Test Maintainability

Improving test maintainability after refactoring with AST-Rewrite based advice

S.J. Hanique

Student: 851749513
Date: 12/06/2017

TEST MAINTAINABILITY

IMPROVING TEST MAINTAINABILITY AFTER REFACTORING WITH AST-REWRITE BASED ADVICE

by

S.J. Hanique

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering
to be defended publicly on Monday June 26, 2017 at 15:00 PM.

Student number:	851749513	
Course code:	IM9906	
Thesis committee:	Dr. ir. H. Passier (chairman),	Open University
	Dr. ir. S. Stuurman (supervisor),	Open University

Open Universiteit
www.ou.nl

CONTENTS

List of Figures

List of Tables

1 INTRODUCTION

Nowadays, many software systems are evolving continuously, with maintenance modifications to the source code as a result. Due to the large numbers of code modifications, it is difficult to maintain code quality. Changes could reduce the code quality even if no errors are induced.

There are several software maintenance activity types [International Standards Organisation (ISO), 1999]:

- **Adaptive maintenance**: software adaptation to remain functional in case of a changing software environment. For example, to upkeep the interaction with an operating system or other external software packages.
- **Corrective maintenance**: software error identification and resolving.
- **Perfective maintenance**: software functionality implementation due to changing user requirements.
- **Preventative maintenance**: software structure improvements by cleaning and reorganization the source code to increase reliability and readability among others.

Preventative maintenance can be necessary as a result of so called bad smells in the source code.

A bad smell [1] (also known as code smell) is a certain code indication that hints to a possible flaw in design or a structural problem [Fowler et al., 1999]. Some examples of bad smells are: duplicated code, large/god classes, long methods, call/message chains and overuse of comments.

Code containing a bad smell is not necessarily incorrect; it can still function as intended. Code smells are considered 'bad' because of the possible future issues they may cause. Note that the emphasis lies on the word 'possible', as not all code smells lead to a negative impact on the software [Zhang et al., 2011]. An example of a bad code smell is a code duplication situation where a duplicated part needs to be updated. If one overlooks

[1] M. Fowler, "*CodeSmell*". Retrieved on 17-08-2016 from http:martinfowler.com/bliki/CodeSmell.html

a problem with another duplication part, a bug may have been induced. Another example is a long method. In this case, the concern is that its internal functionality is difficult to comprehend and thus to maintain. The method is too complex and has too much information clumped together [Fowler et al., 1999]. Bad smells can be treated by a process called refactoring [Fowler et al., 1999].

Refactoring is "*the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure*" [Fowler et al., 1999]. This definition consists of two important parts: (1) maintaining external code behaviour and (2) improving the program structure.

The first part of the definition describes that the preservation of behaviour is key. The functioning of the code has to remain the same, because it is the structure that is incorrect, not the functioning. This is opposed to changes, for example, fixing bugs, where behaviour of code is changed because it is proven faulty. The second part describes the goal of improving the structure of the software system. A better structure improves maintainability, reduces software complexity and thus overall makes the system easier to understand [Fowler et al., 1999, Mens and Tourwé, 2004].

Another common way to improve software maintenance is by using tests. The meaning and correctness of source code is often checked by (unit) tests, especially in Test-Driven Software Development (TDD). TDD is an agile software development method, and its essential part is to write tests beforehand [Beck, 2002]. Other agile development methods, for example eXtreme Programming (XP), use the TDD approach. In short, the cycle of TDD is as follows: at first a test is created to test a future requirement. The tests are then executed and the new test should fail, because the actual code has not yet been written. The next step is to write the code for the requirement, but just enough to make it pass the test. Then the tests are executed again, and now all tests should pass. The last step is to rewrite and clean the code; this is where the refactoring process takes place. When all steps are performed, the cycle starts again with a new iteration [Beck, 2002]. Refactoring and (unit)tests are a key aspects of XP.

A problem arises when refactoring code with corresponding tests. Applying certain refactorings on the code can affect its tests. When, for example, a refactoring introduces a new method, the program's tests are no longer complete. Therefore, changing the test code after a refactoring seems inevitable [Passier et al., 2016, Moonen et al., 2008]. But, what if we could ease the process of revising tests?

The main purpose of this research is to create advice based on the results of a refactoring on the tests. This is a complement to a prior study [Passier et al., 2016]. In short, this prior study describes the need for tests to change subsequent to a refactoring in its corresponding code. Both refactoring and TDD are key aspects of agile development, preserving test code quality and coverage in association with refactoring is important.

Currently, a Java Eclipse plug-in has been created that is capable of detecting refactorings based on AST-rewrites. As output, a list of AST-rewrites for the involved JUnit test cases are listed. These AST-rewrites are similar to what is also known as micro-refactorings [Passier et al., 2016, Schäfer et al., 2009]. More on AST-Rewrites and micro-refactorings will follow in a later chapter.

The presentation of the advice given by the plug-in is currently a mere draft [Passier et al., 2016]. One refactoring could result in more than one smaller piece of advice, which may be more difficult to comprehend. In addition, at the moment, this advice has to be processed manually.

