

# Improving Guidance when Restructuring Variabilities in Software Product Lines

Márcio Ribeiro    Paulo Borba  
Federal University of Pernambuco  
Recife, Brazil  
{mmr3, phmb}@cin.ufpe.br

## Abstract

*Software Product Lines (SPLs) encompass a family of software systems developed from reusable assets. One issue during SPL maintenance is the decision about which mechanism should be used to restructure variabilities aiming at improving the modularity of the SPL artifacts. Due to the great variety of mechanisms (Inheritance, Configuration Files, Aspect-Oriented Programming), selecting the incorrect ones may produce negative effects on the cost to evolve the SPL. To reduce this problem, we propose a decision model to help developers to choose mechanisms to restructure variabilities in SPLs. The domain analyzed by this work consists of test scripts. We also developed a prototype tool to support developers by recommending mechanisms according to the decision model. Using our model and tool may improve the tests variabilities' modularity and remove bad smells such as cloned code.*

## 1 Introduction

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 [17]. On the other hand, implementation activities become more complex because they also have to realize variabilities [15].

In this context, reasoning about how to combine both core assets and product variabilities is a challenging task [4]. In other words, the challenge consists of understanding the available mechanisms (such as Inheritance, Configuration Files, Aspect-Oriented Programming [12], and so forth) for realizing variability and knowing which of them fits best for a given variability [15]. Previous work [13, 3] 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 [8]. 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.

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, we have defined a Decision Model to help developers on the task of choosing mechanisms to restructure variabilities in SPLs. To construct our decision model, we analyzed variabilities found in Motorola mobile phone test scripts. 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.

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 SPL variabilities implemented using *if-else* statements.

The main contributions of our work are:

- A decision model for improving guidance of developers in SPL maintenance. Existing models [4, 15] consider a high level approach that rely basically on feature types. On the other hand, we provide a decision model which is code-centric and more fine-grained. Because we consider not only the feature type, but also the exactly variability location at the source code and some criteria, our recommendations may be more pre-

cise. In addition, being code-centric makes easier the task of understanding how to apply the recommended mechanism through Fowler-like refactorings [9] to restructure variabilities.

- A prototype tool to recommend automatically mechanisms for restructuring SPL variabilities, improving the developer’s productivity when evolving them.

## 2 Motivating Examples

In this section we provide some motivating examples extracted from real Motorola mobile phone test scripts. The SPL variabilities are handled by using *if-else* statements so that there is no modularity in the tests.

### 2.1 Example 1

The first variability kind analyzed in this work is at the **End of Method Body**. Our example of this test variability (illustrated in Figure 1) 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*. Notice that 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.

```
public class TC_065 extends TestCase {
    public void preconditions() {}
    public void procedures() {
        ...
        if (has(PhoneCapability.TRANSFLASH)) {
            ...
        }
        if (has(PhoneCapability.BLUETOOTH)) {
            ...
        }
    }
    public void postconditions() {}
}
```

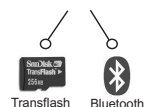


Figure 1. End of Method Body.

Since the *if* statements are tangled in this particular test, we can not reuse the *transflash* and *bluetooth* code throughout other tests. Obviously, cloning the code in other tests is not worthwhile to modularity. Due to the great variety of available mechanisms, deciding which one to use in this particular case is not straightforward, being time-consuming. In addition, such decision may be error-prone, producing negative effects on the product line, like cloned code and an explosion in the number of classes, as we will see later in Section 3.1.

### 2.2 Example 2

The second example of variability kind showed here occurs at the **Middle of Method Body**. Figure 2 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.

```
public class TC_064 extends TestCase {
    public void procedures() {
        ...
        navigationTk.launchApp(PhoneApplication.BROWSER);
        browserTk.takeWebPageScreenshot();
        if (has(PhoneFunctionality.CAN_STORE_WEBPAGE)) {
            block("Store web page");
            browserTk.storeWebPage();
        }
        ...
        browserTk.goToURL("google.com");
        browserTk.takeWebPageScreenshot();
        if (has(PhoneFunctionality.CAN_STORE_WEBPAGE)) {
            block("Store web page");
            browserTk.storeWebPage();
        }
        ...
        browserTk.goToURL("gmail.com");
        browserTk.takeWebPageScreenshot();
        if (has(PhoneFunctionality.CAN_STORE_WEBPAGE)) {
            block("Store web page");
            browserTk.storeWebPage();
        }
        ...
        browserTk.takeWebPageScreenshot();
        ...
    }
}
```




Figure 2. Middle of Method Body.

