Universidade Federal de Pernambuco - UFPE

Centro de Informática - CIn

Pós-graduação em Ciência da Computação

RESTRUCTURING TEST VARIABILITIES IN SOFTWARE PRODUCT LINES

Márcio de Medeiros Ribeiro

DISSERTAÇÃO DE MESTRADO

Recife - PE

25 de Fevereiro de 2008

Universidade Federal de Pernambuco - UFPE
Centro de Informática - CIn

Márcio de Medeiros Ribeiro

RESTRUCTURING TEST VARIABILITIES IN SOFTWARE PRODUCT LINES

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática - CIn da Universidade Federal de Pernambuco - UFPE como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Prof. Dr. Paulo Henrique Monteiro Borba*

Recife - PE
25 de Fevereiro de 2008

To the three most important persons of my life: mother, father, and step-father. I love you so much!

AGRADECIMENTOS

Ao Deus eterno e todo poderoso, pois tudo é do pai, toda honra e toda glória alcançada em minha vida.

Agradeço e dedico este trabalho às três pessoas mais importantes da minha vida: (i) mãe, por seu infinito amor, carinho e dedicação que transcende qualquer distância, seja ela Maceió-Recife ou Maceió-Plutão; (ii) pai, pelo seu incentivo constante cheio de otimismo em qualquer obstáculo da minha vida; (iii) padrasto e segundo pai, por tanta confiança, carinho e amor depositados em minha pessoa. Agradeço também aos demais membros da família que, embora não citados, têm tamanha importância nesta minha vitória.

Ao meu orientador, Professor Paulo Borba, de quem sou fã inconteste, por tanta competência, dedicação e paciência para com o este aluno estressado. Certamente, sem ele, este sonho não estaria sendo concretizado.

Ao apoio técnico fundamental do Fúlvio Silvestre, Silvia Sampaio, Anne Ximenes e Gisele Leal, todos do CIn/Motorola, no uso do *framework* da Motorola. Adicionalmente, agradeço ao Fernando Calheiros da *Meantime Mobile Creations*, pela ajuda fundamental sobre como estender a ferramenta FLiPEx.

À minha namorada e companheira, Helô, por tanto carinho e dedicação, além de toda a confiança depositada em mim. Agradeço sobremaneira pelo amparo fundamental nos momentos tristes (sumariamente por estar longe da família) e pelos belos sorrisos dos momentos felizes, deixando o coração alegre e pronto para enfrentar as batalhas da vida.

Por fim, agradeço a todos os mestres que fizeram parte do meu Mestrado bem como aos meus queridos amigos dos grupos SPG e CIn/Motorola, em especial, Rodrigo Bonifácio (o famoso Tese Pronta), Pedro Matos Júnior (POA), Alberto Costa Neto, Marcos Dósea e Ivan Cardim (Grande Líder). Mais aprendi do que ensinei a estas pessoas maravilhosas...

I think 99 times and do not discover anything. I stop thinking, dive into a deep silence and the truth is revealed to me.

— ALBERT EINSTEIN

RESUMO

Linhas de Produto de Software (LPS) englobam famílias de sistemas desenvolvidos a partir de artefatos reusáveis. Um fator importante durante a manutenção de LPS consiste em decidir sobre qual mecanismo deve ser utilizado para reestruturar suas variações objetivando melhorar a modularidade de seus artefatos. Devido à grande variedade de mecanismos, selecionar os corretos pode ser uma tarefa difícil. Por outro lado, selecionar os incorretos pode produzir efeitos negativos no custo de desenvolver a LPS.

É importante salientar que este problema existe não somente no nível de código fonte, mas também em outros artefatos como requisitos de software e testes. Assim sendo, para reduzir tal problema no nível de testes automatizados, este trabalho propõe um modelo de decisão que ajuda desenvolvedores a escolher mecanismos para reestruturar variações de testes em LPS. Para construir o modelo, algumas variações encontradas em casos de teste automatizados reais desenvolvidos pela Motorola foram analisadas. Neste caso, os testes servem para testar os sistemas de software dos telefones celulares da Motorola. Os testes lidam com as variações dos diferentes celulares usando condicionais *if-else*. Portanto, dada uma variação baseada em condicionais *if-else*, o modelo sugere um mecanismo para prover uma melhor modularidade da variação. Adicionalmente, uma ferramenta para dar suporte aos desenvolvedores de LPS foi desenvolvida. A ferramenta recomenda os mecanismos de acordo com o modelo de decisão proposto.

Aplicando o modelo de decisão e os mecanismos sugeridos por ele pode melhorar a modularidade das variações dos casos de teste e remover problemas como códigos duplicados. Ademais, mostra-se que a tarefa de reestruturar variações torna-se mais rápida e precisa quando a ferramenta é utilizada.

Palavras-chave: Linhas de Produto de Software, Modularidade, Testes de Software

ABSTRACT

Software Product Lines (SPLs) encompass a family of software-intensive systems developed from reusable assets. One issue during SPL maintenance is the decision about which mechanism should be used to restructure product line variabilities aiming at improving the modularity of the SPL artifacts. Due to the great variety of mechanisms, selecting the correct ones may be a difficult task. On the other hand, selecting the incorrect ones may produce negative effects on the cost to evolve the SPL.

It is important to note that this problem exists not only at source code, but also in other artifacts such as requirements and tests. Therefore, in order to reduce the problem at the testing level, we propose in this work a decision model to help developers on the task of choosing mechanisms to restructure test variabilities in SPLs. In order to construct our decision model, we have analyzed some variabilities of real automated test cases developed by Motorola Industrial for testing mobile phone software. The test cases handle the phone's variabilities by using *if-else* statements. Therefore, given an *if-else* based variability, the model is able to suggest mechanisms which provide better modularity. We also developed a tool to support developers when restructuring these test variabilities by recommending mechanisms according to the decision model.

Applying the decision model and the mechanisms suggested by it may improve the tests variabilities' modularity and remove bad smells such as cloned code. Furthermore, we illustrate in this work that when using the tool, the task of restructuring some variabilities may be realized faster and precisely.

Keywords: Software Product Lines, Modularity, Software Testing

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Software development is considered a challenging task. In particular, there is a critical need to reduce cost, effort, and time-to-market of software. At the same time, systems complexity and size are increasing and customers are requesting products that fit their individual needs. In this context, Software Product Lines (SPLs) is an approach aiming at improving software productivity, reducing costs, efforts, and development time [58].

Software Product Lines encompass a family of software-intensive systems developed from reusable assets (known as core assets). By reusing such assets it is possible to construct a large scale of products through specific variabilities defined according to customers' requirements. On the other hand, implementation activities become more complex because they also have to realize variabilities [56].

In this context, reasoning about how to combine both core assets and product variabilities is a challenging task [16]. In other words, the challenge consists of understanding the available mechanisms (such as Inheritance, Configuration Files, Conditional Compilation, Aspect-Oriented Programming, and so forth) for realizing variability and knowing which of them fits best for a given variability at hand [56]. Previous work [47, 15] has structured product line variabilities by using only one mechanism. Because each mechanism has strengths and weaknesses, not all variabilities were well structured when considering modularity criteria, for example.

Due to the great variety of available mechanisms, selecting an incorrect mechanism may produce negative effects on the cost to maintain the SPL [33]. For example, cloned code and concerns not modularized may appear, affecting independent evolution of SPL artifacts, increasing developer's effort, and consequently decreasing productivity when evolving the SPL.

It is important to note that the problem of combining core assets and SPL variabilities exists not only at source code, but also in other artifacts such as requirements and tests. In order to reduce the aforementioned problems at the testing level, we have

defined a Decision Model [33, 32] to help developers on the task of choosing mechanisms to restructure test variabilities of existing SPLs. To construct our decision model, we analyzed variabilities found in Motorola mobile phone integration test cases. The test variabilities analyzed were handled by using *if-else* statements. For example, the *if* body is executed to test product A, whereas the *else* body tests product B. The motivation to restructure them is that such approach does not provide modularity at all; variabilities are not separated from core assets. The model aims at suggesting mechanisms so that these test variabilities can be modularized, removing the *if-else* statements from the test cases.

Besides, our decision model is code-centric because the variability is identified by its exact location at the source code. For example: the variability is at the beginning of a method body. Being code-centric allows SPL developers to choose more easily which mechanism to use when handling variabilities at the source code level. Another advantage is that since the model considers fine-grained code, it is easier to apply refactorings [35], whose target result uses the mechanism recommended by the decision model. Contrasting previous work [16, 56, 30, 60] provide a high-level approach when compared to our proposal. Another particular difference of our model is that, because it considers test variabilities, the mechanisms used to restructure the variabilities must preserve the order of the test steps: changing their order of execution may break the tests.

The methodology to construct the decision model is explained as follows. Each analyzed test variability of the Motorola mobile phone integration test cases was restructured using different mechanisms. Differently of existing models [16, 56, 30, 60], our decision model is based not only on qualitative but also on quantitative studies. For this reason, we apply some software metrics to compare the implementation of each mechanism and then analyze its advantages and disadvantages. Finally, we adapt the decision model to encompass such new variability.

In order to improve even more developer's productivity, a tool for supporting the decision model is essential. For example, when evolving product line variabilities, a tool may reduce developer's effort, avoiding time consuming and error-prone tasks like finding where an existing cloned variability is. For this reason, we have developed a prototype tool for supporting developers when restructuring product line variabilities implemented using *if-else* statements. The tool, named `FLiPRec`, is an extension of the `FLiPEx` tool [26].

`FLiPEx` is able to extract product line variabilities from Java classes to AspectJ aspects. Our tool only recommends (the extraction is not realized) mechanisms to restructure variabilities according to the decision model. However, differently of `FLiPEx`, `FLiPRec` searches for cloned code (and consequently recommends a mechanism which deals with such cloning), being important to modularize the cloned variability as a whole. Thus, although `FLiPEx` uses aspects to restructure the variabilities, the pointcuts produced by such tool is not always as suitable as the ones `FLiPRec` recommends (obviously, when `FLiPRec` recommends the Aspect-Oriented Programming [46] mechanism).

The assumption for using our model and tool is that the test variabilities in the SPL are not well structured (they are not separated from core assets); and are handled by using *if-else* statements. The hypothesis of our work is that putting both model and tool together may provide several benefits such as:

- bad smells such as cloned code can be identified faster and precisely. Accepting and applying the mechanism suggested to remove them may avoid future inconsistences or even bugs [35];
- using the mechanisms recommended by the decision model may improve the variabilities' modularity, allowing developers to work in parallel and increasing their productivity;
- due to the improved modularization, time consuming and error-prone tasks may be avoided, increasing developer's productivity and reducing the time-to-market.

In what follows, we present the summary of goals and the organization of this work.

1.1 SUMMARY OF GOALS

Our work has the following main goals:

- construct a decision model for restructuring test variabilities in SPL; and
- develop a tool for supporting developers when restructuring SPL variabilities by recommending suitable mechanisms to implement them.

1.2 ORGANIZATION

The remainder of this work is organized as follows:

⟹ **Chapter 2** reviews the essential concepts used throughout the work: Software Product Lines, Software Modularity, Software Testing, Software Metrics, and Software Refactoring;

⟹ **Chapter 3** discusses some available variability implementation mechanisms used in this work, ranging from simple ones like Inheritance to more complex ones like Aspect-Oriented Programming (AOP);

⟹ **Chapter 4** presents the Motorola framework where the tests are implemented, then we show how we have found the test variabilities in such framework, and finally we present the construction of the proposed decision model;

⟹ **Chapter 5:** presents the tool that we have developed to support developers when restructuring product line variabilities by recommending mechanisms to implement them;

⟹ **Chapter 6:** presents two scenarios based on the real Motorola tests aiming at evaluating our work;

⟹ **Chapter 7:** describes the concluding remarks, discussing our contributions, the related work, the limitations of our approach, and the future work.

CHAPTER 2

BACKGROUND

In this chapter we review essential concepts explored in this work. First, we discuss concepts related to Software Product Lines (SPL) in Section 2.1. In this context, in order to restructure test variabilities in SPLs, modularity plays an important role. This way, we also discuss some concepts related to Software Testing in Section 2.2 and Software Modularity in Section 2.3. Next, we provide an illustrative example to show the Software Metrics used in this work in Section 2.4. The metrics will be used to compare mechanisms that implement SPL variabilities. Last but not least, Software Refactoring is discussed in Section 2.5. This topic is important for this work since after deciding which mechanism should be used to restructure the test variabilities, refactorings must be done.

2.1 SOFTWARE PRODUCT LINES

Software Product Lines (SPL) is a paradigm to develop software applications using platforms and mass customization [58]. The platforms represent the core assets presented in products (being reused throughout the SPL), whereas the mass customization represents the large scale production concerning the products variabilities according to the customers' requirements.

Basically, SPL define two processes. The first one comprehends the core assets development. It establishes the reusable platform and consequently the commonalities and variabilities of the SPL, being known as *Domain Engineering*. The second one is responsible for deriving product line applications based on the predefined core assets. Therefore, it ensures the correct binding of the variabilities according to the applications specific needs. Such product development process is known as *Application Engineering*.

Some advantages of adopting the SPL approach are outlined below:

- **Reduction of development costs:** when artifacts are reusable in several different kinds of systems, this implies a cost reduction for each system, since there is no

need to develop such components from scratch;

- **Enhancement of quality:** the core assets of a SPL are reused in many products. In this way, they are tested and reviewed many times, which means that the higher chance of detecting faults and correcting them improves the quality of the products;
- **Reduction of time-to-market:** initially, the time-to-market of the SPL is high, because the core assets must be developed first. Afterwards, the time-to-market is reduced, because many components previously developed might be reused for each new product.

However, in order to gain all of these advantages, managing in a suitable way the product variabilities of the SPL is essential. Such variabilities play a key role in the SPL, since different applications of the SPL can be distinguished in terms of these variabilities [14]. In addition, they enable the development of customized applications by reusing some predefined artifacts.

Figure 2.1 illustrates three Motorola mobile phones. The common artifact reused throughout the phones is the software responsible for playing the video. However, each phone has specific variabilities. Notice that *Phone A* does not provide the *stop* button whereas *Phone B* and *Phone C* do. On the other hand, *Phone B* provides the *camera landscape view*, differently from *Phone A* and *Phone C*. This simple example shows that the variability is the basis for mass customization.

Figure 2.1 Variabilities in Motorola Mobile Phones.

Variabilities encountered in the SPL context can be mapped in features present in the products. The Feature-Oriented Domain Analysis (FODA) is a domain analysis method

focused on the description of commonalities and variabilities by means of features, where a feature is a prominent and user-visible aspect, quality, or characteristic of a software system or systems [42].

Knowing the type of a feature may be important for the implementation, since it can indicate which mechanism is recommended or not for implementing the feature. In what follows, we describe the main feature types:

- **Mandatory:** the feature must be always present;
- **Optional:** the feature is an independent complement that may be included or not;
- **Alternative:** the feature replaces another feature when included.

A FODA feature model consists basically of feature diagrams and other components not discussed here. The full treatment of the FODA approach is beyond the scope of this work; details can be found elsewhere [42]. The feature diagram depicts a hierarchical decomposition of features with mandatory (filled circle), optional (open circle), and alternative (open arc) relationships.

Figure 2.2 illustrates part of a Multimedia feature diagram of a mobile phones family. *Figure List Navigation* and *Media State* are mandatory features. The list navigation of figures can have one or two dimensions. They are alternative features: if one is present, the other must not be. *Play* and *Pause* features are mandatory, which means that they must be present in the mobile phones of the family. On the other hand, *Stop*, *Media Finder*, and *Category* are optional features, since some phones do not provide them.

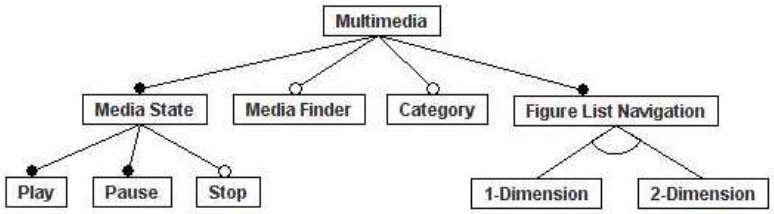


Figure 2.2 Multimedia Feature Diagram.

2.2 SOFTWARE TESTING

Nowadays, software is growing in complexity and size. Typically, software products are developed according to some requirement specification. This way, when delivering some product, developers and designers must ensure that the software is according to these requirements. In addition, the software must be acceptable to the customers and end users. This is where *Software Testing* comes into play. Software testing is a process to not only find bugs in the software, but also a dedicated discipline to evaluate its quality.

The primary task in software testing is defining effective test cases [52]. A test case is a sequence of steps that performs a specific task or a group of tasks. Furthermore, a test case defines expected results. If the artifact under test does not produce the expected results, the test has just found a defect in the artifact. To a tester, it is important to define a suite of tests that, when applied to the artifact under test, exposes a defect.

There are many levels of testing software. We outline three of them in what follows.

- **Unit testing:** tests a minimal component of software. It is used to verify if individual units of source code are working properly. Unit testing facilitates the refactoring process (see Section 2.5): after applying a refactoring, unit tests help on making sure that the code still works as desired. Unit tests are often known as *white-box* tests, since to perform such tests some knowledge about the source code is needed. In the Object-Oriented (OO) approach, such unit of source code to be tested is a method. A popular framework to perform unit testing in Java is JUnit [10];

- **Integration testing:** exercises the combination of individual modules and tests them as a group. Notice that it takes as input modules that have been unit tested. An example of integration testing is described as follows. Sending a picture in a mobile phone message requires using the Multimedia Message Service (MMS). This way, a possible test case consists of attaching the file and sending the message. Therefore, in this scenario, two features are under test: multimedia (to render the picture in the message) and messaging (to send the message with the picture).

- **System testing:** consists of testing the whole system to verify if it is according to its requirements specification. System testing falls in the *black-box* scope, since no

knowledge about the code or design of the system is needed.

In this work, we analyze variabilities found in mobile phone test cases proprietary of Motorola Industrial. The test cases are at the integration level and their variabilities are handled by using *if-else* statements.

Figure 2.3 illustrates commonalities and variabilities when considering the test case approach for SPL. Each circle consists of a test step. Notice that the common steps are used to test any mobile phone. Due to the phone's variabilities, the other steps are executed depending on the phone under test (A or B). This decision is handled by the aforementioned *if-else* statements.

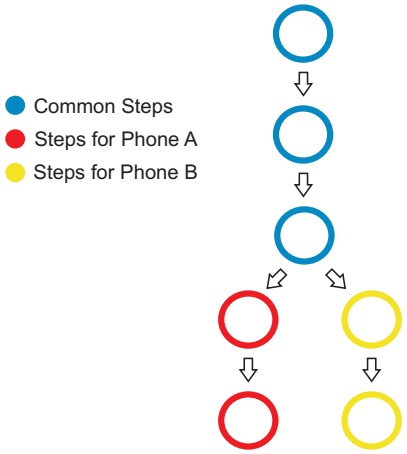


Figure 2.3 Test Steps: Commonalities and Variabilities.

In this work, we aim at separating the common steps from the steps that may vary. The motivation is that there is no modularity in the current approach. For example, Figure 2.4 illustrates two different test cases with variabilities cloned. When separating them, we can have benefits such as reusing the variabilities steps in many test cases and avoiding cloning throughout the test cases. These benefits are worthwhile for modularity.

2.3 SOFTWARE MODULARITY

The concept of modularity applied for software development was first introduced by Parnas [55]. Modularity is closely related to design decisions that decompose and organize the system into a set of modules. The attributes expected in a modular design are outlined as follows:

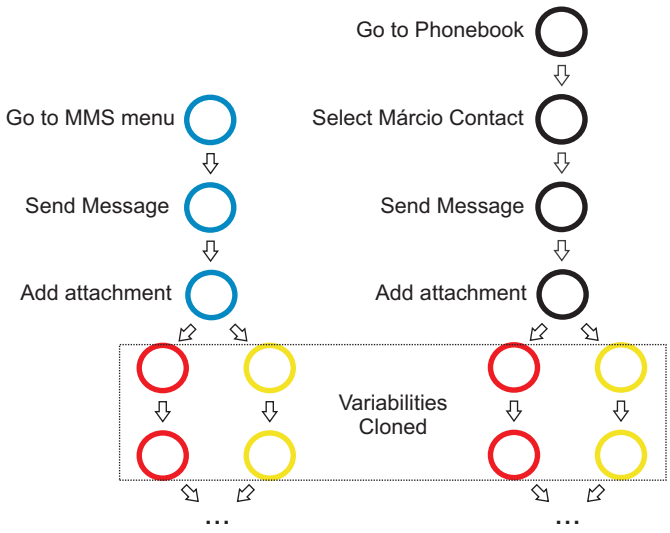


Figure 2.4 Variabilities Cloned in Two Different Test Cases.

