Recommending Refactorings when Restructuring Variabilities in Software Product Lines

Márcio Ribeiro Paulo Borba

Federal University of Pernambuco, Informatics Center, Recife, Brazil

mmr3@cin.ufpe.br/phmb@cin.ufpe.br

Abstract

When restructuring variabilities in Software Product Lines (SPL), due to the great variety of existing mechanisms - such as Inheritance, Configuration Files, Aspect-Oriented Programming etc, developers may spend time and effort to decide which mechanism to use and which refactorings should be applied. To help on this task, we propose in this paper a tool capable of recommending refactorings based on some mechanisms. By applying the recommendations of our tool, bad smells such as cloned code may be removed and the modularity of the features is improved.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques

General Terms Design

Keywords Refactoring, Software Product Line, Modularity

1. Introduction

Software Product Lines encompass a family of software-intensive systems developed from reusable assets (known as core assets). One issue during SPL maintenance is the decision about which mechanism should be used to restructure variabilities. In other words, the challenge consists of understanding the available mechanisms (such as Inheritance, Configuration Files, Aspect-Oriented Programming (AOP), and so forth) for realizing variability and knowing which of them fits best for a given variability [8, 3].

Due to the great variety of available mechanisms, selecting an incorrect mechanism may produce negative effects on the cost to maintain the SPL [10, 9]. For example, cloned code and concerns not modularized may appear, affecting the independent evolution of SPL artifacts, increasing developer's effort, and consequently decreasing productivity when evolving the SPL.

In this context, in order to improve the developer's productivity when restructuring product line variabilities, a tool to avoid repetitive tasks and recommend refactorings may be useful. For example, it may reduce developer's effort, avoiding time consuming and error-prone tasks like finding where an existing cloned variability is. Afterwards, the tool may recommend a suitable refactoring to remove the cloned code. This way, we present in this paper a prototype tool for supporting developers when dealing with tasks like this. Nowadays, the tool is capable of recommending refactorings to restructure a given variability (cloned or not).

We describe our tool through variabilities found in proprietary Motorola mobile phone test scripts. The test variabilities used are handled by using *if-else* statements. For example, the *if* body is executed to test product *A*, whereas the *else* body tests product *B*. Some of these variabilities are scattered or even cloned throughout many test scripts, being difficult to identify and refactor them adequately without tool support.

The main contribution of this paper is to describe how we developed the tool and how it can help developers when restructuring variabilities in product lines through refactorings recommendations.

2. Tool overview

In this section we provide an overview of our tool. Such a tool is an extension of `FLiPEx` [6]. `FLiPEx` is a refactoring tool for extracting product variabilities from Java classes to AspectJ aspects [2]. It is based on the Eclipse plug-in platform [7] and uses the Eclipse infrastructure to perform source code refactorings that extract product variabilities.

The `FLiPEx` tool extracts product variabilities only to aspects. However, previous researchers [8, 3, 5, 10] showed that the AOP mechanism is not always the best one to implement SPL variabilities. This motivated us to write a `FLiPEx` extension to recommend refactorings not only in AOP, but also in another mechanisms such as Inheritance[1], Config-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WRT'08, October 19, 2008, Nashville, Tennessee, USA.

Copyright © 2008 ACM 978-1-60558-339-6/08/10. . . $5.00

[1] Due to space restrictions, we do not cover this mechanism in this paper.

uration Files, and Tracematches [1]. Our tool aims at recommending refactorings to restructure variabilities implemented using *if-else* statements in existing SPLs. It is important to note that the tool is a prototype and does not realize refactorings.

In order to get a recommendation in our tool, the user must first select the desired *if-else* statement. Afterwards, the tool starts the recommendation process, by searching 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 before the selected code, it means that such code is at the beginning of the method body.

Our recommendations are based on the *Neighbors* object. For example, if all found clones are at the beginning of their respective method's bodies, the tool recommends a refactoring based on Aspect-Oriented Programming through a before advice (obviously, according to some preconditions that were checked before recommending).

3. Recommending Refactorings Examples

Now, we provide some examples of refactoring recommendations provided by our tool. The examples consist of variabilities found in Motorola mobile phone test scripts. At the end of this section, we present some screen shots of the tool.

3.1 Variability Cloned

Figure 1 illustrates the *TC_003* test case and an optional feature called *Transflash*. *Transflash* is optional because some phones do not provide such a feature. If does, some additional steps must be performed to test the phone. In addition, Figure 1 shows that the *if* statement of the *Transflash* feature is cloned in the following tests: *TC_009*, *TC_019*, *TC_032*, and *TC_055*.

