Automated Generation of Unit Tests for Refactoring

Bartosz Walter and Błażej Pietrzak

Institute of Computing Science, Poznań University of Technology, Poland
{Bartosz.Walter, Blazej.Pietrzak}@cs.put.poznan.pl

Abstract. The key issue for effective refactoring is to ensure that observable behavior of the code will not change. Use of regression tests is the suggested method for verifying this property, but they appear to be not well suited for applying refactorings. The tests verify the domain-specific code relations, and often neglect properties important for the given refactoring or check the ones that actually change. In the paper we present a concept of generic refactoring tests, which are oriented toward the refactorings rather than the code. They can be generated automatically, are able to adapt to changes introduced by refactorings and learn the specifics of the tested code.

Keywords. Refactoring, unit testing, automation

1 Introduction

Software refactoring is a key to agile software maintenance [1]. It not only helps to find bugs more easily, but, what is more important, to keep it readable and ready for changes [3]. If applied regularly, it benefits in shorter learning curve and easy accommodation of possible changes. However, the necessary condition for effective refactoring is to ensure its correctness. Any modification that may introduce new bugs to the software is more dangerous than untidy code: it requires additional time wasted on debugging the formerly working code.

Unit testing [2] is commonly suggested as the primary method of verification for refactoring. Typical test cases are used to check if domain-specific relations in the code persist and bug has been re-injected . Refactorings, however, aim at different goal. Although they have to preserve the software functionality (and the regression tests as well), but they also introduce new, independent from the business domain properties that must be satisfied or break some of some existing ones. Effectively, many of ordinary test cases fail after refactoring, and some other need to be created. The considerable effort required to adjust the testing suite at every single refactoring indicates that unit tests are inappropriate for refactoring. There is a need for tests suited exclusively for refactoring that would fix the deficiencies of unit tests.

In the paper we present a concept of refactoring tests, which are intended to ease the process of testing the refactorings. These tests are created exclusively for the refactoring purposes and check the properties specific for the transformation being applied. In subsequent sections we describe the requirements for them and their suggested implementation.

J. Eckstein and H. Baumeister (Eds.): XP 2004, LNCS 3092, pp. 211–214, 2004.

2 Concept of Refactoring Tests

Unit testing plays an important role in refactoring. It comes from the observation that the testing suite collected during the everyday development is a powerful tool preventing from unwanted functional changes[1]. According to the *green bar rule* [2], all tests must run every time they are executed, so they act like anchors: protect the ship (code) from drifting out. Since XP assumes that tests are developed along with code, refactoring based on them should be relatively inexpensive and error-resistant.

Unfortunately, it is not always the case. The belief is based on an optimistic assumption that the testing suite is complete and consistent. Actually, it is not the number of tests that matters; it depends rather on the invariants being tested.

There are a few reasons why ordinary test cases are not well suited for refactoring:

- **They are developed for the code**, and therefore are focused on retaining relations specific to that code and, consequently, are closely related to it. Refactorings, however, may require different properties from those in the business area.
- The primary use of unit tests is to protect the code from injecting regressive bugs. In other words, every test **remains unchanged and verifies the same invariant** forever. Refactoring, while preserving some code properties, breaks other, and also break the unit tests. This is misleading for programmers.
- Unit tests used for refactoring purposes are assumed to **run both before and after the change is made** to verify if a given property still exists. However, in two cases the tests are legitimate to fail after the code is refactored: either the tested property is changed or removed, or is expressed now in a way that breaks a test.
- **Unit tests usually do not change**. On the other hand, refactoring enforces changes in the code (and in tests as well) not only to accommodate simple renames or replacements, but it may also introduce semantic changes.

These deficiencies show that there is a need for tests suited exclusively for refactoring, which would overcome some of the limitations mentioned above. The requirements for the proposed *refactoring tests* follow:

- **Refactoring tests should be related to a given refactoring**, not to the code. They represent invariants and properties resulting directly from the transformation.
- **They are logically independent from the ordinary, domain-oriented test cases, and are executed only while applying the refactoring**. Obviously, the refactoring tests can be later used for regression testing, but it is not the primary use of them.
- However, **their interface should be compliant with unit tests**. It allows running all tests in the same environment.
- **Refactoring tests may change during the transformation**. Unit tests are expected to run regardless from the code changes, and the refactoring tests should also comply with that rule.

The aim for the refactoring tests is to support the code analysis in verifying correctness of the source code transformations. Several preconditions for many refactorings cannot be proved statically, and the gap may be filled by refactoring tests. Obviously, they only provide support for detecting functional changes, but usually it appears sufficient for effective programming. Among refactorings provided by Fowler there are a few that can be verified with a well-known, fixed sets of tests [5]. Refactorings of that group are intended beneficiaries of the proposed refactoring tests.

