The Video Store Revisited - Thoughts on Refactoring and Testing

Arie van Deursen
CWI
The Netherlands
http://www.cwi.nl/~arie/
arie@cwi.nl

Leon Moonen
CWI
the Netherlands
http://www.cwi.nl/~leon/
leon@cwi.nl

Abstract

Testing and refactoring are core activities in extreme programming (XP). In principle, they are separate activities where the tests are used to verify that refactorings do not change behavior of the system. In practice however, they can become intertwined when refactorings invalidate tests. This paper explores the precise relationship between the two. First, we identify which of the published refactorings affect the test code. Second, we observe that if test-first design is a way to arrive at well-designed code, "test-first refactoring" *is a way to arrive at a better design for existing code. Third, some refactorings improve testability, and should therefore be followed by improvements of the test code. To emphasize this, we propose the notion of* "refactoring session" *which includes changes to the code followed by changes to the tests. To guide the developer in the steps to take, we propose to extend the description of the mechanics of individual refactorings with consequences for the corresponding test code.*

Keywords

Refactoring, unit testing, extreme programming.

INTRODUCTION

Two key activities in extreme programming (XP) are testing and refactoring. In this paper, we explore the relationship between these two.

In XP, tests are fully automated, self-checking the validity of their outcome. Besides for checking correct behavior, tests are intended for *documentation* purposes. A test case is a simple scenario with a known outcome, and can be used to understand the code being tested. Since the tests are required to be run upon every change, their documentation value is guaranteed to remain up to date [3]. Code development in XP is done through *test-first design:* Structuring the test cases guides the design of the production code.

Extreme programmers improve the design of the system through frequent refactoring. Refactorings improve the internal structure of the code without changing its external behavior.

This is done by removing duplication, simplification, making code easier to understand, and adding flexibility. "Without refactoring, the design of software will decay. Regular refactoring helps code retain its shape." [5, p.55].

One of the dangers of refactoring is that a programmer unintentionally changes the system's behavior. Ideally, it can be verified that this did not happen by checking that all the tests pass after refactoring. In practice however, there are refactorings that will invalidate tests (e.g., when a method is moved to another class and the test still expects it in the original class). In this paper, we explore the relationship between unit testing and refactoring. In Section 2, we provide a classification of the refactorings described by Fowler [5], identifying exactly which of the refactorings affect class interfaces, and which therefore require changes in the test code as well. In Section 3 we take the video store example from [5], and assess the implications of each refactoring on the test code. In Section 4, we propose *test-first refactoring,* which analyzes the test code in order to arrive at code level refactorings. In Section 5, we discuss the relationship between code-level refactorings and test-level refactorings. In Section 6 we integrate these results via the notion of a *refactoring session* which is a coherent set of steps resulting in refactoring of both the code and the tests. In Section 7 we present a summary and draw our conclusions.

TYPES OF REFACTORING

Refactoring a system should not change its observable behavior. Ideally, this is verified by ensuring that all the tests pass before and after a refactoring [1, 5].

In practice, it turns out that such verification is not always possible: some refactorings restructure the code in such a way that tests only can pass after the refactoring if they are modified. For example, refactoring can move a method to a new class while some tests expect it in the original class (in that case, the code will probably not even compile). Nevertheless, we do not want to change the tests together with a refactoring since that will make them less trustworthy for validating correct behavior afterwards.

In the remainder of this section, we will look in more detail at the refactorings described by Fowler [5] to analyze in which case problems might arise because the original tests need to be modified.

Change Bi-directional Association to Unidirectional	Remove Assignments to Parameters	Preserve Whole Object
Replace Magic Number with Symbolic Constant	Replace Data Value with Object	Remove Control Flag
Replace Nested Conditional with Guard Clauses	Introduce Explaining Variable	Substitute Algorithm
Consolidate Duplicate Conditional Fragments	Replace Exception with Test	Introduce Assertion
Replace Conditional with Polymorphism	Change Reference to Value	Extract Class
Replace Delegation with Inheritance	Split Temporary Variable	Inline Temp
Replace Inheritance with Delegation	Decompose Conditional	
Replace Method with Method Object	Introduce Null Object	

