Making Program Refactoring Safer

Gustavo Soares[1], **Rohit Gheyi**[1], **Dalton Serey**[1]

[1]Department of Computing and Systems – UFCG
58429-900 – Campina Grande – PB – Brazil

{gsoares,rohit,dalton}@dsc.ufcg.edu.br

Abstract. *Developers rely on compilation, test suite and tools to preserve observable behavior during refactorings. However, most of the refactoring tools do not implement all preconditions that guarantee behavioral preservation, since formally identifying them is cost-prohibitive. Therefore, these tools may change the behavior of the user's code. We propose two approaches for making program refactoring safer. We present an approach and its implementation (*SAFEREFACTOR*) useful for improving the developer's confidence that the refactoring was correctly applied. It analyzes a transformation and generates tests specific for detecting behavioral changes. Moreover, we propose an approach to help refactoring tool developers to test their implementations. This approach evaluates an implementation by automatically generating programs (test inputs), and checking the results with* SAFEREFACTOR. *We evaluated* SAFEREFACTOR *in transformations applied to seven real programs ranging from 3 to 100 KLOC. In one of them (JHotDraw), we identified a behavioral change that was not revealed so far. On the other hand, we evaluated the second approach by testing 22 refactoring implementations in two Java refactoring tools: the Eclipse Refactoring Module and the JastAdd Refactoring Tool. We analyzed more than 80K transformations, and identified 67 bugs.*

Keywords: Refactoring, Behavior-preservation, Testing, Program generation

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 and improves its internal structure [Fowler 1999, Mens and Tourwé 2004]. In practice, developers perform refactorings either manually – error-prone and time consuming – or with the help of IDEs that 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, the program shown in Listing 1 containing the `A` class and its subclass `B`. The `m` method yields 10. When we apply the pull up refactoring to the `k(int)` method using Eclipse 3.6, the IDE moves it to class `A`. After the transformation, the `m` method yields `20`, instead of `10`. Therefore, the transformation does not preserve behavior using the Eclipse 3.6 IDE. Notice that, originally, the expression inside `m` calls `A.k(long)`. However, after pulling up `k(int)`, this method is called instead.

Listing 1. Pulling up B.k(int) using Eclipse 3.6 changes program behavior.

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

The current practice to avoid behavioral changes in refactorings relies on solid tests [Fowler 1999]. 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 [Mens and Tourwé 2004]. In this case, the tool changes the method invocations on the test suite, and the original and the refactored programs are checked against different test suites. This scenario is undesirable since the refactoring tool may change the test suite meaning [Schäfer et al. 2008].

Moreover, testing refactoring tools is not simple. It requires complex test inputs (Java programs). In practice, refactoring tool developers manually write these inputs. As a result, the test suite may leave hidden bugs that the testers are unaware of. In this work, we aim at making program refactoring safer by helping not only refactoring practitioners but also refactoring tool developers.

2. SAFEREFACTOR

We propose an approach (SAFEREFACTOR) [Soares et al. 2010, Soares et al. 2009b, Soares 2010a, Soares 2010b, Soares et al. 2009a] for checking refactoring safety in sequential Java programs. It analyzes a transformation, and generates a test suite useful for detecting behavioral changes.

We implemented SAFEREFACTOR as an Eclipse plugin. Suppose that we use it to evaluate the previous transformation (see Listing 1). Next we explain the whole process (Figure 1). 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 developer can have more confidence that the transformation does not introduce behavioral changes.

