
<td>S40M2</td>
<td>74.6</td>
<td>80.8</td>
</tr>
<tr>
<td>S40M2V3</td>
<td>71.1</td>
<td>80.3</td>
</tr>
<tr>
<td>S60M1</td>
<td>83.9</td>
<td>93.3</td>
</tr>
<tr>
<td>S60M2</td>
<td>82.8</td>
<td>91.9</td>
</tr>
<tr>
<td>SAM1</td>
<td>65.3</td>
<td>73.4</td>
</tr>
<tr>
<td>SAM2</td>
<td>78.9</td>
<td>87</td>
</tr>
<tr>
<td>SE02</td>
<td>82.7</td>
<td>91.8</td>
</tr>
<tr>
<td>SE03</td>
<td>71.2</td>
<td>80.4</td>
</tr>
<tr>
<td>SE04</td>
<td>71.1</td>
<td>80.3</td>
</tr>
<tr>
<td>SIEM02</td>
<td>80.3</td>
<td>80.3</td>
</tr>
<tr>
<td>SIEM03</td>
<td>71.8</td>
<td>81</td>
</tr>
<tr>
<td>SIEM04</td>
<td>71.1</td>
<td>80.2</td>
</tr>
</tbody>
</table>
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 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 a novel 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 adequate for extracting and evolving SPLs, thereby meeting the hypothesis of the thesis. 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.

Indeed, the method provides concrete guidelines for extracting and evolving SPLs at the implementation and feature model levels 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
transformations at either level of abstraction and which promotes their legibility and
potential applicability in other domains. Moreover, each transformation focuses on few
language constructs and has a specific goal, thus enabling its composition in consecutive
use. The case studies conducted show recurring strategies that can be used with the
refactorings. In addition, refactoring preconditions (a frequently subtle issue) are more
clearly organized and not tangled with the transformation itself.

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.

Although our program refactorings are described in an AOP language, the descrip-
tion of the core of our method (Section 4.1) does not bind a specific a technique for
implementation of product line adoption strategies and thus the method is generic. In
fact, this allows combining different techniques in a project. Accordingly, as pointed
out in the case studies (Section 6.3), some variabilities not solved with AOP could be
solved with other techniques surveyed and compared in Chapter 3. For example, AOP
does not handle very fine-grained variability, which is better addressed by conditional
compilation. Furthermore, we also showed how to enhance AOP to handle additional
variability issues.

The method supports reuse at the implementation and feature model levels. In-
deed, at the implementation level, our refactoring catalog defines refactorings separ-
ating product-specific code from core SPL code. Although in RoF there is only one
reusable aspect and this represents a reduced fraction of the SPL LOC, considerable
reuse was achieved (48% reduction in the total LOC of the SPL implementation), which
as expected since a SPL was extracted from existing applications. Additionally, the con-
figuration knowledge in RoF is coarse-grained and relates sets of features to classes and
aspects. In Best Lap, the LOC of each SPL implementation is approximately the same
as in the CC implementation, but the AO implementation has improved reusability,
locality, adaptability, plugability, and independent development. In contrast to the con-
figuration knowledge in RoF, 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 associ-
ated with a significant number of reusable aspects (56% of the total aspects), and these
compose a tangible amount of SPL LOC (21%). At the feature model level, since each
transformation in our feature model refactoring catalog either maintains or increases
configurability, the resulting feature model does not have worse configurability than at
the beginning of the extractive/reactive process. Therefore, the resulting model repre-
sents a domain model that is more reusable than the pre-existing configurations/feature
models.

The method supports safety at the implementation and feature model levels, since
the refactorings at either level 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 and reactive transformations. In fact, according to the developers involved in the case studies, the bugs found during such case studies related only to how the bytecode optimization tool [136] worked together with AspectJ, an orthogonal issue to our method. Nevertheless, additional empirical evidence needs to be carried out to further assess this hypothesis.

Despite being evaluated only in one domain, the method is still adequate for extracting and reacting SPLs. As explained in Section 2.3, the domain used for evaluation—the mobile games domain—is representative of mobile applications in general and is also rich in variability, which is a key concern in SPLs. Moreover, despite the fact that mobile applications are not large scale applications, the former are non-trivial and offer considerable complexity because they must adhere to stronger portability requirements than desktop applications (Section 2.3). Regarding other domains, we believe that our method is still useful for extracting and reacting SPLs, but possibly need to be extended with more refactorings to address further variability. However, more empirical studies are needed in order to assess completeness in other domains.

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. Although increase on binary application size was noticed in the case studies we conducted, that has not prevented installation of all of the games in the phones and their proper functioning. Further, optimization techniques can reduce this potential issue.

In addition to the benefits of the method outlined above, which are the core contributions in the scope of this work, we have also designed and implemented a prototype of a tool for supporting variability management in the SPL context [11]. 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 [32]. We do not describe it here, since it is not in the scope of this work.

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 variation 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 [137].

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 [112]. 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 transac-
tion 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 [12].
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. In particular, controlled experiments
on software maintenance could be planned to validate the comparisons between the AO
and CC versions (Section 6.2.5). Ultimately, this could lead to the definition of a pattern
language to support the method, whereby program and feature model refactoring would
each be related to one another in an explicit strategy; this could also be explored for
the definition of a tactics language.

