Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Refactoring test code

A. van Deursen, L.M.F. Moonen, A. van den Bergh, G. Kok

Software Engineering (SEN)

SEN-R0119 July 31, 2001

Report SEN-R0119
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Refactoring Test Code

Arie van Deursen

Leon Moonen

http://www.cwi.nl/∼{arie,leon}/
{arie,leon}@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Alex van den Bergh

Gerard Kok

http://www.software-improvers.com/
{alex,gerard}@software-improvers.com

Software Improvement Group
Kruislaan 419, 1098 VA Amsterdam, The Netherlands

ABSTRACT

Two key aspects of extreme programming (XP) are unit testing and merciless refactoring. Given the fact that the ideal test code / production code ratio approaches 1:1, it is not surprising that unit tests are being refactored. We found that refactoring test code is different from refactoring production code in two ways: (1) there is a distinct set of bad smells involved, and (2) improving test code involves additional test-specific refactorings. To share our experiences with other XP practitioners, we describe a set of bad smells that indicate trouble in test code, and a collection of test refactorings to remove these smells.

2000 ACM Computing Classification System: D.2.3, D.2.5, D.2.7, D.2.9, K.6.3
Keywords and Phrases: Refactoring, unit testing, extreme programming
Note: Published in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, 20–23 May 2001, Cagliari, Italy.
Note: Work carried out under projects SEN-1.1, *Software Renovation* and SEN-1.2, *Domain-Specific Languages*.

1. INTRODUCTION

"If there is a technique at the heart of *extreme programming* (XP), it is unit testing" [1]. As part of their programming activity, XP developers write and maintain (white box) unit tests continually. These tests are automated, written in the same programming language as the production code, considered an explicit part of the code, and put under revision control.

The XP process encourages writing a test class for every class in the system. Methods in these test classes are used to verify complicated functionality and unusual circumstances. Moreover, they are used to document code by explicitly indicating what the expected results of a method should be for typical cases. Last but not least, tests are added upon receiving a bug report to check for the bug and to check the bug fix [2]. A typical test for a particular method includes: (1) code to set up the fixture (the data used for testing), (2) the call of the method, (3) a comparison of the actual results with the expected values, and (4) code to tear down the fixture. Writing tests is usually supported by frameworks such as *JUnit* [3].

The test code / production code ratio may vary from project to project, but is ideally considered to approach a ratio of 1:1. In our project we currently have a 2:3 ratio, although others have reported a lower ratio.[1] One of the corner stones of XP is that having many tests available helps the developers to overcome their fear for change: the tests will provide immediate feedback if the system gets broken at a critical place. The downside of having many tests, however, is that changes in functionality will typically involve changes in the test code as well. The more test code we get, the more important it becomes that this test code is as easily modifiable as the production code.

The key XP practice to keep code flexible is “refactor mercilessly”: transforming the code in order to bring it in the simplest possible state. To support this, a catalog of *code smells* and a wide range of refactorings is available, varying from simple modifications up to ways to introduce design patterns systematically in existing code [5].

When trying to apply refactorings to the test code of our project we discovered that refactoring test code is different from refactoring production code. Test code has a distinct set of smells, dealing with the ways in which test cases are organized, how they are implemented, and how they interact with each other. Moreover, improving test code involves a mixture of refactorings from [5] specialized to test code improvements, as well as a set of additional refactorings, involving the modification of test classes, ways of grouping test cases, and so on.

The goal of this paper is to share our experience in improving our test code with other XP practitioners. To that end, we describe a set of *test smells* indicating trouble in test code, and a collection of *test refactorings* explaining how to overcome some of these problems through a simple program modification. This paper assumes some familiarity with the xUnit framework [3] and refactorings as described by Fowler [5]. We will refer to refactorings described in this book using *Name (F:page#)* and to our test specific refactorings described in section 3 using *Name* (#).

2. TEST CODE SMELLS

This section gives a overview of bad code smells that are specific for test code.

Smell 1: *Mystery Guest*.
When a test uses external resources, such as a file containing test data, the test is no longer self contained. Consequently, there is not enough information to understand the tested functionality, making it hard to use that test as documentation. Moreover, using external resources introduces hidden dependencies: if some force changes or deletes such a resource, tests start failing. Chances for this increase when more tests use the same resource. The use of external resources can be eliminated using the refactoring *Inline Resource (1)*. If external resources are needed, you can apply *Setup External Resource (2)* to remove hidden dependencies.

Smell 2: *Resource Optimism*.
Test code that makes optimistic assumptions about the existence (or absence) and state of external resources (such as particular directories or database tables) can cause non-deterministic behavior in test outcomes. The situation where tests run fine at one time and fail miserably the other time is not a situation you want to find yourself in. Use *Setup External Resource (2)* to allocate and/or initialize all resources that are used.

Smell 3: *Test Run War*.
Such wars arise when the tests run fine as long as you are the only one testing but fail when more programmers run them. This is most likely caused by resource interference: some tests in your suite allocate resources such as temporary files that are also used by others. Apply *Make Resource Unique (3)* to overcome interference.