3. A technique for testing refactoring tools

We also propose an approach to help refactoring tool developers to test their implementations [Soares 2010a, Soares 2010b]. First, it automatically generates a large number of small programs as test inputs. This step aims at generating inputs that testers may be unaware of. Second, it applies the refactoring implementation to each one of them. It

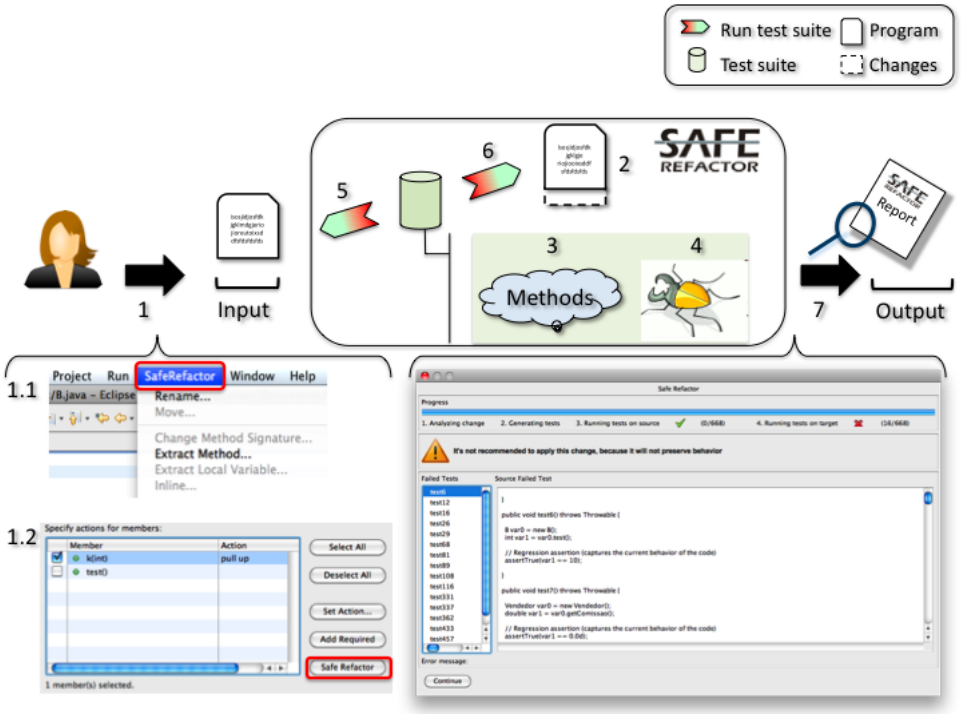

Figure 1. SAFEREFACTOR. 1. The user selects a refactoring in the menu to apply (1.1) and click on the SAFEREFACTOR button (1.2), 2. The tool generates the target program using the Eclipse refactoring API, 3. It identifies common methods in the source and target programs, 4. The tool generates unit tests, 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 developer can see some unit tests that fail.

then uses SAFEREFACTOR to evaluate the correctness of the transformations. To perform the first step, we developed JDolly, a Java program generator. It contains a subset of the Java metamodel specified in Alloy, a formal specification language [Jackson 2006], and uses the Alloy Analyzer, a tool for analysis of Alloy models, for generating solutions for this metamodel. Each solution is translated to a Java program. JDolly receives as input the scope of the generation, that is, the maximum number of elements (packages, classes, fields, and methods) that generated programs must have, and additional constraints for generating test inputs specific for each implementation.

4. Evaluation

We evaluated SAFEREFACTOR against 24 transformations applied to Java programs[1]. They were divided in two categories: *Refactorings in real Java applications*, and *Catalog of Defective Refactorings*. The former (Table 1) consists of refactorings applied to seven real Java applications (3-100 KLOC). Developers used tools or manual steps for applying these transformations, and the test suites of the programs for guaranteeing behavioral preservation. All of the transformations were considered behavior preserving by them. The latter is a catalog of non-behavior-preserving transformations applied by tools

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

Subject	Program	KLOC	Refactoring	Result
1	JHotDraw	23	Extract Exception Handler	Behavior Change
2	CheckStylePlugin	20	Extract Exception Handler	-
3	JUnit	3	Infer Generic Type	-
4	Vpoker	4	Infer Generic Type	-
5	ANTLR	32	Infer Generic Type	Compilation Error
6	Xtc	100	Infer Generic Type	Compilation Error
7	TextEditor	15	Replace Deprecated Code	-

Table 1. Refactorings in real Java applications.