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 engineer-
ing 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 metrics
suite for SPLs at different levels of abstraction, 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 future research is extending the prototype to
support variability management, which we have developed. In order to 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 [128].
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 tool
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 [15]. 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 [116] 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.

Monteiro and Fernandes [102] have proposed a catalog for object-to-aspect and aspect-to-aspect refactorings [102]. Their Extract Feature into Aspect is similar to our Refactoring 1 (Extract Resource to Aspect - after): as part of their goal, both use inter-type declaration to move state to the aspect; differently, behavior is moved into aspect using their Move Method From Class to Inter-type and Extract Fragment into Advice refactorings, whereas Refactoring 1 uses after construct; however, our catalog also includes Refactoring 2, which allows moving behavior in a method to inter-type declaration. Their research also presents refactorings for refactorings to deal with generalization. In particular, Extract Superaspect moves code related to common features to a supersaspectbut. Our Refactoring 8 has the similar purpose of extracting aspect commonality, but it does not extract such commonality into a superaspect; also, our refactoring does not require the pointcuts corresponding to such extracted piece of advice to be the same, but rather that such pointcuts be disjoint. Further, such research does not present equivalents of Refactorings 3 and 6.

Hanenberg et al. [67] have defined aspect-aware refactorings to be ones that can be applied to the base program of an aspect-oriented system [67]. They have also defined enabling conditions to ensure behavior preservation, and then illustrate aspect-aware versions of Rename Method [53] and Extract Method [53]. Finally, they have presented two object-to-aspect refactorings and one aspect-to-aspect refactoring. In the former group, Extract Introduction is similar to part of Refactoring 1, which also use advice construct. Extract Advice has a similar role to Refactoring 4, but uses an around construct instead. Finally, in the second group, Separate Pointcut performs pointcut simplification and does not have a corresponding in our catalog. However, such refactoring corresponds to a combination of the programming laws Extract Named Pointcut [40] and Use Named Pointcut [40], on which our refactoring catalog is built.

Hannenmann et al. [69] provide an abstract representation of object-to-aspect refactorings as roles. However, there are fundamental differences from our refactorings to these refactorings as well as those by Monteiro and Fernandes, and those by Hanenberg et al: 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. Furthermore, our refactorings are derived from elementary AOP programming laws 4.2.1.

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 [140]. 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.

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 [91] builds on the notion of crosscutting interfaces [65, 127] 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 refactorings 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 object-oriented languages [28]. 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) [108], 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 needs 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 of 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 [107], which encompasses a formal specification language and a theorem prover.

7.2.3 Refactoring Product Lines

A related approach [92] proposes 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 [86] 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 [20] 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 seman-
tics 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 [129] 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 [23] 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 [93], an efficient optimization algorithm could be applied in order to extract the product line core from these specifications with the Shared Class Extractor operator [93]. 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 [134] and J2ME Polish [135] 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 [103]. 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 [132] 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 [35, 54, 71] 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 [141]. 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


[71] Hao hua Chu, Henry Song, Candy Wong, Shoji Kurakake, and Masaji Kata- 
giri. Roam, a seamless application framework. Journal of Systems and Software, 

[72] Paul Hudak. Building domain-specific embedded languages. ACM Computing 


[74] Ivar Jacobson, Grady Booch, and James Rumbaugh. The Unified Software Devel-

[75] Ivar Jacobson, Martin Griss, and Patrik Jonsson. Software Reuse: Architecture, 

[76] 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 

[77] Ralph E. Johnson. Documenting frameworks using patterns. In OOPSLA ’92: 
conference proceedings on Object-oriented programming systems, languages, and 


Oriented Domain Analysis (FODA) Feasibility Study. Technical Report 
CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[80] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moon-
hang Huh. Form: A feature-oriented reuse method with domain-specific reference 

of aspect-oriented software with model-based pointcuts. In European Confer-

aspect mining techniques. Transactions on Aspect-Oriented Software Development 

[83] Gregor Kiczales. Aspect–oriented programming. ACM Computing, 28(154), De-
ceember 1996.

[84] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and 
William G. Griswold. Getting Started with AspectJ. Communications of the 


Tese de Doutorado apresentada por Vander Ramos Alves à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título “Implementing Software Product Line Adoption Strategies”, orientada pelo Prof. Paulo Henrique Monteiro Borba e co-orientação do Prof. Geber Lisboa Ramalho e aprovada pela Banca Examinadora formada pelos professores:

Prof. Silvio Romero de Lemos Meira  
Centro de Informática / UFPE

Prof. André Luís de Medeiros Santos  
Centro de Informática / UFPE

Prof. Alessandro Fábricio Garcia  
Departamento de Computação / Universidade de Lancaster - Inglaterra

Profa. Rosana Teresinha Vaeceare Braga  
Departamento de Ciências da Computação e Estatística / USP

Prof. Sérgio Castelo Branco Soares  
Departamento de Sistemas Computacionais / UPE

Visto e permitida a impressão.  
Recife, 23 de março de 2007.

Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO  
Coordenador da Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco.