# Making Program Refactoring Safer

Gustavo Soares, *Member, IEEE,* Rohit Gheyi, Dalton Serey, and Tiago Massoni

***Abstract*—Developers rely on compilation, test suite and tools to preserve observable behavior during refactoring. However, most of the refactoring tools do not implement all preconditions that guarantee the refactoring correctness, since formally identifying them is cost-prohibitive. Therefore, these tools may perform non-behavior preserving transformations. We present a tool for improving safety during refactoring. It automatically generates a test suite that is suited for detecting behavioral changes. We used our tool to evaluate seven real case study refactorings (from 3 to 100 KLOC). We reason about a JHotDraw (23 KLOC) and its refactored version, and automatically detected a behavioral change. This problem was not identified by developers. Finally, we also evaluated our tool against 17 defective refactorings that are not detected by refactoring tools.**

***Keywords*—Refactoring, Behavior-preservation, Unit-testing.**

## I. 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 and improves its internal structure [1], [2]. In practice, developers perform refactorings either manually – error-prone and time consuming – or with the help of IDEs, which can support refactoring, such as Eclipse, Netbeans, JBuilder and IntelliJ.

In general, each refactoring may contain a number of preconditions to preserve the observable behavior. For instance, to rename an attribute, name conflicts cannot be present. However, mostly refactoring tools do not implement all preconditions, because formally establishing all of them is not trivial. Therefore, often refactoring tools allow wrong transformations to be applied with no warnings whatsoever. For instance, Figure I shows a transformation [3] presenting subtle errors in maintaining program behavior when applying it using the Eclipse 3.4.2 IDE. Listing 1 shows a program containing the class `A` and its subclass `B`. The method `test` yields 10. When we apply the pull up refactoring to the method `k(int)` using Eclipse, the resulting code is presented in Listing 2. The method `test` in the target program yields `20`, instead of `10`. Therefore, the transformation does not preserve behavior using the Eclipse 3.4.2 IDE.

The current practice to avoid behavioral changes in refactorings relies on solid tests [1]. However, often test suites do not catch behavioral changes during transformations. They may also be refactored (for instance, rename method) by the tools since they may rely on the program structure that is modified by the refactoring [2]. In this case, the tool changes the method invocations on the test suite, and the original and refactored programs are checked against different test suites. This scenario is undesirable since the refactoring tool may change the test suite meaning [4].

In this article, we describe and evaluate the SafeRefactor, a tool for checking refactoring safety in sequential Java programs using Eclipse IDE. For each transformation, it generates a test suite useful for detecting behavioral changes.

## II. SafeRefactor

SafeRefactor is an Eclipse 3.4.2 plugin [1] that receives a source code and a refactoring to be applied (input). It reports whether it is safe to apply the transformation (output).

Suppose that we use SafeRefactor in Listing 1 program. Next we explain the whole process, which has seven sequential steps for each refactoring application (Figure 2). First the developer selects the refactoring to be applied on the source program (Step 1.1) and uses SafeRefactor (Step 1.2). The plugin starts checking the refactoring safety (Steps 2-7).

It generates a target program based on the desired transformation using Eclipse refactoring API (Step 2). In Step 3, a static analysis automatically identifies methods in common in both source and target programs. Step 4 aims at generating unit tests for methods identified in Step 3. In Step 5, the plugin runs the generated test suite on the source program. Next, it runs the same test suite on the target program (Step 6). If a test passes in one of the programs and fails in the other one, the plugin detects a behavioral change and reports to the user (Step 7). Otherwise, the programmer can have more confidence that the transformation does not introduce behavioral changes.

The goal of the *static analysis* (Step 3) is to identify *methods in common*: they have exactly the same modifier, return type, qualified name, parameters types and exceptions thrown in source and target programs. For example, Listings 1 and 2 contain `A.k(long)` and `B.test()` in common.

