SAFEREFACTOR – Tool for Checking Refactoring Safety

Gustavo Soares[1], **Diego Cavalcanti**[1], **Rohit Gheyi**[1],
Tiago Massoni[1], **Dalton Serey**[1], **Márcio Cornélio**[2]

[1]Department of Systems and Computing – UFCG

[2]Department of Systems and Computing – UPE

{gsoares,diegot,rohit,dalton,massoni}@dsc.ufcg.edu.br, marcio@dsc.upe.br

Abstract. *Despite several evidences that refactorings are not sound and safe, refactoring tools are widely used and trusted upon. Several evidences have been published that these tools may perform erroneous transformations that do not preserve behavior. In order to detect some of these errors, developers may rely on compilation and tests to attest that behavior is preserved. Compilation errors, for instance, are simple to identify. However, changes in behavior very often go undetected. In this paper, we present* SAFEREFACTOR *– an Eclipse plugin to identify behavioral changes in transformations. We evaluate it against 9 transformations (5 of them do not preserve behavior and are not detected by the best program refactoring tools). Our tool has been successful in detecting all behavioral changes and not producing false alarms.*

1. Introduction

Refactoring is defined as the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [Fowler 1999]. In order to help the refactoring process, several tools (Eclipse, JBuilder, IntelliJ, Netbeans) automate a number of refactorings. Each refactoring may contain a number of enabling conditions. Each condition must be verified in order to guarantee that a program change preserves its original behavior. For instance, to apply the Extract Class refactoring [Fowler 1999], naming conflicts must be avoided.

The refactoring enbling conditions are commonly implemented in an *ad hoc* way in tools. Formally verifying refactorings is indeed a challenge [Schäfer et al. 2009], and this effort will surely take a long time before it becomes industry standard. Currently, we need a more practical way to deal with erroneous transformations in refactoring tools. The current practice allows transformations that result in programs that change the original behavior (Section 2). To avoid refactoring errors, developers must rely on compilation and tests to assure behavior preservation. Compilations errors are simple to detect by IDEs; the tool must check whether the target program is amenable to correct compilation. However, rarely test suites are good at catching behavioral changes in the refactoring context, because the test suite itself may also be refactored. For instance, a simple renaming may change the test suite. The implementation of this refactoring in some tools may introduce defects [Ekman et al. 2008]. This scenario is undesired since we do not want to compare the behavior of two programs with respect to two different test suites.

In this paper we present the SAFEREFACTOR[1] – an Eclipse Plugin for detecting behavioral changes in transformations of sequential Java programs. It extends the implementation of some refactorings in Eclipse, such as the Push Down Method and Rename

[1]The tool is available at: http://www.dsc.ufcg.edu.br/~gsoares/saferefactor

Local Variable [Fowler 1999], to incorporate our test based technique to help check the safety of the transformation (Section 3). The tool generates a test suite that is useful to pinpoint non behavior-preserving transformations. In each transformation, we identify program parts that are preserved by the transformation, by comparing the original and the refactored program, and automatically generate a comprehensive set of unit tests aided by a random test generator tool to exercise and assess the refactoring. The same test suite must be run on the original and target programs.

We evaluated our SAFEREFACTOR tool by analysing 9 transformations, from which 4 are correct refactorings, and 5 are erroneous refactorings that were manually identified by Ekman et al. [Ekman et al. 2008]. These behavioral changes are not captured by the current best Java refactoring tools (Eclipse, JBuilder, IntelliJ, Netbeans) during the refactorings tasks. Our plugin correctly identified all behavioral changes. Moreover, we tested it against four correct refactorings and the tool did not produce any false alarms. The main contributions of this paper are the following:

- An Eclipse plugin for checking Java sequential program refactorings (Section 4);
- The plugin evaluation in 9 transformations (Section 4).

2. Motivating Example

In this section, we present an example of the Push Down Method refactoring applied by tools like Eclipse, JBuilder, IntelliJ and Netbeans. The example shows a case in which these tools apply a transformation that is intended to be a refactoring but it does not preserve behavior.

