International Journal of Current Engineering and Technology
©2019 INPRESSCO®, All Rights Reserved

E-ISSN 2277 – 4106, P-ISSN 2347 – 5161
Available at http://inpressco.com/category/ijcet

Research Article

Optimal Refactoring of Test Case at the Design Level

Abdulrahman Issa[*]

Department of Computer, Higher Institute of Science and Technology, Nalut, Libya

Received 05 May 2019, Accepted 06 July 2019, Available online 08 July 2019, **Vol.9, No.4 (July/Aug 2019)**

Abstract

Refactoring is the process of changing a software system aimed at organizing the design of source code, making the system easier to change and less error-prone, while preserving observable behavior. This concept has become popular in Agile software methodologies, such as eXtrerne Programming (XP), which maintains source code as the only relevant software artifact. Refactoring was originally conceived to deal with source code changes. Two key aspects of eXtreme Programming (XP) are unit testing and merciless refactoring. We found that refactoring test code is different from refactoring production code in two ways: (M. B. Cohen et al, 2003) there is a distinct set of bad smells involved, and (John A. Fodeh et al, 2002) improving test code involves additional test code refactoring's. we describe a set of code smells indicating trouble in test code and a collection of test code refactoring explaining how to overcome some of these problems through a simple program modification. The goal of our present investigations is to share our experience in improving test code with other XP practitioners.

Keywords: *Test Smell, Test Case, Refactoring, Unit Testing eXtreme Programming, TDD.*

1. Introduction

Computer software is an engine of growth of soei-economy development which requires new techniques and strategies. The demand for quality in software applications has grown. lienee testing becomes one of the essential components of software development which is the indicator of quality (John A. Fodeh *et al, 2002*).

"Testing proves the presence, not the absence of bugs" -- E.W.Dijkstra

The unit test provides the lowest level of testing during software development, where the individual units of software are tested in isolation from other parts of program/software system. Automated Testing is the other program that runs the program being tested, feeding it with proper input, and thus checking the output against the expected. Once the test case is written, no human intervene is needed thus the test case does all and indicate (Eugene Volokh, VESOFT, *1990*). Adequate testing of software trials prevent this tragedies to occur. Adequate testing however, can be difficult if the software is extremely large and complex. This is because the amount of time and efforts required to execute a large set of test cases or regression test cases be significant (S. Elbaum *et al, 2000*). Therefore, the more testing can be done with accuracy of test cases which assist in corresponding rise in program transformation.

Amongst different types of program transformation, behavior preserving source-to- source transformations are known as refactorings (Don Roberts, *1999*). Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure (Martin Fowler, *1999*).

The refactoring concept was primarily assigned to source code changes. The refactoring of test case may bring additional benefits to software quality and productivity, vis-avis cheaper detection of design flaws and easy exploration of alternative design decisions. Consequently, The term code refactoring and test case refactoring can be made distinct. Thus, one of the main reasons for wide acceptance of refactoring as a design improvement technique and its subsequent adoption by Agile software methodologies, in particular eXtreme Programming (XP(M. B. Cohen *et al, 2003*). The XP encourages the development teams to skip comprehensive initial architecture or design stages, guiding them its implementation activities according to user requirements and thus promoting successive code refactorings when inconsistencies are detected.

2. Test-Driven Development

Test Driven Development (TDD) is the core part of the Agile code development approach derived from

*Corresponding author's ORCID ID: 0000-0003-3602-2095
DOI: https://doi.org/10.14741/ijcet/v.9.4.5

eXtreme Programming (XP) and the principles of the Agile manifesto. It provides to guarantee testability to reach an extremely high test coverage, to enhance developer confidence, for highly cohesive and loosely coupled systems, to allow larger teams of programmers to work on the same code base, as the code can be checked more often. It also encourages the explicitpess about the scope of implementation. Equally it helps separating the logical and physical design, and thus to simplify the design, when only the code needed.