- ✓ **Comprehensibility:** a modular design allows developers to understand a module looking only at: (1) the implementation of the module itself; and (2) the interfaces of the other modules referenced by it;
- ✓ **Changeability:** a modular design enables local changes. If changes are necessary in the internal implementation of a module A, the other modules that depend exclusively on A's interface will not need to change, since there is no modification in the module interface;
- ✓ **Parallel Development:** the specification of module interfaces enables the parallel development of modules. Different teams might only focus on their own modules, reducing the time-to-market and the need of communication.

Parnas proposed the information hiding principle as the criteria to be used in decomposition of systems into modules. According to him, the parts of the system that are susceptible to changes must be hidden into modules with stable interfaces. The Object-Oriented (OO) approach enforces such decomposition criteria aiming at hiding the implementation details of classes and exposing only their interfaces.

In this context, another research made by Baldwin and Clark [22] considers modularity as a key factor to innovation and market growth, independent of the industry area.

Their theory uses Design Structure Matrixes (DSMs)[1] to reason about dependencies among artifacts. In addition, they claim that the task structure organization is closely related to them. In this way, if two modules are coupled, their parallel and independent development is impossible, requiring communication between the different teams, or the implementation of both modules by a single team.

In their theory, DSMs are used to visualize dependencies among *design parameters*. These parameters correspond to any decision that needs to be made along the product design. Design parameters may have different abstraction levels. In software industry, some design decisions are related to process development, language, code/architectural style, and so forth. Moreover, if we consider implementation as design activities, software components like classes, interfaces, packages, and aspects should be represented as design parameters.

The notion of dependency arises whenever a design parameter depends on another. Each design parameter appears in both the row headings and the columns headings of the DSM. A dependency between two parameters is marked with a *X*.

Figure 2.5 represents software components as parameters in a DSM. A mark in row *B*, column *A* represents that component *B* depends on component *A*. In the same way, a *X* in row *A*, column *B* represents that component *A* depends on component *B*. Whenever this mutual dependency occurs, we have an example of *cyclical dependency*, which implies that both components can not be independently addressed, meaning that their parallel development is compromised.

	A	B	C
A		x	
B	x		x
C			

Figure 2.5 Example of Dependencies in a DSM.

Additionally, component *B* depends on *C* (expressed by a *X* in row *B*, column *C*) but *C* does not depend on any other component. Therefore, *C* can be independently developed but *B* can not be completely developed until *C* has been concluded. Aiming at removing these dependencies, Baldwin and Clark [22] propose defining Design Rules.

[1] *Dependency Structure Matrix* is another term used.

Design Rules are parameters used as interfaces between modules that are less likely to be changed [51]. In this way, they can promote decoupling of design parameters, like interfaces decrease the coupling between software components. Such design rules establish strict partitions of knowledge and effort at the outset of a design process. They are not just guidelines or recommendations: they must be rigorously obeyed in all phases of design and production [22]. Figure 2.6 illustrates components *A* and *B* being decoupled by using design rules. Since *A* and *B* do not depend each other anymore, it is possible for example to change *A* for *A'* as long as *A'* respects the design rule. However, notice that the coupling does not disappear. Its place has been changed instead: *A* is coupled to the design rule. The same happens to *B*.

	DRs	A	B	C
DRs				
A	x			
B	x			x
C				

Figure 2.6 Design Rules decoupling components *A* and *B*.

In our previous works [54, 31], we observed that design rules may reduce a new kind of dependency between classes and aspects named *Semantic Dependency*. We also shown that this dependency may have impact on the modularity of Aspect-Oriented software. Since this work uses Aspect-Oriented Programming (AOP) to modularize SPL features, we discuss such dependency in detail in Section 3.4.2.

2.4 SOFTWARE METRICS

The usefulness of design and implementation practices can be evaluated through empirical studies. Software metrics are often used in empirical studies as indicators of the strengths and weaknesses of the studied approach [59]. Since this work aims at comparing different mechanisms to implement SPL variabilities, software metrics become an important tool to guide our decision model. In this context, they should point out the strengths and weaknesses of each mechanism, being useful to indicate the appropriate mechanism to implement a given variability. The metrics used in this work are the same of [59]. They are divided in Separation of Concerns (*SoC*), *size*, and *coupling*.

For *SoC*, we used the *Concern Diffusion over Components* (CDC) metric to measure

concern diffusion. It counts the number of components which contributes to the implementation of a concern. We also measure the number of lines of code whose main purpose is to implement a concern, which is represented by the *Concern Diffusion over Lines of Code* (CDLOC) metric. Moreover, we used the *Number of Concerns per Component* (NCC) metric to count, for each component, the number of concerns it implements.

For *size* metrics, we considered *Lines of Code* (LOC) and *Vocabulary Size (VS)*, which count the number of source code lines (ignoring comments and blank lines) and the number of system components (classes, aspects, interfaces, and configuration files), respectively.

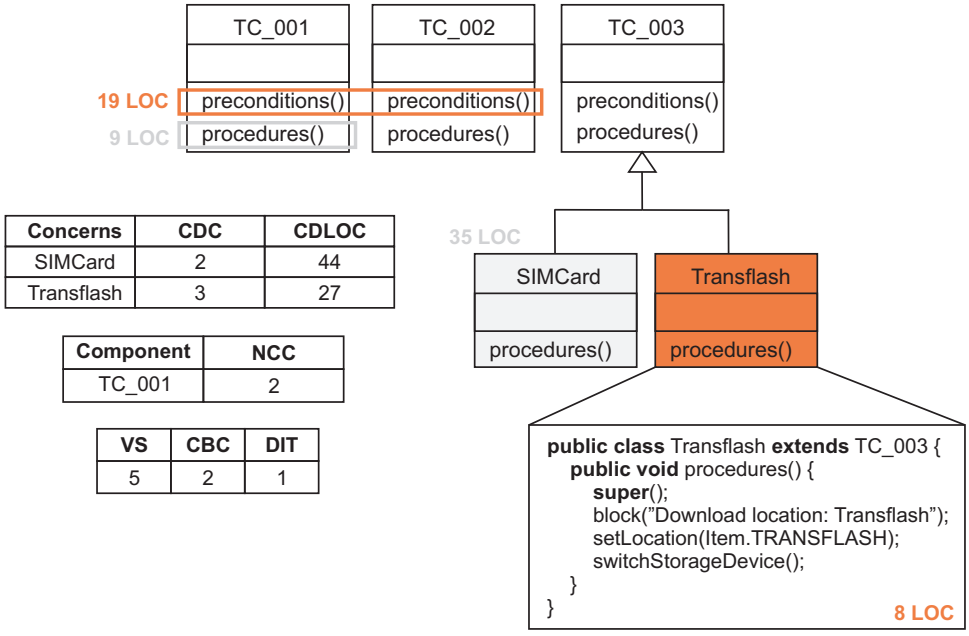
To measure *coupling*, we considered the *Coupling between Components* (CBC) metric, which counts, for each component, the number of other components to which it is coupled. This metric also considers coupling dimensions in AOP (discussed in Section 3.4): relationships between aspects and classes are considered by the metric. We also used the *Depth of Inheritance Tree* (DIT) metric to measure coupling. This metric counts how far down the inheritance hierarchy a class or aspect is declared.

It is important to note that we have chosen these metrics since the majority address modularity. This way, when analyzing them we may conclude which mechanism provides better modularity.

In what follows, we explore all of these metrics by a simple example. Figure 2.7 illustrates three classes which represent test cases (*TC_001*, *TC_002*, and *TC_003*). Two classes inherit from *TC_003*: *SIMCard* and *Transflash*. In this context, they represent variable mobile phone storage devices (in this particular example, this variability is implemented by using the Inheritance mechanism). Notice that these concerns are also scattered throughout the test cases: *SIMCard* is in *TC_001* whereas *Transflash* is in *TC_001* and *TC_002*.

Because *SIMCard* is scattered in two components, $CDC_{SIMCard} = 2$. Analogously, for *Transflash*, $CDC_{Transflash} = 3$, since three components contribute to the implementation of this concern. In order to implement the *SIMCard* and *Transflash* concerns, 44 and 27 *LOC* are necessary, respectively ($35 + 9 = 44$ and $19 + 8 = 27$). This way, the diffusion over lines of code of them are $CDLOC_{SIMCard} = 44; CDLOC_{Transflash} = 27$.

There is only one component which implements more than one concern: *TC_001*. For this reason, $NCC_{TC\_001} = 2$. Because the example uses an inheritance, we calculated how



Concerns	CDC	CDLOC
SIMCard	2	44
Transflash	3	27

Component	NCC
TC_001	2

VS	CBC	DIT
5	2	1

Figure 2.7 Metrics Example.

deep it is ($DIT = 1$). The $CBC = 2$ value shown represents the sum of coupling of all components (*SIMCard* and *Transflash* are coupled to *TC_003*). It is important to note that the CBC metric will be calculated in this way throughout the work. Finally, since this whole example provides five classes, $VS = 5$.

2.5 SOFTWARE REFACTORING

Last but not least, we discuss Software Refactoring. This topic is important for our work because after recommending a mechanism to restructure a given product line variability, refactorings must be done to realize the restructuring.

In the refactoring context, Fowler and Beck [35] have proposed bad smells as a way to diagnose problems in existing code. These bad smells aim at suggesting symptoms that may be indicative of something wrong in the code. In order to remove them, such work claims that Software Refactoring is essential.

Refactoring is a process of changing a software system aiming at improving its internal design but maintaining its external behavior intact. It is a disciplined way for improving the source code and minimizing the chance of introducing errors into the system. In his classic Refactoring book [35], Fowler provides a catalog of refactorings for OO systems.

Each refactoring has a structure which seems to the design patterns catalog defined by [37]. The structure of each refactoring consists of:

- **Name:** the refactoring name. Although it seems unimportant, the name expresses the essence of the refactoring, being useful for a known vocabulary used by the developers;
- **Abstract:** represents the situation that the refactoring might be useful;
- **Motivation:** describes why the refactoring should be made;
- **Mechanics:** a detailed description on how to execute the refactoring;
- **Examples:** illustrates some examples of using the refactoring.

Figure 2.8 depicts an example of refactoring. On the top, it shows the code before applying the refactoring. Analyzing such code, we can see that it is not flexible at all: it is hard to reuse the code of the *printTotal* method. If one wonder to output the results by using HTML instead of plain text, the only recourse is to write a whole new method that duplicates much of the behavior of the *printTotal* method. On the other hand, after applying some refactorings such as *Extract Method* and *Move Method*, the code becomes flexible. Now, it is possible to add new types of outputs without changing the base code and not duplicating code.

According to the Fowler's book, the catalog is not definitive. It is just the beginning, since emerging technologies might introduce new concepts which consequently might require new refactorings. An example for that is the research towards a catalog of Aspect-Oriented refactorings [53, 49]. Such works provide a collection of refactorings that includes the extraction of aspects from OO legacy code and the subsequent tidying up of the resulting aspects.

[53] analyzed the bad smells proposed by Fowler and Beck and reported that some of them might be used by AOP programmers as symptoms of the presence of crosscutting concerns. This particularly applies to the following refactorings: *Divergent Change* (one class that suffers many kinds of changes, i.e. a symptom of code tangling) and *Shortgun Surgery* (one change that alters many classes, i.e. a symptom of code scattering). In addition, they propose new bad smells for crosscutting concerns detection.

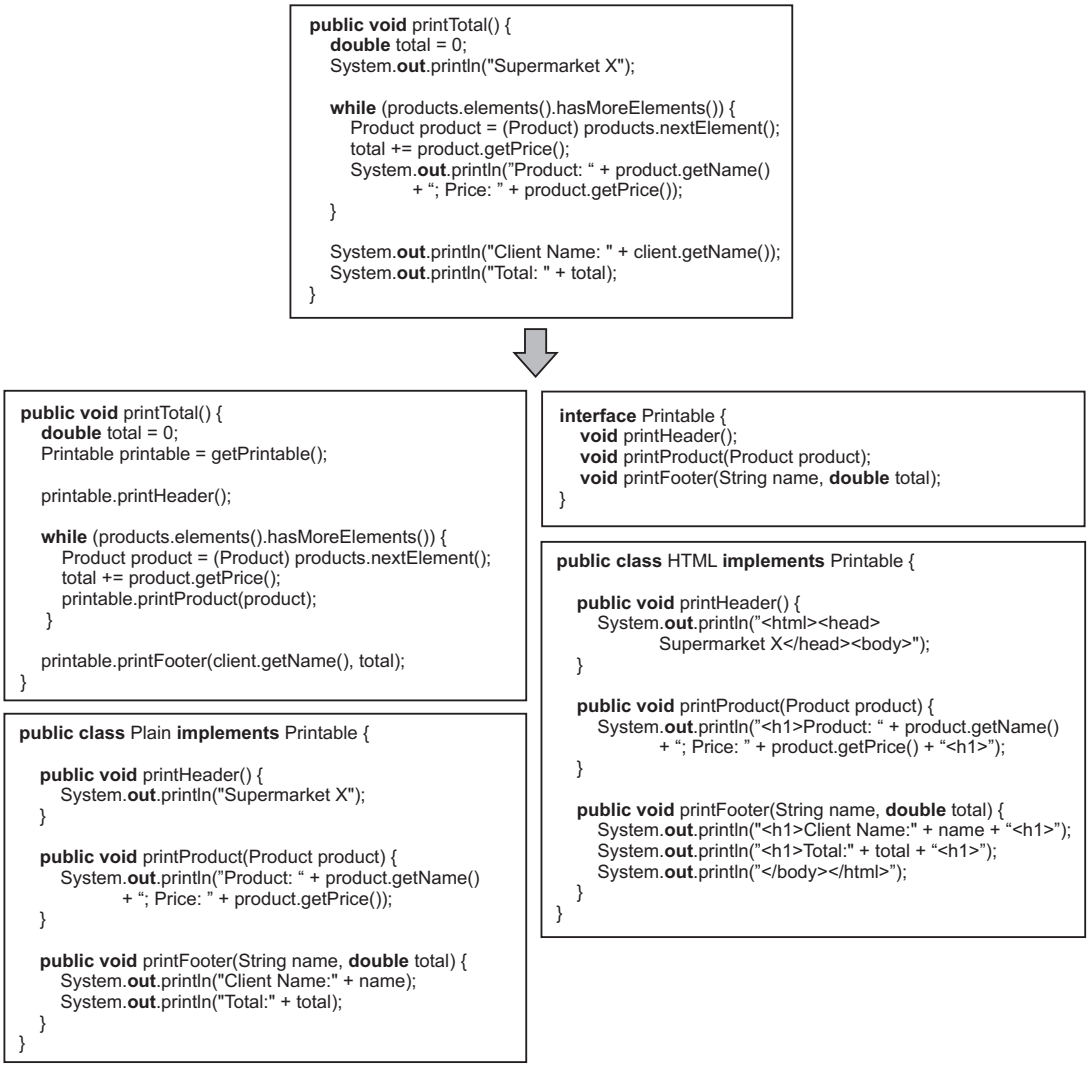


Figure 2.8 Refactoring Example.

It is important to note that the precondition for refactoring is having solid tests. Although tests mean a lot of extra code to write, they guarantee that the refactorings made do not change the expected behavior of the system. This notion of testing is an important part of eXtreme Programming (XP) [23], since extreme programmers are very dedicated testers.

CHAPTER 3

VARIABILITY IMPLEMENTATION MECHANISMS

In this chapter, we review some mechanisms used to implement SPL variabilities. The chapter is organized as follows. We first consider the Inheritance mechanism in Section 3.1. Then Section 3.2 presents Configuration Files, a mechanism that provide runtime binding. The Conditional Compilation approach is detailed in Section 3.3. Next, Aspect-Oriented Programming and Tracematches are presented in Section 3.4 and 3.5, respectively. Next, in Section 3.6 we discuss how Mixins can address SPL variabilities. Finally, the Component Technology is presented in Section 3.7.

3.1 INHERITANCE

Inheritance is a relationship to represent classes hierarchically. It can be used to assign base functionality for superclasses (ancestors) and variabilities for subclasses (children). In this context, the base class (superclass) may be reusable. Because a class may have multiple subclasses, it is possible to define multiple variations of the same base functionality. Moreover, subclasses can override the reusable functionality when defining new methods with the same signature of the superclass' ones.

One of the advantages of inheritance is the support to unanticipated variation: superclasses are unaware of their subclasses. In addition, inheritance requires a very minimal infrastructure of code. On the other hand, inheritance may have a high coupling: changing the superclass may require changing all subclasses. Furthermore, the growth amount of different variabilities carries along automatically growth of the subclasses which, in many cases, leads to a complex and unclear inheritance tree.

Inheritance is limited to static variation. In order to achieve dynamic variation, techniques such as late binding can be used. For example, the Decorator Design Pattern [37] is a solution for dynamic refinements in OO languages by using inheritance. Nevertheless, this solution invalidates some of the advantages of inheritance: it requires a lot of

infrastructural code (see Figure 3.1) and is much less efficient [57].

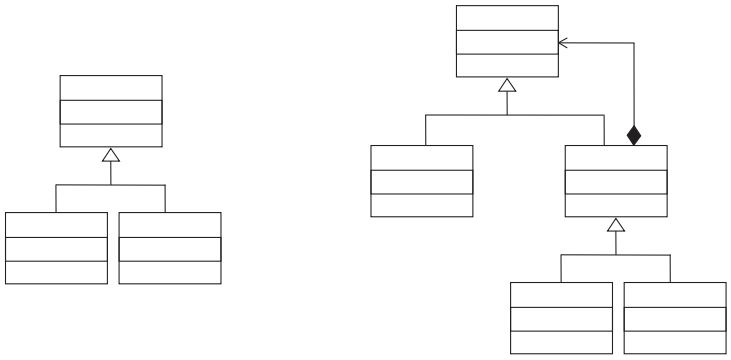


Figure 3.1 Infrastructural code: Inheritance *versus* Decorator Design Pattern.

Inheritance and late binding are also fundamental in OO frameworks. A framework is a set of cooperating classes that make up a reusable design for a specific class of software [40]. One important characteristic of frameworks is the inversion of control. Instead of calling the framework, applications are called by it. Thus, the application specific behavior is triggered at predetermined extension points (also known as hot spots) of the framework. Notice that some of these extension points may be provided by the inheritance and late binding mechanisms. Section 4.1 presents the Motorola framework where the tests analyzed in this work are implemented: *Test Automation Framework* (TAF). It deals with product line variabilities by using basically inheritance, late binding, and *if-else* statements.

3.2 CONFIGURATION FILES

Configuration Files are useful to separate the source code from values that may change the behavior from one version of the application to another [14]. Each value of the configuration file is mapped to a parameter. When executing, the application loads the configuration files and searches for the values of the parameters encountered in the source code.

Figure 3.2 depicts the mapping between parameters and values. Two configuration files and a tester are shown as well. Now, suppose that the tester would like to test the *Motorola Browser* in the mobile phone. In this case, the *motorola.cfg* file should be selected to successfully test the browser. Selecting the incorrect file (*opera.cfg*) may

break the test case. For example, the test case might break when trying to save pages in the cache of the *Motorola Browser*, since it does not provide this feature (*CACHE = false*).

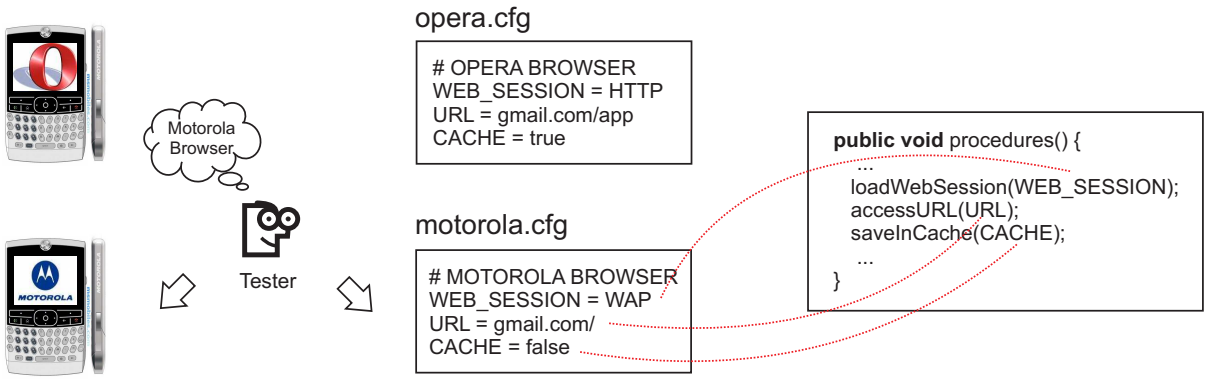


Figure 3.2 Configuration Files Example.

In particular, it is desirable that after delivering the application to the customers, no extra development effort needs to be done when configuring the system, since customers may not have knowledge about the source code of the software. Configuration Files are important in this context because they enable customers to configure some aspects of the application according to their needs without changing any line of code. Notice that some applications provide Graphical User Interfaces (GUI) to help on configuring them, being easily and useful for final users.

3.3 CONDITIONAL COMPILATION

Conditional Compilation is a well-known and widely used mechanism for handling variabilities in languages such as C and C++. This mechanism is used at pre-compile time. Basically, the preprocessor analyzes the code that should be compiled or not based on variables. In this way, if the variable is true, the code must be compiled. Otherwise, the preprocessor will ignore such code snippet and it is not compiled. This mechanism uses a special comment symbol (//#), indicating that a code line will be preprocessed.

