How to Test the Extract Method Refactoring

Ioannis Megas
Department of Computer Science and Engineering, University of Ioannina, Greece
johnybg80@gmail.com

Apostolos V. Zarras
Department of Computer Science and Engineering, University of Ioannina, Greece
zarras@cs.uoi.gr

Chris Karelis
Department of Computer Science and Engineering, University of Ioannina, Greece
chKarelis@gmail.com

ABSTRACT

Extract Method is probably the most commonly used refactoring. Although it may seem quite simple at a first glance, there are many cases that it can become fairly complex. Local variables and parameters can make this refactoring difficult to implement. The whole procedure can also become quite error-prone. The only way to be sure that we performed the refactoring without introducing bugs is by means of testing. Testing, however, seems to many application developers a very bothersome procedure. To facilitate their work we discuss patterns that customize existing testing techniques to the specificities of the Extract Method refactoring.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**;

KEYWORDS

Extract Method, Testing, Patterns

ACM Reference Format:
Ioannis Megas, Apostolos V. Zarras, and Chris Karelis. 2020. How to Test the Extract Method Refactoring. In *European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20), July 1–4, 2020, Virtual Event, Germany.* ACM, New York, NY, USA, 7 pages.

1 INTRODUCTION

One of the most common [7–10, 12] and, at the same time, one of the most interesting refactorings is Extract Method [4]. Extract Method is probably the first refactoring that comes to mind when we decide to clean our code. It helps us make our methods smaller and our code significantly more readable. Extract Method is frequently used to facilitate other, more complex refactorings [4, 11, 13]. *The idea is to extract a code fragment from a source method in a new target method and call the target method in the body of the source method.* In many cases, Extract Method is simple and straightforward. However, this is not always true. Local variables and parameters increase the complexity of the refactoring. There is even the possibility that the extraction becomes so awkward that we need to resort to other refactorings, before using Extract Method [4, 11, 13]. The mechanics of Extract Method are discussed in detail in Martin Fowler's catalog of refactorings, along with examples and hints on how to combine Extract Method with other refactorings (ch.6 in [4]).

Nowadays, several popular IDEs provide automated support for Extract Method. Automated tools reduce the risk of introducing bugs in the code, while refactoring. However, several empirical studies show that often the automated refactoring tools are underused [7–10, 12]. Even if we use an automated refactoring tool, it is always possible that the tool is buggy, or that we accidentally introduce a bug while using it. Moreover, the refactoring tools do not always perform exactly as we expect [3].

In any case, the only way to verify that we did not introduce bugs, while refactoring our code is to test the code after refactoring. Consequently, the last step in the Extract Method mechanics [4] concerns running tests. The problem, however, is that the mechanics of the pattern do not provide us with many details regarding what to test and how. The answers to these issues are not always obvious. In the state of the art, there are techniques, guidelines and tips for testing code after change [1–3, 5, 6]. Choosing amongst them and using them in the context of Extract Method refactorings, is not straightforward, especially for inexperienced developers.

To deal with this issue, in this paper we discuss patterns that combine existing testing techniques and explain how to use them for testing Extract Method refactorings.

The target audience for the patterns is the developer who wants to test his code after having applied the Extract Method refactoring. The starting point is conventional testing techniques, which are customized to the specificities of the Extract Method refactoring. The patterns focus on testing the extracted method and the refactored method. Concerning the testing of the refactored method, existing tests are exploited, along with the development of new tests to guarantee a desired level of coverage.

In Section 2, we discuss Test the Method Behavior; this is the core pattern that allows testing whether the behavior of the code is preserved after the Extract Method refactoring. In Section 3, we discuss Test the Modifications that complements the core pattern; this pattern provides further guarantees that all the modifications to the code have been covered with tests. Finally, in Section 4 we provide a summary of our contribution and discuss future perspectives.

2 TEST THE METHOD BEHAVIOR

Context

An application developer wants to refactor a method that is long and complex. To this end, he wants to use Extract Method to extract a code fragment from `sourceMethod()` to a new method, `extractedMethod()`.

