Refactoring Test Code Safely

Eduardo Martins Guerra
Aeronautical Institute of Technology
guerraem@gmail.com

Clovis Torres Fernandes
Aeronautical Institute of Technology
clovistf@uol.com.br

Abstract

Test Driven Development is a technique in which the refactoring occurs all the time, in the application code and in the test code. But there is not a method to guarantee that the test code behavior after one refactoring remains unchanged. This paper presents a representation based on the JUnit unit test structure, as well as a classification of test code refactorings that may ease the analysis to verify if the test code refactoring was carried out safely, i.e., if the observable behavior of the refactored test code has been kept unchanged. The use of this proposed technique may safely improve and speed up the production of test code refactorings.

1. Introduction

Test Driven Development (TDD) is a software development technique where the unit tests are coded before the application code [1]. Refactoring is another development technique proved useful to change non-functional characteristics of a code, like quality and readability, without changing its behavior [2]. Refactoring is used by TDD developers to eliminate duplicated, unclear, and complicated code, besides allowing design evolution step by step in the application code.

In this scenario, where the test code is very important, the test code refactoring is also important to provide flexibility in the unit tests. Refactoring test code is different from refactoring application code [3], but studies about this topic are based only on practical experiences, without identifying clearly why the test code refactoring is different or how to do it safely [4].

This paper presents the result of a research about test code refactoring, whose objective was identify what is the test code behavior and how to refactor it in a safe way. To achieve this goal, a graphical representation for the unit test structure was developed and used in the analysis to verify if the observable behavior of the refactored test code has been kept unchanged. A classification of test code refactorings by scope is presented and has been also proved useful to differentiate some distinctive kinds of test code refactorings and how the observable behavior of test code refactorings should be safely verified.

The Section 2 explains the difference between test code refactoring and application code refactoring. The Section 3 shows the test code structure and Section 4 presents the graphical notation to represent test suites. The Section 5 defines the concept of verification, which is the base for the equivalence of test suites. Section 6 presents a classification of test refactorings and how to verify the behavior maintenance for each kind. Section 7 presents an analysis on related work. Section 8, 9 and 10 presents some results of this work: respectively a test code refactoring catalog, a refactoring Eclipse plugin for test code and practical results in a real project. Section 11 summarize this paper’s main contributions and show some possibilities of future studies.

2. Test code behavior

When a refactoring is done in an application code, the unit tests can be used to check if the behavior of the refactored code has been kept unchanged. But when the refactoring is made in a test code, the concept of behavior is different from the application code and there is no sense in creating tests to verify the test code behavior. The application code behavior are the effects of its execution, like return values, changes in variables and impacts in external resources, like files or databases. In a unit test, to verify the behavior of an application code piece, an initial context is developed. Then the tested functionality is performed and the effects of its execution are checked by using assertion functions.

A unit test should not cause any collateral effect on the application code after its execution, because the test code meaning is to make verifications in the application code. To guarantee a safely test code refactoring, there should be a way to keep the same verifications made in the application code. To define the concept of behavior in a test code, first one should
study the test suite structure and define what is a verification. Using this concept, it can be made analyzes to verify if a test code refactoring changes the verifications; in other words, to verify if it changes the unit test behavior. Refactoring test code safely means that the results of applying the same test suite before and after refactoring a test code must be kept unchanged.

3. Test suite structure

The goal of this section is to describe each part of a test suite. A representative test class will be used to exemplify all the relevant parts in a test suite. The Java programming language and the JUnit framework structure will be used for this study [5]. In Figure 1 there is a representative unit test and next there is a list of the relevant parts of a test suite:

- **Test Target** – It is the instance or class with the behavior that will be verified by the unit tests. In the example, the test target is the instance variable, or fixture, 'obj' defined in line 03.
- **Action** – It is any object manipulation in the test code that is not an assertion. There are examples of actions in the lines 09 and 13.
- **Assertion** – It is the comparison between the expected behavior and the obtained behavior, which does not make any change in the test target. There are examples of assertions in the lines 10 and 14.
- **Initialization** – It is a group of actions executed before each unit test in a test class. These actions can be considered part of all unit tests in the test class. The initialization method is declared in the line 05.
- **Finalization** – It is a group of actions executed after each unit test in a test class. These actions can be considered part of all unit tests in the test class. The finalization method is declared in the line 16.
- **Unit Test** – It is a group of at least one action and at least one assertion in a certain order. The test methods ‘testModifyObj1()’ and ‘testModifyObj2()’, declared respectively in the lines 08 and 12, are example of unit tests. The actions in the initialization and in the finalization, in the example in the lines 06 and 17, are parts of each unit test in the class. Inside a unit test all the actions and assertions are sequentially executed and the execution order is important for the verifications made in a unit test.
- **Test Suite** – It is a group of unit tests executed independently. A test class can be considered a test suite, but a test suite can be also a group of test classes. Inside a test suite the unit tests are executed independently and the execution order does not matter.

```java
01. public class TestTestedObj extends TestCase {
02.     private TestedObj obj;
03.     protected void setUp() throws Exception {
04.         obj = new TestedObj();
05.     }
06.     public void testModifyObj1(){
07.         obj.modify1();
08.         assertEquals(obj.getValue(),"Value 1");
09.     }
10.     public void testModifyObj2(){
11.         obj.modify2();
12.         assertEquals(obj.getValue(),"Value 2");
13.     }
14.     protected void tearDown() throws Exception {
15.         obj.releaseResources();
16.     }
17. }
```

Figure 1 - Source code for a representative unit test.

4. Test suite representation

To simplify the representation of test suites, both a formal and graphical notation were developed. In this paper only the graphical notation is presented. This graphical notation represents a test suite showing explicitly its parts, turning easier the structure manipulation to refactor the test code. The description of the notation can be followed observing the examples presented in Figure 2.

Figure 2 - Test suite representation.
The test target is represented by a letter “O” followed by a subscript number to identify different targets. An action is represented using a square and an assertion is represented using a circle, and both have inside the targets affected. Colored and graphical patterns are used to distinguish one action or assertion from the others. One unit test is represented by the actions and assertions disposed horizontally in order and linked by a line. A test suite is represented by a box with all unit tests inside. The initialization and the finalization actions are linked with a line with the test suite box respectively in the left and in the right.

5. Verification concept

All concepts presented previously can be easily identified in the test suite representation. The verification concept are more abstract and it represents a check made in the application code by the test code. A unit test can have many verifications which on the other hand represent its behavior.

A verification is composed by one assertion and all the actions that appear before the assertion that deals with the same assertion’s test target. For a better understanding, the Figure 3 shows a unit test representation and the identification of three verifications.

![Figure 3 - An example of how to identify verifications.](image)

With the verification concept, can be defined the test suite equivalence. If two test suites do the same verifications in the application code, those suites can be considered equivalents. So, if one test suite is refactored in an equivalent test suite, this refactoring can be considered safe, because the test code behavior has been kept unchanged. In Figure 4 there is an example of test suite equivalence. In the first test suite there is just the unit test represented in the Figure 3. In the second test suite, there are tree unit tests, one for each verification identified. The equivalence can be confirmed because both test suites have the same verifications.

![Figure 4 - An example of two equivalents test suites.](image)

6. Kinds of test code refactorings

In the study of test code refactoring, tree different kinds of refactoring were identified. The criterion for this classification is the refactoring scope. The tree kinds of refactorings are the following:

- Refactoring inside a test method
- Refactoring inside a test class
- Structural refactoring of test classes.

This classification is important for the analysis of the observable behavior of test codes, because each kind of test code refactoring has a different way to verify the equivalence of the original test suite and the refactored test suite.

The following subsections describe each kind of test refactoring along with an example, for a better comprehension. The refactorings showed as examples are just for illustration and will not be explained in detail.

6.1. Refactoring inside a test method

The simplest kind of refactoring in a test code is the one that occurs inside a test method. In this kind of refactoring, there is a substitution of an action or an assertion for equivalent ones. The main goal of these kind of refactoring is to simplify the test scenario and improve the readability of the test code.