In order to illustrate the problem, consider a company which employs ordinary employees who can get a bonus at the end of the month and special kinds of employees (namely Engineers and Software Engineers) who have a different bonus. Listing 1 shows a program representing this scenario: the class `Employee` represents the ordinary employees and the classes `Eng` and `SwEng` represent the special ones. The method `test` in `SwEng` calls the inherited method `defaultBonus` to get the default value of the bonus. In its turn, `defaultBonus` yields 40. One can use an IDE, such as the Eclipse, to push down `defaultBonus` from `Eng` to `SwEng`. So, the method `test` in the resulting program (presented in Listing 2) yields 80 instead of 40. Therefore, the transformation applied by Eclipse does not preserve behavior. The example presented is small, likely to be detected promptly by a developer or a test suite. However, more subtle behavioral changes may be very difficult to pinpoint, specially in larger programs. Moreover, as test suites are commonly refactored as a result, they may fail to detect errors introduced by IDEs. For example, a simple Rename Method refactoring may change the test suite. Additionally, this refactoring may not preserve the behavior of a program in some situations, as shown by Ekman et al. [Ekman et al. 2008]. Therefore, we should be careful in these situations.

In this paper, we propose a plugin for Eclipse (Section 4) which implements our technique explained in Section 3. Our goal is to avoid running different test suites to the original and target programs. In Section 4, we show how our tool detects the behavioral change in the previous example.

3. Technique

Our technique randomly generates a test suite that can be run in both the original and refactored programs [Soares et al. 2009] to detect unexpected changed behavior. Cur-

Figure 1. Push Down Method - Incorrect handling of super accesses

Listing 1. Source Program

```
class Employee {
   int bonus() { return 40; }
}
class Eng extends Employee {
   int defaultBonus() {
      return super.bonus();
   }
   int bonus() { return 80; }
}
class SwEng extends Eng {
   public int test() {
     return defaultBonus();
   }
}
```

Listing 2. Target Program

```
class Employee {
   int bonus() { return 40; }
}
class Eng extends Employee {
   int bonus() { return 80; }
}
class SwEng extends Eng {
   int defaultBonus() {
      return super.bonus();
   }
   public int test() {
     return defaultBonus();
   }
}
```

rently, it is specific for Java, although it can be similarly implemented for other object-oriented languages. Our technique identifies which methods are affected by changes made in the source program. We generate a test suite that exercises the changes. We explain our technique by splitting it in five sequential steps for each refactoring application, as depicted in Figure 2.

Firstly (Step 1), a static analysis is applied to identify methods presented in the source and target programs that exercise the classes affected by the desired transformation. Step 2 aims at randomly generating unit tests for methods identified in Step 1. Notice that both steps guarantees that the same tests can be run on the source and target programs. In Step 3, the generated test suite is run on the source. If no test fails, the same test suite is also run on the target program (Step 4). If the tests run successfully on the target program, we conclude that the transformation does not introduce behavioral changes (Step 5).

4. SAFEREFACTOR

In this section, we present the SAFEREFACTOR tool, which is useful for detecting behavioral changes in program transformations. It is an Eclipse plugin under the Eclipse Plugin License (EPL), and implements the technique presented in Section 3. Next we present the plugin functionalities and how to use it, besides its architecture.

Consider the example from Section 2 to illustrate how to apply the Push Down Method refactoring on SAFEREFACTOR. The user will do the same initial steps of the traditional way in Eclipse: (1) select `defaultBonus` on the Java Editor; (2) go to menu and choose “Push Down”; (3) Eclipse opens a dialog with parameters for further configuration. In the original Eclipse functionality, after setup, the user can see a preview, or just press “OK” and the IDE applies the refactoring.

SAFEREFACTOR users can additionally improve confidence whether the transformation preserves behavior by clicking the “Safe Refactor” button depicted in Figure 3. Another window is opened (Figure 3), and the plugin starts the steps of our technique (Section 3): Step 1 identifies methods in common in both versions of the code (source and target programs), Step 2 generates unit tests, Step 3 runs tests on current code and