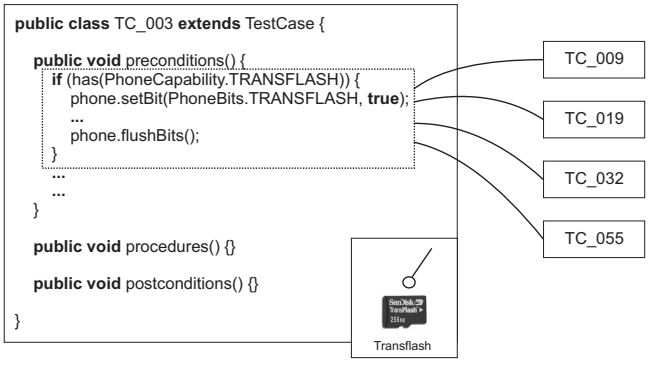

Figure 1. Beginning of Method Body Cloned.

Refactoring only part of the *Transflash* feature is not desired because it will remain not modularized. In order to remove the clones and modularize the whole feature, an initial 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 (there are hundreds of test scripts). Instead, by using our tool, after selecting the *if* statement in *TC_003* and clicking on the recommendation button, it will search for clones of the selected code automatically in all available *.java* files of an Eclipse project. Because the variability is cloned at the beginning of their respective methods, the tool recommend the AOP mechanism through a before advice.

The second example (Figure 2) shows two test cases. The *Dedicated Keys* optional feature is cloned in both tests. Notice that the method calls before the clones are different. After finding the clones, our tool will try to find a unique aspect pointcut to refactor such variability adequately. In this particular case, the tool will search for another call to the *goToApplication* method within the *TC_015.procedures* method body. The same happens to the *goToIdle* method call. The tool would recommend the AOP mechanism since the preconditions were checked: there is no additional calls to *goToApplication* within *TC_015.procedures*; and the same holds to *goToIdle* within *TC_016.procedures*.

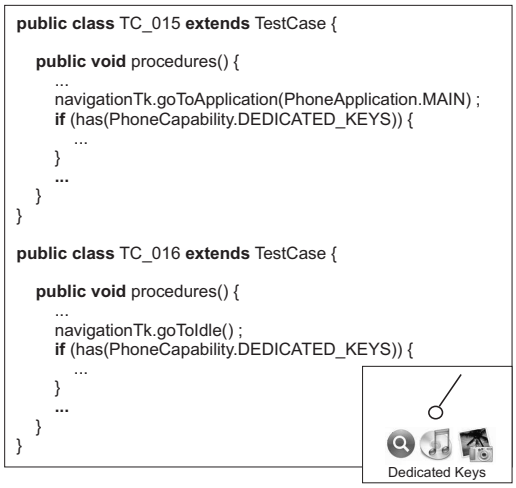

Figure 2. After Method Calls.

The third example of this section considers the *if* and *else* bodies. Figure 3 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 3: either *Opera* or a proprietary *Motorola* browser is selected in the product line instance).

In this context, our tool will search for clones within the *if* and *else* statements. Since only the method parameter value's are different, the tool recommends Configuration Files and intertype declarations of Aspect-Oriented to provide such values and remove the *if-else* statements. Consequently, the cloning within them is removed as well.

```
public class TC_107 extends TestCase {

   public void preconditions() {}

   public void procedures() {
      ...
      if (has(PhoneFunctionality.OPERA_BROWSER)) {
         browserTk.loadWebSession(Session.WEB_HTTP);
         browserTk.setWebSessionAsDefault(Session.WEB_HTTP);
         editorTk.typeText(BrowserURLContent.URL_013);
         browserTk.scrollAndSelectLink(BrowserLink.PAGE_036);
      } else {
         browserTk.loadWebSession(Session.WEB_WAP);
         browserTk.setWebSessionAsDefault(Session.WEB_WAP);
         editorTk.typeText(BrowserURLContent.URL_017);
         browserTk.scrollAndSelectLink(BrowserLink.PAGE_026);
      }
      ...
   }

   public void postconditions() {}

}
```

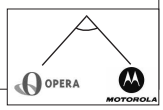

Figure 3. Method Parameter.

3.2 Searching for a Valid Tracematch

Figure 4 shows *if* statements cloned at the middle of the *procedures* body. Notice that the optional *Store Web Page* feature can not be refactored 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).

```
public class TC_064 extends TestCase {

   public void procedures() {
      ...
      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.goToURL("gmail.com/app");
      browserTk.takeWebPageScreenshot();
      if (has(PhoneFunctionality.CAN_STORE_WEBPAGE)) {
         block("Store web page");
         browserTk.storeWebPage();
      }
      ...
      browserTk.takeWebPageScreenshot();
      ...
   }

}
```

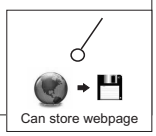

Figure 4. Middle of Method Body.