Figure 3.3 illustrates an example of this process extracted from a J2ME Mobile Games Product Line[1]. The block of code that actually plays the desired sound varies depending

[1]Developed by Meantime Mobile Creations/CESAR.

on the sound API provided by the phone. This variability exists because phone manufacturers provide different proprietary APIs to deal with sounds. The alternative sound APIs are detached in Figure 3.3. Notice that the code snippet to be compiled depends on variables used in *if directives*. For example, when selecting the *device_sound_api_nokia* variable, the preprocessor compiles the *Code_Block_A* and ignores both *Code_Block_B* and *Code_Block_C*.

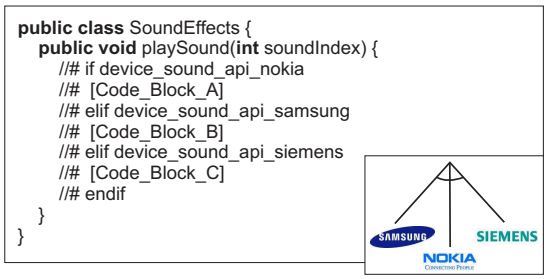


Figure 3.3 Conditional Compilation Example.

The advantage of this mechanism is that it can address fine-grained variabilities. On the other hand, it does not address modularity, since the different variants are not separated from the base source code.

3.4 ASPECT-ORIENTED PROGRAMMING

Aspect-Oriented Programming (AOP) is a well-known mechanism that addresses the separation of concerns which are scattered and tangled throughout other modules [46]. The separation of such concerns (known as crosscutting concerns) avoids the scattering and tangling, promoting a more modular system. Logging, distribution, concurrency, tracing, security, and transactional management are accepted as examples of crosscutting concerns well addressed by AOP. Further, previous works [17, 15, 56, 33] showed that AOP is useful to implement optional and alternative features in a product line approach.

Such crosscutting concerns are encapsulated in units called aspects. The process of composing aspects and classes is known as weaver and often is supported by tools such as AJDT [3]. The AOP language used in this work is AspectJ [45].

3.4.1 AspectJ

AspectJ is an aspect-oriented extension to Java. AspectJ consists of two parts: the language specification and the language implementation [48]. The language specification part defines the language in which the code is written; the language implementation part provides tools for compiling, debugging, and integrating with popular integrated development environments (IDEs) such as Eclipse [9].

The exact location where aspects may act in AspectJ depends on defining *joinpoints*, *pointcuts*, and *advices*. *Joinpoints* are a set of points during the program execution flow where one may want to execute a piece of code. In order to group a set of *joinpoints*, a *pointcut* can be used. *Advices* define the code to be executed in determined *pointcuts*.

We illustrate such concepts by an example. When testing a mobile phone feature in TAF (the framework where the tests analyzed in this work are implemented, see Section 4.1 for details), sometimes it is necessary to enable this feature. This task is performed by turning on the flex bit of the feature. However, to maintain the initial state of the phone, when the test is finished, the flex bits must be reset. This operation (represented by calling the *resetFlexBits* method) is scattered in the test cases. In order to avoid the scattering of the *resetFlexBits* calls throughout the test cases, the aspect *ResetFlexBits* was created (Listing 3.1).

The *execution* joinpoint (line 4) is a point where aspects may execute: during the execution of methods. The *reset* pointcut groups all executions of methods named *postconditions* present in any class (see at line 4 the asterisk of the **.postconditions* expression). The piece of code that should be accomplished after the execution of every *postconditions* method is defined in the *after* advice (lines $6-8$). After applying this aspect, the *resetFlexBits* calls do not need to be scattered throughout the test cases anymore.

Listing 3.1 Aspect Example.

```
1 public aspect ResetFlexBits {
2
3     pointcut reset(TestCase testCase):
4             execution(void *.postconditions()) && this(testCase);
5
6     after(TestCase testCase): reset(testCase) {
7         testCase.resetFlexBits();
8     }
9 }
```

The AspectJ language also supports *around* advices. An around advice surrounds a joinpoint and has control whether the joinpoint's execution should proceed or not. A method with an empty body can be advised by an around advice and some piece of code can be executed inside that method. In addition, the language defines intertype declarations. By using this functionality, attributes and methods can be introduced into a class. For example, the following intertype

public static final *String TC_065.BROWSER_WEB_SESSION;*

introduces the *BROWSER_WEB_SESSION* attribute into the *TC_065* class.

Although the crosscutting concern to reset flex bits of Listings 3.1 is now modularized, we observed in our previous work [31] that aspects (aimed to support *crosscutting modularity*) might actually break *class modularity*. In the presence of aspects, class modularity is compromised because, when evolving a class, it may be necessary to analyze the implementation of existing aspects, instead of analyzing only the class and the interface of other referred classes. It happens because of *Semantic Dependencies* between classes and aspects. Since such dependencies may have impact on modularity analysis of AO systems, we explore them in the next section.

3.4.2 Semantic Dependencies in Aspect-Oriented Software

After discussing some essential concepts of AOP, we present a kind of dependency that can compromise modularity in AO software. In our previous works [54, 31], we have presented an approach of modularity analysis that considers both syntactic and semantic dependencies between classes and aspects. In what follows, we discuss such dependencies using a detailed example.

Syntactic dependency in OO software components (classes and interfaces) occurs when there is a direct reference between them, such as inheritance, composition, methods signatures (parameters, return types, exceptions throwing), class instantiations, and so forth. This dependency causes compile errors whenever a component is modified or removed from the system, being thus easily detected. In the same way of classes and interfaces, direct references can appear between aspects and other components, which means that aspects can also have syntactic dependency.

However, there is another kind of dependency that is not so easy to realize because it occurs without explicit references between system components (classes, interfaces, and aspects - in AO systems). We call this kind of coupling as *Semantic* dependency. Besides, this kind of dependency does not cause compilation errors when removing or modifying components.

As an example of semantic dependency in AO systems, suppose the requirement of synchronizing all methods of a class (concurrency management). Such requirement consists of encompassing with synchronized blocks the bodies of all methods. Consider that the aspect developer is responsible for implementing this concern in an aspect and that a class developer, which is oblivious about this aspect, decides to implement methods in the same class. In this situation, at least three problems can occur:

1. The methods created by the class developer might not need concurrency management, but they will be synchronized by the aspect;

2. If the class developer, oblivious about the aspect, implements concurrency management on methods class, these methods would be synchronized twice; and

3. Depending on how the synchronization approaches are implemented by the aspect and class developers, together they might lead the system to a deadlock or a livelock situation [50].

In such cases, the expected behavior of the system could be compromised, since some additional synchronization would be created or, even worst, the system might reach a deadlock or a livelock.

The situation above exposes that modularity problems have occurred: (1) the comprehensibility is compromised, since two modules should be studied in order to understand the concern; and (2) the parallel development is problematic, because one developer can implement unintended behavior into a module which, although it is not under his responsibility, might break the system.

These problems are caused by semantic dependencies, because the class depends on the aspect to work correctly. They occur since the class developer is oblivious about the aspect implementation (there is no syntactic definition that the concurrency concern is woven in the class by the aspect). Such problem can also appear when, for example,

the aspect developer changes the concurrency implementation. In this case, the aspect developer might break the expected behavior of the class.

To avoid the aforementioned problems, we proposed to use design rules in order to reduce not only syntactic but also semantic dependencies between aspects and classes. Using design rules requires both aspects and classes developers agreement. In this way, they promote the decoupling between such components, promoting modularity.

Figure 3.4 illustrates the development using design rules between classes and aspects. The design rule makes explicit that every method in the *EmployeeRepository* class will be synchronized by the *Concurrency* aspect and that the *EmployeeRepository* class can not implement any synchronization mechanism.

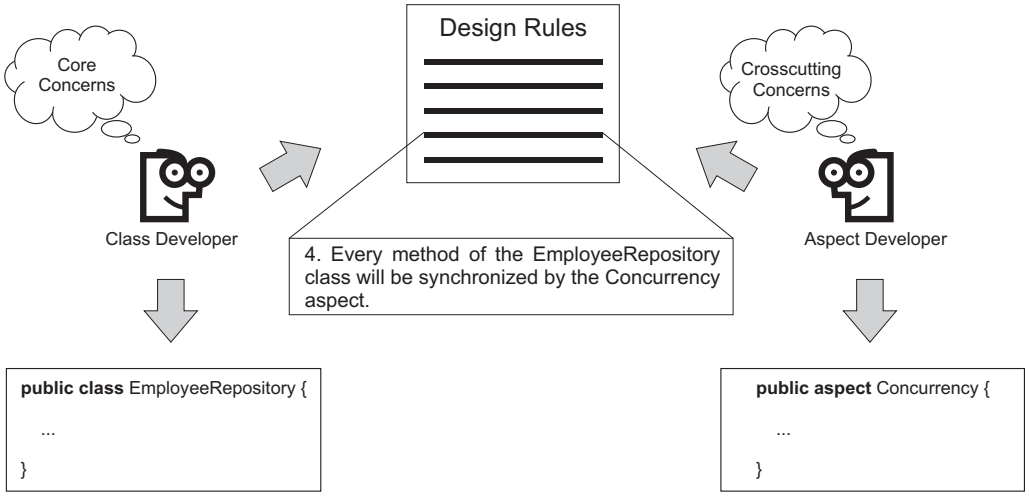


Figure 3.4 Aspect and Class developers using Design Rules.

Hence, the class developer is not oblivious anymore when considering the implementation of the concurrency concern. The design rule will constrain the developer while creating methods in the *EmployeeRepository* class, eliminating the semantic dependencies between the aspect and the class. If the methods must be synchronized, the developer just create them and leave the concurrency responsibility for the aspect. Otherwise, he must communicate to the aspect developer that some methods will not need concurrency management.

This way, we consider throughout the work that when recommending the AOP mechanism, design rules must be present to guarantee the modularity.

3.5 TRACEMATCHES

An aspect observes the execution of a base program; when certain actions occur, the aspect runs some extra code of its own. In the AspectJ language, the observations that an aspect can make are confined to the current action: it is not possible to directly observe the history of a computation [13].

Tracematches instead allow one to write an advice that is triggered by a pattern that ranges over the whole history of the computation. For example, suppose that one wants to execute some piece of code after calling the method f followed by g without any method call between them. Listing 3.2 shows how to do this using tracematches. The reserved word *sym* defines symbols that may be included in the alphabet of the tracematch. In the example, notice that all method calls should be included (line 7). Lines 3 and 5 define specific symbols for calls to f and g, respectively. The regular expression on line 9 specifies that the advice is triggered on traces that end with a call to f and g in this order. Since all method calls are in the alphabet of the tracematch, a trace like *f, a, b, g* would not match *f, g* (calls to a and b are not ignored). On the other hand, removing line 7 causes a reduction to the alphabet to only two symbols (f and g). This way, a trace like *f, a, b, g* will match *f, g* because calls to a and b are now ignored.

Listing 3.2 Tracematch Example.

```
1  tracematch() {
2
3     sym f after : call(* *.f(..));
4
5     sym g after : call(* *.g(..));
6
7     sym anyMethodCall after : call(* *.*(..));
8
9     f g {
10        System.out.println("fg");
11    }
12 }
```

The main shortcoming of this mechanism is the high coupling between the tracematches and classes. In this case, changing minimal statements in the class might break the tracematch. For example, a class developer oblivious to the tracematch of Listing 3.2 may put a new method call between f and g because of maintenance reasons. As we explained earlier, in this situation the tracematch stops working.

As discussed in Section 3.4.2, design rules may be used to promote modularity in AO software. Likewise, we argue that such design rules must also be present in tracematches to avoid the aforementioned problem.

The tracematches implementation presented here is an extension of the abc compiler [21].

3.6 MIXINS

A mixin is an abstract subclass; i.e. a subclass definition that may be applied to different superclasses to create a related family of modified classes [24]. In other words, a mixin is an abstract subclass that may be used to specialize the behavior of a variety of parent classes. Mixins can also be used to express a simple form of compositional variability: different variations can be produced by combining different lists of mixins [57].

We illustrate the mixins mechanism in the following simple text editor example. In this example, many objects like texts and figures can be placed into the editor. *Text* and *Figure* are classes that extend from *EditorObject*. In this editor, it is desirable to compose texts and figures, which means that figures may contain texts. Listing 3.3 illustrates the *EditorObject*, *Text*, and *Figure* classes written using the CaesarJ [1] language (the mixins approach used in this work relies on this language). The *EditorObject* class stores the object position and defines the *draw* method, which should be overwritten for drawing specific objects.

Listing 3.3 Mixins Example.

```
1  cclass EditorObject {
2      protected int positionX;
3      protected int positionY;
4      public void draw(Graphics graphics) {
5          ...
6      }
7  }
8
9  cclass Text extends EditorObject {
10     protected String text;
11     public void draw(Graphics graphics) {
12         super.draw(graphics);
13         graphics.drawText(positionX, positionY, text);
14     }
15 }
16
```

```
17 cclass Figure extends EditorObject {
18     protected String path;
19     public void draw(Graphics graphics) {
20         super.draw(graphics);
21         graphics.drawFigure(positionX, positionY, path);
22     }
23 }
24
25 cclass TextFigure extends Text & Figure {}
```

As mentioned, composing texts and figures may be possible. Thus, different *EditorObject* subclasses such as *Figure*, *Table*, or *Rectangle*, should be decorated with the *Text* functionality. In other words, the *Text* class should add text drawing to any *EditorObject* subclass. This way, we conclude that the *Text* class is a mixin, which can be composed with *EditorObject* subclasses.

To compose texts and figures, a new mixin is defined (line 25). The functionality obtained is similar to the following inheritance hierarchy: *TextFigure* extends *Text*; *Text* extends *Figure*; and *Figure* extends *EditorObject*. Calling the *draw* method from a *TextFigure* object will produce a chain of supercalls that will draw a text within a figure at the end.

When comparing to mixins, the main disadvantage of plain classes is that they have fixed parents and can not be reorganized to different inheritance hierarchies [6]. In contrast, mixins are more flexible due to their capability of composing classes. Nevertheless, when considering more objects, the complexity of the compositions increases and sometimes can get confused.

In the SPL context, mixins may be useful to implement optional features. Mixins can be used to compose them in many ways and produce a set of different products.

3.7 COMPONENT TECHNOLOGY

By component we mean any reusable piece of software with a well-defined interface. Nevertheless, there is no consensus about the definition of a component, since other definitions consider that a component must be independently deployable [57], for instance. In this work we do not cover such definitions.

In the context of component technology, we only discuss *Service-Based Composition*. When considering this approach, developers do not need to explicit dependencies between

components and services. The binding of the components to the services is determined automatically and at runtime. On the other hand, there is no static validation, which guarantees that all components will be connected to the services adequately [57]. Technically, such binding may be achieved either by Services Registers or by Dependency Injection [36]. In this particular work, we use dependency injection as a mechanism to restructure test variabilities in SPL.

Using dependency injection, a set of services may handle software variabilities as long as they implement a defined interface. This way, such approach is useful as a mechanism to implement SPL variabilities at runtime and without explicit dependencies. Dependency injection requires much less infrastructural code when comparing to an approach which uses inheritance in combination with the Strategy/Decorator and Factory design patterns [37]. However, dependency injection depends on a container responsible for finding and passing the necessary services to the component.

Listing 3.4 shows how to use dependency injection in the Spring container [4].

Listing 3.4 Dependency Injection Example using the Spring Container.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
4  "http://www.springframework.org/dtd/spring-beans.dtd">
5
6  <beans>
7      <bean id="MovieLister" class="br.cin.ufpe.moviesystem.MovieLister">
8          <property name="finder">
9              <ref local="MovieFinder"/>
10         </property>
11     </bean>
12
13     <bean id="MovieFinder" class="br.cin.ufpe.moviesystem.finder.DatabaseMovieFinder"/>
14 </beans>
```

In order to list the available movies, the *MovieLister* component (line 7) depends on a *MovieFinder* (lines $8-10$). The container will create and pass to the *MovieLister* component an instance of the *MovieFinder* configured. In this particular example, the *DatabaseMovieFinder* will be instantiated (line 13). The *finder* property is an attribute of the *MovieLister* class. This class must provide the *setFinder* method so that the container can set up the *MovieFinder* previously instantiated.

Since there is no explicit dependencies, all we have to do to take another product such

as *FileMovieFinder* into consideration is changing the configuration file. Notice that this task can be done even at runtime.

CHAPTER 4

DECISION MODEL FOR RESTRUCTURING TESTS IN SOFTWARE PRODUCT LINES

Chapter 3 presented some existing variability implementation mechanisms. In this chapter we use them to construct a decision model for recommending those mechanisms to restructure test variabilities. Our decision model is code-centric (because the variability is identified by its exact location at the source code), being important for (i) allowing developers to decide which mechanism to use more easily when dealing with these variabilities at the source code; and for (ii) refactoring reasons. Besides, the model is based on both qualitative (using DSMs and discussing advantages and disadvantages of each mechanism) and quantitative (using metrics) studies.

Before presenting the decision model, we first present the framework where the tests analyzed in this work are implemented in Section 4.1 named TAF. Then we discuss in Section 4.2 how we have found the TAF test variabilities by using cloned code detection tool. The definition of the decision model is presented in Section 4.3. Initially, we depict some test variabilities. For each analyzed variability, we implement it using different mechanisms such as inheritance, mixins, configuration files, and so forth. Afterwards, we calculate the metrics discussed in Section 2.4 for each mechanism to indicate which one fits better to the given variability, being a guide to define the decision model. Last but not least, according to the obtained results, we present our decision model in Section 4.4.

4.1 TEST AUTOMATION FRAMEWORK (TAF)

When testing mobile phone software, test engineers may execute test cases manually. Since a test case may be repeated several times during software development, executing them manually is both time consuming and error-prone. In order to minimize these problems, automated test cases may be used. An automated test case can automatically reproduce the steps that would be performed manually by the test engineer [43, 44].

Test Automation Framework (TAF) is an OO framework developed by Motorola Industrial that supports the creation of automated test cases for mobile phone software. It was designed to allow reuse of automated test cases across distinct phone models of a given product family. Once a test case is automated, it is ready to be executed as many times as needed, through the phone development life cycles, reducing the time to market [43, 44].

To test the mobile phone software, TAF simulates key pressings at the phone keyboard. Moreover, TAF checks the expected results after such key pressing. In this context, it may check if some display content is showed correctly, for example. If not, the framework raises an exception and the test case fails.

TAF encapsulates test input actions (such as a sequence of key pressings) and test output analysis (such as checking the phone display contents) into a so-called Utility Function (UF). While test cases tell "what" to test, UFs tell "how" to perform the steps. UFs are entities that isolate a functionality. For example, *launchApp* is an UF that initiates a determined application. Depending on the phone's families, the task of launching an application may be different. To handle such product line variabilities, UFs are implemented hierarchically by using the inheritance mechanism (see Figure 4.1). Since TAF has potentially more than one implementation for each UF, the appropriate UF implementation for the phone family under test must be previously known. This information is encoded within an XML configuration file, which is maintained by TAF developers [43, 44]. Thus, TAF seems to be a Domain-Specific Language (DSL) for tests.

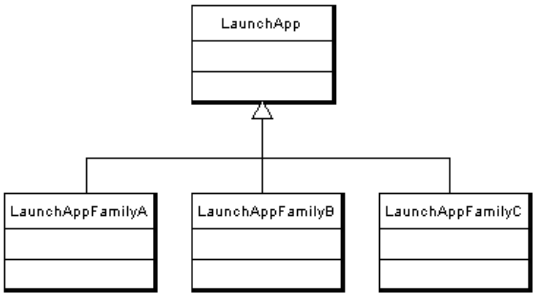


Figure 4.1 LaunchApp UF: variabilities implemented using Inheritance.

On the other hand, some variabilities are not handled by using inheritance (in the UFs layer). Often, phones of the **same family** have different capabilities[1] and function-

[1]In terms of hardware.

alities[2]. For these cases, instead of Inheritance, *if-else* statements are used to address these variabilities. Figure 4.2 illustrates the feature diagrams of phone's capabilities and functionalities used in TAF test cases.

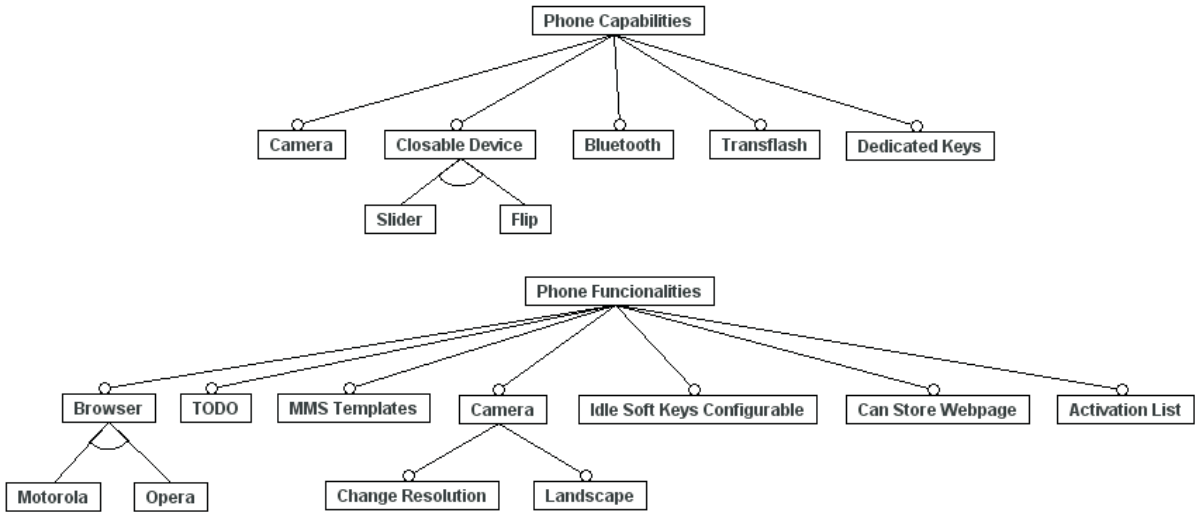


Figure 4.2 TAF Feature Diagrams: Phone Capabilities and Phone Functionalities.

Listing 4.1 shows a snippet test case. Notice that a TAF test case consists of a set of UF calls (i.e., method calls). Line 3 launches the *Browser* application. The *URL* to be accessed depends on the phone's browser (Lines $5-9$). Afterwards, the "*OK*" button is pressed through the *acceptDialog* UF (Line 11).

Listing 4.1 Code snippet of a TAF Test Case.