In Figure 5, there is an example of a refactoring named here Decompose Assertion. In this refactoring, a composed assertion is decomposed in two or more simple assertions. In Figure 6 there is a representation of the original unit test and the refactored unit test, using the notation defined in Session 3.

To verify if the behavior has been kept unchanged after the refactoring, the new actions should have the
same effect over the test target and the new assertions should do the same checks on the application code. Running the test suite and having all the tests executing successfully is a good indicator that the refactoring does not have changed the test behavior. But it should be used carefully, because it’s not 100% guaranteed.

6.2. Refactoring inside a test class

This kind of refactoring deals with elements of a test class, such as test methods, initialization, finalization and fixtures. In these refactorings, as the example illustrated in Figure 7, the elements are rearranged inside the test class to avoid any kind of code duplication and make it easier to add new tests. The example in Figure 7 illustrates the refactoring Join Incremental Tests. In this refactoring, many methods that makes verifications in an incremental way are joined into a single one, removing the code duplication. In Figure 8, using the graphical notation, it is represented the equivalent test suites that represents the test code before and after refactoring.

The behavior verification in this kind of test code refactoring is made using the concept of verification. As this kind of refactoring does not change the actions and assertions, but just reorganize them inside the test class, if the refactored test suite is equivalent to the original test suite, than the behavior is considered unchanged.

```java
public void testObjEmpty(){
    TestedObj obj = new TestedObj();
    assertEquals("Test 0",obj.getSize(),0);
}
public void testObjAddOne(){
    TestedObj obj = new TestedObj();
    obj.add(1);
    assertEquals("Test 1",obj.getSize(),1);
}
public void testObjAddTwo(){
    TestedObj obj = new TestedObj();
    obj.add(1);
    obj.add(1);
    assertEquals("Test 2",obj.getSize(),2);
}
public void testObjAdd(){
    TestedObj obj = new TestedObj();
    assertEquals("Test 0",obj.getSize(),0);
    obj.add(1);
    assertEquals("Test 1",obj.getSize(),1);
    obj.add(1);
    assertEquals("Test 2",obj.getSize(),2);
}
```

Figure 7 – An example of Join Incremental Tests refactoring.

```
public void testObjModifyObj(){
    TestedObj obj = new TestedObj();
    obj.modify();
    assertTrue("Composed Assertion",
        obj.isModified() && obj.getValue() == 0);
}
public void testObjModifyObj(){
    TestedObj obj = new TestedObj();
    obj.modify();
    assertTrue("Modified", obj.isModified());
    assertEquals("Comparison",obj.getValue(),0);
}
```