After identifying a set of useful methods, the plugin uses Randoop [5] to generate unit tests (Step 4). Randoop generates tests for classes within a time limit. A unit test typically consists of a sequence of method and constructor invocations that creates and mutates objects with random values, plus a JUnit assertion. Randoop executes the program to receive a feedback gathered from executing test inputs as they are created, to avoid generating redundant and illegal inputs [5]. It creates method sequences incrementally, by randomly selecting a method call to apply and selecting arguments from previously constructed sequences. Each sequence is executed and checked against a set of contracts. For instance, an object must be equal to itself. Our tool uses the Randoop default contracts. We modified Randoop to remove some defects, and to pass a set of methods as parameter. All tests generated only contain calls to the methods identified in Step 3. The default

G. Soares, R. Gheyi, D. Serey and T. Massoni are affiliated to the Department of Computing and Systems, Federal University of Campina Grande, Campina Grande, PB, 58429-900 Brazil e-mail:{gsoares,rohit,dalton,massoni}@dsc.ufcg.edu.br.

[1] Available at http://www.dsc.ufcg.edu.br/~spg/saferefactor/

Listing 1. Source Program

```
public class A {
  public int k(long i) {
    return 10;
  }
}
public class B extends A {
  public int k(int i) {
    return 20;
  }
  public int test() {
    return new A().k(2);
  }
}
```

Listing 2. Target Program

```
public class A {
  public int k(long i) {
    return 10;
  }
  public int k(int i) {
    return 20;
  }
}
public class B extends A {
  public int test() {
    return new A().k(2);
  }
}
```