```
1  public void procedures() {
2
3          navigationTk.launchApp(PhoneApplication.BROWSER);
4
5          if (has(PhoneFunctionality.OPERA_BROWSER)) {
6              editorTk.typeText(BrowserURLContent.URL_13);
7          } else {
8              editorTk.typeText(BrowserURLContent.URL_17);
9          }
10
11         phoneTk.acceptDialog();
12 }
```

By using *if-else* statements, the variabilities are not separated from the core assets: they are tangled and scattered throughout the test cases. Additionally, some of them

[2]In terms of software.

are cloned in many test cases, being time consuming and error-prone changing them. Consequently, the *if-else* mechanism does not provide feature modularity. In Section 4.3, we restructure some of these variabilities through many mechanisms such as inheritance, mixins, configuration files, and so forth. After comparing the mechanisms using some DSMs and the metrics showed in Section 2.4, we present our decision model: given a TAF test case variability, we suggest a mechanism to implement it to achieve modularity. The test cases used in this work are at the integration level.

We finish this section describing an important observation. The mechanisms used to restructure the test variabilities must take care to the order of the statements. Otherwise, the test cases do not work as expected. For example: suppose a test case that sets the alarm clock on a phone. The first step is to access the alarm application, and the second is to set it up. The reverse order is not possible.

4.2 FINDING PRODUCT LINE VARIABILITIES

Section 4.1 presented the framework where the tests are implemented. Due to its complexity and size (the analyzed TAF integration test cases have 414 classes and $56.325 LOC$), the first challenge aiming at restructuring their variabilities is actually to find themselves[3]. A possible way to do that is looking at each class to find variabilities at the source code. However, given the high number of classes, this is a hard and painful task, being time consuming and error-prone.

In order to minimize the costs of this task, we performed just the opposite: instead of searching for variabilities, we searched for possible commonalities. Afterwards, looking at the commonalities, the variabilities were easily found. To perform this task, we used a cloned code detection tool. In what follows, we show an example of variability found by using this technique. The tool used in this task is named CCFinder [2, 38].

Once the process of finding clones is finished, the tool lists the classes used to search clones in the left side of the window. When selecting two or more classes, the tool shows their code at the same time in the *Source Text* edge. Clones are marked as gray blocks as illustrated in Figure 4.3. The figure also detaches two methods of different classes. They seem to be cloned, but actually there is a little bit difference in them.

[3]Due to the high LOC size, we neither found nor restructured all TAF test variabilities.

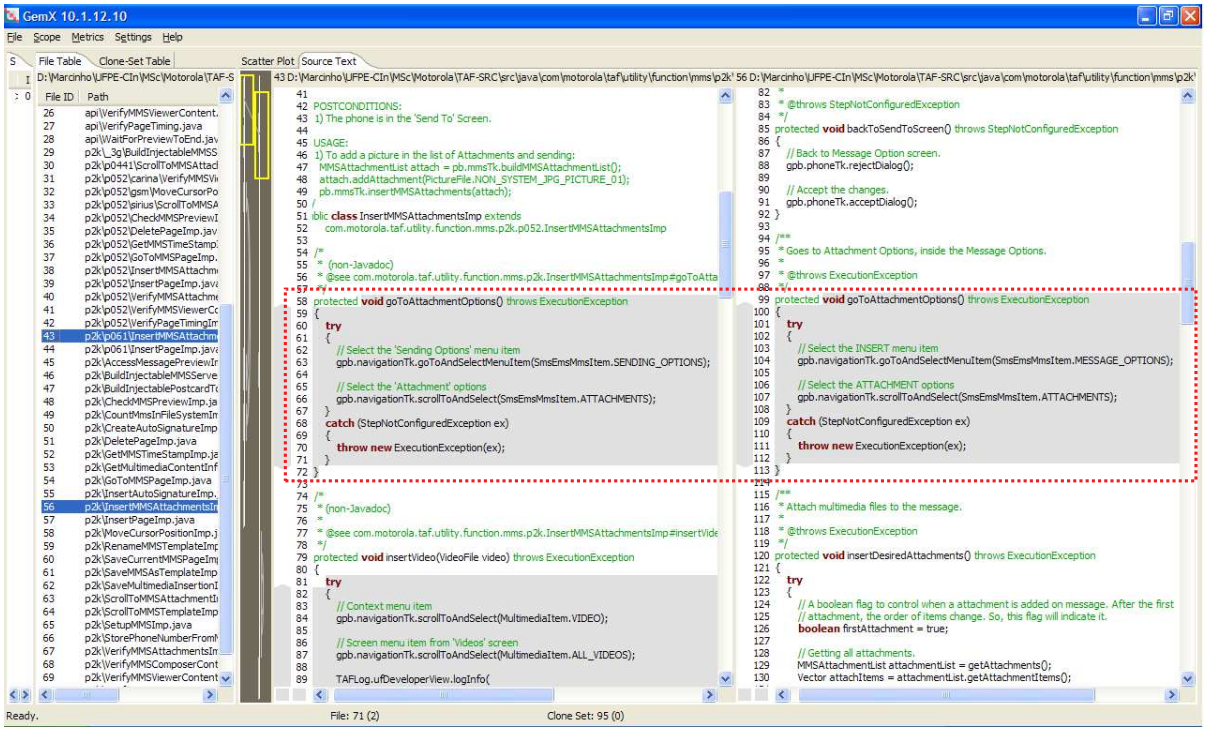


Figure 4.3 GUI of the CCFinder tool.

Figure 4.4 shows the difference between the methods detached in Figure 4.3. The method *goToAttachmentOptions* aims at going to the attachment option of a Multimedia Message Service (MMS). However, depending on the phone, the path to get into such option may be different. As illustrated in Figure 4.4, for *Phone A*, the path must contain the *Sending Options* menu, whereas for the *Phone B* the path must contain the *Message Options* menu. Notice that the different paths are defined as constants in the source code (see the calls to the *goToAndSelectMenuItem* methods).

Finding product line variabilities by using the cloned code technique was very useful in this work. Although we do not provide an empirical study in this context, we believe that this technique may be efficient for finding product line variabilities in existing not well structured SPLs. It is important to note that others [25, 62] suggest that such technique is also useful to identify crosscutting concerns.

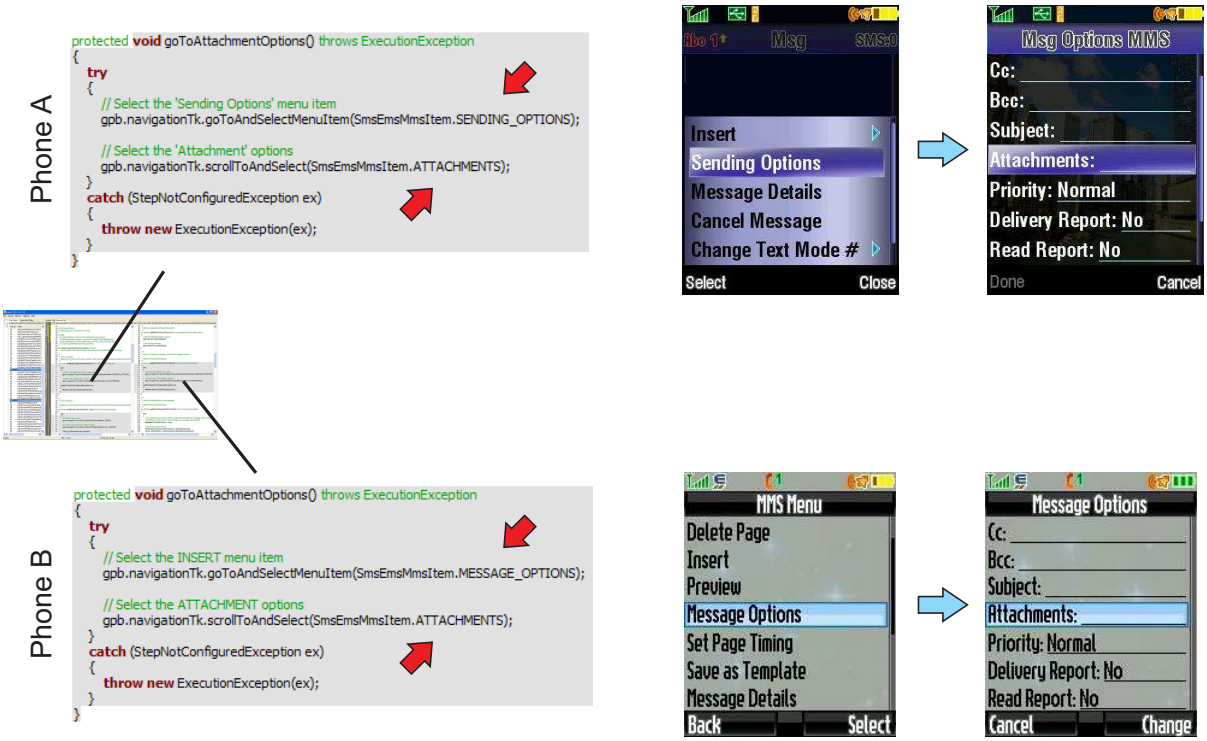


Figure 4.4 Variabilities found using the Cloned Code Detection Technique.

4.3 TOWARDS A DECISION MODEL

As mentioned in Section 2.1, the SPL approach contains, besides common, variable elements. Hence, implementation activities become more complex because they have to realize variabilities as well [56]. In particular, reasoning about how to combine both core assets and product variabilities is a challenging task [16].

Many mechanisms can be used to combine these artifacts. They range from simple ones like Inheritance (see Section 3.1) to more complex ones like Aspect-Oriented Programming (see Section 3.4). Due to the great variety of mechanisms, selecting the correct ones may be a difficult task. On the other hand, selecting the incorrect ones may produce negative effects on the cost to evolve the SPL [33]. For example, cloned code and concerns not modularized may appear, affecting the independent evolution and decreasing productivity when evolving the SPL.

In order to reduce the aforementioned problems at the testing level, we have defined a Decision Model [33, 32] to help developers on the task of choosing mechanisms to restructure test variabilities in SPL. To construct our decision model, we analyzed some

TAF integration test variabilities. In this way, the decision model is based on variabilities handled by using *if-else* statements found in TAF integration test cases. The motivation is that such approach does not provide modularity at all; variabilities are not separated from core assets. Therefore, the model aims at suggesting mechanisms so that variabilities can be modularized.

In what follows, we explain the inputs and the output of the decision model. Since our model is code-centric, the first input is the exact **location** of the variability at the source code. For example, the variability appears at the end of a method body. The second input consists of the **feature type** (variability type). In this work, we have considered the optional and alternative types. The last input is some **criteria** used to compare the mechanisms. The strengths and weaknesses of each mechanism will be analyzed through these criteria. Each criteria is detailed as follows.

- ✓ **Modularity:** the primary goal of every variability mechanism is to improve reusability by enabling separation of reusable assets from their variabilities [57]. In this way, the Modularity criteria represents the separation of assets that a mechanism provides according to the concepts presented in Section 2.3. Possible values: *yes* or *no*;

- ✓ **Source Code Size:** represents the increasing/decreasing of lines of code when applying the mechanism. For example, if the number of lines of code of the mixins approach increased 9% when compared to the original test case code, we write +9%. If it decreased 9%, we write −9%.

- ✓ **Scalability:** tries to discover if the mechanism provides modularity when implementing more features than the original code. For example: the browser alternative features *Opera* and *Motorola* appear in the original code. To analyze if the mechanisms are scalable, we suppose more than two browsers. Possible values: *yes* or *no*;

- ✓ **Time:** represents the binding time provided by the mechanism. Possible values: *compile-time* or *runtime*.

Based on the aforementioned inputs, the decision model should be able to **recommend mechanisms** for restructuring a given test variability. The mechanisms suggested

by the model represent its output. Figure 4.5 outlines an application of our decision model.

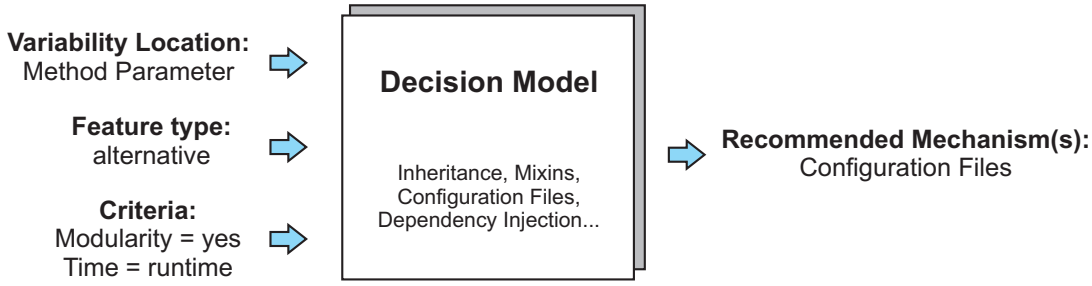


Figure 4.5 An application of the Decision Model.

It is important to note that not always we will be able to decide which mechanism fits best to a given variability and a set of criteria. For example, if the `Modularity`, `Source Code Size`, and `Scalability` criteria are considered and two mechanisms are similar according to those criteria, but the `Time` is not took as input of the model, we recommend both mechanisms and leave the final decision to the user. He may prefer the mechanism that provides runtime binding instead of compile-time binding, for instance. Also, notice that the mechanisms are not mutually exclusive. Each one has strengths and weaknesses, as discussed in Chapter 3.

The methodology to construct the decision model is explained as follows. For each analyzed variability of the TAF test cases, we restructured it using different mechanisms. Differently of existing models [16, 56], our decision model is based not only on qualitative but also on quantitative studies. For this reason, we calculate some software metrics presented in Section 2.4 to compare the implementation of each mechanism and analyze the advantages and disadvantages of the mechanisms as well. Finally, we adapt the decision model to encompass such new variability.

The remainder of this section considers each variability separately, based on its location at the source code.

4.3.1 Beginning/End of Method Body

⇛ **Feature type analyzed:** optional.

The first variability kind analyzed in this work is at the **Beginning/End of Method Body**. Our example of this test variability (illustrated in Figure 4.6) consists of two optional features implemented at the end of the *procedures* method. Hence, four instances of the product line are possible: (i) neither *transflash* nor *bluetooth* are present in the phone; (ii) both *transflash* and *bluetooth* are present in the phone; (iii) phones with only *transflash*; and (iv) phones with only *bluetooth*. As mentioned in Section 4.1, the order of execution of the steps (in this case, features) must be preserved: changing it might break the test case. Therefore, to restructure these features, the mechanisms must take their order into consideration.

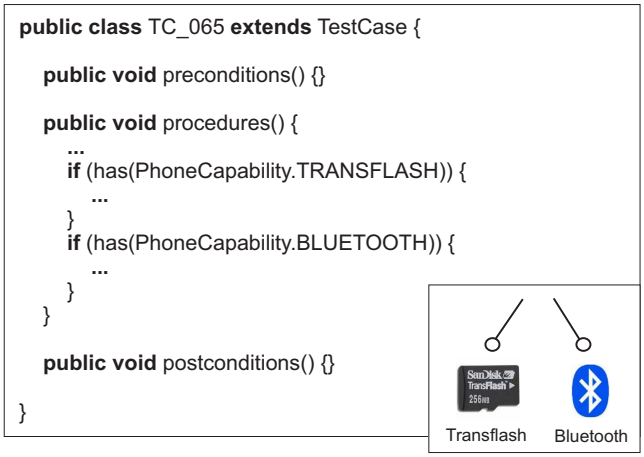


Figure 4.6 End of Method Body.

We have implemented this test variability using four mechanisms. Since this section deals with two optional features, the mechanisms must be able to compose them in case of phones which have both features. This way, beyond **Inheritance**, we have used the **Decorator** design pattern, the **Mixins** approach, and **AOP**. The details of each implementation are presented as follows:

- **Inheritance:** this implementation consists of overriding the *procedures* method. Therefore, two classes inherits from *TC_065*. Each class overrides the aforementioned method and calls the super method followed by the specific feature code (*transflash* or *bluetooth*). Last but not least, another subclass is considered to implement both features;

- **Decorator:** relies on the Decorator design pattern [37]. To compose both features,

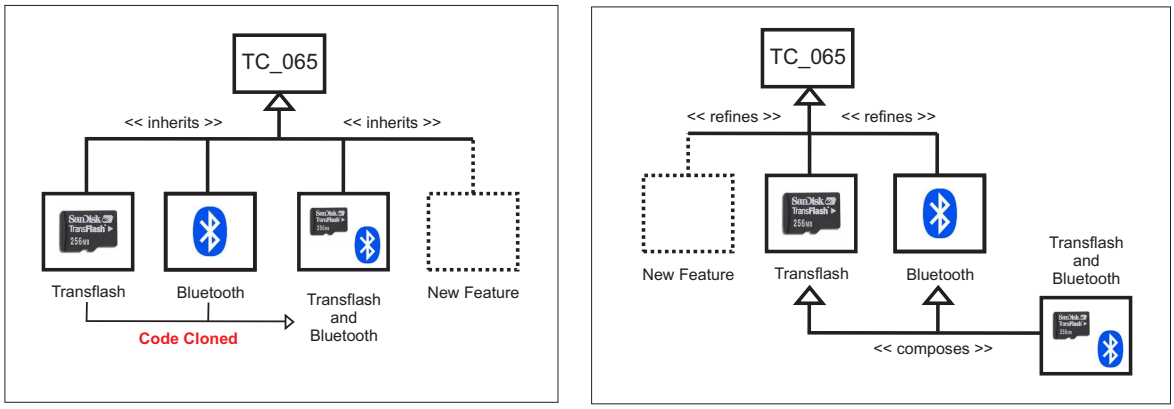
clients must instantiate the composition of the *transflash* and *bluetooth* classes;

- **Mixins:** analogous to Inheritance;
- **AOP:** two aspects (one for each feature) are created to crosscut the *procedures* method using an *after advice*. Further, an extra aspect declares the precedence of the features, being essential to define the features' order.

After detailing each implementation, we analyze them by using both qualitative and quantitative studies. Basically, when we describe some qualitative characteristic, we will try to reflect it in the metrics presented in Section 2.4 and vice-versa. For example, if we described that a mechanism produces cloned code, we will show the increasing in the $CDLOC$ and LOC metrics. We have used the metrics to compare each mechanism implementation.

Table 4.1 shows the metrics for the **Beginning/End of Method Body** mechanisms. For these mechanisms, consider that DIT represents the maximum DIT value found among all components. Figure 4.7(a) illustrates that the **Inheritance** mechanism does not enable feature compositions suitably: for phones with both *Transflash* and *Bluetooth*, it is necessary to create a new class which clones the source code of the two classes responsible for implementing each feature separately. In other words, the mechanism does not address Separation of Concerns (SoC) adequately. As depicted in Table 4.1, the cloned code is reflected in the $CDLOC$ and LOC metrics: the impact on the `Source Code Size` is high (+180% for $CDLOC$ and +48% for LOC). These metrics show how the amount of lines of code is higher when comparing to the **Mixins**, **Decorator**, and **AOP** mechanisms. Also, the **Inheritance** mechanism does not provide `Scalability`. For example, if we consider three features (*Transflash*, *Bluetooth*, and *Infrared*), the amount of cloned code increases and the hierarchy may get complicated due to the growing on the number of classes and compositions. As illustrated in Figure 4.7(b), although **Mixins** solves the cloned code problem, the `Scalability` problem remains when considering at least one more feature.

Although some metrics show that the **Mixins** mechanism is slightly better than the **Decorator** and **AOP** (see $CDLOC$ and LOC metrics), we concluded that this holds only for two features. Again, if we consider one more feature (*Infrared*, for example),



(a) End of Method Body using Inheritance.

(b) End of Method Body using Mixins.

Figure 4.7 End of Method Body: Inheritance x Mixins.

the number of classes and compositions increases significantly when using the **Inheritance** and **Mixins** mechanisms, being hard to maintain the whole application. In this case, instead of defining only one class for composing the features, the developer must implement four: *Transflash* & *Bluetooth*, *Transflash* & *Infrared*, *Bluetooth* & *Infrared*, and *Transflash* & *Bluetooth* & *Infrared* (see Figure 4.8). Because of the last case, the DIT metric would be 3 for the **Mixins** mechanism. According to Table 4.1, the NCC metric is the same for all mechanisms: *Class*, *Mixins*, and *Extra Aspect* implement the *Transflash* and *Bluetooth* features ($NCC = 2$). However, in the *Infrared* scenario, both **Inheritance** and **Mixins** mechanisms would have four classes where $NCC > 1$ (Figure 4.8). In other words, these four components do not separate the three aforementioned features. The CDC and VS metrics also increase using these mechanisms: before *Infrared*, $CDC_{Transflash} = CDC_{Bluetooth} = 2$ and $VS = 4$; after *Infrared*, $CDC_{Transflash} = CDC_{Bluetooth} = CDC_{Infrared} = 4$ and $VS = 8$.

The **AOP** mechanism does not have such `Scalability` problem because of the *Extra Aspect* responsible for declaring the precedence among the features (only this component have $NCC > 1$). Whenever a new feature must be considered, the aspect for that feature is written, and the existing precedence aspect (Listing 4.2) is modified to take the new feature into consideration (in the *Infrared* scenario, $VS = 5$). The CDC metric remains the same for all features: for example, the *Infrared* feature is diffused in its own

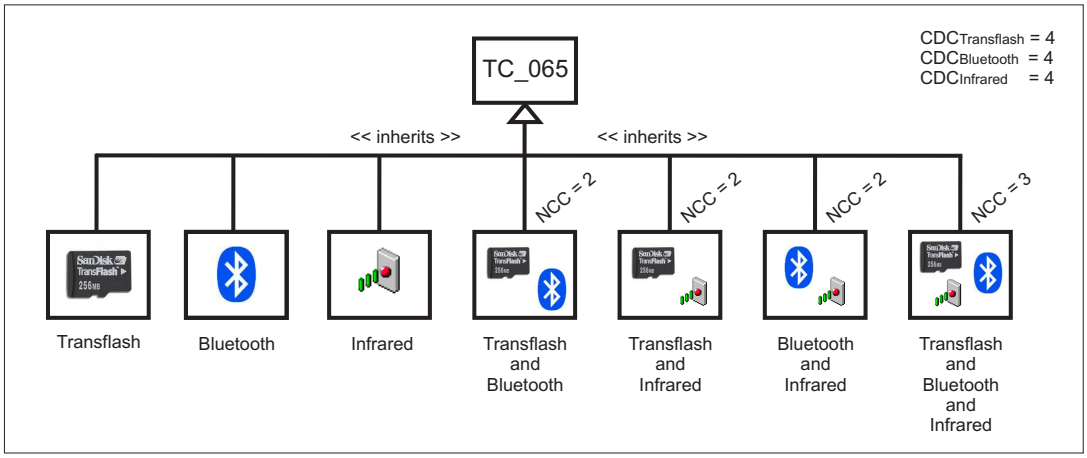


Figure 4.8 Three optional features implemented using Inheritance. The impact on Scalability is analogous when considering Mixins.

aspect and in the *Extra Aspect* ($CDC_{Infrared} = 2$). According to the concepts discussed in Section 3.4.2, design rules are essential to guarantee the `Modularity` criteria in the **AOP** approach. Since the design rules language presented in [34] is under construction, a simple document can make explicit the design rules: "*TC_065 must provide the procedures method; Aspects are responsible for introducing in the correct order the Transflash, Bluetooth, and Infrared features at the end of procedures body*".

Listing 4.2 Declare Precedence Extra Aspect.

