Systematic Testing of Refactoring Tools

Andreas Thies · Friedrich Steimann
FernUniversität in Hagen

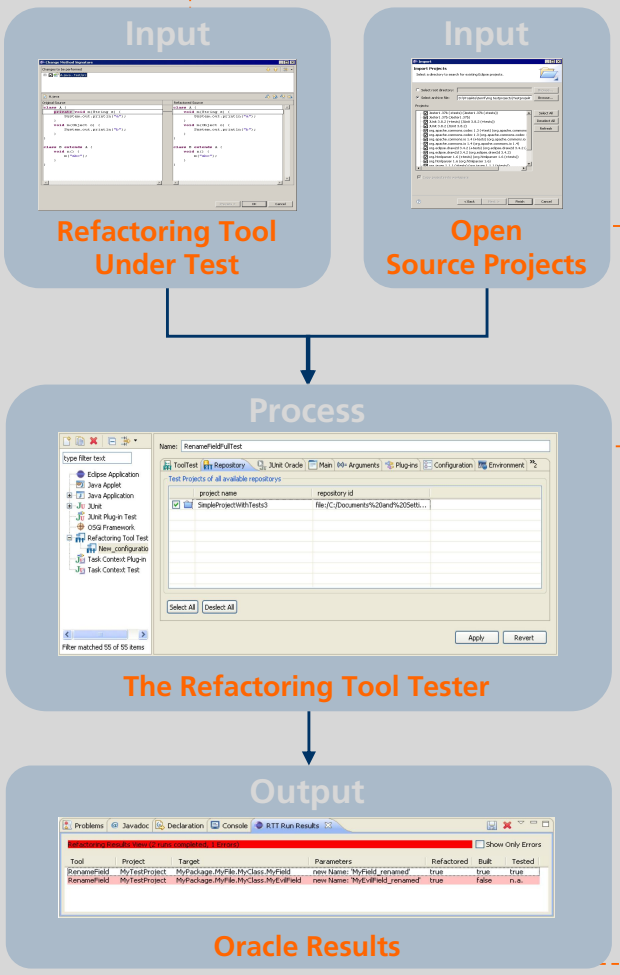
The Problem:
============

Refactorings offer the chance to improve code quality, but they involve the risk to introduce new bugs. Originally, tool support was meant to make refactorings more reliable, but current state of the art is that using refactoring tools risks breaking the code.

Refactoring tool developers spend much effort in testing; the Eclipse Language Toolkit [3] (home of Eclipse's refactoring tools) has thousands of test cases publicly available. Nevertheless, a notable number of refactoring tools are flawed! [1]

Before writing a test case the programmer must be aware of what exactly needs to be tested. Unfortunately, modern object oriented languages come with large numbers of language concepts (such as overriding, overloading, hiding, shadowing, obscuring, ambiguous access, name clashes, type compatibility…). Refactoring tools (as well as their tests) need to cover not only every single of those but also their combinations. Good test coverage is therefore intrinsically hard to achieve.

The Approach



➢ Each **Refactoring Tool Under Test** is plugged into the Refactoring Tool Tester using a specific *adapter*. An adapter specifies where a refactoring may be performed and which parameters are supplied. Typically, writing an adapter requires less than 50 lines of code.

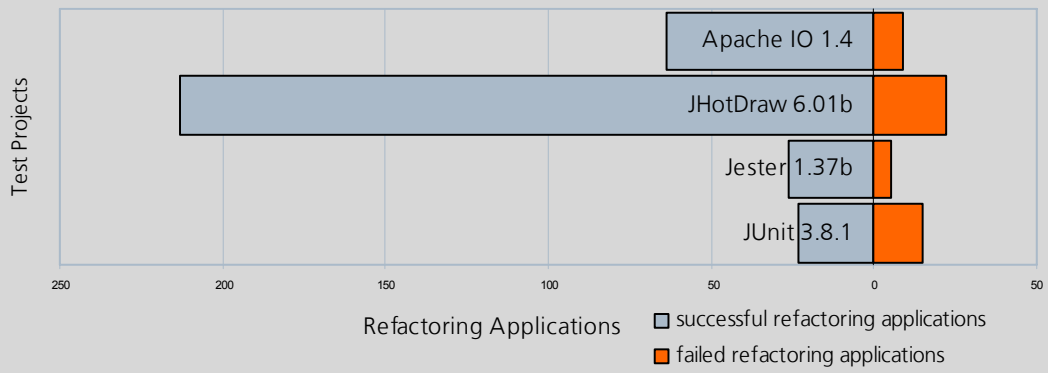
➢ Large **Open Source Projects** are the most realistic scenario for testing refactoring tools because they are representative of the real world of software development.

The Refactoring Tool Tester tests refactoring tools systematically by automatically applying refactorings to large open source code bases, and checking their unchanged behavior.