The TDD is not a testing technique, rather a development and design technique in which the tests are written prior to the production code. The tests are added its gradually during its implementation and when the test is passed, the code is refactored accordingly to improve the efficacy of internal .structure of the code. The incremental cycle is repeated until all functionality is implemented to final. The TDD cycle consists of six fundamental steps:

1) Write a test for a piece of functionality,
2) Run all tests to see the new test to fail,
3) Write corresponding code that passes these tests,
4) Run the test to see all pass,
5) Refactor the code and
6) Run all tests to see the refactoring did not change the external behavior.

The first step involves simply writing a piece of code to. ensure the tests of desired functionality. The second is required to validate the correctness of test, i.e, the test must not pass at this point, because the behavior under implementation must not exist as yet. Nonetheless, if the lest passes over, means the test is either not testing correct behavior or the TDD principles have not been strictly followed. The third step is the writing of the code.

However, it should be kept in mind to only write as little code as possible to enable to pass the test (Astels, *2003*). Next, step is to see that the change has not introduced any of the problems somewhere else in the system. Once all these tests are passed, then the internal structure of the code should be improved by refactoring. The above mentioned cycle is presented in Figure 1.

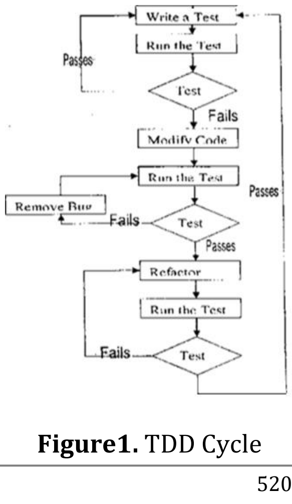

Figure1. TDD Cycle

7) Refactoring

Program restructuring is a technique for rewriting software may be useful either for legacy software as well as for the production of new systems (Robert S. Arnold, *1989*; William G. Griswold, *1991*; B.-K. Kang, *1999*). If the internal structure is changed, although the behavior (what the program is supposed to do) is maintained. Restructuring re-organizes the logical structure of source code in order to improve specific attributes (B.-K. Kang, *1999*) or to make it less error-prone when future changes are introduced (Robert S. Arnold, *1989*).

Behavior preserving program changes are known as refactorings which was introduced by Opdyke (William Opdyke, *1992*). Yet its gaining importance by Fowler's work (Martin Fowler, *1999*) and eXtreme Programming (XP) [IJ, an Agile software development in context of object-oriented development. In this context, a refactoring is usually composed of a set of small and atomic refactorings, after which the largest source code is better than the original with respect to particular quality attributes, such as readability and modularity.

Thus, refactoring can be viewed as a technique for software evolution through-out software development and maintenance. Software evolution can be classified into the following types (Sheena R *et al, 2003*):

- Corrective evolution: correction of errors;
- Adaptive evolution: modifications to accommodate requirement changes;
- Perfective evolution: modifications to enhance existing features.

Refactoring is mostly applied in perfective software evolution, though it also affects corrective and adaptive evolution. First, well- organized and flexible software allows one to quickly isolate and correct errors. Second, such software ensures that new functionality can be easily added to address changing user requirements.

A known issue about refactorings is automatization. Small steps of refactoring have usually been performed manually using primitive tools such as text editors with search and replace functionality. This situation eventually leads to corrupt the design of source code, mostly due to the fact that manual refactoring is tedious and prone to errors (Don Roberts, *1999*). Although the choice of which refactoring to apply is naturally made by human, automatic execution of refactorings might result in a major improvement in productivity.

In addition, concerning behavior preservation, TDD informally guides refactoring assisted by unit tests, increasing the correctness of a sequence of transformations. Furthermore, verification of object-oriented programs is highly nontrivial. A number of recent research initiatives have pointed out directions for formally justifying refactorings In Opdyke's work, preconditions for refactorings are analyzed (William Opdyke, *1992*), whereas Robert's work formalizes the effect of refactorings in terms of pastconditions, in order to build efficient refactoring tools (Don Roberts, *1999*). In contrast, Mens (Tom Mens *et al, 2002*), apply graph representation to I.peets that should be preserved and graph rewriting rules as formal specification for refactorings.