```
1 public aspect ExtraAspect {
2     declare precedence : TransflashAspect, BluetoothAspect, InfraredAspect;
3 }
```

When compared to the other mechanisms, the **Decorator** mechanism is worse in the following metrics: $CDLOC_{Transflash}$, $CDLOC_{Bluetooth}$, VS, and CBC. This happens because it requires a lot of infrastructural code as illustrated in Figure 3.1. Nevertheless, the mechanism does not need any artifact to implement both features ($CDLOC_{Both} = 0$). Therefore, only one class is responsible for implementing the *Transflash* feature, being important to the `Modularity` criteria. Analogously, only one class implements the *Bluetooth* feature. The same happens in the *Infrared* scenario, which means that the **Decorator** mechanism does not have the aforementioned `Scalability` problem.

So far, we have analyzed how to structure the test variabilities showed in this section. No discussion about how to instantiate the variabilities was considered yet. In

End of Method Body		Inheritance	Decorator	Mixins	AOP
CDLOC	Bluetooth	36	40	36	36
	Transflash	21	25	21	21
	Both	44	0	2	4
	Total	101 (+180%)	65 (+80%)	59 (+63%)	61 (+69%)
CDC	Bluetooth	2	1	2	2
	Transflash	2	1	2	2
NCC	Both (Class)	2	-	-	-
	Both (Mixins)	-	-	2	-
	Both (Extra Aspect)	-	-	-	2
LOC	Commonalities	75	75	75	75
	Variabilities	101	65	59	61
	Total	176 (+48%)	140 (+17%)	134 (+12%)	136 (+14%)
VS		4	5	4	4
CBC		3	6	4	4
DIT		1	1	2	0

Table 4.1 Beginning/End of Method Body metrics.

what follows, we outline such topic. The **Inheritance**, **Mixins**, and **Decorator** mechanisms need some code infrastructure like a Factory Method [37] to instantiate the correct product line variability according to the phone that is under test. When taking this infrastructure into consideration, the following metrics increase for these three mechanisms: CDC, LOC, VS, and CBC . For example, when considering the **Decorator** infrastructure, $CDC_{Transflash} = CDC_{Bluetooth} = 2$, since references to the features will be diffused in some other class of the infrastructure. The **AOP** mechanism also needs some infrastructure like a *makefile* to insert/remove the aspects from the classpath before starting the compilation.

Figure 4.9 summarizes the mechanisms and criteria used in this section to implement optional test variabilities at the **Beginning/End of Method Body**.

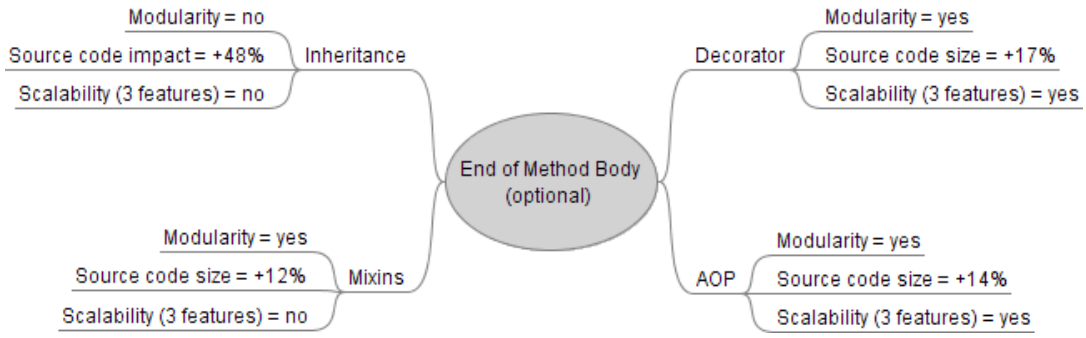


Figure 4.9 Summary: End of Method Body mechanisms.

4.3.2 Beginning/End of Method Body Cloned

⇛ **Feature type analyzed:** optional.

As discussed in Section 3.4.1, some optional features in TAF framework must be enabled before being tested. Figure 4.10 illustrates the *Transflash* feature being enabled. This task is done by turning on the flex bit of the feature through the *setBits* method.

The next variability kind discussed in this work is cloned at the beginning/end of method body. For this reason, we call this variability **Beginning/End of Method Body Cloned**. As showed in Figure 4.10, the code responsible for turning on the *Transflash* feature is cloned in many test cases, being time consuming and error-prone maintaining it.

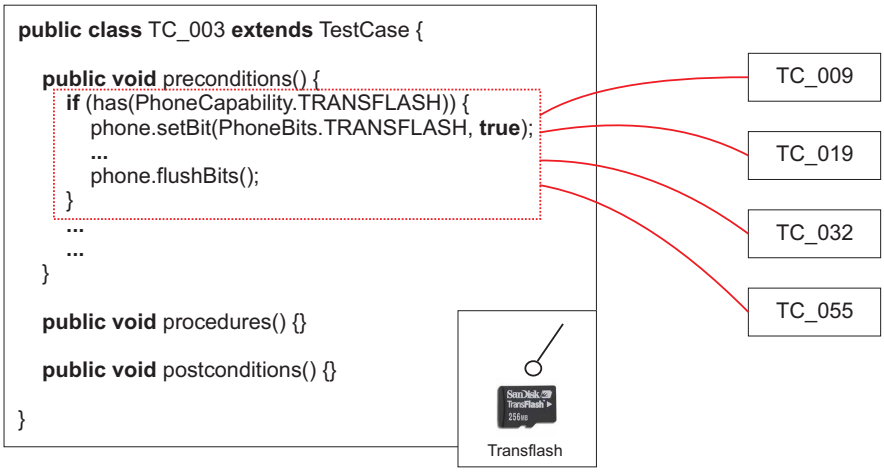


Figure 4.10 Beginning of Method Body Cloned.

Since the feature is crosscutting throughout many classes, we have used the **AOP** mechanism, which is a well known mechanism to implement crosscutting concerns [46]. Besides, some researchers [19] claim that **AOP** implements *Homogeneous Crosscuts* elegantly. Because homogeneous crosscuts means "multiple join points with a single piece of advice", the *Transflash* example falls in such definition: multiple *preconditions* methods with the same *Transflash* code.

Another possible approach consists of using **Inheritance**. In this case, a new class *TransflashFlexBit* (which extends *TestCase*) may be created to implement the optional code responsible for setting the *Transflash* flex bit. The classes which need such functionality (*TC_003*, *TC_009*, *TC_019*, *TC_032*, and *TC_055*) might inherit *TransflashFlexBit*. However, we found at least two problems in this approach: (i) such new hierarchy is not according to the TAF framework definition, which says that each test case must directly inherit from *TestCase*; and (ii) the *if* statement can not be removed from the *TransflashFlexBit* class. Otherwise, instead of optional, the *Transflash* flex bit code will become mandatory when instantiating any *TransflashFlexBit* subclass.

Therefore, we have only used the **AOP** mechanism in this variability. According to the metrics showed in Table 4.2, the **AOP** approach is better than the **Original** one. First, it removed the cloned code, decreasing the `Source Code Size` (-70% for $CDLOC$ and -6% for LOC). In addition, the *Transflash* flex bit code is not scattered in many classes ($CDC = 1$), being worthwhile to the `Modularity` criteria. Notice that there is an increasing[4] on the total coupling: $CBC = 5$. It happens since the aspect created has direct references to the five aforementioned test cases.

Begin of Method Body Cloned	**Original**	**AOP**
CDLOC	40	12 (-70%)
CDC	5	1
LOC	460	432 (-6%)
VS	5	6
CBC	0	5

Table 4.2 Beginning/End of Method Body Cloned metrics.

[4]It seems that this coupling metric is inconsistent. Despite the gains with respect to modularity, the coupling has increased. We discuss more about this in the Concluding Remarks of the work.

In summary, the metrics have just confirmed that the **AOP** mechanism is suitable for implementing crosscutting concerns [46], homogeneous crosscut [19], and for removing cloned code [61].

4.3.3 Whole Method Body

⇛ **Feature type analyzed:** alternative.

The **Whole method body** variability occurs when the whole body of a method differs among product line instances. Figure 4.11 illustrates an example of such variability: it is at the whole *preconditions* method body. The alternative closable devices (*FLIP* and *SLIDER*) are depicted. Only one device can be used in each product line instance.

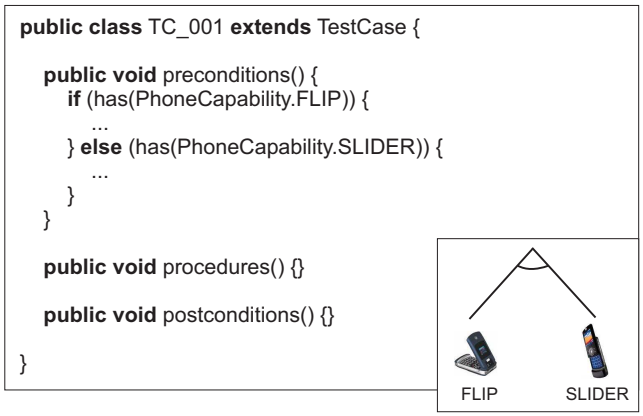


Figure 4.11 Whole Method Body.

We have implemented this test variability using four mechanisms. We did not use the **Decorator** and **Mixins** mechanisms since no feature should be composed in this variability. Next, we present the implementation details of the four mechanisms.

- **Strategy:** relies on the Strategy design pattern [37]. An interface *Closable* is defined and two classes (*FLIP* and *SLIDER*) implement it;
- **Dependency Injection:** analogous to Strategy. In addition, a set method (*setClosable*) is defined in the test case. It is used by a container (here, *Spring*) to set the closable device available to be used in the test;

- **AOP - Intertype:** this implementation relies on intertype declarations of AspectJ. The method *preconditions* is omitted from the class body. Next, a set of aspects become responsible for introducing the missing method into the class using an intertype declaration. Each aspect introduces a different version of such method (*FLIP* or *SLIDER*);

- **AOP - Around:** this implementation uses around advices. This way, the *preconditions* method body is removed. Afterwards, aspects (one for each alternative feature, *FLIP* or *SLIDER*) advise the *preconditions* method and execute the code defined in the around advice.

Table 4.3 shows the metrics for the **Whole Method Body** mechanisms. Notice that the metrics contain information about the infrastructure to instantiate the features in the **Dependency Injection** mechanism (such as the *spring.xml* file, the *setClosable* method, and some classes of the container). To maintain equality, we calculated metrics of a Factory Method [37] used by the **Strategy** mechanism as well.

Because of these infrastructures, the **Strategy** and **Dependency Injection** mechanisms have higher impact on the `Source Code Size` than the **AOP - Intertype** and **AOP - Around** mechanisms. For the same reason, VS and CBC are also higher. In particular, when considering the **Dependency Injection**, we have $CBC = 6$, but actually one may consider 5, since the class used to instantiate the correct service (in this case, the closable device) is provided by the *Spring Framework*. This way, the class is not susceptible to change frequently. We are reporting this situation because metrics tools such as *Metrics* [11] and *Eclipse Metrics Plug-in* [8] discard the classes from packages like *java.lang* when calculating the total coupling of the system.

In both approaches, the *FLIP* feature is implemented in a separated class. However, there is a direct reference to such class in the Factory Method (**Strategy**) and in the *spring.xml* file (**Dependency Injection**). Thus, $CDC_{FLIP} = 2$. Analogously, $CDC_{SLIDER} = 2$.

The metrics have showed that both **AOP** approaches are very similar. The decision about which mechanism to use may depend on the user: if he prefers to maintain the *preconditions* method, he should use the around approach. Otherwise, he should use the intertype approach. Notice that the intertype approach is not according to the TAF

Whole Method Body		**Strategy**	**Dep. Injection**	**AOP (it)**	**AOP (a)**
CDLOC	Flip	10	10	11	12
	Slider	13	13	14	15
	Total	23 (+53%)	23 (+53%)	25 (+67%)	27 (+80%)
CDC	Flip	2	2	1	1
	Slider	2	2	1	1
LOC	Commonalities	239	228	211	213
	Variabilities	23	23	25	27
	Total	262 (+16%)	251 (+11%)	236 (+4%)	240 (+6%)
VS		5	5	3	3
CBC		6	6	2	2

Table 4.3 Whole Method Body metrics.

framework definitions, which says that each test case must implement the *preconditions*, *procedures*, and *postconditions* methods.

In particular, around advices may be used to remove some functionality. Such functionality is known as negative features [16]. Nevertheless, it is important to note that AspectJ does not remove the code from the resulting bytecodes [17]. Therefore, developers must be careful when implementing negative features using around advices in limited devices such as mobile phones.

Basically, all analyzed mechanisms provide SoC, being important to the `Modularity` criteria. However, because of semantic dependencies (presented in Section 3.4.2), the **AOP** approaches depend on design rules. Furthermore, there is no impact on `Scalability` in all mechanisms. To add a new closable device in the **AOP** mechanisms, just a simple aspect is necessary. When considering the **Strategy** and **Dependency Injection** mechanisms, a little task is needed beyond creating the new class: in order to be used, the new feature must be added in the Factory Method or in the *spring.xml* file. Figure 4.12 confirms that the **Strategy** and **Dependency Injection** mechanisms are similar. The same happens with the **AOP** approaches. Notice that Figure 4.12(b) does not contain design rules: there are cyclical dependencies between the components of the **AOP** approaches.

Figure 4.13 summarizes the mechanisms and criteria used in this section to implement alternative test variabilities at the **Whole Method Body**.

		1	2	3	4	5
Closable Interface	1					
FLIP	2	x				
SLIDER	3	x				
Factory / Container	4		x	x		
TC_001	5	x			x	

(a) Strategy and Dependency Injection.

		1	2	3
TC_001	1		x	x
FLIP Aspect	2	x		
SLIDER Aspect	3	x		

(b) AOP Intertype and AOP Around.

Figure 4.12 DSMs of the Whole Method Body mechanisms.

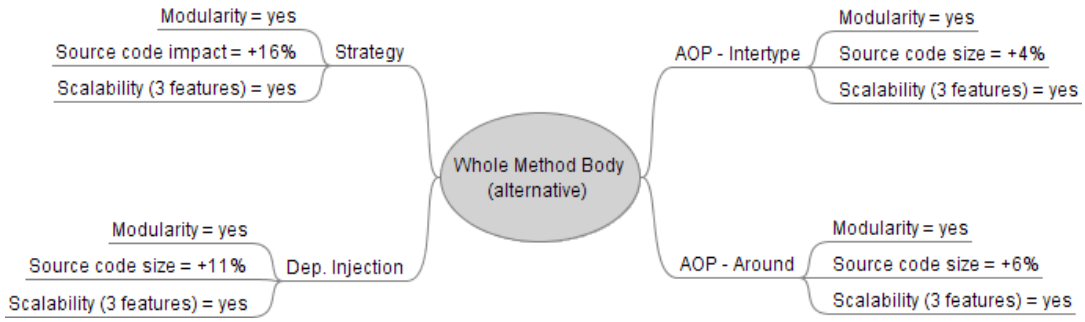


Figure 4.13 Summary: Whole Method Body mechanisms.

4.3.4 Middle of Method Body

⇛ **Feature type analyzed:** optional.

In this section, we present the variability kind that occurs at the **Middle of Method Body**. Figure 4.14 illustrates an optional feature of mobile phone browsers. Some browsers are limited and can not store web pages. On the other hand, if such functionality is present, it should be tested.

Notice that the variability presented in Figure 4.14 can not be addressed by using pure aspects. For example: if we use an aspect to weave the store web page code after calling the *takeWebPageScreenshot* method, the aspect will weave the feature in five places, which is incorrect (there are five *takeWebPageScreenshot* calls).

Since calls to methods such as *launchApp*, *goToURL*, and *takeWebPageScreenshot* often happen more than once in TAF test cases, **Tracematches** may be an useful mechanism to address these variabilities. This way, instead of considering only the *takeWebPageScreenshot* call, we can use tracematches to create a regular expression, as showed

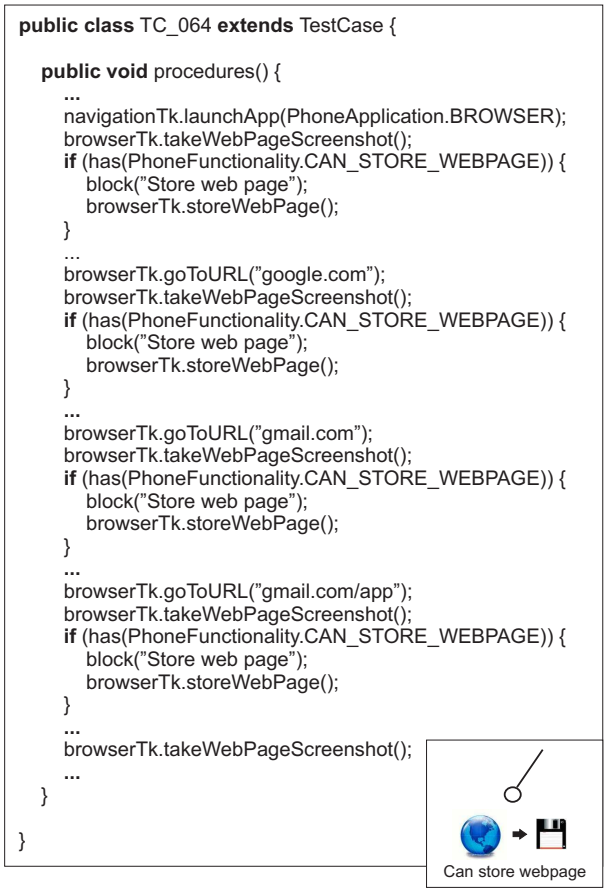


Figure 4.14 Middle of Method Body.

in Listing 4.3. Now, after any call to *launchApp* or *goToURL* followed by a call to the *takeWebPageScreenshot* method will be advised by the *tracematch* and the code to store the web page will be executed.

Listing 4.3 Store Web Page Tracematch.