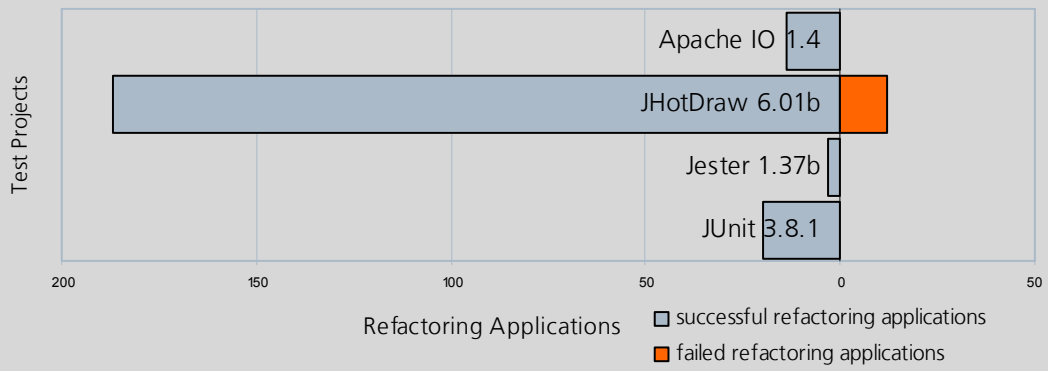
➢ While adapters and open source projects provide the test input, the compiler and the unit tests of the projects provide the **Oracle**:
After application of a refactoring by the Refactoring Tool Tester, the sample program must still compile, and its test cases must still pass.

Test Results

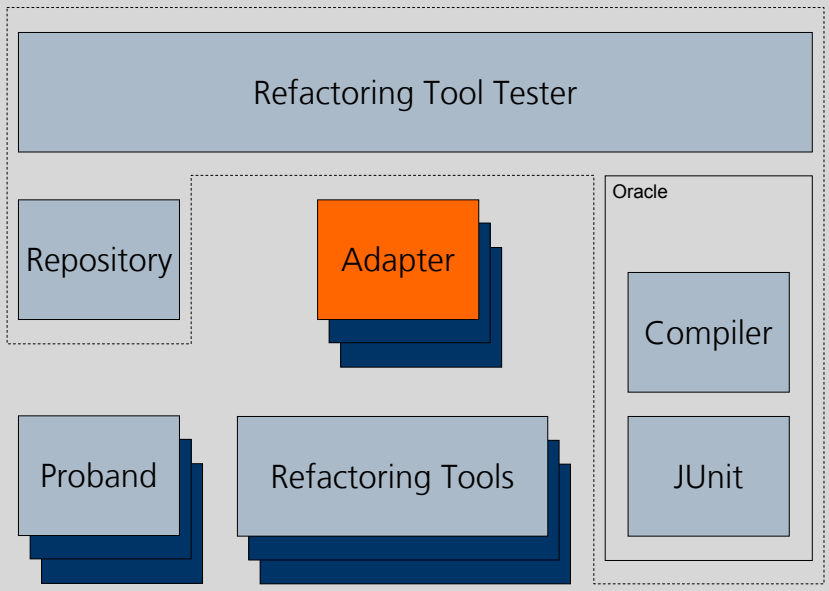
Testing the JDT Move Class Refactoring Tool



Testing the JDT Pull Up Method Refactoring Tool



Tool Architecture



Related Approaches

Formal proofs of correctness are desirable – but seem infeasible for refactoring tools. A formal proof needs a complete specification of the desired behavior which in the specific case of refactoring requires a complete spec of the target language.

A promising approach for **automated testing** was presented by Daniel et al. [1]. Their ASTGen-Framework offers comfortable libraries to write *imperative generators* whose execution results in abstract syntax trees used as test inputs for refactoring tools. ASTGen helped to uncover dozens of bugs in current refactoring tools. But again, when implementing the imperative generator the programmer must be aware what he is testing for. E.g. when testing our own refactoring tools with ASTGen, we did not recognize the relevance of @Override annotations for the *RIWD*-Refactoring [2] – so the imperative generator did not produce any annotations – leaving a corresponding bug undiscovered. When we used the Refactoring Tool Tester, the bug emerged immediately – because of the frequent usage of annotations in open source projects.

Evaluation

The Refactoring Tool Tester's **results are promising**. The Tester helped us to find bugs in existing refactoring tools as well as to confirm results of ongoing research on refactoring tools: In [4] we developed a set of constraints to respect accessibility during the refactoring process. By testing refactoring tools equipped with our constraints we discovered the need for several constraints we had failed to derive from the language spec.

Nevertheless, there are cases in which the Refactoring Tool Tester's approach has disadvantages. While it turns out to test refactoring tools with a small set of possible parameters quite efficiently it is rather inefficient when parameters may be chosen almost arbitrarily. A refactoring tool pulling up methods or moving classes may be tested exhaustively by the Refactoring Tool Tester as the number of possible locations for each declaration is small. The picture changes for refactorings requiring a new identifier, such as *extract method* or *rename*. Here, exhaustive testing of all possible identifiers is futile. Even though a refactoring adapter may limit the set of used identifiers to an adequate subset (e.g. to provoke name clashes), it may also ignore relevant cases – limiting the use of the refactoring tool tester's advantages.

References

[1] B. Daniel, D. Dig, K. Garcia, D. Marinov. "Automated testing of refactoring engines" in: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2007)
[2] H. Kegel, F. Steimann. "Systematically refactoring inheritance to delegation in Java" in *Proceedings of the 30th international conference on Software engineering* (2008) 431–440.
[3] The Eclipse Language Toolkit (LTK): http://www.eclipse.org/articles/Article-LTK/ltk.html
[4] F. Steimann, A. Thies. "From public to private to absent: Refactoring java programs under constrained accessibility" in: *Proceedings of 23rd European Conference on Object-Oriented Programming* (2009) 419 – 443.

Contact

Andreas Thies · Friedrich Steimann
Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
FernUniversität in Hagen
D-58084 Hagen

andreas.thies@fernuni-hagen.de · steimann@acm.org