Notice that the code to store the web page is cloned throughout the test case (and throughout many other tests, not shown in Figure 2). There is no reuse, which means that maintaining this code may be time consuming and error-prone. Also, due to the hundreds of tests available at Motorola, finding out where the cloned code is may affect the developers productivity. Again, deciding which mechanism to use to deal with this cloning may be a difficult task.

### 2.3 Summary

In summary, two problems have arisen: how to decide which mechanism should be used to restructure a given variability; and how to improve developers productivity by reducing the time spent when finding cloned code and at the same time recommending automatically mechanisms able to deal with such cloning. This way, in order to improve the guidance of developers to restructure SPL variabilities, we present a decision model (Section 3) and a tool (Section 4).

## 3 Towards a Decision Model

As mentioned in Section 1, the SPL approach contains, besides common, variable elements. Hence, implementation activities become more complex because they have to

realize variabilities as well [15]. In particular, reasoning about how to combine both core assets and product variabilities is a challenging task [4].

Many mechanisms can be used to combine these artifacts. They range from simple ones like Inheritance to more complex ones like Aspect-Oriented Programming (AOP) [12]. Due to this great variety, selecting the incorrect ones may produce negative effects on the cost to evolve the SPL [8]. As showed in Section 2, 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, we have defined a Decision Model to help developers on the task of choosing mechanisms to restructure variabilities in SPL. To construct our decision model, we analyzed variabilities found in Motorola mobile phone test scripts like the ones presented in Section 2. In this way, the decision model is based on variabilities handled by using *if-else* statements. The model aims at suggesting mechanisms so that variabilities can be modularized.

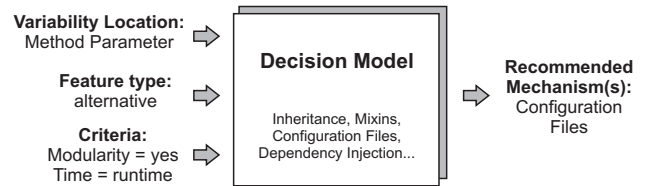
In what follows, we explain the inputs and the output of our model. Since it is code-centric, the first input is the exact **location** of the variability at the source code. For example, the variability of Section 2.2 appears at the middle of a method body. The second input consists of the **feature type** (variability type). In this work, we have considered the optional<sup>1</sup> and alternative<sup>2</sup> 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 [16]. In this way, the Modularity criteria represents the separation of concerns that a mechanism provides. 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 inheritance approach increased 9% when compared to the original *if-else* 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. Possible values: *yes* or *no*;
- **Time:** represents the binding time provided by the mechanism. Possible values: *compile-time* or *runtime*.

<sup>1</sup>As showed in Section 2, the feature may be included or not.

<sup>2</sup>The feature replaces another feature when included, like a XOR.

The decision model should be able to **recommend mechanisms** based on the aforementioned inputs. The mechanisms suggested by the model represent its output. Figure 3 outlines an application of our decision model.



**Figure 3. Application of the Decision Model.**

Notice 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, for instance.

The methodology to construct the decision model is explained as follows. For each analyzed test variability, we restructured it using different mechanisms. Differently of existing models [4, 15], our decision model is based not only on qualitative but also on quantitative studies. For this reason, we calculate some software metrics of Separation of Concerns (SoC), size, and coupling 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 metrics used in this work are presented in Table 1. They are an important tool to guide our decision model, pointing out the strengths and weaknesses of each mechanism, being useful to indicate the appropriate mechanism to implement a given variability. More details of these metrics can be found elsewhere [18]. 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.