```
1  tracematch() {
2
3      sym launchApp after : call(* *.launchApp(..)) && within(TC_064);
4
5      sym goToURL after : call(* *.goToURL(..)) && within(TC_064);
6
7      sym takeWebPageScreen after : call(* *.takeWebPageScreenshot(..)) && within(TC_064);
8
9      (launchApp | goToURL) takeWebPageScreen {
10         block("Store web page");
11         browserTk.storeWebPage();
12     }
13 }
```

According to the metrics showed in Table 4.4, the **Tracematches** approach is better than the **Original** one. The `Source Code Size` has been decreased because there is no cloned code throughout the test case anymore (-50% for $CDLOC$ and -6% for LOC). In the `Modularity` context, any unanticipated change in the store web page code is now localized: only the *tracematch* of Listing 4.3 needs changing.

However, in order to guarantee the `Modularity` criteria when using **Tracematches**, design rules must be present. Defining **Tracematches** produces a strong coupling between the *tracematch* and the *TC_064*. Any unanticipated change in the *TC_064*, like changing the order of some statements, may break the *tracematch*. In this way, we have a cyclical dependency situation: the *tracematch* depends on the *TC_064*; and to change the *TC_064*, the developer must be aware about the tracematch.

Middle of Method Body	**Original**	**Tracematches**
CDLOC	24	12 (-50%)
CDC	1	1
LOC	194	182 (-6%)
VS	1	2
CBC	0	1

Table 4.4 Middle of Method Body metrics.

Figure 4.15(a) illustrates the cyclical dependency through a DSM (presented in Section 2.3), whereas Figure 4.15(b) shows design rules coming into play to remove the dependency between the *tracematch* and *TC_064*. Again, a document can make explicit the design rules: "*TC_064 must provide the procedures method; A tracematch is responsible for introducing the store web pages feature according to the following regular expression: after calling (launchApp || goToURL) takeWebPageScreenshot*".

Due to refactoring reasons, one may change the order of some statements of the program. As discussed, changing the statements' order of a program may stop **Tracematches** working. Nevertheless, this problem is minimized when considering test cases, since they have a defined and strict order. For instance, changing the order of the *goToURL* and *takeWebPageScreenshot* methods does not make sense: it is not possible to take a web page screen shot before accessing it. Indeed, we believe that even in this domain, design rules might be useful to avoid problems.

	TC_064	Tracematch
TC_064		x
Tracematch	x	

(a) Cyclical Dependency.

	DRs	TC_064	Tracematch
DRs			
TC_064	x		
Tracematch	x		

(b) Design Rules removing Cyclical Dependency.

Figure 4.15 Tracematches with/without Design Rules.

4.3.5 Method Parameter

⇛ **Feature type analyzed:** alternative.

Last but no least, we present the **Method Parameter** variability. It occurs whenever the value of a method parameter differs according to the selected product. Figure 4.16 illustrates this situation. The parameters of the *loadWebSession*, *setWebSessionAsDefault*, *typeText*, and *scrollAndSelectLink* methods vary depending on the currently installed browser (the alternative feature is depicted in Figure 4.16: either *Opera* or a proprietary *Motorola* browser is selected in the product line instance).

We have implemented this variability using two mechanisms. The details of each implementation are depicted as follows.

- **AOP:** relies on intertype declarations. Two aspects implement the method parameter's values for each browser separately. This way, the test case uses constants introduced by the aspects as parameters of methods, eliminating the *if-else* statements of the test. For example, the *loadWebSession* method would use as parameter a constant introduced by the aspect.

- **Configuration Files:** relies on configuration files to provide the method parameter's values. In order to load the files, the test case must use some infrastructure responsible for reading them. One file is required by browser.

Table 4.5 shows the metrics for the **Method Parameter** mechanisms. Both mechanisms provide a suitable SoC since the variabilities of each browser are implemented by separated aspects and files, which means that the test case no longer handles variability using *if-else* statements.

```
public class TC_055 extends TestCase {

   public void preconditions() {}

   public void procedures() {
      ...
      if (has(PhoneFunctionality.OPERA_BROWSER)) {
         browserTk.loadWebSession(Session.WEB_SESSION_HTTP);
         browserTk.setWebSessionAsDefault(Session.WEB_SESSION_HTTP);
      } else {
         browserTk.loadWebSession(Session.WEB_SESSION_WAP);
         browserTk.setWebSessionAsDefault(Session.WEB_SESSION_WAP);
      }
      ...
      If (has(PhoneFunctionality.OPERA_BROWSER)) {
         editorTk.typeText(BrowserURLContent.URL_013);
      } else {
         editorTk.typeText(BrowserURLContent.URL_017);
      }
      ...
      If (has(PhoneFunctionality.OPERA_BROWSER)) {
         browserTk.scrollAndSelectLink(BrowserLinkContent.PAGE_036);
      } else {
         browserTk.scrollAndSelectLink(BrowserLinkContent.PAGE_026);
      }
      ...
   }

   public void postconditions() {}

}
```

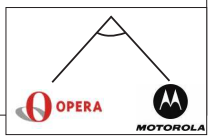


Figure 4.16 Method Parameter.

The $LOC_{Variabilities}$ metric for the **AOP** mechanism is higher when comparing to the **Configuration Files** mechanism. It happens because configuration files do not have lines to define *package*, *import*, and *aspect* signatures. In contrast, the $LOC_{Commonalities}$ metric for the **AOP** mechanism is smaller, since the **Configuration Files** mechanism relies on an additional infrastructure for loading the file (for this purpose, we have considered a Factory Method [37]. Thus, $VS_{Config.Files} = 4$). It is worthwhile to note that when considering such infrastructure, $CDC_{Opera} = CDC_{Motorola} = 2$.

Although $CBC_{Config.Files} = 3$, one may consider 2, because the class responsible for loading the configuration file is not susceptible to change frequently (for example, it may be the *java.util.Properties* class).

To evaluate the `Scalability` criteria, we supposed 6 browsers, instead of 2. Notice that the CDC metric of each mechanism remains the same for all 6 browsers: $CDC_{AOP} = 1$ and $CDC_{Config.Files} = 2$ (considering the factory).

Despite the very similar LOC_{Total} (87 and 88), when considering 6 browsers we

Method Parameter		AOP	Conf. Files
CDLOC	Opera	10	3
	Motorola	10	3
	Total	20 (-28%)	6 (-78%)
CDC	Opera	1	2
	Motorola	1	2
LOC	Commonalities	67	82
	Variabilities	20	6
	Total	87 (-6%)	88 (-5%)
VS		3	4
CBC		2	3

Table 4.5 Method Parameter metrics.

have observed a higher increasing in the **AOP** mechanism: $LOC_{AOP} = +38\%$ *versus* $LOC_{Config.Files} = +16\%$. As soon as new browsers are considered, such difference tends to be higher.

Figure 4.17 summarizes the mechanisms and criteria used in this section to implement alternative test variabilities at the **Method Parameter**.

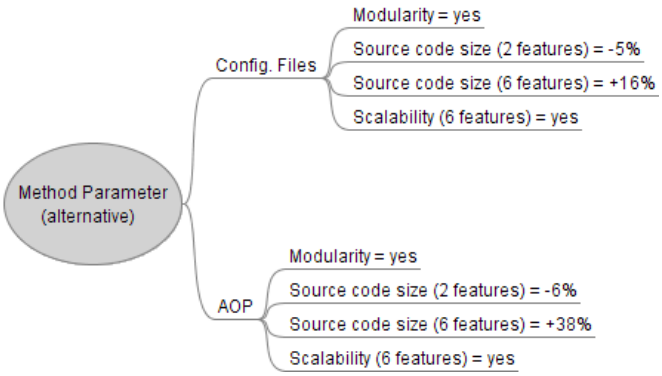


Figure 4.17 Summary: Method Parameter mechanisms.

4.4 SUMMARY: DECISION MODEL

After analyzing each variability by its exact location at the source code and considering the features types, we finally present our decision model. Since such model aims at

restructuring test variabilities in SPL, the assumption for using the model is that the tests where it will be applied are not well structured and use *if-else* statements to handle their variabilities. By "not well structured", we basically mean that the features are not modularized.

Although our model was defined based on some examples of test variabilities, we believe that the DSM analysis and metrics results of the mechanisms would be similar when considering other examples. Of course, metrics like LOC and $CDLOC$ would be different, which can affect slightly the `Source Code Size` criteria. In addition, depending on the number of variabilities analyzed, the total system coupling (CBC) would not be the same either. However, CDC and NCC would be similar, which may guarantee the `Modularity` and `Scalability` criteria in those other examples.

In summary, the decision model presented in this work **suggests** some mechanisms to restructure test variabilities in SPL. Applying the recommended mechanisms in accordance to the aforementioned assumption may provide several benefits, such as:

- **Eliminating cloned code:** cloned code is nothing more than a breeding ground for bugs in the future [35]. Eliminating them means avoiding these bugs when evolving the SPL. Moreover, the code becomes more modular;
- **Independent evolution:** because the features may be modularized using the mechanisms recommended by the decision model, developers might work in parallel when evolving the tests;
- **Productivity increasing:** time consuming and error-prone tasks like evolving cloned code are avoided. Moreover, due to the provided modularization, many developers may work in parallel, reducing the time-to-market.

All of these benefits will be evaluated in Chapter 6.

Finally, Figure 4.18 illustrates our decision model. As explained in Section 4.3, the exact variability location at the source code, the feature type, and some criteria are the model's inputs.

Figure 4.18 Decision Model.

CHAPTER 5

SUPPORTING THE VARIABILITY IMPLEMENTATION MECHANISMS RECOMMENDATION

As we discussed earlier, choosing an appropriate mechanism to restructure a given variability may be a difficult task. If no tool support is available, such task may get worse. For example, if a variability is scattered throughout many classes, discovering in what classes the variability is may be time consuming and error-prone.

To minimize such difficulties, we present in this chapter a prototype tool that we have developed. The tool is able to **recommend mechanisms** aiming at restructuring SPL variabilities (cloned or not). This way, given a variability implemented using *if-else* statements, the tool may recommend a suitable mechanism to restructure such variability according to the decision model depicted in Section 4.4. Our tool is an extension of the `FLiPEx` [26] tool. Thus, firstly we present such tool and explore how it works. Afterwards, we present our `FLiPEx` extension tool, named `FLiPRec`.

5.1 FLIPEX TOOL

`FLiPEx` [26] is a refactoring tool that implements code transformations [15] for extracting product variabilities from Java classes to AspectJ aspects. `FLiPEx` is based on the Eclipse plug-in platform [28] and uses the Eclipse infrastructure to perform source code refactorings that extract product variabilities.

We explore the `FLiPEx` tool by an example. Listing 5.1 illustrates a class responsible for managing the graphical part of an application. The class extends *FullCanvas*. Depending on the graphics API supported by the specific device, the class may extend another canvas.

Listing 5.1 MainCanvas class.

```
1 import com.nokia.mid.ui.FullCanvas;
2 public class MainCanvas extends FullCanvas {
```

```
3       ...
4   }
```

In this context, only one graphical manager is available: for *Nokia* phones. To bootstrap an existing product into a SPL, refactorings are useful to guide the SPL derivation process by extracting product variabilities and appropriately structuring them [26]. Therefore, we might use the `FLiPEx` tool to transform such *Nokia* product into a SPL based on the aforementioned refactorings.

Using the `FLiPEx` tool, the process begins when the user selects the code to be extracted. In this case, "`extends FullCanvas`". Afterwards, the refactoring is presented to the user as a wizard that he interacts with to provide all the information required to perform the refactoring (for example, the name of the resulting aspect). The refactoring consists of checking the preconditions of the selected code, removing the original code from the *MainCanvas* class, and generating the AspectJ code [26]. The result of applying the refactoring is showed in Listing 5.2.

Listing 5.2 Extracting code from Java class to AspectJ aspects using the FLiPEx tool.

```
1   // core
2   public class MainCanvas {
3       ...
4   }
5
6   // Nokia variability
7   import com.nokia.mid.ui.FullCanvas;
8   public aspect NokiaCanvasAspect {
9       declare parents: MainCanvas extends FullCanvas;
10  }
```

Notice that the SPL *extractive* [29] approach has been used: we bootstrapped an existing product into a very simple SPL. Now, we can also use the *reactive* [29] approach to extend the existing SPL to encompass other products (see the *Siemens* variability in Listing 5.3).

Listing 5.3 Extending the SPL to encompass a new product.

```
1   // Siemens variability
2   import com.siemens.mp.color_game.GameCanvas;
3   public aspect SiemensCanvasAspect {
4       declare parents: MainCanvas extends GameCanvas;
5   }
```

The refactoring above is known as "*Move Extends Declaration to Aspect*". `FLiPEx` provides more 7 refactorings. Each `FLiPEx` refactoring has an associated *Extractor*, which is responsible for removing the code from the Java class and creating the AspectJ code that inserts the variation [26]. Each *Extractor* has a corresponding *Validator*, responsible for analyzing the selected code to check if it meets all the preconditions of its refactoring.

So far, we have showed that `FLiPEx` is able to extract existing products into a SPL. Nevertheless, `FLiPEx` is also able to restructure a SPL based on **Conditional Compilation** (discussed in Section 3.3) to **AOP**. For instance, suppose the code of Figure 3.3. After selecting the whole *playSound* body, the `FLiPEx` wizard will give an option to create three aspects (one for each alternative sound API). If the user agrees with such information, the code is extracted into three aspects.

5.2 FLIPREC: A FLIPEX EXTENSION TOOL

The `FLiPEx` tool extracts product variabilities only to aspects. However, previous researches [16, 56, 19] as well as our decision model [33, 32] showed that the **AOP** mechanism is not always the best one to implement SPL variabilities.

For this reason, we have implemented an extension to the `FLiPEx` tool. Our tool (named `FLiPRec`) aims at **recommending mechanisms** to restructure variabilities implemented using *if-else* statements in existing SPLs. The available mechanisms to be recommended are **Inheritance**, **Configuration Files**, **AOP**, and **Tracematches**. It is important to note that our tool does not realize refactorings.

In order to get the `FLiPRec` recommendation, the user must perform basically two steps: (i) select the desired *if-else* statement; and (ii) click on the recommendation button. However, differently of `FLiPEx`, `FLiPRec` searches for clones of the selected code before starting the process of recommending mechanisms. In addition, `FLiPRec` generates primitive design rules. We consider primitive since they rely on plain text, instead of a design rules language (the design rules language presented in [34] is under construction). Figure 5.1 details the main differences between `FLiPEx` and its extension `FLiPRec`. Due to the clones searching, `FLiPRec` recommends better pointcuts than the `FLiPEx` tool.

The `FLiPEx` tool provides Eclipse Extension Points [28] related to the *Validator* and *Extractor* interfaces, which means that we can create new validators and extractors.

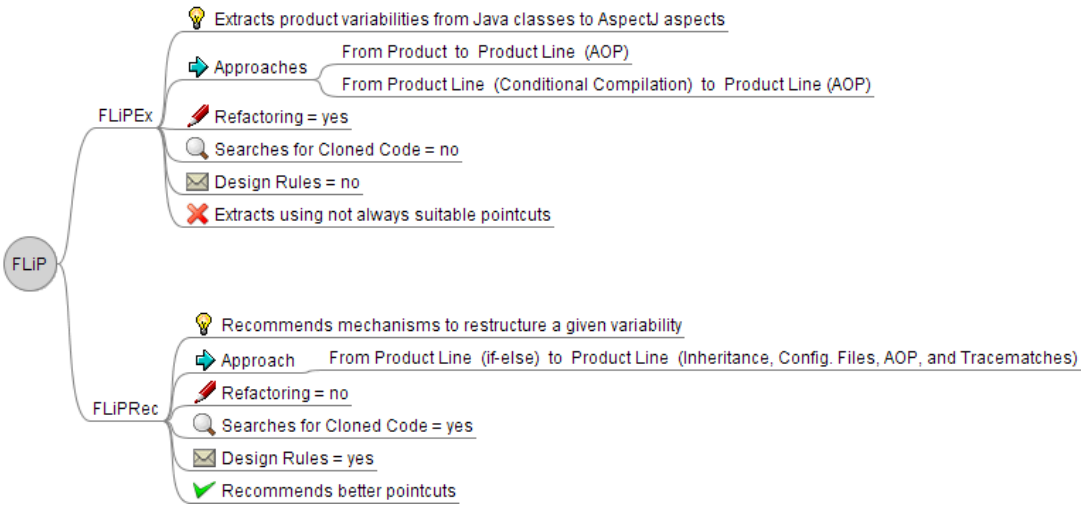


Figure 5.1 Differences between FLiPEx and FLiPRec.

Since `FLiPRec` does not realize refactorings, we have only used the *Validator* extension point. Four validators have been created: *InheritanceValidator*, *ConfigurationFilesValidator*, *AOPValidator*, and *TracematchesValidator*. One important characteristic of these extension points is the inversion of control: the `FLiPRec` validators are called by `FLiPEx`. Therefore, if the `FLiPRec` plug-in is installed in Eclipse, running the `FLiPEx` tool implies running the `FLiPRec` validators as well.

When clicking on the recommendation button, `FLiPRec` firstly searches for clones of the selected code. For each clone, an object of the *Neighbors* class is created. Such object stores the statements which are before and after the clones. The selected code also has an associated *Neighbors* object. Notice that the *Neighbors* objects are important for determining the exact location of the selected code and its clones. For instance, if there is no statement after the selected code, it means that such code is at the end of the method body.

Each `FLiPRec` validator depends on the locations presented in Section 4.3. For example, the *AOPValidator* depends on the *BeginningOfMethodBodyCloned* location, as illustrated in Figure 5.2. As mentioned, by using the *Neighbors* object, it is possible to determine the location of the selected code and its clones. Based on these *Neighbors* objects, the *BeginningOfMethodBodyCloned* location verifies if all clones are at the beginning of their respective method's bodies. If positive, the *AOPValidator* returns *true* and then the **AOP** mechanism is recommended to restructure the selected code and its

clones. Notice that a validator may depend on many locations.

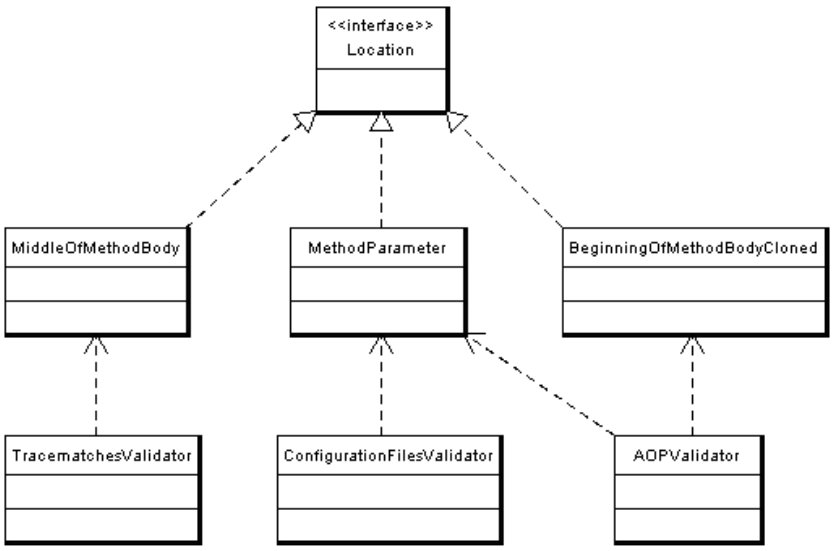


Figure 5.2 Part of the FLiPRec Class Diagram: some locations and validators.

Our tool uses the Eclipse Java Development Tool (JDT) [7], Abstract Syntax Trees (AST) [12], and Visitors [37] to perform the aforementioned analysis.

In what follows, we provide four examples of using our tool.

5.2.1 Example 1

Listing 5.4 illustrates two test cases: *TC_009* and *TC_055*. The *if* statement of both tests is cloned. Now, suppose that the user does not know this information. After selecting the *if* statement in *TC_009* and clicking on the recommendation button, `FLiPRec` searches for clones in all available *.java* files and then all validators are executed. Because the variability is cloned at the beginning of two methods, the *BeginningOfMethodBodyCloned* location will be validated. Consequently, the *AOPValidator* returns *true*. Finally, the result is showed to the user (Figure 5.3).

Listing 5.4 If statement cloned in two test cases.