EuroPLoP '20, July 1–4, 2020, Virtual Event, Germany

ACM ISBN 978-1-4503-7769-0/20/07…$15.00
DOI: 10.1145/3424771.3424793

Problem

The developer wants to execute tests, to make sure that he will not break anything after the refactoring, but he does not know what to test.

Forces

- The developer may perform the refactoring manually or automatically. Manual refactoring increases the risk of introducing bugs in the code. However, automated refactoring tools may also be buggy, or they may simply not work as expected by the developer.
- `sourceMethod()` has certain responsibilities that characterize the method's behavior. Refactoring `sourceMethod()` may introduce bugs that lead to unexpected behavior.
- The test base of the project may include test cases for `sourceMethod()`.
- Writing and maintaining tests is a time and resource consuming process. Testing everything is not realistic.

Solution

The developer should run tests that check if the behavior of `sourceMethod()` after the refactoring is the same as before the refactoring. To achieve this, the developer identify in the test base of the project tests that record the actual behavior of `sourceMethod()`, before the refactoring. Typically, these are called *characterization tests*. If the test base does not include tests for `sourceMethod()`, the developer should write characterization tests, before the refactoring. After the refactoring, the developer should run the characterization tests to check if the behavior of `sourceMethod()` remains unchanged.

Hints

How to develop characterization tests. The development of a characterization test can be based on the application requirements or the method documentation. However, the characterization test does not aim at finding a bug. Its purpose is to document the actual method behavior, no matter if this is correct or not. Therefore, a better way to record the actual behavior of the method is to rely on the method itself.

Specifically, a technique for developing a characterization test (ch. 13 [3] for a method is to write an assertion that will fail for sure. Let the failure report what the actual behavior should be and then change the test so that it expects the behavior that the method produces. Depending on the effects of the method, the assertion may check *a return value, the state of an instance variable of the class, the state of an object that is passed as parameter to the method, the value of some global variable/object that is modified by the method* [3].

Another technique for developing a characterization test (ch. 6 [2]) is to make a hypothesis about how the method behaves, encode the hypotheses in a test and run the test to see if the hypothesis holds. If it does the test qualifies as a characterization test.

How to find existing characterization tests in the test base. A simple way for finding in the test base tests that concern `sourceMethod()`, is to track the callers of the method, using the facilities of the IDE.

How to check if the characterization tests account for all the code modifications. A way for checking whether the existing tests cover all the code modifications that heve been performed in `sourceMethod()` and `targetMethod()`, is to instrument the code of `sourceMethod()` with sensing variables [3], before running the tests, to trace the parts of the code that execute when running the tests, and remove the sensing variables after testing. Another way for checking, is to use use the debugger, or other IDE facilities for test coverage calculation.

Consequences

- The pattern increases the developer's confidence that the behavior of `sourceMethod()` is preserved after the refactoring, no matter if he performs it automatically, or manually.
- The pattern facilitates the reuse of existing tests and the incremental expansion of the test base.
- The tests do not target specific statements or execution paths. Hence, they can be reused for testing future refactorings, even if the implementation of `sourceMethod()` is significantly changed.
- Recording the actual behavior of `sourceMethod()` may not be easy; the related documentation may be incomplete or inconsistent, with respect to the actual implementation. The implementation of the method may be hard to comprehend. Hence, the tests may not cover all the responsibilities of `sourceMethod()`.
- The refactoring of `sourceMethod()` may involve changes to responsibilities that are not covered by characterization tests.

Example

To illustrate the pattern, we re-implement the triangle example from [5] (Figure 1). The `Triangle` class is used for the manipulation of `Triangle` objects. The class has four instance variables: `a`, `b`, `c` correspond to the sides of a `Triangle` object; the `type` of a `Triangle` object can be EQUILATERAL, ISOSCELES, or SCALENE.

The `Triangle` constructor is quite long with complex conditional statements. The developer wants to refactor the constructor using Extract Method, to make the code simpler and more readable.

To make the constructor simpler and more readable, the developer plans to extract the complex nested conditional statement that sets `type` to a new method, called `setType()`.