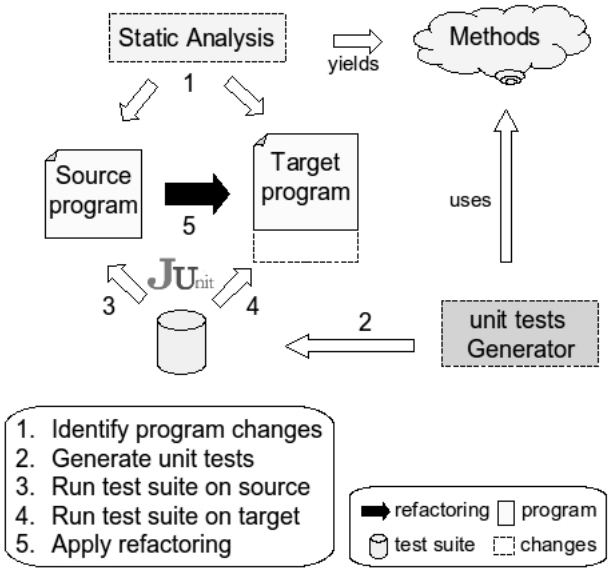

Figure 2. Our Refactoring Technique

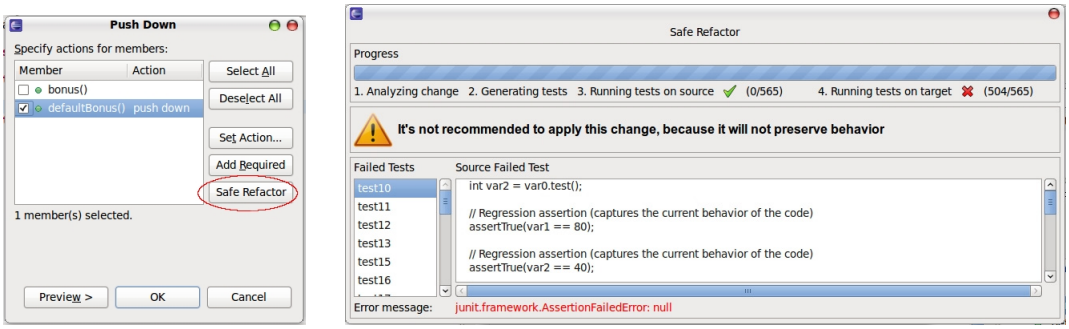

Figure 3. The SAFEREFACTOR Window

Step 4, if no tests fail, runs tests on target program. The user can monitor the process on the top of the window. In Figure 3, Step 4 shows that 504 out of 565 units tests fail, thus the desired program does not have the same behavior of the original one. A message advises that the refactoring should not be applied. Also, the tool shows the lists of tests that fail, and the source code to help the user to identify which sequence of method calls change the program behavior. In the example depicted in Figure 3, it presents a test case that unveils the problem explained in Section 2.

SAFEREFACTOR connects on Eclipse through the Plugin Architecture (Figure 4) [Clayberg and Rubel 2004]. Eclipse provides the Language Toolkit API for automated refactorings, that was implemented by the JDT Team to create the Eclipse Refactoring tool [Widmer 2007]. The plugin contains two main modules: `GUI` and `Core`. The first brings the refactoring functionality in wizard mode. `Wizard` is a set of connected windows that helps a user to do a specific refactoring, this is the goal of class `RefactoringWizard`. This class holds an object `Change`, which contains the transformations that will be made in the code. The class `UserInputPage` creates the controls for further setup of the refactoring. In these controls, there is a button to start our

technique.

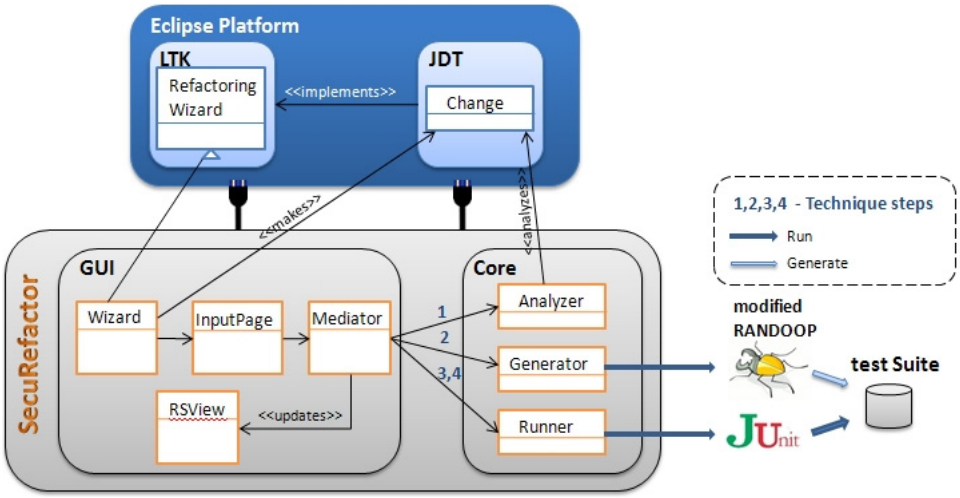

Figure 4. The SAFEREFACTOR Architecture

The technique execution is designed with the Mediator pattern. The object `Mediator` is responsible for calling each class in the Core (`Analyzer`, `Generator` and `Runner`) to complete the steps showed in Section 3 and update the informations to the user. First `Mediator` calls `Analyzer` that is responsible for the first step. It examines the program and the object `Change` and returns classes and methods that will remain with the same signature in the transformed code. Next, `Mediator` uses `Generator` (Step 2) that receives these entities and generates a JUnit test suite for them with RANDOOP [Pacheco et al. 2007], which automatically generates a test suite. It is a random tester tool for Java programs. Finally, in Steps 3 and 4, `Mediator` uses `Runner` to execute the suite of tests on source and target programs.