```
1 public class TC_009 extends TestCase {
2     public void preconditions() {
3         if (has(PhoneCapability.TRANSFLASH)) {
4             phone.setBits(PhoneBits.TRANSFLASH, true);
5             phone.flushBits();
6         }
```

```
7           ...
8       }
9   }
10
11  public class TC_055 extends TestCase {
12      public void preconditions() {
13          if (has(PhoneCapability.TRANSFLASH)) {
14              phone.setBits(PhoneBits.TRANSFLASH, true);
15              phone.flushBits();
16          }
17          ...
18      }
19  }
```

In order to extract these variabilities to an aspect, the design rules showed in the *Extractor Description* panel of Figure 5.3 must be followed: the classes *TC_009* and *TC_055* must provide the *preconditions* methods; and to work correctly, they require an aspect to advise the beginning of their *preconditions* methods.

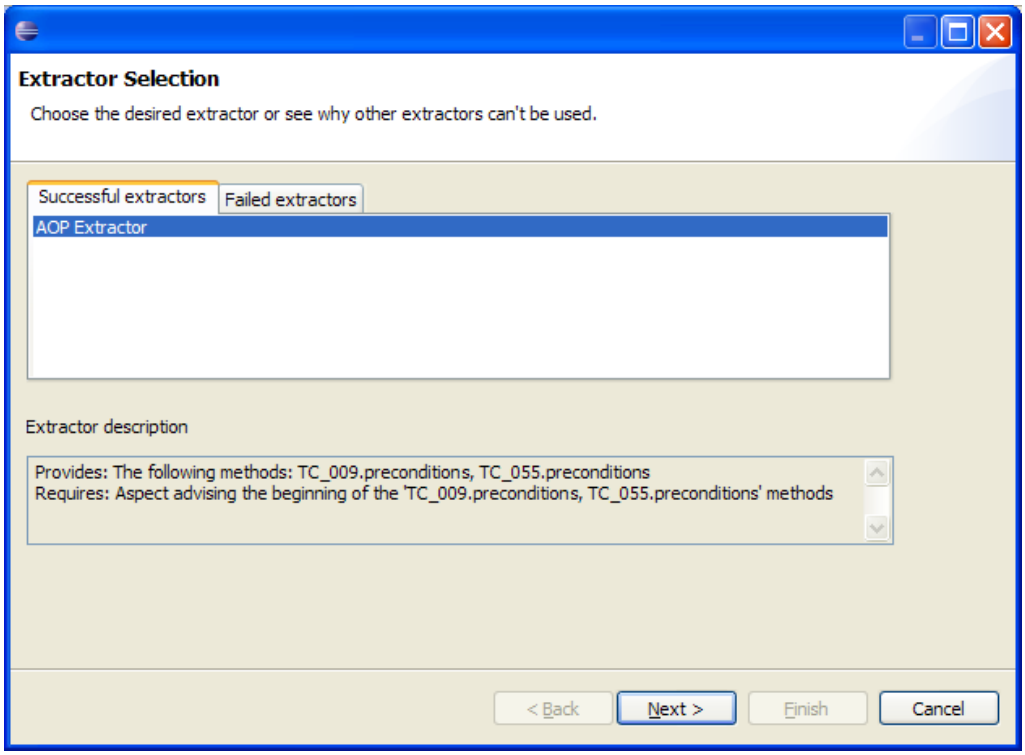


Figure 5.3 FLiPRec recommending the AOP mechanism.

5.2.2 Example 2

Listing 5.5 shows an *if* statement at the middle of the *procedures* body. As mentioned in Section 4.3.4, **Tracematches** can be useful to restructure this kind of variability. In order to avoid problems like introducing the code into wrong places, `FliPRec` must guarantee that there is a unique tracematch to be applied. Otherwise, although the *if* statement falls in the *MiddleOfMethodBody* location, the **Tracematches** mechanism is not recommended.

Listing 5.5 If statement at the middle of procedures body.

```
1  public class TC_064 extends TestCase {
2      public void procedures() {
3          ...
4          browserTk.verifyBrowserHistory();
5          browserTk.goToURL("gmail.com");
6          browserTk.takeWebPageScreenshot();
7          if (has(PhoneFunctionality.CAN_STORE_WEBPAGE)) {
8              block("Store web page");
9              browserTk.storeWebPage();
10         }
11         ...
12         browserTk.goToURL("gmail.com/app");
13         browserTk.takeWebPageScreenshot();
14         ...
15     }
16 }
```

Figure 5.4 summarizes how `FLiPRec` searches for a valid and unique tracematch. Notice that `FLiPRec` is using the *Neighbors* object to find the unique trace. The first upper neighbor is considered: *takeWebPageScreenshot*. Next, `FLiPRec` verifies if there is another call to the *takeWebPageScreenshot* method (*Step 1*). Since there are two calls to this method, after calling the *takeWebPageScreenshot* method is not a unique trace because the variability code would be wrongly introduced into two places, instead of only one. Thus, the tool takes another neighbor into consideration. In this case, it verifies if the next trace (*goToURL* followed by *takeWebPageScreenshot*) exists (*Step 2*). Because such trace already exists, `FLiPRec` repeats the process and uses another neighbor (*Step 3*). Since the trace *verifyBrowserHistory* followed by *goToURL* followed by *takeWebPageScreenshot* is unique, the tool recommends this trace. Notice that the quantity of these loops is configurable. Figure 5.5 illustrates such recommendation.

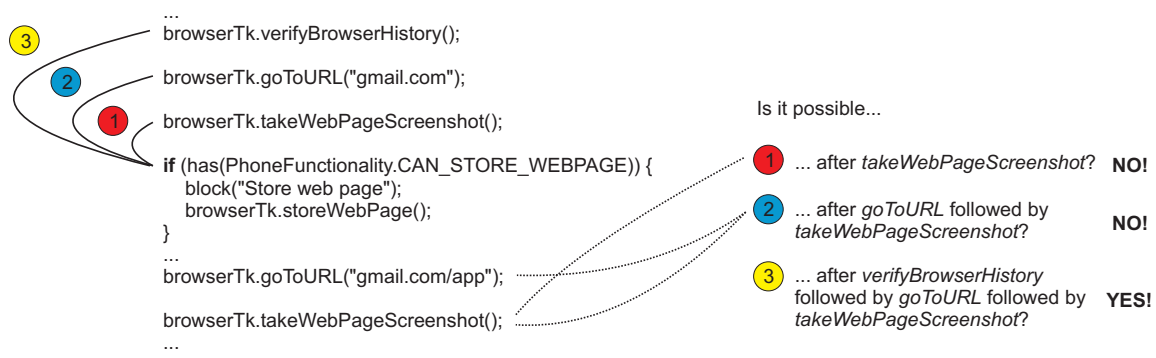


Figure 5.4 FLiPRec searching for a unique tracematch.

5.2.3 Example 3

Listing 5.6 illustrates part of the *Opera* and *Motorola* browser alternative features. There is only one difference in the *if* and *else* bodies: the value of the method parameter.

Listing 5.6 If-else statement implementing an alternative feature.

```
1  public class TC_055 extends TestCase {
2      public void procedures() {
3          ...
4          if (has(PhoneFunctionality.OPERA_BROWSER)) {
5              editorTk.typeText("gmail.com/app");
6          } else {
7              editorTk.typeText("gmail.com");
8          }
9          ...
10     }
11 }
```

Figure 5.2 shows that more than one validator may depend on a specific location (both *AOPValidator* and *ConfigurationFilesValidator* depends on *MethodParameter* location). Thereby, the `FLiPRec` tool can recommend more than one mechanism to implement the selected variability, as depicted in Figure 5.6. Notice that this is according to our decision model: albeit both mechanisms provide `Modularity` and `Scalability`, they differ in the binding time so that we leave the final decision to the user.

5.2.4 Example 4

Listing 5.7 shows a variability cloned in two test cases. Notice that the method calls before the clones are different. After finding the clones, `FLiPRec` will try to find a unique pointcut

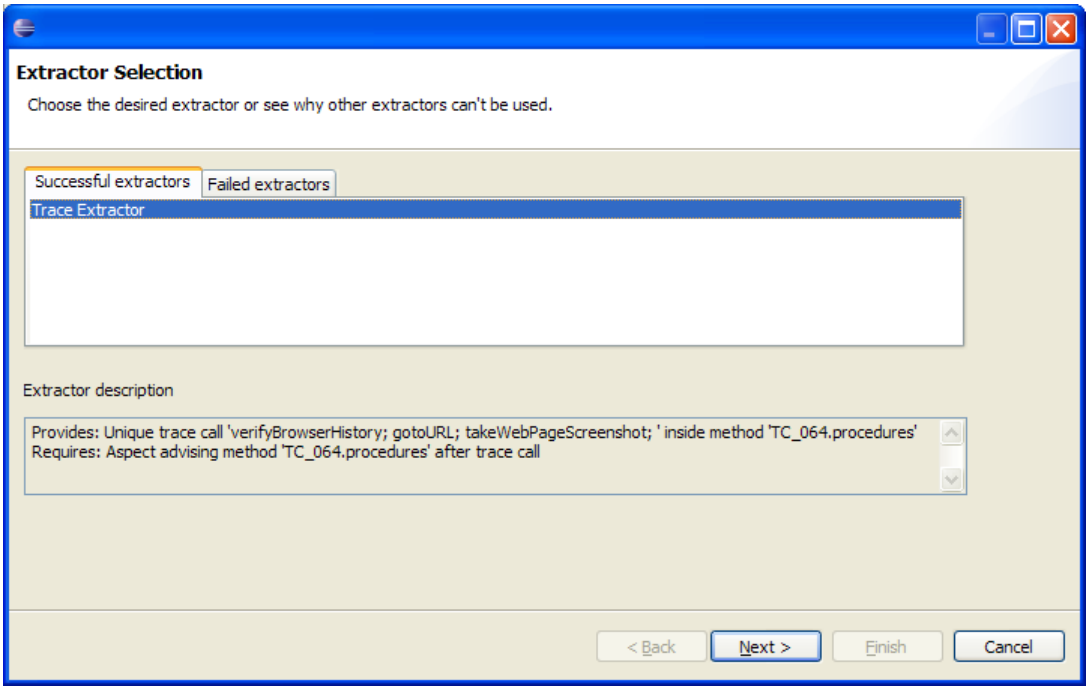


Figure 5.5 FLiPRec recommending the Tracematches mechanism.

to restructure such variability adequately. In this particular case, the tool will search for another call to the *goToApplication* method inside the *TC_015.procedures* method body. The same happens to the *goToIdle* method call. Since there is no additional calls in this example, the tool recommends the **AOP** mechanism with the following pointcut: after calling *goToApplication* or *goToIdle* within *TC_015.procedures* and *TC_016.procedures* methods. Otherwise, it tries to find a valid trace for recommending the **Tracematches** mechanism.

Listing 5.7 Different method calls before the clones.

```
1  public class TC_015 extends TestCase {
2      public void procedures() {
3          ...
4          navigationTk.goToApplication(PhoneApplication.MAIN_MENU);
5          if (has(PhoneCapability.DEDICATED_KEYS)) {
6              ...
7          }
8          ...
9      }
10 }
11
12 public class TC_016 extends TestCase {
13     public void procedures() {
```

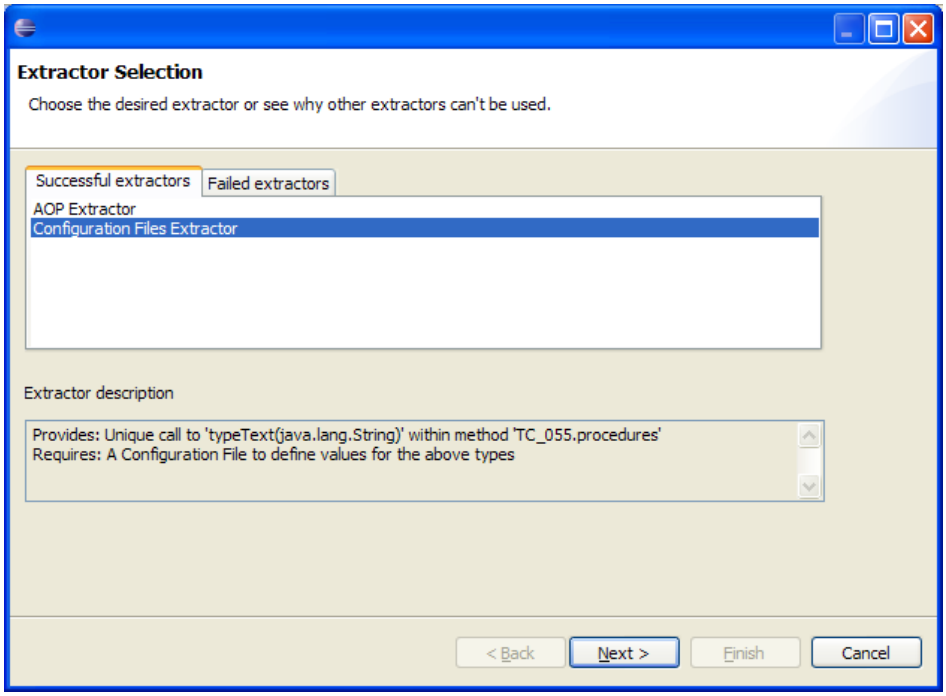


Figure 5.6 FLiPRec recommending both AOP and Configuration Files mechanisms.