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

### 3.1 End of Method Body

Since the example of Figure 1 consists of 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:

Metric	Definition
Concern Diffusion over Components (CDC)	Counts the number of components that implements a concern
Concern Diffusion over Lines of Code (CDLOC)	Counts the number of lines of code responsible for implementing a concern
Number of Concerns per Component (NCC)	For each component, counts the number of concerns it implements
Lines of Code (LOC)	Counts the number of source code lines (ignoring comments and blank lines)
Vocabulary Size (VS)	Counts the number of components like classes, aspects, and configuration files
Coupling Between Components (CBC)	The total coupling among the components
Depth of Inheritance Tree (DIT)	Counts how far down the inheritance hierarchy a class

**Table 1. Software Metrics used to compare the mechanisms implementations.**

- **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 [10]. To compose both features, clients must instantiate the composition of the *transflash* and *bluetooth* classes;
- **Mixins:** analogous to Inheritance. A mixin is an abstract subclass that may be applied to different superclasses to create a related family of modified classes. It composes related classes like a multiple inheritance. Details about mixins can be found elsewhere [7];
- **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.

Table 2 shows the metrics for the **End of Method Body** mechanisms. For these mechanisms, consider that *DIT* represents the maximum *DIT* value found among all components. Figure 4(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 2, 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 other 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

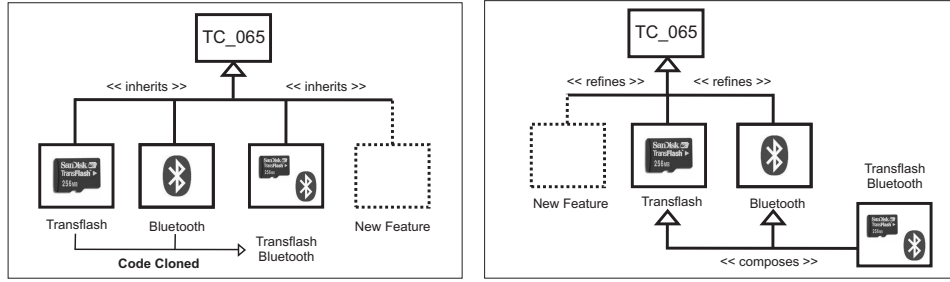
Figure 4(b), although **Mixins** solves the cloned code problem, the *Scalability* problem remains when considering at least one more feature.

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), 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(c)). Because of the last case, the *DIT* metric would be 3 for the **Mixins** mechanism. According to Table 2, 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(c)). 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 1) 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 aspect and in the *Extra Aspect* ( $CDC_{Infrared} = 2$ ).

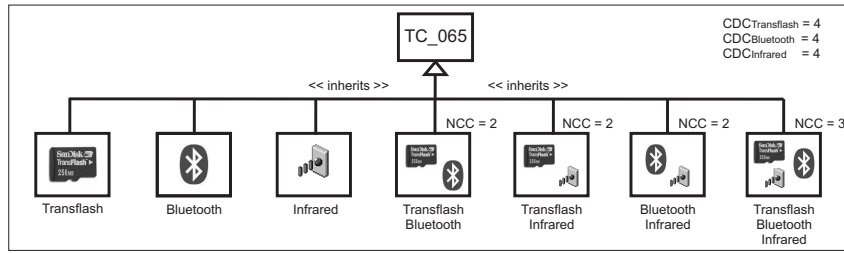
**Listing 1. Declare Precedence Extra Aspect.**

```
public aspect ExtraAspect {
  declare precedence : TransflashAspect , BluetoothAspect ; }
```



(a) End of Method Body using Inheritance.

(b) End of Method Body using Mixins.



(c) Three optional features using Inheritance. The impact on Scalability is analogous for Mixins.

**Figure 4. End of Method Body: Inheritance x Mixins.**

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 an extra infrastructural code: an interface and an abstract class. 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 *Scalability* problem.

### 3.2 Middle of Method Body

The variability presented in Figure 2 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 four places, which is incorrect (there are four *takeWebPageScreenshot* calls, but only three *if* statements).

Since calls to methods such as *launchApp*, *goToURL*, and *takeWebPageScreenshot* often happen more than once in the test scripts, **Tracematches** [2] may be an useful mechanism to address these variabilities. This way, in-

stead of considering only the *takeWebPageScreenshot* call, we can use tracematches to create a regular expression, as showed in Listing 2. 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.

According to the metrics showed in Table 3, 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 2 needs changing.