Refactoring	Programs	Time (mm:ss)		Failures				Bugs			
				CE		BC		CE		BC	
		Eclipse	JRRT	Eclipse	JRRT	Eclipse	JRRT	Eclipse	JRRT	Eclipse	JRRT
Rename Class	7200	03:12	03:48	194	0	145	0	1	0	1	0
Rename Method	13464	07:54	11:30	549	0	0	0	1	0	0	0
Rename Field	6080	05:18	05:58	6	76	0	16	1	3	0	1
Push Down Method	5880	04:48	04:36	595	618	186	105	2	2	6	2
Push Down Field	7488	04:42	04:26	340	0	92	0	1	0	1	0
Pull Up Method	8760	06:24	06:06	132	296	203	74	2	2	7	2
Pull Up Field	6624	05:36	06:30	222	72	546	0	3	2	3	0
Encapsulate Field	8832	05:26	05:34	2000	0	0	108	1	0	0	1
Move Method	8938	08:31	05:39	889	922	3586	1422	2	3	4	2
Add Parameter	15808	13:41	14:18	706	0	1116	137	1	0	3	2
Remove Parameter	15808	14:18	-	706	-	190	-	1	-	2	-
Change Modifier	7224	11:42	-	602	-	703	-	1	-	3	-
Total of Bugs								**17**	**12**	**28**	**10**

Table 2. Test of refactoring implementations of Eclipse and JRRT; CE = compilation error; BC = behavioral change.

manually catalogued in the literature [Schäfer et al. 2008, Steimann and Thies 2009]. In the first category, SAFEREFACTOR detected a behavioral change in one transformation and two compilation errors (Table 1). Regarding the *Catalog of Defective Refactorings*, it detected all behavioral changes but one.

Moreover, we evaluated our technique for testing refactoring tools by testing 22 refactoring implementations in two Java refactoring tools: the Eclipse Refactoring Module and the JastAdd Refactoring Tool (JRRT) [Schäfer et al. 2010]. Table 2 summarizes the results. It shows the number of programs generated by JDolly, the testing time, and the number of failures (compilation errors and behavioral changes) of each implementation. Many of these failures are related to the same bug. We analyzed them and identified 29 bugs related to compilation errors, and 38 bugs related to behavioral changes (Column Bugs). JDolly and SAFEREFACTOR were useful for detecting bugs that have not been revealed so far. Additionally, many of the bugs that we found are also bugs in Netbeans 6.7 (we manually checked them).

5. Related work

Schäfer et al. [Schäfer et al. 2008] propose a Rename refactoring implementation. It is based on the name binding invariant: each name should refer to the same entity before and after the transformation. Furthermore, Schäfer et al. [Schäfer et al. 2009b, Schäfer et al. 2010] propose a number of Java refactoring implementations using an enriched language. Despite their technique turns easier to implement the transformation itself, it is necessary an additional effort to translate the program back. As correctness criterion, they proposed other invariants such as control flow and data flow preservation. We evaluated ten of these implementations using our technique. In the sample used in our evaluation, the JRRT outperformed Eclipse with respect to refactoring correctness: we found 22 bugs in JRRT and 45 bugs in Eclipse.

Daniel et al. [Daniel et al. 2007] proposed an approach for automated testing refactoring tools. They propose a program generator called ASTGen. It allows developers to guide the program generation by extending Java classes. Our generator allows this by specifying Alloy constraints. To evaluate the refactoring correctness, they implemented six oracles that evaluate the output of each transformation. While their oracles could only syntactically compare the programs to detect behavioral changes, SAFEREFACTOR generates tests that do compare program behavior. They found one bug related to behavioral change (we found 28 bugs). Moreover, both techniques found the same number of bugs related to compilation errors (17).