Table 1. Compatible refactorings (type B)

Consolidate Conditional Expression	Pull Up Constructor Body	Extract Superclass	Pull Up Method
Replace Delegation with Inheritance	Replace Temp with Query	Extract Interface	Pull Up Field
Replace Inheritance with Delegation	Duplicate Observed Data	Push Down Method	
Replace Record with Data Class	Self Encapsulate Field	Push Down Field	
Introduce Foreign Method	Form Template Method	Extract Method	

Table 2. Backwards compatible refactorings (type C)

Taxonomy

If we start with the assumption that refactoring does not change the behavior of the system, then there is only one reason why a refactoring can break a test: *because the refactoring changes the interface that the test expects.* Note that the interface extends to all visible aspects of a class (fields, methods, and exceptions). This implies that one has to be careful with tests that directly inspect the fields of a class since these will more easily change during a refactoring. [1]

So, initially, we distinguish two types of refactorings: refactorings that do not change any interface of the classes in the system and refactorings that do change an interface. The first type of refactorings have no consequences for the tests: since the interfaces are kept the same, tests that succeeded before refactoring will also succeed after refactoring (if the refactoring indeed preserves the tested behavior).

The second type of refactorings can have consequences for the tests since there might be tests that expect the old interface. Again, we can distinguish two situations:

Incompatible: the refactoring destroys the original interface. All tests that rely on the old interface must be adjusted.

Backwards Compatible: the refactoring extends the original interface. In this case the tests keep running via the original interface and will pass if the refactoring preserves tested behavior. Depending on the refactoring, we might need to add more tests covering the extensions.

A number of incompatible refactorings that normally would destroy the original interface can be made into backwards compatible refactorings. This is done by extending the refactoring so it will retain the old interface, for example, using the Adapter pattern or simply via delegation. As a side-effect, the new interface will already partly be tested. Note that this is common practice when refactoring a published interface to prevent breaking dependent systems. A disadvantage is that a larger interface has to be maintained but when delegation or wrapping was used, that should not be too much work. Furthermore, language features like deprecation can be used to signal that this part of the interface is outdated.

Classification

We have analyzed the refactorings in [5] and divided them into the following classes:

A. *Composite:* The four big refactorings Convert Procedural Design to Objects, Separate Domain from Presentation, Tease Apart Inheritance, and Extract Hierarchy will change the original interface, but we will not consider them in more detail since they are performed as series of smaller refactorings.

B. *Compatible:* Refactorings that do not change the original interface. Refactorings in this class are listed in Table 1.

C. *Backwards Compatible:* Refactorings that change the original interface and are inherently backwards compatible since they extend the interface. Refactorings in this class are listed in Table 2.

[1] In fact, direct inspection of fields of a class is a test smell that could better be removed beforehand [4].

Change Unidirectional Association to Bi-directional	Separate Query from Modifier	Remove Middle Man	Add Parameter
Replace Parameter with Explicit Methods	Introduce Parameter Object	Remove Parameter	Move Method
Replace Parameter with Method	Parameterize Method	Rename Method	

Table 3. Refactorings that can be made backwards compatible (type D)

Replace Constructor with Factory Method	Replace Type Code with Class	Remove Setting Method	Hide Delegate
Replace Type Code with State/Strategy	Change Value to Reference	Encapsulate Downcast	Inline Method
Replace Type Code with Subclasses	Introduce Local Extension	Collapse Hierarchy	Inline Class
Replace Error Code with Exception	Replace Array with Object	Encapsulate Field	Hide Method
Replace Subclass with Fields	Encapsulate Collection	Extract Subclass	Move Field

Table 4. Incompatible refactorings (type E)

D. *Make Backwards Compatible:* Refactorings that change the original interface and can be made backwards compatible by adapting the new interface to the new one. Refactorings in this class are listed in Table 3.

E. *Incompatible:* Refactorings that change the original interface and are not backwards compatible (for example, because they change the types of classes that are involved). Refactorings in this class are listed in Table 4.

Note that the refactorings Replace Inheritance with Delegation and Replace Delegation with Inheritance are listed both in the *Compatible* and *Backwards Compatible* tables since they can be of either category, depending on the actual case.