To make sure that the refactoring will not break anything, he should Test the Method Behavior. The test base of the application does not include tests for the `Triangle` constructor. Hence, he starts by writing some characterization tests using the JUnit framework. *The purpose of the tests is to make sure that after the refactoring the constructor behaves as it did before the refactoring.* To record the actual behavior of the constructor before the refactoring the developer writes three tests that create three `Triangle` objects `t1`, `t2`, `t3` of different type, equilateral, isosceles and scalene, respectively. The developer uses the `toString()` method of the `Triangle` class to write the assertions of the tests that check the state of the created objects.

In a first step, the developer writes the three tests with assertions that would most likely fail (Figure 2). In particular, the assertions assume that the state of the objects is an empty string. He runs the tests before the refactoring and the tests fail. The fail traces

```
public class Triangle {
      private static int EQUILATERAL = 0;
      private static int ISOSCELES = 1;
      private static int SCALENE = 2;

      private int a;
      private int b;
      private int c;
      private int type;

      public Triangle(int sA,int sB,int sC)
                  throws NotTriangleException {

            if(!((sA > 0)&&(sB > 0)&&(sC > 0)))
                  throw new NotTriangleException(
                              "Invalid Side Lengths");

            else if ((sA >= sB + sC) ||
                        (sB >= sA + sC) ||
                        (sC >= sA + sB)) {
                  throw new NotTriangleException(
                        "Inequality Violation");
            } else {

                  if ((sA == sB) && (sB == sC)) {
                        type = EQUILATERAL;
                  } else
                        if ((sA != sB) &&
                              (sB != sC) &&
                              (sA != sC)) {
                        type = SCALENE;
                  } else
                        type = ISOSCELES;

                  a = sA;
                  b = sB;
                  c = sC;
            }

      }

      public int calculatePerimeter() {
            return a + b + c;
      }

      public String toString() {
            return
                  a + " " +
                  b + " " +
                  c + " " + type;
      }
}
```

Figure 1: The implementation of `Triangle` before the extraction of `setType()`.

returned by JUnit report that the actual states of `t1`, `t2`, and `t3`, as set by the constructor, are "10 10 10 0", "10 5 10 1", and "11 5 15 2", respectively.

In a second step, the developer revises the implementation of the tests, with respect to the JUnit fail traces, so that the tests pass (Figure 3). At this point, the tests record the actual behavior of the constructor.

Now the developer can perform the actual refactoring to extract the `setType()` method (Figure 4). After the refactoring the developer re-executes the characterization tests to ensure that the constructor behaves as before.

Known Uses & Techniques

The characterization testing technique is introduced by Michael Feathers (ch. 13 [3]). Fowler also suggests to write tests that record method responsibilities before refactoring (ch. 4 [4]) and use these

```
class TriangleTest{

      @Test
      void test1() {
            try {

                  Triangle  t1 = new Triangle(10, 10, 10);
                  assertEquals(" ", t1.toString());

            } catch (NotTriangleException e) {
                  e.printStackTrace();
            }
      }

      @Test
      void test2(){
            try {

                  Triangle  t2 = new Triangle(10, 5, 10);
                  assertEquals(" ", t2.toString());

            } catch (NotTriangleException e) {
                  e.printStackTrace();
            }
      }

      @Test
      void test3() {
            try {

                  Triangle t3 = new Triangle(11, 5, 15);
                  assertEquals(" ", t3.toString());

            } catch (NotTriangleException e) {
                  e.printStackTrace();
            }
      }
}
```

Figure 2: Failing tests for `Triangle`.

tests to make sure that the behavior remains unchanged after refactoring.

Related Patterns

Write Tests to Understand is a pattern reported by Serge Demeyer et al. (ch. 6 [2]) to facilitate the understanding of a legacy system. The pattern suggests to record hypotheses and conclusions about the code in terms of executable tests. Record Business Rules as Tests is another pattern introduced by the same authors that suggests recording the main business rules of the legacy system as tests and using the tests to make sure that the business rules are still valid during and after the system re-engineering. Serge Demeyer et al. further discuss Grow Your Test Base Incrementally, a pattern that deals with the reuse of existing tests and the incremental growth of the test base.