Fig. 1. Pull Up Method Refactoring Enables Overloading

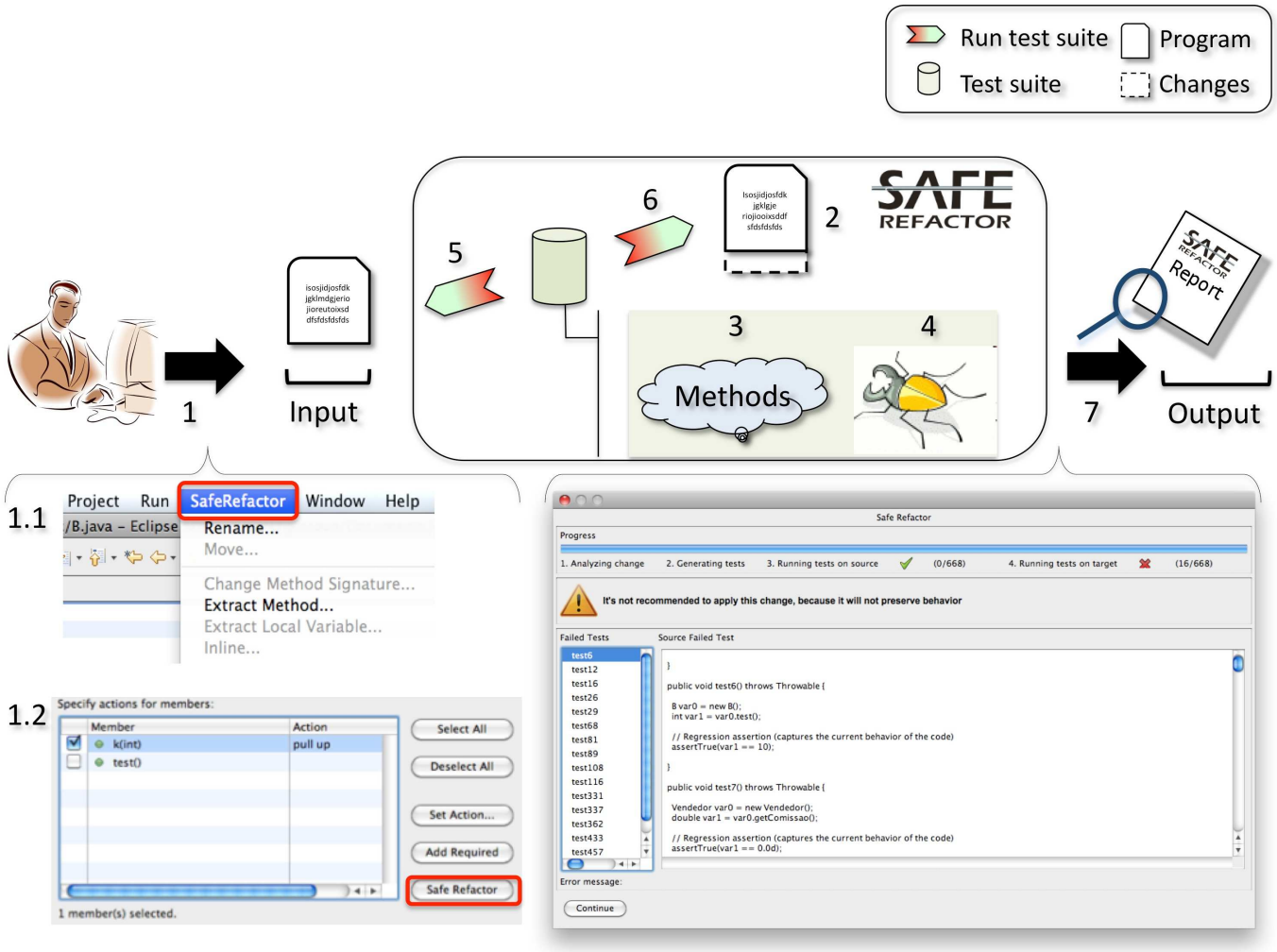


Fig. 2. SAFEREFACTOR. 1. The user selects a refactoring in the menu to apply (1.1) and click on the SAFEREFACTOR button (1.2), 2. The tool generates the target program using the Eclipse refactoring API, 3. It identifies common methods in the source and target programs, 4. The tool generates unit tests using our modified Randoop, 5. It runs the test suite on the source program, 6. The tool runs the test suite on the target program, 7. The tool shows the report to developer. If it finds a behavior change, the user can see some unit tests that fail.

time limit is 2s. Steps 3 and 4 ensure that the same tests can be run on the source and target programs.

The whole process to finish Figure I example takes less than 8 seconds on a dual-processor 2.2 GHz Dell Vostro 1400 laptop with 2 GB RAM, and generates 154 unit tests (151 of them failed in the target program). SAFEREFACTOR reports the user that the refactoring should not be applied. Users can see some tests that expose the behavior change (Step 7). In other situations, SAFEREFACTOR can report compilation errors that may be introduced by refactoring tools. If SAFEREFACTOR

does not find a behavior change or compilation error, it reports that users can improve confidence that the transformation is sound.

Opdyke compares the observable behavior of two programs with respect to the `main` method (a method in common). If it is called twice (source and target programs) with the same set of inputs, the resulting set of output values must be the same [6]. SAFEREFACTOR checks the observable behavior with respect to randomly generated sequences of methods and constructor invocations. They only contain calls to methods in common. If the source and target programs have different results for the same input, they do not have the same behavior.

## III. EVALUATION

We evaluated SAFEREFACTOR in some transformations.

### A. Subject Characterization

Tables I and II show the subjects (pairs of source and target programs) used in the experiment. Each of them is uniquely identified (Subject column). The subjects are divided in two categories: refactoring real applications, and a catalog of defective refactorings [2].

The first category (Subjects 1-7) consists on refactorings performed by developers applied to real Java applications ranging from 3-100 KLOC (non-blank, non-comment lines of code) using tools or manual steps. All of them are considered behavior preserving by developers. We use SAFEREFACTOR to evaluate whether the transformation preserves the observable behavior.

Third-party developers performed a refactoring on JHotDraw and CheckStylePlugin (Subjects 1-2) to modularize exception handling code [7]. Fuhrer et al. [8] proposed and implemented an Eclipse refactoring to apply the infer generic type argument refactoring, enabling applications to use Java generics. They evaluated their refactoring in four real Java applications: JUnit, Vpoker, ANTLR, and Xtc (Subjects 3-6). Murphy-Hill et al. [9] performed some experiments to analyze how developers refactor. They used a set of twenty Eclipse components versions from Eclipse CVS, and manually detected the refactorings applied. We evaluate one transformation in Eclipse TextEditor module (Subject 7).