**Listing 2. Store Web Page Tracematch.**

```

tracematch() {
  sym launchApp after : call(* *.launchApp(...));
  sym goToURL after : call(* *.goToURL(...));
  sym takeWebPageScreen after :
    call(* *.takeWebPageScreenshot(...));
  (launchApp | goToURL) takeWebPageScreen {
    block("Store_web_page");
    browserTk.storeWebPage();
  }
}

```

However, defining **Tracematches** produces a strong coupling between the *tracematch* and the *TC\_064*. Any unan-

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	Bluetooth and Transflash (Class)	2	-	-	-
	Bluetooth and Transflash (Mixin)	-	-	2	-
	Bluetooth and Transflash (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 2. End of Method Body metrics.**

anticipated 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 3. Middle of Method Body metrics.**

This way, we claim in this work that defining design rules [14] or crosscutting programming interfaces (XPIs) [11] should be used to remove the dependency between the *tracematch* and *TC\_064*.

### 3.3 Summary: Decision Model

After analyzing each variability by its exact location at the source code and considering its feature type, we finally present our decision model. The assumption for using the model is that the variabilities where it will be applied use *if-else* statements to handle their variabilities or analogous conditionals, such as “*#ifdef*” and “*#elif*”, broadly used in many product lines.

Although our model was defined based on some examples of test variabilities, we believe that the 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. Also, 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 **recommends** some mechanisms to restructure 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 [9]. Eliminating them means avoiding these bugs when evolving the SPL and makes the code more modular;
- **Independent evolution:** because the features may be modularized using the mechanisms recommended by the decision model, developers might work in parallel when maintaining the tests;
- **Productivity increasing:** time consuming and error-prone tasks like evolving cloned code are avoided. Also, due to the provided modularization, developers may work in parallel, reducing the time-to-market.

Figure 5 illustrates our decision model<sup>3</sup>. Notice that the exact variability location at the source code, the feature type, and some criteria are the model’s inputs.

## 4 Supporting Developers

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 increases the developer’s effort.

<sup>3</sup>Due to space restrictions, we did not detail all locations of the model.

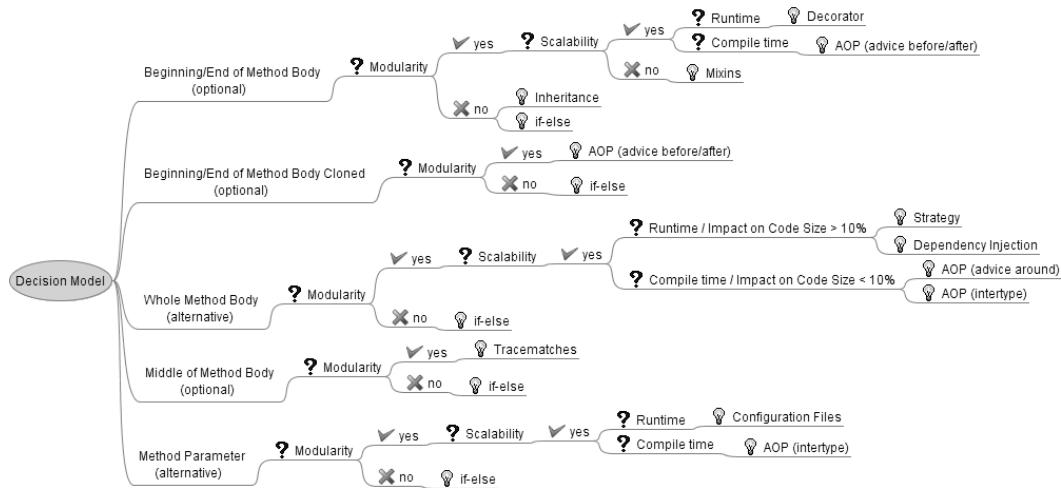


Figure 5. Decision Model.

To minimize such difficulties, we present here a prototype tool that we have developed based on the Eclipse Java Development Tool (JDT) [1]. The tool is able to **recommend mechanisms** aiming at restructuring SPL variabilities. 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 3.3.

To get the recommendation, the user must select the desired *if-else* statement. Afterwards, the tool searches for clones of the selected code and starts the process of recommending mechanisms. In this paper, we provide two examples of using our tool. For more details, some videos of the tool are available in <http://www.cin.ufpe.br/~mmr3/recommender-tool/>.