The goal is to develop the given concept and improve the advice. A possible way of achieving this, is to reassemble the listed AST-Rewrites into a (larger), more readable, advice. If applicable, this advice could be presented as a refactoring (as defined by Fowler [Fowler et al., 1999]). Ideally this method could be extended to also reassemble AST-Rewrites into an advised design pattern, as a design pattern often consist of several refactorings. To accomplish this goal, the current results of the study and the output including the limitations of the current plug-in should be analyzed. Providing an advice, independent of the integrated development environment's implementation of a specific refactoring, is also a necessity. The presentation and way of giving advice are of importance as well. Eventually the refactoring advice could be made executable, which means that the advice can be directly applied if the developer chooses to.

1.1. RESEARCH QUESTIONS

The aforementioned goals are converted and divided into five separate research questions:

- **RQ1: Which different output combinations can the plug-in return?**
  This research question is aimed at examining the data to work with. For example, finding a common pattern in the output combinations, would provide a base for advice determination.

- **RQ2: Can the original refactoring be requested from a set of AST-Rewrites, and if not, can AST-Rewrites be recombined to reveal the initial refactoring?**
  Information on the original refactoring holds valuable data on, among others, the location and parameters of a refactoring. This data is necessary in order to provide a certain detail to the advice.

- **RQ3: Is it possible to group AST-rewrites into a larger standardized advised change, and if so, which standards are applicable to present the advice in?**
  With an advice standard we mainly want to achieve readability, but with future additions in mind, we also want to try and support automation.

- **RQ4: Can the refactored code itself contribute to the advice?**
  As the refactored code itself literally shows the exact change, we would like to research if this data provides additional information for the advice creation.

- **RQ5: Is it possible to automate the advised changes with available refactorings in the Eclipse IDE or JDT?**
  With the automation of advice in mind, we want to research if (and how) the possible standard of RQ3 can be automated.

1.2. RESEARCH APPROACH

During this research we will use the 'Design and Creation' approach [Oates, 2006]. With this approach, an artifact will be delivered with the goal of knowledge contribution. This research will include the following artifacts: constructs, models, methods and instantiations. Constructs include: refactoring, micro-refactoring, abstract syntax tree, AST-Rewrite and JUnit. The models will be UML diagrams to aid the problem understanding and to aid the creation of the plug-in. Methodologies include ways to determine the impact of AST-Rewrites and its corresponding code change on the tests, and to give advice based on the determined impact. The instantiation is a proof of concept plug-in addition which demonstrates the results of this research. During the design and creation process, additional requisite knowledge will be obtained via a literature study.

1.3. RESEARCH DOCUMENT STRUCTURE

This research document is organized as follows. Chapter 2 further describes the concepts of refactoring and AST-Rewrites. In Chapter 3 we take a closer look at the different AST-Rewrite types, by examining several refactoring examples. Chapter 4 discusses how to determine the exact change of a refactoring and describes how certain changes can impact tests. In the fifth Chapter we provide an approach to create an advice based on the determined impact, whilst in Chapter 6 we provide a way to present said advice. Chapter 7 describes the creation of our own advice addition plug-in, which is an extension to the 'Retestoring' plug-in. For Chapter 8 we take a look at other literature that is related to our research. In Chapter 9 we discuss and evaluate our contributions and provide a final conclusion for our research. Lastly, Chapter 10 lists future work.

2

Refactoring and AST-Rewrites

This chapter provides an introduction to the important concepts of this research. First, we provide an introduction to the aspect of refactoring. Additionally, we provide a description of the different refactoring types and the limitations of refactorings in general. Furthermore, we show that a refactoring is essentially a set of smaller changes, which can be described by two different techniques. Next, we provide a definition of the concept 'test', and provide examples of how a test code environment could look like. Lastly, we describe the current status of the 'Retestoring' plug-in.

2.1. Refactoring Examples and Counterparts

A bad smell can be resolved by one or more refactorings. For instance, when the functionality of a method is not clear by its name, the 'Rename Method' refactoring is applicable. This refactoring is nothing more than changing a method's name to reflect its true purpose. In the background, every call to this method has to be changed as well.

Another refactoring is the 'Extract Method' refactoring. This refactoring can be used to shorten the length of a long method. In this case, code in the method's body can be grouped together and extracted as a separate method. Long methods often perform more than one distinct task. These (sub) tasks or functionalities can be separated, which make the code more readable and thus easier to maintain. However, one can get too optimistic and create a method for every few lines of code. When (extracted) methods are "no longer pulling their weight", one can replace the method call with its body [Fowler et al., 1999]. This is the opposite of the extract method refactoring, and is called 'Inline Method'. In Table 2.1, we show a code example of an 'Extract Method' refactoring for a print price calculation method.

Table 2.1 shows a method for calculating the price of a print order. In this example, two 'Extract Method' refactorings take place. Because the calculation method is long and consists of several separable steps, the 'Extract Method' refactoring is applicable. The calculation of the page price and colour price can be extracted from the method as two distinct methods. Although this simple refactoring may not make the code shorter, it is easier to read and better maintainable. When, for example, another colour price variant is introduced, only the calculateColorPrice method has to be edited, while the other pricing methods remain intact.