As we can see in Figure 4, the technique is decoupled from Eclipse's refactoring implementations. The current implementation of our tool implements some refactorings used by Eclipse developers. Third-party developers can extend our plugin with others refactorings by creating news wizards extending the class `RefactoringWizard` of the LTK API. It is important to mention that it is not necessary implement the refactoring itself. We can reuse the JDT implementation.

We evaluate SAFEREFACTOR against 9 transformations: 5 of them identified by Ekman et al. [Ekman et al. 2008] which change the behavior of the target program; and 4 examples in which the transformations do not introduce behavioral changes. The 5 transformations are wrongly applied by the best refactoring tools. They present the following errors: incorrect handling of field accesses, incorrect handling of type accesses, incorrect handling of private method accesses, renaming field can shadow static import and renaming local variable can lead to shadowing by field. Our evaluation has been successful in detecting all of the expected errors and do not produce false alarms. The number of tests generated is proportional to the time allocated to it. The default time limit is 3 seconds. It is recommended to increase the time limit, when the program is large. So, this increases the chances to exercise all changes.

5. Conclusions

In this paper, we present the SAFEREFACTOR tool – an Eclipse plugin for improving confidence that Java transformations does not change behavior. It implements our technique for generating tests in order to find behavioral changes. We generate a test suite that runs on the original and refactored versions of a program. We evaluated the SAFEREFACTOR against 9 transformations. Our approach identified all 5 non behavior-preserving transformations that are not detected by the best Java refactoring tools.

As a future work, we intend to extend our plugin to be used for all refactorings available in Eclipse. Additionally, we aim at evaluating our tool in a real case study. Moreover, the current implementation of our tool only generates tests for public methods. So, adjusting it to consider non-public ones may be another direction. Finally, we intend to improve our static analysis taking into account the classes hierarchy.

Ekman et al. [Ekman et al. 2008] manually catalog a number of non-trivial compilation errors and behavioral changes resulting from refactoring tools (Eclipse, JBuilder, IntelliJ and Netbeans), when trying to apply refactorings such as Rename Local Variable and Push Down Method. We consider all of their examples when evaluating our tool. Our technique is able to automatically pinpoint all behavioral changes introduced. Murphy-Hill and Black [Murphy-Hill and Black 2008b, Murphy-Hill and Black 2008a] characterize a number of problems with current refactoring tools usability. The results of the study show that speed, accuracy, and user satisfaction can be increased. Our work is complementary since we focus on soundness instead of usability.

References

[Clayberg and Rubel 2004] Clayberg, E. and Rubel, D. (2004). *Eclipse: Building Commercial-quality Plug-ins*. Addison-Wesley.

[Ekman et al. 2008] Ekman, T., Ettinger, R., Schafer, M., and Verbaere, M. (2008). Refactoring Bugs in Eclipse, IDEA and Visual Studio. At http://progtools.comlab.ox.ac.uk/refactoring/bugreports.

[Fowler 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Murphy-Hill and Black 2008a] Murphy-Hill, E. and Black, A. P. (2008a). Refactoring Tools: Fitness for Purpose. *IEEE Software*, 25(5):38–44.

[Murphy-Hill and Black 2008b] Murphy-Hill, E. R. and Black, A. P. (2008b). Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In *International Conference on Software Engineering*, pages 421–430.

[Pacheco et al. 2007] Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. (2007). Feedback-Directed Test Generation. In *ICSE*, pages 75–84.

[Schäfer et al. 2009] Schäfer, M., Ekman, T., and de Moor, O. (2009). Challenge proposal: Verification of refactorings. In *Programming Languages meets Program Verification*, pages 67–72.

[Soares et al. 2009] Soares, G., Gheyi, R., Massoni, T., and Cornélio, M. (2009). Generating Unit Tests for Checking Program Refactorings. In *Braz. Symp. on Prog. Languages*.

[Widmer 2007] Widmer, T. (2007). Unleashing the Power of Refactoring. At http://www.eclipse.org/resources/resource.php?id=321.