#### 4.1 Variability Cloned

Listing 3 illustrates two test cases: *TC\_045* and *TC\_063*. According to our decision model showed in Figure 5, the variability is at the **Beginning of Method Body Cloned**: the *if* statement to setup the *Transflash* preconditions is cloned. After selecting the *if* statement in *TC\_045* and clicking on the recommendation button, the tool searches for clones in all available *.java* files. Afterwards, it recommends a mechanism, which in this case was **AOP**.

Listing 3. Cloning in two test cases.

```
public class TC_045 extends TestCase {
    public void preconditions() {
        if (has(PhoneCapability.TRANSFLASH)) {
            phone.setBits(PhoneBits.TRANSFLASH, true);
            phone.flushBits();
        }
        ...
    }
}
```

```
public class TC_063 extends TestCase {
    public void preconditions() {
        if (has(PhoneCapability.TRANSFLASH)) {
            phone.setBits(PhoneBits.TRANSFLASH, true);
            phone.flushBits();
        }
        ...
    }
}
```

Therefore, our tool may improve the productivity by reducing painful tasks like finding cloned variabilities in hundreds of classes. At the same time, the tool recommends a mechanism able to deal with the cloning, which avoids future errors and helps on modularizing the complete feature.

#### 4.2 Searching for a Valid Tracematch

Figure 6 shows an *if* statement at the middle of a certain method body. As mentioned in Section 3.2, **Tracematches** can be useful to restructure this variability. To avoid problems like introducing the code into wrong places, the tool must guarantee that there is a unique tracematch. Otherwise, the **Tracematches** mechanism is not recommended.

Figure 6 also summarizes how the tool searches for a unique tracematch. The first upper neighbor is considered: *takeWebPageScreenshot*. Next, it 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: it verifies if the next trace (*goToURL* followed by *takeWebPageScreenshot*) exists (*Step 2*). Because such trace already exists, it considers another neighbor (*Step 3*). Since the trace of the *Step 3* is unique, the tool recommends this trace.



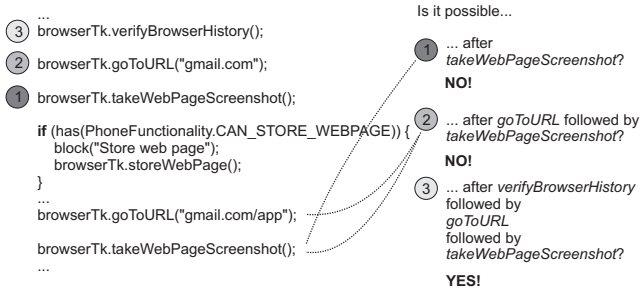


Figure 6. Searching for a unique tracematch.

The biggest test case analyzed has 424LOC. In this case, without tool support, finding valid traces would be much more difficult, being error-prone and impacting directly on the developers productivity. When considering the smallest test case analyzed (42LOC), a tool could not be necessary.

## 5 Evaluation: Comparing Recommendations

Existing models [4, 15] are neither code-centric nor fine-grained. Basically, they consider a high level approach that rely on feature types: given a feature type, they say whether a mechanism is able or not to implement that feature type. Since our model requires not only the feature type, in this section we evaluate how useful our three inputs are.

### 5.1 Location and Feature Type

The models presented in [4, 15] claim that is possible to implement (i) optional features by using **AOP**; and (ii) alternative features by using **Inheritance** or design patterns [10] like the **Strategy**. Notice that our model is in accordance with such information (Figure 5). Despite very important, the feature type is not always enough to recommend mechanisms, as showed in the following scenarios.

**Scenario 1:** Consider the two examples of optional features presented in Figures 1 and 2. Based only on the nature of the features, if we decided to use **AOP** according to [4, 15], we would have problems in the example of Figure 2, since it is not always easy to create valid pointcuts at the middle of method bodies by using pure aspects. A refactoring should be performed before applying **AOP**.