Figure 5 - An example of Decompose Assertion refactoring.

```
Figure 6 - Representation of the original and the refactored test suite.

6.3. Structural refactoring of test classes

This kind of refactoring acts in another level of abstraction, dealing with inheritance of test classes and location of test methods. The goal of this kind of refactoring is to organize the unit tests in the test class
hierarchy to minimize the code duplication and ease the addition of new tests.

Usually, this kind of refactoring is needed after a refactoring in the application code. One example where it occurs is the refactoring that a superclass is created and a method is pulled up. In spite of the unit tests still execute successfully, the test class structure is not the most appropriate.

Sometimes the refactorings made in the production code should be extended to the test code [6]. It is almost always true for application code refactorings that deals with generalization [2], like extract superclass, extract interface, pull up method, pull down method and collapse hierarchy. In the example of application code refactoring above, the test code refactoring should mirror the class hierarchy to conform the test class hierarchy to the new application class hierarchy.

For the sake of keeping the behavior unchanged in this kind of test refactoring, the final test suite structure must not change. The movement of tests in a hierarchy or the creation of superclasses must not change the unit tests in a test suite. If one structural refactoring in a test class have altered the unit tests executed, the behavior would have been probably changed too.

7. A catalog for test code refactorings

It has been also developed a preliminary catalog of test code specific refactorings with 15 refactorings from the tree kinds showed in this paper. This catalog intends to be used by developers as a reference, documenting some test code refactorings. The following list shows the catalog refactorings with a brief explanation:

Refactoring inside a test method
- Add Assertion Explanation - It adds a string parameter in the assertion method with its explanation [4].
- Introduce Equality Method - It substitutes many comparisons in a same object with a comparison method [4].
- Simplify Use Setting – It: substitutes the actions to simpler ones with the same effect on test target.
- Split Action from Assertion - It substitutes an assertion that make modifications on one action and on one assertion.
- Decompose Assertion - It substitutes one composite assertion for equivalent ones.

Refactoring inside a test class
- Add Fixture - It creates a fixture and use it as a test target.
- Extract Setup Method – It creates a setup() method and moves initialization code to it.
- Extract Tear Down Method It creates a teardown() method and moves finalization code to it.
- Join Incremental Tests It joins tests that are created in a incremental way.
- Join Similar Tests With Different Data - It joins all similar tests that use different data in only one.

Structural refactoring of test classes.
- Mirror Hierarchy to Tests - It mirrors to test classes the same hierarchy of the application classes.
- Pull Up Test - It moves a test method up in the hierarchy.
- Pull Down Test - It moves a test method down in the hierarchy.
- Create Template Test - It creates a template test to abstract functions like the template method design pattern [7].
- Split Test from Aggregated Class - It separates the check of the responsibility of an aggregated class into another unit test.

8. A software implementation

Based on the formal and graphical notation, an Eclipse plugin was developed to automate the refactorings [8] inside a test class. All the refactoring mechanisms are implemented based on the test structure described in Section 3. This software helps in the validation of the test structure and its formal representation. This plugin is an academic prototype, still in development phase, but it can be downloaded at http://sourceforge.net/projects/testrefactoring/.

9. Related work

Beck in his TDD examples refactors both application code and test code, but do not show the difference between them [1]. Being one of the first references about TDD, he shows the importance of test code refactoring, despite he does not show that explicitly.

Deursen et al. were the first one to treat application code and test code refactoring as different things [4]. In their paper, they identify test code smells and specific refactorings applied to test code. It focuses on the practice and do not address issues like test code refactorings in a safe way.

In another study, Deursen and Moonen divide the refactorings in many types, and they use this classification to identify which refactorings affect the test code [6]. This is an important contribution, because with this it is possible to define which refactorings are needed to apply to a test code after a refactoring in the corresponding application code.
10. Practical Results

The test code refactoring was experimented in a large project of high complexity in the Brazilian Air Force. A training about test driven development [1], including the practices explained in this paper, was supplied to the development team. The graphical representation helped in a fast comprehension of the technique.

The use of test code refactoring helps in the test code support keeping it cleaner. In Figure 9 there is the TDD cycle used by the developers that include the test code refactoring activity. This cycle represents how test code refactoring fits in the development process.

The graphical notation and the catalogue aim to help testers to improve their understanding of test code refactoring and how to refactor safely unit tests. The use of test code refactoring in a real project shows that it can help in the test code support.

The catalog of test code refactorings identifies the most common and general purpose refactorings. For specific software, like ones that deals with a lot of concurrence, the unit tests should need additional types of refactorings. This catalog does not address also tests that use external resources [4]. In a future work this catalog could be improved with refactorings for tests in a specific environment and for special test techniques.

The development of the Eclipse plugin reveals that this work can be also a valuable input for the development of automated refactoring tools for test code. In future works, an automated refactoring browser [8] for test code could be developed based in this work and used to improve developer’s productivity.

11. Conclusions

In this paper a way to refactor test code safely was showed. The following are considered the most important contributions:

- The creation of a graphical notation to represent test suites, which may ease the comprehension and the manipulation of the test code structure.
- The explicit definition of behavior in a test code and when two test suites are equivalent.
- The classification of test code refactorings in categories by scope. By the classification it is possible to know what is necessary to safely verify the test code behavior.
- A preliminary catalogue of test code refactorings following the proposed classification.
- A development of a tool that automates the refactoring of test code.
- How to fit the test code refactoring in the TDD cycle.

11. References