```
14          ...
15          navigationTk.goToIdle();
16          if (has(PhoneCapability.DEDICATED_KEYS)) {
17              ...
18          }
19          ...
20      }
21  }
```

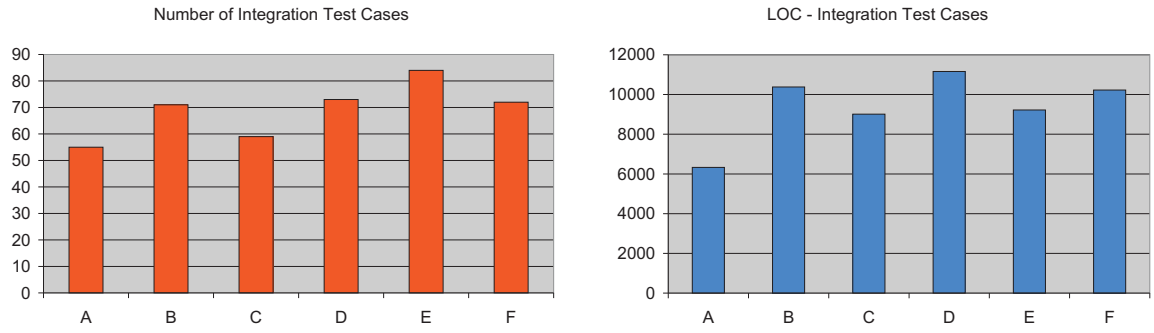
CHAPTER 6

EVALUATION

Before presenting some scenarios to evaluate this work, we firstly present some numbers of the TAF integration test cases. Such numbers will be important for the evaluation since they represent real test cases.

Figure 6.1 illustrates numbers of integration test cases of six families of phones. Figure 6.1(a) shows the number of test cases of each family whereas Figure 6.1(b) depicts the lines of code of each family. Based on these numbers, a test case has on average

$$\frac{56.325}{414} = 136LOC.$$



(a) Total of Integration Test Cases: 414.

(b) Total of LOC: 56.325.

Figure 6.1 Integration Test Cases.

In order to evaluate our approach, we present in this chapter two scenarios. In these scenarios, we will try to evaluate some recommendations of the decision model and how useful `FLiPRec` is to a developer that is restructuring product line variabilities. The scenarios are detailed in what follows.

6.1 SCENARIO 1

For the first scenario, consider a test case which has the number of lines of code next to the average: $150LOC$. Now, suppose that this test case has an optional feature cloned and scattered throughout the test (similar to the code of Figure 4.14). Therefore, if the feature must be updated, many places of the code needs changing, which increases the costs of maintaining such code.

Additionally, suppose that $15LOC$ represent *package* and *import* statements and *class* and *methods* signatures. Lets also assume that $35LOC$ represent the optional and cloned feature. This way, it remains $100LOC$ which represent the common steps (calls to UFs like *launchApp*, *acceptDialog*, and so forth).

Supposing that a developer would like to restructure such variability, the first problem is to define which mechanism to use. This decision is not always easy due to the great variety of existing mechanisms. If he opted to use **Tracematches**, another problem comes into play: finding valid traces in $100LOC$ may be a time consuming and error-prone activity. If the developer uses an incorrect trace, it can affect the test steps order. Consequently, the test may not work as desired.

On the other hand, by using our approach, the developer not only receives the **Tracematches** recommendation, but also a valid and precise trace defined by our tool. Hence, the task of restructuring this test variability is realized faster and is not susceptible to errors either. At the end, both activities - (i) restructuring the feature and (ii) future updates to the feature - will have reduced costs, which may improve the SPL productivity.

Despite the $136LOC$ on average, obviously there are bigger and smaller test cases. For example, the biggest test case analyzed has $495LOC$. In this case, finding valid traces would be much more difficult than the scenario described above, impacting directly on the developer's productivity. On the other hand, when considering the smallest test case analyzed ($42LOC$), this task may be easier. For these small cases, Integrated Development Environments (IDEs) like Eclipse [9] may be enough because of the capability of identifying repeated method calls throughout a determined class.

If no valid trace is found, our approach fails since it may not recommend any mechanism, being an important disadvantage of it. We will try to solve this problem by using other mechanisms like **Program Transformations** as future work.

6.2 SCENARIO 2

In this scenario, we use the phone families illustrated in Figure 6.1. Initially, lets suppose that a developer found an optional variability in a given test of the family *A* and he wants to restructure it. For some unknown reason, he has used the **Inheritance** mechanism to restructure such test variability. Nevertheless, the variability is actually cloned in some tests of the families *C*, *D*, and *F*.

This way, two possibilities appear: either the developer does not know about the cloning or he is aware about this information. In the first one, according to our decision model, he chose the wrong mechanism, since the feature is still not modularized. In the second one, a problem arises: unless someone has a deep knowledge about the feature, a painful and long task should be done to find where the clones are (discarding the known test, it remains 204 tests: 59 (Family *C*) + 73 (Family *D*) + 72 (Family *F*) = 204).

Again, our approach may be useful in scenarios like this. By using our model and tool, the **AOP** mechanism would be selected because of the crosscutting feature nature, instead of **Inheritance**, removing the cloned code and guaranteing the feature modularity. Consequently, future inconsistences are avoided and the costs of maintaining the feature would be reduced. Beyond avoiding the incorrect mechanism choice, our tool would identify the cloned code faster and precisely, reducing costs and improving developer's productivity.

An important limitation of our approach is that `FLiPRec` is capable to find cloned code, not crosscutting concerns. Integrating it with tools like Aspect Mining Tool [5] would improve even more the results provided by our approach.

CHAPTER 7

CONCLUDING REMARKS

This work presented a decision model for recommending mechanisms to restructure test variabilities in SPL. Additionally, we developed a prototype tool aiming at supporting developers when evolving variabilities in SPLs. It can suggest mechanisms to restructure a given variability faster and precisely according to the proposed decision model.

Basically, in order to instantiate our model, the following inputs should be provided: the feature type; the variability location at the source code; and some optional criteria such as `Scalability` and `Source Code Size`. After filling the model with such information, it recommends a suitable mechanism to restructure the given test variability according to the selected criteria.

Although we have discussed throughout the work that our approach targets *if-else* statements encountered in automated integration test cases, applying our work may also be useful in other domains and scenarios. We outline them in what follows:

- Some analogous kind of variabilities found in TAF test cases were encountered in a completely different domain: J2ME Games [33, 32]. Given the different natures of the domains in which the variabilities were found, we believe that such product line variabilities are likely to occur in many other domains as well. According to [41], it seems that our decision model can address variabilities found in other domains beyond testing;

- Since the **Conditional Compilation** mechanism uses analogous conditional statements, like "#ifdef" and "#elif", our approach can be easily applied in product lines which use this mechanism to structure their variabilities. In particular, we have showed in previous work [33] that this is possible. On the other hand, our tool would need some adjustment;

- Although we focused exclusively on **restructuring**, we believe that the decision model can also be used for **structuring** test variabilities. For example, when writ-

ing a variability in the test case, the developer has knowledge about the location where the variability must be implemented and about the feature type. Such information can be inputs of the model. However, we must analyze more deeply if this assumption really holds;

- Because other kinds of tests (like feature[1] tests) implemented with TAF follow the same structure of the integration ones, we would not have problems to use the decision model in those test cases either.

The prototype tool was developed for supporting developers when restructuring product line variabilities implemented using *if-else* statements. The tool can indicate mechanisms faster and precisely according to the decision model, reducing the time spent to restructure variabilities and then increasing developer's productivity.

As we showed in the evaluation of this work (Chapter 6), using our model and tool may provide benefits like: (i) bad smells such as cloned code can be identified; (ii) using the mechanisms suggested may improve the variabilities' modularity, allowing developers to work in parallel; and (iii) due to the improved modularization, time consuming and error-prone tasks are avoided, increasing developer's productivity and reducing the time-to-market.

7.1 CONTRIBUTIONS

The contributions of this work are detailed as follows:

- A code-centric decision model, since the variability is identified by its exact location at the source code. Thereby, our model is more fine-grained than the existing ones, which consider complete algorithms or feature types. The main advantage of being code-centric is for refactoring reasons: since the model consider fine-grained code, it is easier to apply Fowler-like refactorings [35] to restructure variabilities, allowing developers to decide which mechanism to use more easily when dealing with these variabilities at the source code. In addition, existing models only consider qualitative studies, such as the type of features which a mechanism can address adequately. In contrast, our model is also based on quantitative studies by using

[1]Responsible for testing one particular feature such as "multimedia" or "messaging".

metrics of Separation of Concerns (SoC), size, and coupling. The metrics were useful to compare the mechanisms, pointing out their strengths and weaknesses related to some criteria;

- Unique tool to recommend mechanisms for restructuring SPL variabilities. Our tool considers four mechanisms, not only **AOP**. Further, the tool is extensible: to add a new mechanism or location, few interfaces need to be implemented. Besides recommending, `FLiPRec` adds important functionalities to the `FLiPEx` tool, such as searching for cloned code, recommending better pointcuts, and generating design rules.

7.2 RELATED WORK

The first related work discussed here [16] claims that little attention has been given on how to deal with variabilities in SPLs at the source code level. To this end, they examine various implementation approaches with respect to their use in a product line context. They present a model for making a comparison of variability implementation approaches based on the following feature types: positive, negative, optional, alternative, and multioptional (XOR). The work describe, for each mechanism, the possibility of addressing the aforementioned feature types with respect to the following levels: possible, not possible, difficult, and questionable. Additionally, it compares the mechanisms using criteria such as traceability, scalability, and binding time. However, this work neither provides a code-centric study nor quantitative studies as we do, being a high-level approach when compared to our proposal. In contrast, they also make a programming language mapping: for each mechanism, they analyze whether it is possible or not to use the mechanism in the languages Java, C++, Delphi, and Smalltalk. Because TAF test cases are written in Java, we focused on this language only.

Another related approach [56] summarizes product line implementation technologies from a programming language point-of-view. Besides presenting a deep comparison among the available programming languages considering items like domain issues (General-Purpose and Domain-Specific Languages) and paradigms, a framework to compare mechanisms was constructed with respect to features types, similar to [16]. Even though the work provides a little case study with source code, the analysis often remains

strict to the feature types, which means that it does not consider the variability location, much less quantitative studies to compare the mechanisms.

Both previous discussed works motivate the construction of a code-centric decision model as we developed here. They claim that it is a challenging task to understand the mechanisms available for realizing product line variabilities and know which of these mechanisms fits best for a given variability at source code.

The next work [15] proposes a method to address the creation and evolution of SPLs focusing on the implementation and feature level. The method first bootstraps the SPL and then evolves it with a reactive approach. Such work also provides a refactoring catalog obtained from an empirical study of some mobile phone games. The proposed method relies on this catalog to extract product variabilities from Java classes to AspectJ aspects, aiming at improving the SPL design and reducing costs of maintaining it. Although this work provides an initial framework for comparing variability implementation techniques, the proposed method relies only on **AOP** refactorings. These refactorings helped us to learn how to restructure some test variabilities (using the **AOP** mechanism) after analyzing similar variabilities found in mobile phone games. On the other hand, our work proposes a decision model that uses different mechanisms (not only **AOP**), since previous works [16, 56, 19] have showed that **AOP** is not always suitable for addressing SPL variabilities. Nevertheless, we did not provide refactorings, only recommendations (even though they represent an initial step to perform the refactoring. For example: when recommending the **AOP** mechanism, a simple pointcut to be used is indicated).

Another related work [60] provides a study on how to implement variabilities in SPLs. As expected, the work concludes that all mechanisms have some limitations and shortcomings. Although the work did not use **AOP** as a mechanism for implementing variabilities, **Program Transformations** and **Frameworks** are considered. In contrast, the work provides an extensive list of characteristics to evaluate the mechanisms, such as scope, flexibility, efficiency, SoC, tool support, traceability, and so forth. Moreover, it considers four binding times: pre-compile, compile, link, and runtime. On the other hand, the presented study again is neither at the source code level nor quantitative to compare the available mechanisms.

The approach proposed by [30] is a method called Multi-Paradigm Design. This method consists of analyzing commonalities and variabilities of a SPL and implementation

mechanisms. Further, the commonalities and variabilities are mapped on the available mechanisms. Our approach differs from this work because we rely on a decision model that is code-centric and more fine-grained than the domain analysis solution that it proposes, allowing SPL developers to choose more easily which mechanism to use when handling variabilities at the source code level. Moreover, our decision model make the task of defining refactorings easier, which means that the implementation of code variability might migrate from one mechanism to another.

Another work [47] has applied an empirical study consisting of activities performed to systematically improve the design and implementation of an existing software component in order to reuse it in a SPL. The component investigated is the Image Memory Handler (IMH) which is used in current products of office appliances such as copier machines, printers, and multi-functional peripherals. The approach followed to refactor the existing IMH software deals with the component documentation initially. Such documentation encompasses both design and variabilities of the component. Afterwards, an improvement of the design as well as of the implementation is done with respect to the reusability and maintainability of the component. Similarly to our work, to achieve this activity, all "#if" and "#ifdef" macros were analyzed to discover relevant variabilities and candidates to be reused throughout the products. They also made clone detection and restructured existing header files to get a clear SoC. The only mechanism used to restructure the component was **Conditional Compilation**, which is known by not achieving a complete SoC, since variabilities remain tangled to the core assets.

A recent work [18] have proposed an automated approach that identifies aspect candidates in code and infers pointcuts expressions for these aspects. Such approach mines for aspect candidates, identifies the join points for the aspect candidates, clusters the join points, and infers an effective pointcut expression for each cluster of join points. Moreover, they have implemented a tool for supporting the proposed approach. Such tool consists of four main components: the aspect mining module, join point identifier, clustering module, and inference module. As mentioned, the approach is powerful for encountering effective pointcut expressions, including AspectJ wildcards. For example, suppose that the tool found a crosscutting concern at the beginning of both *promptNew* and *promptOpen* methods. This way, the tool identifies that they share the same prefix "prompt" and infers a pointcut expression like this: *before(): execution(* *.prompt*(..))*.

The approach used in this tool is very similar to the `FLiPRec`: the statements which are before and after the identified concern are analyzed. However, since these statements may be repeated more than once in the same method body, the work claims that it is difficult to capture a pointcut. Indeed, we disagree because such task is not always difficult when considering the **Tracematches** mechanism (differently of [18] we have used this mechanism to address this problem in our tool). Besides, `FLiPRec` searches for cloned code, whereas [18] searches for crosscutting concerns.

7.3 OPEN ISSUES AND LIMITATIONS

Our decision model is limited because of several reasons:

- As mentioned several times in this work, the decision model proposed is code-centric and fine-grained. The main disadvantage of having such characteristics is that the model seems to be only useful when the SPL is already implemented[2], which means that basically it serves only to the evolution phase of the SPL life cycle;

- Although we have analyzed a subset of our decision model in the J2ME mobile games domain in [33, 32], we must apply the decision model and the `FLiPRec` tool in others domains to have a more consistent evaluation of the provided benefits;

- There is a great variety of other mechanisms such as **Program Transformations**, **Feature Oriented Programming**, **Generics**, and many other design patterns [37] that we did not consider. Additionally, there are other feature types that we did not addressed, like positive, negative, multioptional (XOR), and mutually inclusive and exclusive. In addition, we have used only four criteria (`Modularity`, `Scalability`, `Source Code Size`, and `Time`). In the binding time context, we did not analyzed mechanisms at pre-compile and link times. Considering a restricted set of mechanisms, feature types, and criteria may also restrict our model and tool. For instance, if no valid trace was found by `FLiPRec` tool, it is possible that any mechanism is recommended. Further, consider two mechanisms that are similar in all four criteria analyzed in this work. Nevertheless, suppose that one is much better in `Performance` than the other. Since we did not study this criteria, we can not

[2]We need to investigate more deeply how useful the model is to structure test variabilities.

recommend the best one. Unfortunately, time restrictions caused these limitations.

- The metrics suite used in this work is limited since we do not calculate items like cohesion [39]. Also, the way we have calculated the coupling metric was quite simple, differently of works like [64]. In addition, such traditional calculus may be considered questionable. For example, suppose two coupled components: *A* and *B*. Putting an interface between them may allow developing *A* and *B* in parallel. Besides, one can change the component *A* for *A'* as long as *A'* respects the interface. Despite the real gains with respect to `Modularity`, tools like *Metrics* [11] and *Eclipse Metrics Plug-in* [8] indicate higher total system coupling when considering the interface. This way, an open issue is how to calculate coupling considering these gains to the `Modularity` criteria. The same situation occurred in Section 4.3.2. After modularizing the crosscutting feature, the total coupling increased ($CBC = 5$). Therefore, this metric may contradict the other ones like CDC, generating doubts when defining/evolving the model and affecting it negatively through a questionable recommendation.

- We did not study all feature types in all proposed locations because of (i) time restrictions; and (ii) because some feature types are not likely to occur at a determined location. For example, all variabilities found at the **Method Parameter** location were alternative. However, we need to explore more deeply this topic to discover how true this information is.

Likewise, the tool also has limitations:

- We did not consider all mechanisms used in the decision model, such as **Mixins**, **Dependency Injection**, and the **Decorator** design pattern;

- The input must be an *if-else* statement;

- The tool does not calculate the metrics used in the the decision model automatically.

7.4 FUTURE WORK

As future work, we intend to apply both the decision model and the `FLiPRec` tool in others SPLs of different domains to evaluate the model. Additionally, we intend to improve our

decision model with:

- more mechanisms such as **Program Transformations**, by using the JaTS/A-JaTS [27, 20] and XVCL [63] technologies. This approach will be useful for addressing variabilities almost at any location of the code, being important for a "location-based" model as we proposed in this work;
- more criteria like `Performance` and `Bytecode Size` (important for limited devices like mobile phones);
- more locations like defining attributes or constants.

Therefore, this work proposed a decision model for restructuring test variabilities in SPL at the source code level and a tool for supporting developers when doing this task. We believe that, when applying both decision model and tool, not well structured SPL variabilities may have benefits, such as improving the SPL design, reducing maintenance costs and increasing developer's productivity.

BIBLIOGRAPHY

[1] CaesarJ Project, December 2007. http://caesarj.org/.

[2] CCfinder Official Site, May 2007. http://www.ccfinder.net/.

[3] Getting started with AJDT, June 2007. http://www.eclipse.org/ajdt/gettingstarted.php.

[4] Spring Framework, December 2007. http://www.springframework.org/.

[5] Aspect Mining Tool, January 2008. http://www.cs.ubc.ca/labs/spl/projects/amt.html.

[6] CaesarJ Project - Mixin Composition, January 2008. http://caesarj.org/index.php/ProgrammingGuide/MixinComposition.

[7] Eclipse Java Development Tools, January 2008. http://www.eclipse.org/jdt/.

[8] Eclipse Metrics Plugin, January 2008. http://eclipse-metrics.sourceforge.net/.

[9] Eclipse.org Home, January 2008. http://www.eclipse.org/.

[10] JUnit, January 2008. http://www.junit.org/.

[11] Metrics, January 2008. http://metrics.sourceforge.net/.

[12] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 2nd edition, 2006.

[13] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 345–364, New York, NY, USA, 2005. ACM Press.

[14] Vander Alves. *Implementing Software Product Line Adoption Strategies*. PhD thesis, Federal University of Pernambuco, Recife, Brazil, March 2007.

[15] Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and Evolving Mobile Games Product Lines. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *Lecture Notes in Computer Science*, pages 70–81. Springer-Verlag, September 2005.

[16] Michalis Anastasopoulos and Cristina Gacek. Implementing Product Line Variabilities. In *Proceedings of the 2001 Symposium on Software Reusability (SSR'01)*, pages 109–117, New York, NY, USA, 2001. ACM Press.

[17] Michalis Anastasopoulos and Dirk Muthig. An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. In *Proceedings of the 8th International Conference on Software Reuse (ICSR'04)*, Lecture Notes in Computer Science, pages 141–156. Springer, July 2004.

[18] Prasanth Anbalagan and Tao Xie. Automated Inference of Pointcuts in Aspect-Oriented Refactoring. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pages 127–136, New York, NY, USA, 2007. ACM Press.

[19] Sven Apel and Don Batory. When to Use Features and Aspects? A Case Study. In *Proceedings of the 5th international conference on Generative Programming and Component Engineering (GPCE'06)*, pages 59–68, New York, NY, USA, 2006. ACM Press.

[20] Roberta Arcoverde, Sergio Soares, Patrícia Lustosa, and Paulo Borba. AJaTS: AspectJ Transformation System. In *Proceedings of the 1st Workshop on Refactoring Tools (WRT'07), in conjunction with the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, pages 35–36, New York, NY, USA, July 2007. ACM Press.

[21] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *Proceedings of the 4th*

International Conference on Aspect-Oriented Software Development (AOSD 2005), pages 87–98, New York, NY, USA, 2005. ACM Press.

[22] Carliss Y. Baldwin and Kim B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, March 2000.

[23] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[24] Gilad Bracha and William Cook. Mixin-Based Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA/ECOOP'90)*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[25] Magiel Bruntink, Arie van Deursen, Tom Tourwe, and Remco van Engelen. An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 200–209, Washington, DC, USA, 2004. IEEE Computer Society.

[26] Fernando Calheiros, Paulo Borba, Sérgio Soares, Vilmar Nepomuceno, and Vander Alves. Product Line Variability Refactoring Tool. In *Proceedings of the 1st Workshop on Refactoring Tools (WRT'07), in conjunction with the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, pages 33–34, New York, NY, USA, July 2007. ACM Press.

[27] Fernando Castor, Kellen Oliveira, Adeline Souza, Gustavo Santos, and Paulo Borba. JaTS: A Java Transformation System. In *Proceedings of the 15th Brazilian Symposium on Software Engineering (SBES'01)*, pages 374–379, October 2001.

[28] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison-Wesley Professional, 2006.

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

[30] James Coplien. *Multi-Paradigm Design*. PhD thesis, Vrije Universiteit Brussel, Etterbeek, Belgium, July 2000.

[31] Márcio de Medeiros Ribeiro, Marcos Dósea, Rodrigo Bonifácio, Alberto Costa Neto, Paulo Borba, and Sérgio Soares. Analyzing Class and Crosscutting Modularity with Design Structure Matrixes. In *Proceedings of the 21th Brazilian Symposium on Software Engineering (SBES'07)*, pages 167–181, October 2007.

[32] Márcio de Medeiros Ribeiro, Pedro Matos Jr., and Paulo Borba. A Decision Model for Implementing Product Line Variabilities. In *Proceedings of the 23rd ACM Symposium on Applied Computing (SAC'08)*, New York, NY, USA, March 2008. ACM Press. To appear.

[33] Márcio de Medeiros Ribeiro, Pedro Matos Jr., Paulo Borba, and Ivan Cardim. On the Modularity of Aspect-Oriented and Other Techniques for Implementing Product Lines Variabilities. In *Proceedings of the 1st Latin American Workshop on Aspect-Oriented Software Development (LA-WASP'07), in conjunction with the 21th Brazilian Symposium on Software Engineering (SBES'07)*, pages 119–130, October 2007.

[34] Márcos Dósea, Alberto Costa Neto, Paulo Borba, and Sérgio Soares. Specifying Design Rules in Aspect-Oriented Systems. In *Proceedings of the 1st Latin American Workshop on Aspect-Oriented Software Development (LA-WASP'07), in conjunction with the 21th Brazilian Symposium on Software Engineering (SBES'07)*, pages 67–78, October 2007.

[35] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.

[36] Martin Fowler. Inversion of Control Containers and the Dependency Injection Pattern. Technical report, http://martinfowler.com/articles/injection.html, January 2004.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, January 1995.

[38] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Refactoring Support Based on Code Clone Analysis. In *Proceedings of the 5th International Conference on Product Focused Software Process Improvement (PROFES'04)*, pages 220–233, 2004.

[39] Baowen Xu Jianjun Zhao. Measuring Aspect Cohesion. In *Proceedings of the Fundamental Approaches to Software Engineering (FASE'2004)*, 2004.

[40] Ralph E. Johnson. Documenting Frameworks Using Patterns. In *Proceedings of the 7th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'92)*, pages 63–76, 1992.

[41] Pedro Matos Jr. Analyzing techniques for implementing product line variabilities. Master's thesis, Federal University of Pernambuco, Recife, Brazil, 2008. To be finished.

[42] Kyo-Chul Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[43] Luiz Kawakami, André Knabben, Douglas Rechia, Denise Bastos, Otavio Pereira, Ricardo Silva, and Luiz Santos. An Object-Oriented Framework for Improving Software Reuse on Automated Testing of Mobile Phones. In *Proceedings of Testing of Software and Communicating Systems - 19th IFIP TC6/WG6.1 International Conference (TestCom'07), 7th International Workshop (FATES'07)*, pages 199–211. Springer, June 2007.

[44] Luiz Kawakami, André Knabben, Douglas Rechia, Denise Bastos, Otavio Pereira, Ricardo Silva, and Luiz Santos. A Test Automation Framework for Mobile Phones. In *Proceedings of the 8th IEEE Latin American Test Workshop*, Washington, DC, USA, March 2007. IEEE Computer Society. To appear.

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

[46] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect–Oriented Programming. In *Proceedings of European Conference on Object–Oriented Programming (ECOOP'97)*, LNCS 1241, pages 220–242, 1997.

[47] Ronny Kolb, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. A Case Study in Refactoring a Legacy Component for Reuse in a Product Line. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 369–378, Washington, DC, USA, 2005. IEEE Computer Society.

[48] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[49] Ramnivas Laddad. *Aspect-Oriented Refactoring*. Addison-Wesley Professional, 2006.

[50] Doug Lea. *Concurrent Programming in Java*. Addison–Wesley, 2nd edition, 1999.

[51] Cristina Videira Lopes and Sushil Krishna Bajracharya. An Analysis of Modularity in Aspect-Oriented Design. In *LNCS Transactions on Aspect-Oriented Software Development I*, pages 1–35. Springer, 2006.

[52] John McGregor. Testing a Software Product Line. Technical Report CMU/SEI-2001-TR-022, Carnegie Mellon University, 2001.

[53] Miguel Monteiro and João Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *Proceedings of the 4th international Conference on Aspect-Oriented Software Development (AOSD 2005)*, pages 111–122, New York, NY, USA, September 2005. ACM Press.

[54] Alberto Costa Neto, Márcio de Medeiros Ribeiro, Marcos Dósea, Rodrigo Bonifácio, Paulo Borba, and Sérgio Soares. Semantic Dependencies and Modularity of Aspect-Oriented Software. In *Proceedings of 1st Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07), in conjunction with the 29th International Conference on Software Engineering (ICSE'07)*, page 11, Washington, DC, USA, 2007. IEEE Computer Society.

[55] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[56] Thomas Patzke and Dirk Muthig. Product Line Implementation Technologies. Technical Report 057.02/E, Fraunhofer Institut Experimentelles Software Engineering, October 2002.

[57] Christoph Pohl, Andreas Rummler, Vaidas Gasiunas, Neil Loughran, Hugo Arboleda, Fabricio Fernandes, Jacques Noye, Angel Nunes, Robin Passama, Jean-Claude Royer, and Mario Sudholt. Survey of existing implementation techniques with respect to their support for the practices currently in use at industrial partners, July 2007.

[58] Klaus Pohl, Gunter Bockle, and Frank J. van der Linden. *Software Product Line Engineering*. Springer, 2005.

[59] Cláudio Santánna, Alessandro Garcia, Christina Chavez, Carlos Lucena, and Arndt von Staa. On the Reuse and Maintenance of Aspect-Oriented Software: A Assessment Framework. In *Proceedings of the 17th Brazilian Symposium on Software Engineering (SBES'03)*, pages 19–34, October 2003.

[60] Antti Tirila. Variability Enabling Techniques for Software Product Lines. Master's thesis, Tampere University of Technology, Tampere, Finland, September 2002.

[61] Arie van Deursen, Marius Marin, and Leon Moonen. Aspect Mining and Refactoring. In *Proceedings of the 1st International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE'03)*, November 2003.

[62] Remco van Engelen. On the Use of Clone Detection for Identifying Crosscutting Concern Code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[63] Hongyu Zhang and Stan Jarzabek. XVCL: A Mechanism for Handling Variants in Software Product Lines. *Science of Computer Programming*, 53(3):381–407, 2004.

[64] Jianjun Zhao. Measuring Coupling in Aspect-Oriented Systems. In *Proceedings of the 10th International Software Metrics Symposium (Metrics'04)*, 2004.

This volume has been typeset in LaTeXwith the UFPEThesis class (www.cin.ufpe.br/~paguso/ufpethesis).