What Next

If the refactoring of `sourceMethod()` involves changes to responsibilities that are not covered by characterization tests the developer can use Test the Modifications that follows.

3 TEST THE MODIFICATIONS

Context

An application developer wants to refactor `sourceMethod()` that is long and complex. To this end, he wants to use Extract Method to extract a code fragment from `sourceMethod()` to a new method,

```
class TriangleTest {

	@Test
	void test1() {
		try {

			Triangle t1 = new Triangle(10, 10, 10);
			assertEquals("10 10 10 0", t1.toString());

		} catch (NotTriangleException e) {
			e.printStackTrace();
		}
	}

	@Test
	void test2(){
		try {

			Triangle t2 = new Triangle(10, 5, 10);
			assertEquals("10 5 10 1", t2.toString());

		} catch (NotTriangleException e) {
			e.printStackTrace();
		}
	}

	@Test
	void test3() {
		try {

			Triangle t3 = new Triangle(11, 5, 15);
			assertEquals("11 5 15 2", t3.toString());


		} catch (NotTriangleException e) {
			e.printStackTrace();
		}
	}

}
```

Figure 3: Revised tests for `Triangle`.

`extractedMethod()`. After the refactoring the developer checks if `sourceMethod()` behaves as before, using Test the Method Behavior. However, the developer realizes that the characterization tests that he used do not cover all the modifications that he performed during the refactoring.

Problem

The developer wants to perform more detailed testing to verify all the modifications that he made in the body of `sourceMethod()` and `extractedMethod()`.

Forces

- The refactoring may involve changing several statements in the body of `sourceMethod()`. Specifically, the developer must move the extracted code statements to `extractedMethod()` and modify the body `sourceMethod()` to invoke `extractedMethod()`. The developer may also have to modify the extracted code statements after moving them to `extractedMethod()`, so as to deal with local variables, parameters and return values that are needed to facilitate the invocation of `extractedMethod()`. Modifying a code fragment, be it the body of `sourceMethod()`, or the extracted code fragment, is a risk for introducing bugs.
- There may be several different execution paths that pass from the modified statements.
- Writing and maintaining tests is time and resource consuming process. Testing everything is not realistic.

```
public class Triangle {
	private static int EQUILATERAL = 0;
	private static int ISOSCELES = 1;
	private static int SCALENE = 2;

	private int a;
	private int b;
	private int c;
	private int type;

	public Triangle(int sA,int sB,int sC)
			throws NotTriangleException {

		if(!((sA > 0) && (sB > 0) && (sC > 0)))
			throw new NotTriangleException(
				"Invalid Side Lengths");

		else if ((sA >= sB + sC) ||
				(sB >= sA + sC) ||
				(sC >= sA + sB)) {
			throw new NotTriangleException(
				"Inequality Violation");
		} else {

			setType(sA, sB, sC);

			a = sA;
			b = sB;
			c = sC;
		}

	}

	private void setType(
			int sA, int sB, int sC) {

		if ((sA == sB) && (sB == sC)) {
			type = EQUILATERAL;
		} else
			if ((sA != sB) &&
				(sB != sC) &&
				(sA!=sC)) {
			type = SCALENE;
		} else
			type = ISOSCELES;
	}

	public int calculatePerimeter() {
		return a + b + c;
	}

	public String toString() {
		return
			a + " " +
			b + " " +
			c + " " + type;
	}
}
```

Figure 4: The implementation of `Triangle` after the extraction of `setType()`.

Solution

The developer should run tests that pass from the all the modified statements. For more detailed testing, he should run tests that execute all the feasible execution paths that pass from the changed statements.