REVISITING THE VIDEO STORE

In this section, we study the relationship between testing and refactoring using a well-known example of refactoring. We revisit the video store code used by Fowler [5, Chapter 1], extending it with an analysis of what should be going on in the accompanying video store test code.

The video store class structure before refactoring is shown in Figure 1. It consists of a *Customer,* who is associated with a series of *Rentals,* each consisting of a *Movie* and an integer indicating the number of days the movie was rented. The key functionality is in the Customer's *statement* method printing a customer's total rental cost. Before refactoring, this statement is printed by a single long method. After refactoring, the statement functionality is moved into appropriate classes, resulting in the structure of Figure 2 taken from [5, p. 51].

Figure 1. Classes before refactoring

Fowler emphasizes the need to conduct refactorings as a sequence of small steps. At each step, you must run the tests in order to verify that nothing essential has changed. His testing approach is the following: "I create a few customers, give each customer a few rentals of various kinds of films, and generate the statement strings. I then do a string comparison between the new string and some reference strings that I have hand checked" [5, p. 8]. Although Fowler doesn't list his test classes, this typically should look like the code in Figure 3.

Studying this string-based testing method, we can make the following observations:

- The setup is complicated, involving the creation of many different objects.
- The documentation value of the test is limited: it is hard to relate the computation of the charge of 4.5 for movie `m1` to the way in which charges are computed for the actual movies rented (in this case a children's and a regular movie, each with their own price computation).
- The tests are brittle. All test cases include a full statement string. When the format changes in just a very small way, all existing tests (!) must be adjusted, an error prone activity we would like to avoid.

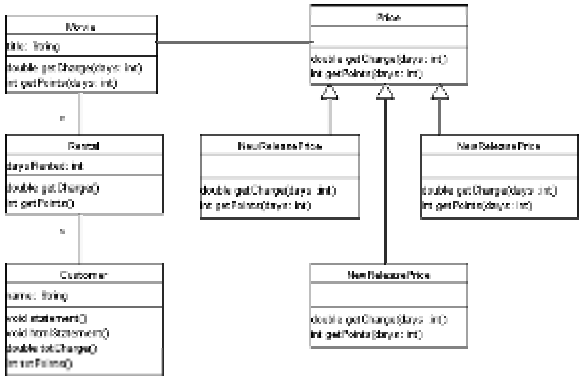

Figure 2. Class structure after refactoring

Unfortunately, there is no other way to write tests for the given code. The poor structure of the long method necessarily leads to an equally poor structure of the test cases. From a testing perspective, we would like to be able to separate computations from report writing. The long statement method prohibits this: it needs to be refactored in order to be able to improve the testability of the code.

This way of reasoning naturally leads to the application of the Extract Method refactoring to the *statement* method. Fowler comes to the same conclusion, based on the need to write a new method printing a statement in HTML format. Thus, we extract *getCharge* for computing the charge of a rental, and *getPoints* for computing the "frequent renter points".

Extract Method is of type B, the *compatible* refactorings, so we can use our existing tests to check the refactoring. However, we have created new methods, for which we might like to add tests that document and verify their specific behavior. To create such tests, we can reuse the setup of movies, rentals, and customers used for testing the *statement* method. We end up with a number of smaller test cases specifically addressing either the charge or rental point computations. Since the correspondence between test code and actual code is now much clearer and better focused, we can apply white box testing, and use our knowledge of the structure of the code to determine the test cases needed. Thus, we see that the *getCharge* method to be tested distinguishes between 5 cases, and we make sure our tests cover these cases.

```
Movie m1 = new Movie("m1", Movie.CHILDRENS);
Movie m2 = new Movie("m2", Movie.REGULAR);
Movie m3 = new Movie("m3",
Movie.NEW_RELEASE);
Rental r1 = new Rental(m1, 5);
Rental r2 = new Rental(m2, 7);
Rental r3 = new Rental(m3, 1);
Customer c1 = new Customer("c1");
Customer c2 = new Customer("c2");
public void setUp() {
c1.addRental(r1);
c1.addRental(r2);
c2.addRental(r3);
}
public void testStatement1() {
String expected =
"Rental Record for c1\n" +
"\tm1\t4.5\
"\tm2\t9.5\n" +
"Amount owed is 14.0\n" +
"You earned 2 frequent renter points";
assertEquals(expected, c1.statement());
}
...
```