8) Causes of refactoring

In computer programming, code smell is any symptom in the source code of a program that possibly indicates a problem at steep level.

Often the deeper problem hinted by a code smell can be uncovered when the code is 'subjected to a short feedback cycle where it is refactored in small, controlled steps, and the resulting design is examined to assist the needs of more rcfactoring. From the programmer's point of view, code smells are forecast to refactor, and what specific refactoring techniques are to be used. Thus, a code smell is a driver for refactoring. Code smell hint that provides can be improved In some where in your code.

Determining a code smell is often a subjective judgment lind will often vary by language, developer and its methodology. There are certain tools, such as Checkstyle, PMD .and FindBugs for Java, to automatically evaluate for certain kinds of code smells. When to apply refactorings to the test code, is different from refactoring production code and the test code has a distinct set of smells dealing with the test cases are organized, to study its implementation and interaction with each other. Moreover, improving test code involves a mixture of refactorings from specialized to test code improvements as well as a set of additional refactorings involving the modification of test classes, ways of grouping lest cases; and so on (M. Fowler, *1999*).

Refactoring (to Patterns)

- Simple Design -> Code Smell -> Refactor
- Refactoring (to Patterns) is the ability to transform a "Code Smell" into a positive design pattern.

Following arc the examples of some of the Dad Resource Interface

Such wars arise when the tests execute you are the only one testing which fails 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. Jdentified Uniquely is one of the test code refactoring method used to overcome Resource Interference.

Code Smells that are encountered in case (unit/class) design

- Duplicated Code
- Methods too big
- Nested "if' statements
- Classes with too many instance variables
- Classes with too much code
- Strikingly similar subclasses
- Too many.private (or protected) methods
- Similar looking code sections
- Dependency cycles
- Passing Nulls To Constructors
- Classes with too little code

9) Test case code smells

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

Self Contained

When a test uses external resources, such as file containing test data, the test is no longer self contained. Consequently, there is no enough information to understand the test functionality, to use it as test documentation.

Moreover, external resources introduces hidden dependencies: if some force mutates such a resource, tests start failing. Chances for this increase becomes more when more tests use the same resource. The use of external resources can be thus eliminated using refactoring Intregral Resource.

Resource Optimism

Test code that makes optimistic assumptions about the existence (or absence) and state of ex tema I resources (such as particular directories or database tables) can cause nondeterministic behavior in test outcomes. The situation where tests run fine at one time and fail miserably at the other time needs to be avoided. Resource Allocation refactoring used to allocate and/or initialize all resources that are to be used.

Resource Interface

Such wars arise when the tests execute you are the only one testing which fails 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. Jdentified Uniquely is one of the test code refactoring method used to overcome Resource Interference.

Setup Method

In the JUnit framework a programmer can write a setUp method that can be executed before each test method to create a fixture for the tests to run. 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.

Splitting Method

When a test method checks 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. Note that splitting into smaller methods Which can slow down the tests due to increased letup/teardown overhead.

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, it becomes difficult to know the cause of concern. Usc Asertion Explanation to remove this smell.

Class-to-be-tested

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 classIo-be-tested) .. I'he smell which arises also indicates the 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.

Duplication across Test Class

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 [3, p. 76]. The most common case for test code will be duplication of code in the same test class. For duplication across test classes, it may prove helpful to mirror the class hierarchy of the production code into the test class hierarchy. A word of caution however can introduce dependencies between tests moving duplicated code from two separate classes to a common class.

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 such refactoring.

10)Test code refactoring

Bad smell seems to arise more often in production code than in test code. The main reason for this is that, production code is adopted 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. 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 lest code, the following refactorings are encountered:

Integral Resource

To remove the dependency between a test method and some external resource, we incorporate the resource in the test code. This is done by setting up fixture in the test code that holds the same contents as the resource , This fixture is then can be used instead Of the resource to run the test . A simple example of this refactoring ss to put the contents of a file that is used into some string in test code.