Hints

Typically, statement and path testing relies on the concept of *a program graph* (e.g., ch. 8 [5], ch. 3 [1]) that represents a method. The program statements are the nodes and the flow of control is depicted by the edges of the program graph. In statement testing, the goal is to execute tests that pass from the nodes of the graph.

```
public class Triangle {
      private static int EQUILATERAL = 0;
      private static int ISOSCELES = 1;
      private static int SCALENE = 2;

      private int a;
      private int b;
      private int c;
      private int type;

      public Triangle(int sA,int sB,int sC)
                        throws NotTriangleException {

            if(!((sA > 0) && (sB > 0) && (sC > 0)))
                  throw new NotTriangleException(
                              "Invalid Side Lengths");
            else if ((sA >= sB + sC) ||
                        (sB >= sA + sC) ||
                        (sC >= sA + sB)) {
                  throw new NotTriangleException(
                        "Inequality Violation");
            } else {

                  setType(sA, sB, sC);

                  a = sA;
                  b = sB;
                  c = sC;
            }
      }

      private void setType(
                  int sA, int sB, int sC) {

            if ((sA == sB) && (sB == sC)) {
                  type = EQUILATERAL;
            } else
                  if ((sA != sB) &&
                        (sB != sC) &&
                        (sA!=sC)) {
                  type = SCALENE;
            } else
                  type = ISOSCELES;
      }
      public int calculatePerimeter() {
            return a + b + c;
      }

      public String toString() {
            return
                  a + " " +
                  b + " " +
                  c + " " + type;
      }
}
```

Figure 5: The implementation of `Triangle` before the extraction of `isNotTriangle()`.

In path testing, the goal is to execute tests that pass from the paths of the graph.

Consequences

- Testing all modifications provides more confidence that the refactoring did not introduce bugs in the implementation of `sourceMethod()` and `extractedMethod()`.
- Developing the tests may not be easy if the method implementation is complex.
- The tests are code based, i.e. they target specific statements and execution paths. Thus, they may become obsolete if the method implementation is significantly changed.

```
public class Triangle {
      private static int EQUILATERAL = 0;
      private static int ISOSCELES = 1;
      private static int SCALENE = 2;

      private int a;
      private int b;
      private int c;
      private int type;

1     public Triangle(int sA,int sB,int sC)
                  throws NotTriangleException {

2           if(!((sA > 0)&&(sB > 0)&&(sC > 0)))
3                 throw new NotTriangleException(
                              "Invalid Side Lengths");
4           else if (isNotTriangle(sA, sB, sC)) {

5                 throw new NotTriangleException(
                        "Inequality Violation");

6           } else {
7                 setType(sA, sB, sC);
8                 a = sA;
9                 b = sB;
10                c = sC;
11          }
12    }

      protected boolean isNotTriangle(
                  int sA, int sB, int sC) {

            return (sA >= sB + sC) ||
                        (sB >= sA + sC) ||
                        (sC >= sA + sB);
      }
      private void setType(
                  int sA, int sB, int sC) {

            if ((sA == sB) && (sB == sC)) {
                  type = EQUILATERAL;
            } else
                  if ((sA != sB) &&
                        (sB != sC) &&
                        (sA!=sC)) {
                  type = SCALENE;
            } else
                  type = ISOSCELES;
      }

      public int calculatePerimeter() {
            return a + b + c;
      }

      public String toString() {
            return
                  a + " " +
                  b + " " +
                  c + " " + type;
      }
}
```

Figure 6: The implementation of `Triangle` after the extraction of `isNotTriangle()`; the line numbers in the constructor body correspond to node labels of the program graph given in Figure 7.

Example

To simplify the constructor of the `Triangle` class, the developer extracted `setType()`. To verify that the refactoring did not affect the behavior of the constructor he developed a number of characterization tests.

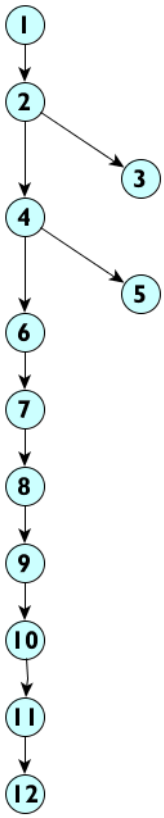

Figure 7: The program graph of `Triangle`; ; the node labels correspond to line number of the constructor body given in Figure 6.