Smell 4: *General Fixture*.
In the JUnit framework a programmer can write a setUp method that will be executed before each test method to create a fixture for the tests to run in.

Things start to smell when the setUp fixture is too general and different tests only access part of the fixture. Such setUps are harder to read and understand. Moreover, they may make tests run more slowly (because they do unnecessary work). The danger of having tests that take too much time to complete is that testing starts interfering with the rest of the programming process and programmers eventually may not run the tests at all.

---

[1] This project started a year ago and involves the development of a product called DocGen [4]. Development is done by a small team of five people using XP techniques. Code is written in Java and we use the JUnit framework for unit testing.

The solution is to use setUp only for that part of the fixture that is shared by all tests using Fowler's *Extract Method (F:110)* and put the rest of the fixture in the method that uses it using *Inline Method (F:117)*. If, for example, two different groups of tests require different fixtures, consider setting these up in separate methods that are explicitly invoked for each test, or spin off two separate test classes using *Extract Class (F:149)*.

Smell 5: *Eager Test*.
When a test method checks several methods of the object to be tested, it is hard to read and understand, and therefore more difficult to use as documentation. Moreover, it makes tests more dependent on each other and harder to maintain.

The solution is simple: separate the test code into test methods that test only one method using Fowler's *Extract Method (F:110)*, using a meaningful name highlighting the purpose of the test. Note that splitting into smaller methods can slow down the tests due to increased setup/teardown overhead.

Smell 6: *Lazy Test*.
This occurs when several test methods check the same method *using the same fixture* (but for example check the values of different instance variables). Such tests often only have meaning when considering them together so they are easier to use when joined using *Inline Method (F:117)*.

Smell 7: *Assertion Roulette*.
"Guess what's wrong?" This smell comes from having a number of assertions in a test method that have no explanation. If one of the assertions fails, you do not know which one it is. Use *Add Assertion Explanation (5)* to remove this smell.

Smell 8: *Indirect Testing*.
A test class is supposed to test its counterpart in the production code. It starts to smell when a test class contains methods that actually perform tests on other objects (for example because there are references to them in the class-to-be-tested). Such indirection can be moved to the appropriate test class by applying *Extract Method (F:110)* followed by *Move Method (F:142)* on that part of the test. The fact that this smell arises also indicates that there might be problems with data hiding in the production code.

Note that opinions differ on indirect testing. Some people do not consider it a smell but a way to guard tests against changes in the "lower" classes. We feel that there are more losses than gains to this approach: It is much harder to test anything that can break in an object from a higher level. Moreover, understanding and debugging indirect tests is much harder.

Smell 9: *For Testers Only*.
When a production class contains methods that are only used by test methods, these methods either (1) are not needed and can be removed, or (2) are only needed to set up a fixture for testing. Depending on functionality of those methods, you may not want them in production code where others can use them. If this is the case, apply *Extract Subclass (F:330)* to move these methods from the class to a (new) subclass in the test code and use that subclass to perform the tests on. You will often find that these methods have names or comments stressing that they should only be used for testing.

Fear of this smell may lead to another undesirable situation: a class without corresponding test class. The reason then is that the developer (1) does not know how to test the class without adding methods that are specifically needed for the test and (2) does not want to pollute his production class with test code. Creating a separate subclass helps to deal with this problem.

Smell 10: *Sensitive Equality*.
It is fast and easy to write equality checks using the toString method. A typical way is to compute an actual result, map it to a string, which is then compared to a string literal representing the expected value. Such tests, however may depend on many irrelevant details such as commas, quotes, spaces, etc. Whenever the toString method for an object is changed, tests start failing. The solution is to replace toString equality checks by real equality checks using *Introduce Equality Method (6)*.

Smell 11: *Test Code Duplication.*
Test code may contain undesirable duplication. In particular the parts that set up test fixtures are susceptible to this problem. Solutions are similar to those for normal code duplication as described by Fowler [5, p. 76]. The most common case for test code will be duplication of code in the same test class. This can be removed using *Extract Method (F:110)*. For duplication across test classes, it may provide helpful to mirror the class hierarchy of the production code into the test class hierarchy. A word of caution however: moving duplicated code from two separate classes to a common class can introduce (unwanted) dependencies between tests.

A special case of code duplication is *test implication*: test A and B cover the same production code, and A fails if and only if B fails. A typical example occurs when the production code gets refactored: before this refactoring, A and B covered different code, but afterwards they deal with the same code and it is not necessary anymore to maintain both tests.

3. REFACTORINGS

Bad smells seem to arise more often in production code than in test code. The main reason for this is that production code is adapted and refactored more frequently, allowing these smells to escape.

One should not, however, underestimate the importance of having fresh test code. Especially when new programmers are added to the team or when complex refactorings need to be performed, clear test code is invaluable. To maintain this freshness, test code also needs to be refactored.

We define *test refactorings* as changes (transformations) of test code that: (1) do not add or remove test cases, and (2) make test code better understandable/readable and/or maintainable.

The production code can be used as a (simple) test case for the refactoring: If a test for a piece of code succeeds before the test refactoring, it should also succeed after the refactoring (and no, replacing all test code by `assert(true)` is not considered a valid refactoring). This obviously also means that you should not modify production code while refactoring test code (similar to not changing tests when refactoring production code).