Figure 3. Initial sample test code

This has solved some of the problems. The tests are better understandable, more complete, much shorter, and less brittle. Unfortunately, we still have the complicated setup method. What we see is that the setup mostly involves rentals and movies, while the tests themselves are in the customer testing class. This is because the extracted method is in the wrong class: applying Move Method to *Rental* simplifies the set up for new test cases. Again we use our analysis of the test code to find refactorings in the production code.

The Move Method is of type D, refactorings that can be made backwards compatible by adding a wrapper method to retain the old interface. We add this wrapper so we can check the refactoring with our original tests. However, since the documentation of the method is in the test, and this documentation should be as close as possible to the method documented, we want to move the tests to the method's new location. Since there is no test class for Rental yet, we create it, and move the test methods for *getCharge* to it. Depending on whether the method was part of a published interface, we might want to keep the wrapper (for some time), or remove it together with the original test.

Fowler discusses several other refactorings, moving the charge and point calculations further down to the *Movie* class, replacing conditional logic by polymorphism in order to make it easier to add new movie types, and introducing the *state* design pattern in order to be able to change movie type during the life time of a movie.

When considering the impact on test cases of these remaining video store refactorings, we start to recognize a pattern:

- Studying the test code and the smells contained in it may help to identify refactorings to be applied at the production code;
- Many refactorings involve a change to the structure of the unit tests of well: in order to maintain the documenting value of these unit tests, they should be changed to reflect the structure of the code being tested.

In the next two sections, we take a closer look at these issues.

TEST-FIRST REFACTORING

In *test-first refactoring,* we try to use the existing test cases in order to determine the code-level refactorings. Thus, we study *test* code in order to find improvements to the *production* code.

This calls for a set of *code smells* that helps to find such refactorings. A first category is the set of existing code smells discussed in Fowler's book [5]. Several of them, such as long method, duplicated code, long parameter list, and so on, apply to test code as well as they do to production code. In many cases solving them involves not just a change on the test code, but first of all a refactoring of the production code.

A second category of smells is the collection of *test smells* discussed in our earlier paper on *refactoring test cases* [4]. In fact, in our movie example we encountered several of them already. Our uneasy feeling with the test case of Figure 3 is captured by the Sensitive Equality smell [4, Smell 10]: comparing computed values to a string literal representing the expected value. Such tests

depend on many irrelevant details, such as commas, quotes, tabs, and so on. This is exactly the reason the customer tests of Figure 3 become brittle.

Another *test smell* we encountered is called Indirect Testing [4, Smell 8]: a test class contains methods that actually perform tests on other objects. Indirect tests make it harder to understand the relationship between test and production code. While moving the *getCharge* and *getPoints* methods in the class hierarchy (using Move Method), we also moved the corresponding test cases, in order to avoid Indirect Testing.

The test-first perspective may lead to the formulation of additional test smells. For example, we observed that setting up the fixture for the CustomerTest was complicated. This indicates that the tests are in the wrong class, or that the underlying business logic is not well isolated. Another smell appears when there are many test cases for a single method, indicating that the method is too complex.

Test-first refactoring is a natural consequence of test-first design. Test-first design is a way to get a good design by thinking about test cases first when adding functionality. Test-first refactoring is a way to improve your design by rethinking the way you structured your tests.

In fact, Beck's recent article on test-first design [2] contains an interesting example that can be transferred to the refactoring domain. It involves testing the construction of a mortality table. His first attempt requires a complicated setup, involving separate "person" objects. He then rejects this solution as being overly complex for testing purposes, and proposes the construction of a mortality table with just an age as input. His example illustrates how test case construction guides design when building new code; likewise, test case refactoring guides the improvement of design during refactoring.