Since calls to methods such as *goToURL* and *takeWebPageScreenshot* often happen more than once in the Motorola test cases, Tracematches [1] 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 in Listing 1 (Line 7). Now, after any call to *goToURL* followed by a call to the *takeWebPageScreenshot* method, the tracematch will weave the *Store Web Page* code.

Listing 1. Store Web Page Tracematch.

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

The biggest test case analyzed in this work has 424LOC. In this case, without tool support, finding valid traces would be much more difficult, being error-prone and impacting directly on the developer's productivity. On the other hand, when considering the smallest test case analyzed (42LOC), a tool could not be necessary.

Figure 5 summarizes how our 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 to be used on the refactoring process.

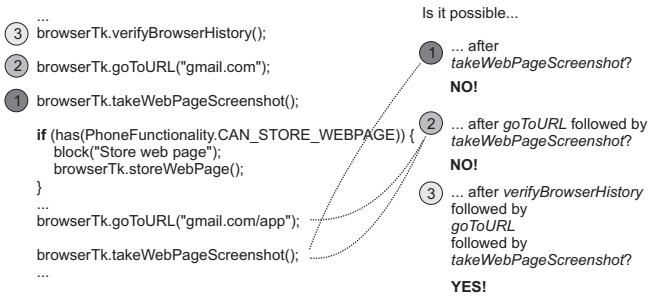

Figure 5. Searching for a unique tracematch.

3.3 Screen shots

Figure 6 shows screen shots of the tool. The main part represents the recommendation of Figure 1. The smaller parts represent the recommendations of Figures 5 and 2, respectively (the complete screen of these examples are not provided due to space restrictions).

4. Concluding Remarks

This paper presented a prototype tool to support developers when evolving variabilities in SPLs. It can recom-

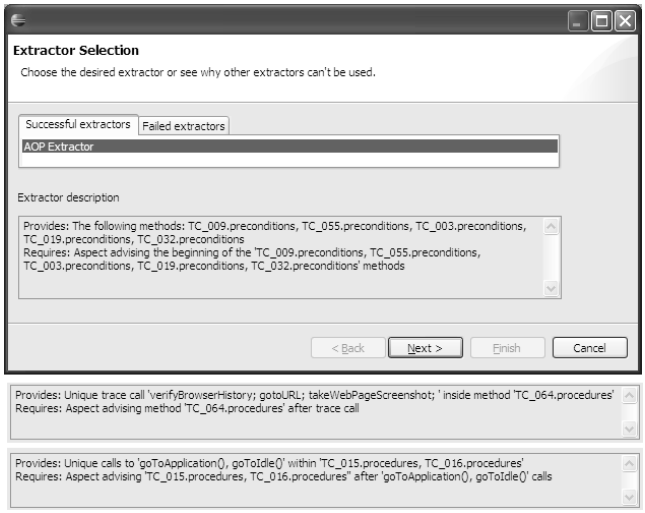

Figure 6. Screen shots: recommending refactorings.

mend refactorings to restructure variabilities faster and precisely, which may reduce developer's effort and increase their productivity. By applying the recommendations, bad smells such as cloned code may be removed and the modularity is improved. For more details, some videos of our tool are available at `http://www.cin.ufpe.br/~mmr3/recommender-tool/`.

On the other hand, our tool has some limitations. The input must be *if-else* statements and many other mechanisms and bad smells are not considered. Also, the provided recommending messages should be improved.

To finish this paper, we discuss a related work [4]. Such work 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 Tracematches mechanism. Besides, our tool searches for cloned code, whereas [4] searches for crosscutting concerns.

Acknowledgments

We would like to thank CNPq, a Brazilian research funding agency, for partially supporting this work. Also, we thank SPG[2] members for fruitful discussions about this work.

[2] http://www.cin.ufpe.br/spg

References

[1] 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.

[2] 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 *LNCS*, pages 70–81. Springer-Verlag, September 2005.

[3] 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.

[4] 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.

[5] 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.

[6] 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), affiliated to 21st European Conference on Object-Oriented Programming (ECOOP'07)*, pages 33–34, New York, NY, USA, July 2007. ACM Press.

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

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

[9] Márcio 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)*, pages 276–277, New York, NY, USA, March 2008. ACM Press.

[10] Márcio 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)*, pages 119–130, October 2007.