**Scenario 2:** Figure 7 illustrates an alternative feature. Either *Opera* or a proprietary *Motorola* browser is selected in the product line instance. Although [4, 15] say that **Inheritance** is able to implement alternative features, we can see that using **Inheritance** in Figure 7 would not be straightforward without previous refactorings [9]. In other words, it is not clear how to apply **Inheritance** is this case. Instead, we recommend **Configuration Files** or **AOP** intertypes due to the fine-grained nature of the variability, which

in this case is at the **Method Parameter** (see Figure 5). No previous refactoring is needed in our recommendations.

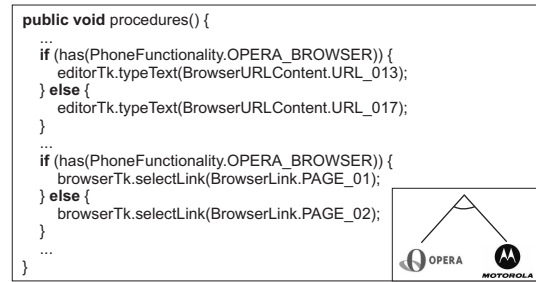


Figure 7. Method Parameter.

Hence, the location of the variability can also be important when deciding which mechanism to use. Thereby, to evaluate if our locations are comprehensive, we analyzed all variabilities of 50 Motorola mobile test scripts and calculated the occurrence of each location. The locations proposed here covered 71% of the variabilities analyzed (the results are detailed in Table 4). The *Not Analyzed* row represents locations that our model does not cover: (i) nested *if-else* statements; and (ii) some locations with other feature type (for example: *Middle of Method Body - Alternative*).

Location	Feature Type	%
Middle of Method Body	Optional	37%
Beginning/End of Method Body	Optional	11%
Beginning/End of Method Body Cloned	Optional	5%
Method Parameter	Alternative	16%
Whole Method Body	Alternative	2%
Not Analyzed	Both	29%

Table 4. Occurrences of each location.

### 5.2 Criteria

The results of the metrics used in this paper can help even more the developer's decision. For example, suppose a SPL of mobile software with limited resources. Although **Decorator** and **AOP** are suitable for restructuring the variabilities of Figure 1 with respect to *Modularity* and *Scalability*, the difference in metrics like *LOC*, *CDLOC*, and *VS* might be crucial to the developer's choice. Due to the limited resources, the *Source Code Size* may be the most important criteria so that he would prefer the **AOP** mechanism instead of **Decorator**. *Modularity* and *Scalability* are considered in [4].

### 5.3 Better Recommendations

We definitely claim that using only the feature type may be insufficient to recommend mechanisms. The examples



showed that the recommendations may change drastically when including the location and/or criteria. We can obtain better recommendations through the three inputs because:

- Based on both location and feature type makes easier the task of understanding how to apply the recommended mechanism. Besides, due to a code-centric and fine-grained recommendation, no refactorings are needed to prepare the code to receive the recommended mechanism. Therefore, the task of applying such mechanism may be realized faster.
- Using the criteria may avoid future problems. For example: consider that only the *Transflash* feature exists in Figure 1. If the developer is aware about new features, he would consider the *Scalability* criteria. Considering only the feature type, **Inheritance** might be recommended, generating cloned code and an unclear and complex hierarchy when dealing with the new features in the future. On the other hand, such problems would be avoided by recommendations that take the *Scalability* criteria into consideration.

Nevertheless, the main disadvantage of being code-centric and fine-grained is that the model seems to be only useful when the SPL is already implemented, satisfying only the evolution phase of the SPL life cycle.

## 6 Related Work

The first related work discussed here [4] examines 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. Also, 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 our test cases are written in Java, we focused on this language only.

Another related approach [15] 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 and paradigms, a framework to

compare mechanisms was constructed with respect to features types, similar to [4]. 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.

The next work [3] 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 variabilities from Java classes to AspectJ aspects. Although [3] provides a framework to compare variability implementation mechanisms, the method relies only on **AOP** refactorings. Such refactorings helped us to learn how to restructure some test variabilities (using **AOP**) after analyzing similar variabilities found in mobile games. In contrast, our work proposes a model that uses different mechanisms (not only **AOP**), since previous work [4, 15, 6] showed that **AOP** is not always suitable for addressing SPL variabilities. However, we did not provide refactorings, only recommendations.

