How to Test Refactoring

by Jeroen Mengerink

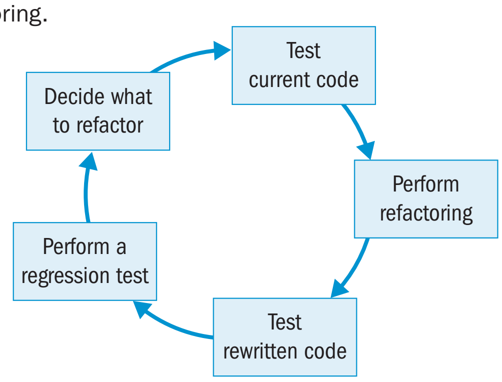
A fundamental part of the Agile methodology is refactoring: rewriting small sections of code to be functionally equivalent but of better quality. Don't forget to test the refactoring! What do you test? The answer is simple: you test whether the code really is functionally equivalent.

To test the rewritten code, you use the unit tests that accompanied the original code. But does unit testing alone prove that you really have functionally equivalent code? No! While refactoring, developers often change more than just the complexity and quality of the code. A tester's nightmare … It appears to be a small change, but the code is quite likely used in several parts of the solution. So you must perform a regression test after testing the changed code itself.

First I will describe how to test the current and rewritten code with unit test. I have identified three scenarios that occur in practice. The code that needs refactoring has:

- no unit tests;
- bad unit tests;
- good unit tests.

After these scenarios, I will go into the regression test and explain the importance of proper regression testing while refactoring.



Unit test the current and rewritten code

Unit tests are tests to test small sections of the code. Ideally each test is independent, and stubs and drivers are used to get control over the environment. Since refactoring deals with small sections of code, unit tests provide the correct scope.

Refactor code that has no existing unit tests

When you work with very old code, in general you do not have unit tests. So can you just start refactoring? No, first add unit tests to the existing code. After refactoring, these unit tests should still hold. In this way you improve the maintainability of the code as well as the quality of the code. This is a complex task. First you need to find out what the functionality of the code is. Then you need to think of test cases that properly cover the functionality. To discover the functionality, you provide several inputs to the code and observe the outputs. Functional equivalence is proven when the code is input/output conformant to the original code.

Refactor to increase the quality of the existing unit tests

You also see code which contains badly designed unit tests. For example, the unit test verifies multiple scenarios at once. Usually this is caused by not properly decoupling the code from its dependencies (Code sample 1). This is undesirable behaviour because the test must not depend on the state of the environment. A solution is to refactor the code to support substitutable dependencies. This allows the test to use a test stub or mock object. As shown in Code sample 2, the unit test is split into three unit tests which test the three scenarios separately. The rewritten code has a configurable time provider. The test now uses its own time provider and has complete control over the environment.

Treat unit tests as code

The last situation deals with a piece of code which has good unit tests. Just refactor and then you are done, right? Wrong! When you refactor this code, the test will pass if you refactor correctly. But do not forget to check the validity of the tests. You might think the tests are good, but the unit tests are code too. Every refactor action incorporates a check, and possibly a refactor, of the unit tests.

Perform a regression test

After unit testing the code, you need to verify if the code works in the solution's context. Remember: In Agile you must provide business value. To show the value, you need to perform a test that relates to the business. A regression test is designed to test the important flows through the solution. And these flows embody the business value. Do you run a complete regression test after each time you refactor? This depends on the risks and on the scalability of the regression test.

Create a scalable regression test

The use case is a common way to describe small parts of functionality. This is a great way to partition your regression test. Create a small set of regression test cases to cover a use case. When you use proper version management for the code, it is easy to see which part of the code belongs to which use case. Whenever a section of code is changed, you can see to which use case it belongs and then execute the regression tests for that use case.

However, when code is reused (another good practice), you target a group of use cases. I generally use mindmaps for tracking dependencies within my projects. The mindmaps provide insight in which code is used by which use cases. This requires a disciplined development team. When you reuse existing code, you need to update the mindmap!

Expand the scope of the regression test

Do you test enough when you scale the regression test to the scope determined in the mindmap? No, the regression test serves a larger goal. You check if the (in theory) unaffected areas of the solution are really unaffected. So you test the part that is affected by the refactoring and you test the main flows through the solution. The flows that provide value to the customer are the most important.