While working on our test code, we encountered the following refactorings:

Refactoring 1: *Inline Resource.*
To remove the dependency between a test method and some external resource, we incorporate that resource in the test code. This is done by setting up a fixture in the test code that holds the same contents as the resource. This fixture is then used instead of the resource to run the test. A simple example of this refactoring is putting the contents of a file that is used into some string in the test code.

If the contents of the resource are large, chances are high that you are also suffering from *Eager Test* (5) smell. Consider conducting *Extract Method (F:110)* or *Reduce Data (4)* refactorings.

Refactoring 2: *Setup External Resource.*
If it is necessary for a test to rely on external resources, such as directories, databases, or files, make sure the test that uses them explicitly creates or allocates these resources before testing, and releases them when done (take precautions to ensure the resource is also released when tests fail).

Refactoring 3: *Make Resource Unique.*
A lot of problems originate from the use of overlapping resource names, either between different tests run done by the same user or between simultaneous test runs done by different users.

Such problems can easily be prevented (or repaired) by using unique identifiers for all resources that are allocated, for example by including a time-stamp. When you also include the name of the test responsible for allocating the resource in this identifier, you will have less problems finding tests that do not properly release their resources.

Refactoring 4: *Reduce Data.*
Minimize the data that is setup in fixtures to the bare essentials. This will have two advantages: (1) it make them better suitable as documentation, and (2) your tests will be less sensitive to changes.

Refactoring 5: *Add Assertion Explanation.*
Assertions in the JUnit framework have an optional first argument to give an explanatory message to the user when the assertion fails.

Testing becomes much easier when you use this message to distinguish between different assertions that occur in the same test. Maybe this argument should not have been optional?

Refactoring 6: *Introduce Equality Method*.
If an object structure needs to be checked for equality in tests, add an implementation for the “equals” method for the object’s class. You then can rewrite the tests that use string equality to use this method. If an expected test value is only represented as a string, explicitly construct an object containing the expected value, and use the new equals method to compare it to the actually computed object.

4. RELATED WORK

Fowler [5] presents a large set of bad smells and refactorings that can be used to remove them. The difference between his work and ours is that we focus on smells and refactorings that are typical for test code whereas his book focuses more on production code. The role of unit tests in [5] is also more geared towards proving that a refactoring didn’t break anything than to be used as documentation of the production code.

Instead of focusing on cleaning test code which already has bad smells, Schneider [6] describes how to prevent these smells right from the start by discussing a number of best practices for writing tests with JUnit.

The C2 Wiki contains some discussion on the decay of unit test quality and practice as time proceeds[2], and on the maintenance of broken unit tests[3] Opinions vary between repairing broken unit tests, deleting them completely, and moving them to another class in order to make them less exposed to changes (which may lead to our *Indirect Testing* (8) smell).

5. CONCLUSIONS

In this paper, we have looked at test code from the perspective of refactoring. While working on our XP project, we observed that the quality of the test code was not as high as the production code. Test code was not refactored as mercilessly as our production code, following Fowler’s advice that it is okay to copy and edit test code, trusting our ability to refactor out truly common items later[5, p. 102]. When at a later stage we started to refactor test code more intensively, we discovered that test code has its own set of problems (which we translated into smells) as well as its own repertoire of solutions (which we formulated as test refactorings).

The contributions of this paper are the following:

- We have collected a series of test smells that help developers to identify weak spots in their test code;
- We have composed a set of specific test refactorings enabling developers to make improvements to their test code in a systematic way;
- For each smell we have given a solution, using either a potentially specialized variant of an existing refactoring from [5] or one of the dedicated test refactorings.

The purpose of this paper is to share our experience in refactoring test code of our ongoing XP project with other XP practitioners. We believe that the resulting smells and refactorings provide a valuable starting point for a larger collection based on a broader set of projects. Therefore, we would like to invite readers interested in further discussion on this topic to the C2 Wiki[4].

An open question is how test code refactoring interacts with the other XP practices. For example, the presence of test code smells may indicate that your production code has some bad smells. So trying to refactor test code may indirectly lead to improvements in production code. Furthermore, refactoring test code may reveal missing test cases. Adding those to your framework will lead to a more complete test coverage of the production code. Another question is at what moments in the XP process test refactorings should be applied. In short, the precise interplay between test refactoring and the XP practices is a subject of further research.

---

[2] http://c2.com/cgi/wiki?TwoYearItch
[3] http://c2.com/cgi/wiki?RefactorBrokenUnitTests
[4] http://c2.com/cgi/wiki?RefactoringTestCode

References

1. K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, October 1999.
2. K. Beck. *Extreme Programming Explained. Embrace Change*. Addison Wesley, 1999.
3. K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.
4. A. van Deursen, T. Kuipers, and L. Moonen. Legacy to the extreme. In *Extreme Programming and Flexible Processes in Software Engineering; Proceedings XP2000*. Addison-Wesley, 2001. To appear.
5. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
6. A. Schneider. JUnit best practices. *Java World*, 12, 2000. `http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-junit.html`.