The second category (Subjects 8-24) includes non-behavior transformations applied by refactoring tools. These bugs were identified in the literature [3], [4], [10]. We use SAFEREFACTOR to evaluate whether it detects the behavior changes.

### B. Experimental Setup

We run the experiment on a dual-processor 2.2 GHz Dell Vostro 1400 laptop with 2 GB RAM and running Ubuntu 9.04. In both categories, we used a command line interface provided by our SAFEREFACTOR. It receives three parameters: source and target program paths, and timeout to generate tests. We used the default timeout of 90s and 2s to generate the tests in the first and second categories, respectively. We did not use the SAFEREFACTOR Eclipse graphical interface in order to automate the experiment.

[2]All subjects are available at: http://www.dsc.ufcg.edu.br/~spg/saferefactor/experiments.htm

### C. Experimental Results

SAFEREFACTOR detected a behavioral change in one refactoring and two compilation errors in less than 4 minutes in the first category. Table I shows the program name, its size in KLOC and the total time in seconds required by SAFEREFACTOR to yield a result in the Program, KLOC and Total Time (s) columns, respectively.

Each line of Tables I and II contains the number of generated tests (Tests column) and the number of tests detecting the behavioral change (Error column) for each subject. The Result column indicates whether SAFEREFACTOR identified a behavior change or a compilation error. The symbol - indicates that no behavior change is detected.

In the second category, it detected all but one behavior change in less than 8s. Table II indicates the defective refactoring applied and a description of the behavior change introduced in the Refactoring and Bug Description columns, respectively.

Developers refactored JHotDraw in order to avoid code duplication with identical exception handlers in different parts of a system [11]. Eight programmers working in pairs performed the change: they extracted the code inside the `try`, `catch`, and `finally` blocks to methods in specific classes that handle exceptions. They relied on refactoring tools, pair review, and unit tests to assure that the behavior was preserved. Some classes that implement `Serializable` were refactored. Developers changed the `clone` method and introduced the `handler` attribute to handle exceptions. However, they forgot to serialize this new attribute. Thus, when the method `clone` try to serialize the object, an exception is thrown. Therefore, the refactored method `clone` has a different behavior.

Moreover, Eclipse wrongly applied a refactoring to ANTLR and Xtc, introducing a compilation error that was not reported [8]. Finally, SAFEREFACTOR did not detect behavior change in Subjects 2-4 and 7. In the second category, SAFEREFACTOR identified all but one behavior change that uses standard output. Tables I and II summarize the results.

### D. Discussion

Our tool cannot detect behavioral changes in the standard output (`System.out.println`) messages and exception messages (Subject 8). Thus, we modify some subjects to include `return` statements with the values of the messages. We intend to generate some test patterns that are useful for detecting changes in the standard output. Additionally, our technique can detect behavior changes in `void` methods only when they change some fields that contain a *getter* method for it. We aim at improving our technique by automatically generating *getter* methods for all attributes. Finally, the current implementation of our technique cannot deal with inner and non-public classes directly. However, if there is a method in common using them, we can exercise them indirectly.

Consider a rename method from `A.m(..)` to `A.n(..)`. The set of methods identified by Step 3 does not include them. A similar thing occurs when renaming a class. We cannot compare the renamed method's behavior directly. However, SAFEREFACTOR compares them indirectly if a method in

| 1st Category: Refactoring Real Applications | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Subject** | **Program** | **KLOC** | **Refactoring** | **Tests** | **Error** | **Total Time (s)** | **Result** |
| 1 | JHotDraw | 23 | Extract Exception Handler | 2245 | 273 | 148 | Behavior Change |
| 2 | CheckStylePlugin | 20 | Extract Exception Handler | 5864 | 0 | 235 | - |
| 3 | Junit | 3 | Infer Generic Type | 1127 | 0 | 99 | - |
| 4 | Vpoker | 4 | Infer Generic Type | 466 | 0 | 109 | - |
| 5 | ANTLR | 32 | Infer Generic Type | - | - | 2 | Compilation Error |
| 6 | Xtc | 100 | Infer Generic Type | - | - | 4 | Compilation Error |
| 7 | TextEditor | 15 | Replace Deprecated Code | 16009 | 0 | 107 | - |