Refactoring requires testing

Every change in the code needs to be tested. Therefore testing is required when refactoring. You test the changes at different levels. Since a small section of code is changed, unit testing seems the most fitting level. But do not forget the business value! Regression testing is of vital importance for the business.

- Refactoring requires testing.
- Testing refactoring requires a good understanding of the code.
- A good understanding of the code requires a disciplined development team.
- A disciplined development team refactors.

Code sample 1: Unit test depending on the environment

From *http://xunitpatterns.com*.

```
public void testDisplayCurrentTime_whenever() {
  // fixture setup
  TimeDisplay sut = new TimeDisplay();
  // Exercise sut
  String result = sut.getCurrentTimeAsHtmlFragment();
  // Verify outcome
  Calendar time = new DefaultTimeProvider().getTime();
  StringBuffer expectedTime = new StringBuffer();
  expectedTime.append("<span class=\"tinyBoldText\">");
  if ((time.get(Calendar.HOUR_OF_DAY) == 0) &&
    (time.get(Calendar.MINUTE) <= 1)) {
    expectedTime.append("Midnight");
  } else if ((time.get(Calendar.HOUR_OF_DAY) == 12) &&
    (time.get(Calendar.MINUTE) == 0)) { // noon
    expectedTime.append("Noon");
  } else {
    SimpleDateFormat fr = new SimpleDateFormat("h:mm a");
    expectedTime.append(fr.format(time.getTime()));
  }
  expectedTime.append("</span>");
  assertEquals( expectedTime, result);
}
```

Code sample 2: Independent unit tests

From *http://xunitpatterns.com*.

```
public void testDisplayCurrentTime_AtMidnight()
    throws Exception {
  // Fixture setup:
  TimeProviderTestStub tpStub = new
    TimeProviderTestStub();
  tpStub.setHours(0);
  tpStub.setMinutes(0);
  // Instantiate SUT:
  TimeDisplay sut = new TimeDisplay();
  sut.setTimeProvider(tpStub);
  // Exercise sut
  String result = sut.getCurrentTimeAsHtmlFragment();
  // Verify outcome
  String expectedTimeString = "<span class=\"tinyBoldText
    \">Midnight</span>";
  assertEquals("Midnight", expectedTimeString, result);
}
public void testDisplayCurrentTime_AtNoon()
    throws Exception {
  // Fixture setup:
  TimeProviderTestStub tpStub = new
    TimeProviderTestStub();
  tpStub.setHours(12);
  tpStub.setMinutes(0);
  // Instantiate SUT:
  TimeDisplay sut = new TimeDisplay();
  sut.setTimeProvider(tpStub);
  // Exercise sut
  String result = sut.getCurrentTimeAsHtmlFragment();
  // Verify outcome
  String expectedTimeString = "<span
    class=\"tinyBoldText\">Noon</span>";
  assertEquals("Noon", expectedTimeString, result);
}
public void testDisplayCurrentTime_AtNonSpecialTime()
    throws Exception {
  // Fixture setup:
  TimeProviderTestStub tpStub = new
    TimeProviderTestStub();
  tpStub.setHours(7);
  tpStub.setMinutes(25);
  // Instantiate SUT:
  TimeDisplay sut = new TimeDisplay();
  sut.setTimeProvider(tpStub);
  // Exercise sut
  String result = sut.getCurrentTimeAsHtmlFragment();
  // Verify outcome
  String expectedTimeString = "<span
    class=\"tinyBoldText\">7:25 AM</span>";
  assertEquals("Non special time", expectedTimeString,
    result);
}
```

> about the author

Jeroen Mengerink

Jeroen works as a test consultant for Polteq. In addition to his work for clients, he is involved in various test innovations. His main area of expertise is Agile, for which he is the person to talk to within Polteq. Jeroen teaches several test courses, e.g. about Agile, SOA and Cloud, and is a teacher of the Certified Agile Tester course (CAT). He is co-author of the book and approach Cloutest® on how to test when cloud computing is involved. He has contributed as a speaker to a number of internal and external events for Polteq and its clients. In several international assignments he has presented the results of TPI assessments to a variety of senior management. He has presented several times at events like ChinaTest, Eurostar and TestNet on a large variety of subjects.

Twitter: @angusvb