Other approaches have proposed refactoring specifications [Borba et al. 2004, Tip et al. 2003, Silva et al. 2008, Steimann and Thies 2009]. They analyze various aspects of Java, as accessibility, types, name binding, data flow, and control flow. However, proving refactoring correctness for the whole Java language is considered a grand challenge [Schäfer et al. 2009a]. We propose a more practical approach for detecting behavioral changes using a tool support, regardless the kind of refactoring.

6. Conclusions

We presented an approach and its implementation (SAFEREFACTOR) for improving safety of the refactoring activity. By using SAFEREFACTOR, developers can have more confidence that a refactoring was correctly applied. We used it in transformations with up to 100KLOC, and it was useful for reporting behavioral changes that were not detected by IDEs and existing test suites. We also proposed an approach that helps refactoring tool developers to test their implementations. This approach evaluates an implementation by automatically generating programs (test inputs), and checking the results with SAFEREFACTOR. It was useful for identifying 67 bugs in 22 refactoring implementations of state-of-the-art industrial and academic refactoring tools.

Acknowledgment

We would like to thank Tiago Massoni, Paulo Borba, Augusto Sampaio, and David Naumann. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES)[2], funded by CNPq grants 573964/2008-4, 477336/2009-4 and 304470/2010-4.

[2] http://www.ines.org.br

References

[Borba et al. 2004] Borba, P., Sampaio, A., Cavalcanti, A., and Cornélio, M. (2004). Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100.

[Daniel et al. 2007] Daniel, B., Dig, D., Garcia, K., and Marinov, D. (2007). Automated testing of refactoring engines. In *FSE '07*, pages 185–194.

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

[Jackson 2006] Jackson, D. (2006). *Software Abstractions: Logic, Language and Analysis*. MIT press.

[Mens and Tourwé 2004] Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

[Schäfer et al. 2010] Schäfer, M., , and de Moor, O. (2010). Specifying and implementing refactorings. In *OOPSLA '10*, pages 286–301.

[Schäfer et al. 2008] Schäfer, M., Ekman, T., and de Moor, O. (2008). Sound and extensible renaming for Java. In *OOPSLA '08*, pages 277–294.

[Schäfer et al. 2009a] Schäfer, M., Ekman, T., and de Moor, O. (2009a). Challenge proposal: Verification of refactorings. In *PLPV '09*, pages 67–72.

[Schäfer et al. 2009b] Schäfer, M., Verbaere, M., Ekman, T., and Moor, O. (2009b). Stepping stones over the refactoring rubicon. In *ECOOP '09*, pages 369–393.

[Silva et al. 2008] Silva, L., Sampaio, A., and Liu, Z. (2008). Laws of object-orientation with reference semantics. In *SEFM '08*, pages 217–226.

[Soares 2010a] Soares, G. (2010a). Making program refactoring safer. In *Student Research Competition at ICSE '10*, pages 521–522.

[Soares 2010b] Soares, G. (2010b). Making program refactoring safer. In *XVII Latin-American Master Thesis Contest CLTM 2010 at CLEI '10*.

[Soares et al. 2009a] Soares, G., Cavalcanti, D., Gheyi, R., Massoni, T., Serey, D., and Cornélio, M. (2009a). SafeRefactor - tool for checking refactoring safety. In *Tools Session at SBES*, pages 49–54.

[Soares et al. 2009b] Soares, G., Gheyi, R., Massoni, T., Cornélio, M., and Cavalcanti, D. (2009b). Generating unit tests for checking refactoring safety. In *SBLP*, pages 159–172.

[Soares et al. 2010] Soares, G., Gheyi, R., Serey, D., and Massoni, T. (2010). Making program refactoring safer. *IEEE Software*, 27:52–57.

[Steimann and Thies 2009] Steimann, F. and Thies, A. (2009). From public to private to absent: Refactoring Java programs under constrained accessibility. In *ECOOP '09*, pages 419–443.

[Tip et al. 2003] Tip, F., Kiezun, A., and Baumer, D. (2003). Refactoring for Generalization Using Type Constraints. In *OOPSLA '03*, pages 13–26.