A recent work [5] has proposed an approach that identifies aspect candidates in code and infers pointcuts expressions for these aspects by using clusters of join points. The proposed tool is very 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. Thus, the tool identifies that they share the same prefix “prompt” and infers a pointcut like this: *before(): execution(\* \*.prompt\*(..))*. The approach used is very similar to our tool: the statements before and after the identified concern are analyzed. However, since these statements may be repeated in the same method body, the work claims that it is difficult to capture a pointcut. Indeed, we showed that such task is not always difficult when considering the **Trace-matches** mechanism. Besides, our tool searches for cloned code, whereas [5] searches for crosscutting concerns.

## 7 Concluding Remarks

This paper presented a decision model to guide developers when restructuring variabilities in SPL. Differently of existing ones, our model is code-centric and fine-grained. Further, to compare the mechanisms, we used not only qualitative, but also quantitative studies through metrics. Besides, we developed a prototype tool to support developers when evolving variabilities in SPLs. It can recommend mechanisms to restructure variabilities faster and precisely according to the decision model, reducing developer’s effort and increasing their productivity.

Although we have discussed throughout the paper that our approach targets *if-else* statements encountered in test scripts, applying our work may also be useful in other domains/scenarios: (i) analogous variabilities found in the test scripts were also encountered in a different domain: J2ME Games [8]. Given the different natures of the domains in which the variabilities were found, we believe that they are likely to occur in other domains as well. Thus, it seems that our model can address variabilities found in other domains beyond testing; (ii) since the **Conditional Compilation** mechanism uses analogous conditionals, like “`#ifdef`” and “`#elif`”, our approach is easily applied in product lines that use this mechanism to structure their variabilities;

As we showed in the evaluation of this work, the three inputs of our model may be useful when recommending mechanisms. For example, the criteria might be useful to analyze the `Source Code Size` of a mechanism, helping developers when working on limited devices. Besides, we showed that the proposed locations of our decision model cover 71% of all analyzed test script variabilities.

On the other hand, our approach has some limitations:

- There is a great variety of other mechanisms such as **Program Transformations** and **Generics**, and many other design patterns [10] that we did not consider. Additionally, there are other feature types that we did not address, like positive, negative, and multioptional (OR). In addition, we have used only four criteria.
- We did not study all feature types in all proposed locations because some of them are not likely to occur at a determined location. For example, all variabilities found at the **Method Parameter** were alternative. However, we need to analyze more deeply this topic.

We intend to address such limitations as future work.

## References

- [1] Eclipse Java Development Tools, January 2008. <http://www.eclipse.org/jdt/>.
- [2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. 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.
- [3] V. Alves, P. M. Jr., L. Cole, P. Borba, and G. Ramalho. Extracting and Evolving Mobile Games Product Lines. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, volume 3714 of LNCS, pages 70–81. Springer-Verlag, September 2005.
- [4] M. Anastasopoulos and C. 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.
- [5] P. Anbalagan and T. 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.
- [6] S. Apel and D. 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.
- [7] G. Bracha and W. 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.
- [8] M. de Medeiros Ribeiro, P. M. Jr., P. Borba, and I. 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)*, pages 119–130, October 2007.
- [9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, January 1995.
- [11] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular Software Design with Cross-cutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.
- [12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, pages 220–242, 1997.
- [13] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A Case Study in Refactoring a Legacy Component for Reuse in a Product Line. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, pages 369–378, Washington, DC, USA, 2005. IEEE Computer Society.
- [14] C. V. Lopes and S. K. Bajracharya. An Analysis of Modularity in Aspect-Oriented Design. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 15–26, New York, NY, USA, March 2005. ACM Press.
- [15] T. Patzke and D. Muthig. Product Line Implementation Technologies. Technical Report 057.02/E, Fraunhofer Institut Experimentelles Software Engineering, October 2002.
- [16] C. Pohl, A. Rummler, V. Gasiunas, N. Loughran, H. Arboleda, F. Fernandes, J. Noye, A. Nunes, R. Passama, J.-C. Royer, and M. Sudholt. Survey of existing implementation techniques with respect to their support for the practices currently in use at industrial partners, July 2007.
- [17] K. Pohl, G. Bockle, and F. J. van der Linden. *Software Product Line Engineering*. Springer, 2005.
- [18] C. Santánna, A. Garcia, C. Chavez, C. Lucena, and A. 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.