3 Test Adaptation Mechanism

Although refactoring tests verify well-known, fixed properties in the code affected by the transformation, it is still the programmer who is responsible for writing them, which requires considerable effort.

Fortunately, it can be significantly minimized. A refactoring test actually builds up on two pillars: (1) a property it verifies and (2) actual code-related data: class names, methods, interfaces etc. Such generic, code independent tests are called *test templates*. Unlike an ordinary test, a test template has to be implemented once only and later can be reused for generating actual, code-specific tests.

The need for tests adaptation is another issue. As an example, let's consider *Encapsulate Collection* refactoring [3]. It replaces direct access to a collection with delegate methods provided by the collection owner. After the refactoring is complete, the destructive methods on the collection are disabled. Instead, newly created delegating methods become available. A test case for *add()* method, although syntactically correct, would therefore fail for the refactored code. To avoid this, the refactoring tests should seamlessly adapt to the changed interface.

The proposed mechanism assumes that a single refactoring test is actually composed of two test case sets: *pre-tests* and *post-tests*, executed before and after the change is made, respectively. Since they also can be expressed as test templates, the doubling the number of tests is meaningless. The subtests are invisible from outside: the refactoring test implements the *Strategy* pattern [4] and chooses one of them for execution, depending on the phase of refactoring.

It is important to notice that the multiplicity of the pre- and post-tests may differ from 1:1. Coming back to the *Encapsulate Collection* example, a pre-test for *add()* method checks if an object is successfully added to the collection. The refactoring, however, is expected to disable direct *add()* calls on collection and produce a new method *addClass()* in the owner class, that will take its responsibility. Thus, two post-tests are required: one for the existing *add()* method (that should leave collection unchanged) and another for a newly created method *addClass()* (which should succeed).

4 Learning Tests

The test templates are sufficient for generating simple tests. However, some properties of the tested code that may lead to differences in results of pre- and post-tests, are related to the specific implementation. They cannot be hardcoded in templates and must be determined at runtime. To avoid running the software, a new kind of pre-tests is introduced: *the learning tests*.

Unlike other pre-tests they are not required to pass. Their sole responsibility is to learn specific implementations by sampling the code if it behaves in a particular way. A failure is not an error indication, but a source of information about the code. The acquired results are used for generating post-tests that better fit the code internals. Learning tests help to ensure that the post-tests will expect the same behavior as the one that program presented before the transformation.

Again, the *add()* method in *Collection* interface will serve as an example. Different implementations of that interface may vary, and the programmer does not need to know how particular *add()* method in the code acts. S/he rather expects the code to preserve the behavior after the refactoring is complete. To make the post-tests immune from possible failures at inserting *null* values, a learning test checks if the collection accepts *null*s. Depending on the result, the post-test considers the failure of the same operation as an error (if pre-test succeeded) or a success (if it failed before).

The overall algorithm for using the refactoring tests is following:

1. Programmer generates pre-tests and learning-tests from existing templates for that transformation. The tests are then executed and their results are stored.
2. Programmer refactors (manually or automatically) the code.
3. The post-tests are generated and executed. The results are compared to the pre-tests footprint and evaluated. Any difference indicates that refactoring was incorrect.

5 Conclusions

The presented concept of refactoring tests has been developed as a plug-in for Eclipse platform for *Encapsulate Collection* refactoring. It is a step towards automation of refactoring in the area that previously was subject to manual-only manipulations. They are created and executed for the sake of a given refactoring, not for regression purposes, which makes them different from ordinary unit tests. They exploit the idea of test templates, which relax the programmer from coding them. Refactoring tests act like proxies for pre-tests and post-tests, which allow keeping them working regardless of the changes resulting from a refactoring. And finally, they can learn the program specific implementation and adapt to its expected behavior. Thanks to these features, testing the refactorings can be easier and less error-prone.

Acknowledgements. This work has been supported by the Polish State Committee for Scientific Research as a part of the research grant KBN/91-0824.

References

1. Beck K.: Extreme Programming Explained. Embrace Change. Addison-Wesley, 2000.
2. Beck K.: Gamma E.: Test infected: Programmers love writing tests. Java Report, 3(7), 1998, pp. 51-56.
3. Fowler M.: Refactoring. Improving the Design of Existing Code. Addison-Wesley, 1999.
4. Gamma E. et al.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
5. Walter B.: Extending Testability for Automated Refactoring, in: Succi G., Marchesi M. (Eds.): Extreme Programming and Agile Processes in Software Engineering, Lecture Notes in Computer Science 2675, Springer Verlag, 2003, pp. 429-430.