Resource Allocation

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 allocate these resources before testing and releases them when done (take precautions to ensure the resource is also released when tests fail).

Identified Uniquely

Lot of problems originate from tho use of overlapping resource names; either between different tests run done by the same user or between simultaneous tests run done by different users. Such problems can easily be overcome using unique identifiers for all resources that are allocated, such as 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.

Minimize Data

Minimize the data that is setup in fixtures to bare essentials, this will have two advantages :

1) In making them better suitable for documentation and consequently.
2) The tests will be less sensitive to changes.

Assertion Explanation

Assertions in the Unit framework have an optional first argument to give an expanatory 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 Ihe saem test. May be this argument should not have been optional,

Add Equality Method

If an object structure needs to be checked for equality in test . an implementatlon for the "equals' method for the object's class needs to be added, you then can rewrite the tests that use string equality to to use htis method. If an expected test value is only represented as astring. explicitly construct an object containing the expected value and use the new equals method to compare it to the actually computed object.

Conclusions

By end large, refactoring can improve overall quality of a test case using these set of smells choices. The only concern needs to be understand is the selection of refactoring chices. But which refactoring choices should be implemented? We advocates program slicing in conjunction with code smell' to guide refactoring process. By slicing the software system one or more bad smells, different refactoring options can examined and evaluated using these sets of smells. Thus the combination of program slicing and set of code srhells guides the refactoring process.

A software system essentially needs the refactoring systems for its better performance. Thus this refactoring process assist in its high quality and can prove to be more maintainable techniques. This ref acto ring process thus can be executed in lower error rates, fewer test cases per module and to increased over all understandability and maintainability in return. In both the design and maintenance phases, these advantages can be realized almost immediately.

References

M. B. Cohen, P. B. Gibbons, W. B. Mugridge and C.J. Colbourn. (2003), Constructing Test Suites for Interaction Testing, *25th International Conference on Software Engineering (ICSE'30),* pp. 38-49, Portland, Oregon, United States, IEEE Computer Society.

John A. Fodeh and Niels B. Svendsen. (2002), Release Metrics: When to Stop Testing with a clear conscience, Journal of Software Testing Professionals, March.

Eugene Volokh, VESOFT. (1990), Automated Testing When and How, Interact Magazine.

Don Roberts. Practical Analysis for Refactoring. (1999), PhD thesis, University of Illinois at Urbana-Champaign.

S. Elbaum, A. G. Malishevsky and G. Rothermel. (2000), Prioritizing Test Cases for Regression Testing, *ACM SIGSOFT International Symposium on Software Testing and Analysis,* pp. 102-112, Portland, Oregon, United States, ACM Press.

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

Robert S. Arnold. (1989), Software Restructuring. Proceedings of the IEEE, 77(4):607{617, April}.

William G. Griswold. (1991), Program Restructuring as an Aid to Software Maintenance. PhD thesis, University of Washington.

B.-K. Kang and J. M. Bieman. (1999), A Quantitive Framework for Software Restructuring. Journal of Software Maintenance, 11 :245 {284}.

Martin Fowler. (1999), Refactoring improving the design of existing code. Addison Wesley.

William Opdyke. (1992), Refactoring Object-Oriented Frameworks. PhD thesis, University of JII inois at Urbana-Champaign.

Tom Mens, Serge Demeyer, and Dirk Janssens. (2002), Formalising Behaviour Preserving Program Transformations. In Proceedings of the First International Conference on Graph Transformation, pages 286{301. Springer-Verlag.

Sheena R. Judson, Doris L. Carver, and Robert France. (2003), A Metamodeling' Approach to Model Refactoring, Submitted to UML.

Frederick P. Brooks Jr. (1995),The Mythical Man-Month (Anniversary Ed.). Addison- Wesley Longman Publishing Co., Inc.

Martin Fowler. (1999), Refactoring: Improving the Design of Existing Code, Addison Wesley Longman, Inc., Pub.