REFACTORING TEST CODE

In our study of the video store example, we saw that many refactorings on the code level can be completed by applying a corresponding refactoring on the test case level. For example, to avoid Indirect Testing, the refactoring Move Method should be followed by "Move Test". Likewise, in many cases Extract Method should be followed by "Extract Test". To retain the documentation value of the unit tests, their structure should be in sync with the structure of the source code.

In our opinion, it makes sense to extend the existing descriptions of refactorings with suggestions on what to do with the corresponding unit tests, for example in the "mechanics" part.

The topic of refactoring test code is discussed extensively in [4]. An issue of concern when changing test code is that we may "loose" test cases. When refactoring production code, the availability of tests safeguards us from accidentally loosing code, but this is not the case when modifying tests. A solution is to measure coverage before and after changing the tests. As an example, this can be done through *mutation testing* using a tool such as Jester [6]. Jester automatically makes changes to conditions and literals in Java source code. If the code is well-tested, such changes should lead to failing tests. Running Jester before and after test case refactorings helps to verify that the changes did not affect test coverage.

REFACTORING SESSIONS

The meaningful unit of refactoring is a sequence of steps involving changes to both the code base and the test base. We propose the notion of a *refactoring session* to capture such a sequence. It consists of the following steps:

1. Detect *smells* in the code or test code that need to be fixed. In test-first refactoring, the test set is the starting point for finding such smells.
2. Identify candidate refactorings addressing the smell.
3. Ensure that all existing tests run.
4. Apply the selected refactoring to the code. Provide a backwards compatible interface if the refactoring falls in category D. Only change the associated test classes when the refactoring falls in category E.
5. Ensure that all existing tests run. Consider applying mutation testing to assess the coverage of the test cases.
6. Apply the testing counterpart of the selected refactoring.
7. Ensure that the modified tests still run. Check that the coverage has not changed.
8. Extend the test cases now that the underlying code has become easier to test.
9. Ensure the new tests run.

The integrity of the code is ensured since (1) all tests are run between each step; (2) each step changes either code or tests, but never both at the same time (unless this is impossible).

CONCLUSIONS

In this paper we have taken a close look at the interplay between testing and refactoring. We consider the following as our most important contributions:

- We have analyzed which of the documented refactorings necessarily affect the test code. It turns out that the majority of the refactorings are in category D (requiring explicit actions to keep the interface compatible) and E (necessarily requiring a change to the test code).
- We have studied Fowler's video store example from the point of view of unit tests included for documentation purposes. We have shown the test

case implications of each refactoring conducted.

- We have proposed the notion of *test-first refactoring,* which uses the existing test cases as the starting point for finding suitable code level refactorings.
- We have argued for the need to extend the descriptions of refactorings with a section on their implications on the corresponding test code. If the tests are to maintain their documentation value, they should be kept in sync with the structure of the code.
- We have proposed the notion of a *refactoring session,* capturing a coherent series of separate steps involving changes to both the production code and the test code.

Our observations and proposals help us in understanding the interaction between testing and refactoring. Moreover, they constitute valuable input when developing advanced refactoring tools that need to be tightly integrated with (JUnit) test suites.

ACKNOWLEDGMENTS
We would like to thank Frank Westphal for valuable feedback on our paper. We also would like to thank the members of the University of Illinois Software Architecture Group for their feedback on our paper provided through an MP3 recording of their discussion!

REFERENCES

[1] K. Beck. *Extreme Programming Explained. Embrace Change.* Addison Wesley, 2000.

[2] K. Beck. Aim, fire: Kent Beck on test-first design. *IEEE Software Community Chest,* 18(5), September/October 2001. http://www.computer.org/software/homepage/2001/05Design/index.htm.

[3] A. van Deursen. Program comprehension risks and benefits in extreme programming. In E. Burd, P. Aiken, and R. Koschke, editors, *Proceedings 8th Working Conference on Reverse Engineering (WCRE 2001),* pages 176–185. IEEE Computer Society, 2001.

[4] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In M. Marchesi and G. Succi, editors, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001),* pages 92–95, May 2001.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[6] I. Moore. Jester — a JUnit test tester. In M. Marchesi and G. Succi, editors, Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), pages 84–87, May 2001.