TABLE I
SUMMARY OF SAFEREFACTOR EVALUATION IN REFACTORING REAL CASE STUDIES

| 2nd Category: Catalog of Defective Refactorings | | | | | |
|---|---|---|---|---|---|
| **Subject** | **Refactoring** | **Bug Description** | **Tests** | **Error** | **Result** |
| 8 | Push Down Method | Incorrect handling of super accesses | 488 | 0 | - |
| 9 | Rename Class | Renaming a class leads to undiagnosed shadowing | 102 | 95 | Behavior Change |
| 10 | Rename Variable | Renaming a local variable leads to shadowing by field | 494 | 492 | Behavior Change |
| 11 | Rename Method | Renaming a method leads to shadowing of statically imported method | 93 | 91 | Behavior Change |
| 12 | Encapsulate Field | No check for overriding problems | 474 | 464 | Behavior Change |
| 13 | Extract Method | Incorrect dataflow analysis | 558 | 554 | Behavior Change |
| 14 | Push Down Method | Incorrect handling of super accesses | 486 | 404 | Behavior Change |
| 15 | Push Down Method | Incorrect handling of field accesses | 78 | 75 | Behavior Change |
| 16 | Push Down Method | Pushing down a method enables overloading | 101 | 99 | Behavior Change |
| 17 | Move Class | Move a class to another package disables overriding | 101 | 99 | Behavior Change |
| 18 | Move Class | Move a class to another package disables overloading | 79 | 77 | Behavior Change |
| 19 | Change Method Signature | Increasing method visibility enables overriding | 214 | 40 | Behavior Change |
| 20 | Change Method Signature | Increasing method visibility enables overloading | 79 | 76 | Behavior Change |
| 21 | Change Method Signature | Increase method visibility enables overriding to another package | 121 | 40 | Behavior Change |
| 22 | Pull Up Method | Pulling up a method enables overloading | 101 | 99 | Behavior Change |
| 23 | Pull Up Method | Pulling up a method enables overriding | 170 | 88 | Behavior Change |
| 24 | Pull Up Method | Pulling up a method to a class in another package enables overriding | 167 | 163 | Behavior Change |

TABLE II
SUMMARY OF SAFEREFACTOR EVALUATION IN THE CATALOG OF DEFECTIVE REFACTORINGS

common (`x`) calls them. Step 4 generates tests including `x` in the randomly generated sequence of method calls. It is similar to Opdyke's notion. If `main` calls them, we compare them indirectly. Moreover, a simple rename method may enable or disable overloading [10]. This feature is a potential source of problems. Step 4 generates specific tests for exercising every method (not affected by the transformation) named `m` or `n` in the superclasses or subclasses of `A`. Table II shows three defective renaming refactorings that our tool detected.

### *E. Related Work*

Daniel et al. [12] proposed a technique for automated testing refactoring tools. They found some compilation errors introduced by Eclipse and Netbeans. Steimann and Thies [3] and Ekman et al. [4] manually catalogued behavioral changes problems (for some kinds of refactorings) in refactoring tools. Moreover, Schäfer et al. [10] explained an approach to solve problems on the rename refactoring. We propose a more practical approach for detecting behavioral changes using a tool support, regardless the kind of refactoring.

## IV. FINAL REMARKS

We presented a tool for improving safety during refactoring activities. Our earlier works present our technique [13], and a tool [14]. Here we evaluate our technique with empirical studies. We intend to create a plugin for other IDEs, such as Netbeans. Additionally, besides using it in more real case studies, we aim at testing refactoring tools in order to automatically find non-behavior transformations following a similar approach of Daniel et al. [12].