However, the code of the constructor is still quite complicated (Figure 5). The developer decides to extract another method, named `isNotTriangle()` that checks if the triangle inequality holds, to make sure that the constructor's parameters define a valid triangle (Figure 6).

To test the constructor he uses once more Test the Method Behavior. This time, the test base of the project includes the characterization tests that he implemented for the extraction of `setType()`. The developer runs the tests that execute successfully. The existing characterization tests pass from the invocation of `isNotTriangle()`.

Nevertheless, the developer wants to be sure that he tested all the statements that have been modified and all the execution paths that pass from these statements. To this end, he draws a simple program graph for the constructor of the `Triangle` class (Figure 7). Based on the graph, he finds two paths that pass from the invocation of `isNotTriangle()`: 1, 2, 4, 6, 7, 8, 9, 10, 11, 12 and 1, 2, 4, 5. The second path throws an exception if the triangle inequality is violated. The developer realizes that this path is not covered by the existing tests, as they all pass from the first path. To cover this part of the code he develops an additional test (Figure 8). He runs the test that executes successfully. The new test records the constructor behavior in case

```
class InequalityTest {

      @Test
      void testTriangle() {
            assertThrows(
              NotTriangleException.class,
              () -> new Triangle(1000, 10, 20));
      }
}
```

Figure 8: Testing triangle inequality violations.

of invalid parameters. Hence, it can also be part of the application test base, to be used as a characterization test for future refactorings.

Known Uses and Techniques

Statement and path testing are two well-known code based testing techniques that can be found in almost every software engineering text book (e.g., ch. 8 [5], ch. 3 [1]). Michael Feathers introduced the notion of targeted testing (ch. 13 [3]). The idea is to focus statement and path testing on the modified method statements and paths, respectively.

4 CONCLUSION

In this paper, we introduced testing techniques in the form of patterns for testing the Extract Method refactoring. The proposed patterns customize conventional testing techniques, with respect to the Extract Method refactoring.

Concerning the future perspectives of this work, a interesting challenge is to provide automated support that facilitates the application of the proposed testing patterns. Another possible research issue is to look for further testing patterns for Extract Method, or other popular refactorings.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Uwe van Heesch and the members of our writers workshop for their constructive comments and suggestions, during the preparation of this paper.

REFERENCES

[1] 2014. *SWEBOK v3: IEEE Software Engineering Body of Knowledge.* IEEE.
[2] S. Demeyer, S. Ducasse, and O. Nierstrasz. 2009. *Object-Oriented Reengineering Patterns.* Square Bracket Associates.
[3] M. C. Feathers. 2009. *Working Effectively with Legacy Code.* Prentice Hall.
[4] M. Fowler. 2000. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.
[5] C. P. Jorgensen. 2014. *Software Testing, a Craftsmans Approach.* Taylor and Francis Group.
[6] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. 2017. Effective Regression Test Case Selection: A Systematic Literature Review. *ACM Compututing Surveys* 50, 2 (2017), 29:1–29:32.
[7] M. Kim, T. Zimmermann, and N. Nagappan. 2014. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649.
[8] E. Murphy-Hill, C. Parnin, and A. Black. 2009. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE).* 287–297.
[9] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle. 2013. A Multidimensional Empirical Study on Refactoring Activity. In *Proceedings of the ACM Conference of the Center for Advanced Studies on Collaborative Research (CASCON).* 132–146.
[10] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. 2012. Use, Disuse, and Misuse of Automated Refactorings. In *Proceedings of the 34th International Conference on Software Engineering (ICSE).* 233–243.

[11] T. Vartziotis, A. V. Zarras, A. Tsimakis, and P. Vassiliadis. 2020. Recommending Trips in the Archipelago of Refactorings. In *Proceedings of the 46th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*. to appear.

[12] Z. Xing and E. Stroulia. 2006. Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM)*. 458–468.

[13] A. V. Zarras, T. Vartziotis, and P. Vassiliadis. 2015. Navigating through the Archipelago of Refactorings. In *Proceedings of the the Joint 23rd ACM SIGSOFT Symposium on the Foundations of Software Engineering and 15th European Software Engineering Conference (FSE/ESEC)*. 922–925.