## ACKNOWLEDGMENT

We gratefully thank the guest editors and the anonymous referees for useful suggestions. This work was partially supported by the National Institute of Science and Technology

for Software Engineering (INES), funded by CNPq grants 573964/2008-4 and 477336/2009-4.

## REFERENCES

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
[2] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE TSE*, vol. 30, no. 2, pp. 126–139, 2004.
[3] F. Steimann and A. Thies, "From public to private to absent: Refactoring java programs under constrained accessibility," in *ECOOP*, 2009, pp. 419–443.
[4] T. Ekman, R. Ettinger, M. Schafer, and M. Verbaere, "Refactoring bugs in eclipse, idea and visual studio," 2008.
[5] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*, 2007, pp. 75–84.
[6] W. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, UIUC, 1992.
[7] J. Taveira, C. Queiroz, R. Lima, J. Saraiva, S. Soares, H. Oliveira, N. Temudo, A. Araújo, J. Amorim, F. Castor, and E. Barreiros, "Assessing intra-application exception handling reuse with aspects," in *SBES*, 2009, pp. 22–31.
[8] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller, "Efficiently refactoring java applications to use generic libraries," in *ECOOP*, 2005, pp. 71–96.
[9] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," in *ICSE*, 2009, pp. 287–296.
[10] M. Schäfer, T. Ekman, and O. Moor, "Sound and extensible renaming for java," in *OOPSLA*, 2008, pp. 277–294.
[11] B. Cabral and P. Marques, "Exception handling: A field study in java and .net," in *ECOOP*, 2007, pp. 151–175.
[12] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *FSE*, 2007, pp. 185–194.
[13] G. Soares, R. Gheyi, T. Massoni, M. Cornélio, and D. Cavalcanti, "Generating unit tests for checking refactoring safety," in *SBLP*, 2009, pp. 159–172.
[14] G. Soares, D. Cavalcanti, R. Gheyi, T. Massoni, D. Serey, and M. Cornélio, "Saferefactor - tool for checking refactoring safety," in *Tools Session at SBES*, 2009, pp. 49–54.

**Gustavo Soares** is a PhD student in the Department of Computer Science at Federal University of Campina Grande. His research interests include refactorings and formal methods. He holds a MSc in Computer Science from the Federal University of Campina Grande, and is a member of the IEEE and ACM. Contact him at DSC/UFCG, 882 Aprígio Veloso, Bodocongó, Campina Grande, Brazil; Phone: +55 83 3310 1122; gsoares@dsc.ufcg.edu.br.

**Rohit Gheyi** is a professor in the Department of Computer Science at Federal University of Campina Grande. His research interests include refactorings, formal methods and software product lines. He holds a Doctoral degree in Computer Science from the Federal University of Pernambuco, and is a member of the ACM. Contact him at DSC/UFCG, 882 Aprígio Veloso, Bodocongó, Campina Grande, Brazil; Phone: +55 83 3310 1122 (ext. 2202); rohit@dsc.ufcg.edu.br.

**Dalton Serey** is a professor in the Department of Computer Science at Federal University of Campina Grande. His research interests include software evolution. He holds a Doctoral degree in Computer Science from the Federal University of Campina Grande, and is a member of the ACM. Contact him at DSC/UFCG, 882 Aprígio Veloso, Bodocongó, Campina Grande, Brazil; Phone: +55 83 3310 1122; dalton@dsc.ufcg.edu.br.

**Tiago Massoni** is a professor in the Department of Computer and Systems at the Federal University of Campina Grande. His research interests include software design and evolution, and formal methods. In addition to his academic posts he also worked as a programmer at IBM in California. He holds a Doctoral degree in Computer Science from the Federal University of Pernambuco, and is a member of the ACM. Contact him at DSC/UFCG, 882 Aprígio Veloso, Bodocongó, Campina Grande, Brazil; Phone: +55 83 3310 1122 (ext. 2202); massoni@dsc.ufcg.edu.br.