Extract method refactoring example

Before refactoring

```
public double calculatePrintPrice(int amountPages, boolean colored,
    int quality) {
    double price = 0;
    double colorPrice = BLACK_WHITE_PRICE;
    if (colored) {
      colorPrice = COLOR_PRICE;
    }
    double pagePrice = 0;
    switch (quality) {
    case 1:
      pagePrice = BASE_RATE * colorPrice;
    case 2:
      pagePrice = PLUS_RATE * colorPrice;
    case 3:
      pagePrice = HIGH_RATE * colorPrice;
    }
    price = amountPages * pagePrice;
    return price;
  }
```

After refactoring

```
public double calculatePrintPrice(int amountPages, boolean colored,
    int quality) {
    double price = 0;
    double colorPrice = calculateColorPrice(colored);
    double pagePrice = calculatePagePrice(quality, colorPrice);
    price = amountPages * pagePrice;
    return price;
  }

  private double calculateColorPrice(boolean colored) {
    double colorPrice = BLACK_WHITE_PRICE;
    if (colored) {
      colorPrice = COLOR_PRICE;
    }
    return colorPrice;
  }

  private double calculatePagePrice(int quality, double colorPrice) {
    double pagePrice = 0;
    switch (quality) {
    case 1:
      pagePrice = BASE_RATE * colorPrice;
    case 2:
      pagePrice = PLUS_RATE * colorPrice;
    case 3:
      pagePrice = HIGH_RATE * colorPrice;
    }
    return pagePrice;
  }
```

Table 2.1: Extract Method Refactoring example

2.2. MICRO-REFACTORINGS

Performing refactorings manually shows that these refactorings actually consist of several sub steps. These (sub) steps of a refactoring are called micro-refactorings and each perform a small defined task [Schäfer et al., 2009]. As an example, we use the 'Rename Method' refactoring again, where the actual renaming is only a part of the refactoring. This refactoring also includes: finding all method references, keeping these references intact (locking and unlocking bindings) and verifying the new name [Schäfer et al., 2009]. Something that is simple to understand can thus still be a lot of (programming) work. In Table 2.2, we show a decomposition of the extract method refactoring as an example.

Table 2.2 illustrates the sub steps of the extract method refactoring. The first piece of code shows the highlighted code that is going to be extracted into a separate method. The first step is to create a new method and set the selected code as its body. The creation of the new method is not complete yet. The second step is to analyse which variables the new method needs in order to function. In this case it needs the variables `total` and `person` from the original method, which are set as the new method's parameters. The next step is to check which variables are referenced or need to be returned to the original method. The variable `total` is changed in the new method and used later on in the method. Therefore the new method must return the total value and change its return type. The last step is complete the extraction; give the new method a proper name and call the new method in the original method.

2.3. REFACTORING LIMITATIONS

The decomposition of a refactoring into smaller steps makes the limitations more clear. The example of Table 2.2 shows a limitation of the extract method refactoring. When the extracted part modifies two or more local variables of the method, the extract method refactoring is not applicable. Java has no keyword to solve this problem, in contrast with C#, where the '`ref`' keyword can be used. Solutions to this problem are other refactorings, for example, 'Replace Temp with Query', which extracts a local variable expression as a method [Fowler et al., 1999]. If the variable is used more than once, the refactoring 'Split Temporary Variable' can be used. This refactoring separates each temporary variable used in an expression. When too many local variables are used in the code to be extracted, the 'Replace Method with Method Object' is the final solution. This refactoring creates an object and turns the needed local variables into its fields [Fowler et al., 1999].

2.4. REFACTORING AUTOMATION

Refactorings as 'Extract Method' or 'Rename' are relative simple code changes for a programmer. But decomposition proves them to be more complex. When we take a look at large refactorings, it becomes even more complex, for instance, the separation of the domain logic from the presentation logic refactoring. This is a refactoring, which consists of several smaller refactorings. Even for humans, these large refactorings are difficult to perform.

Nowadays, there are many Integrated Development Environments (IDE's) that offer (partly) automated refactoring support. For example, Eclipse, IntelliJ and Visual Studio offer the programmer a fair amount of ready to use refactorings. This support can make refactoring more accessible and less human error prone [Schäfer et al., 2009]. However,

Extract method refactoring in micro-refactorings

0: Original code

```
public void calculateTotalSalary() {
    int total = 0;
    for(Person p : getAllPersonnel()) {
      printLine("name:  " + p.name);
      total += p.salary;
      }
    printLine("total salary: " + total);
  }
```

1: New method creation

```
private void newMethod() {
    printLine("name: " + p.name);
    total += p.salary;
  }
```

2: Method parameters

```
private void newMethod(int total, Person p) {
  printLine("name: " + p.name);
  total += p.salary;
}
```

3: Return type and value

```
private int newMethod(int total, Person p) {
    printLine("name: " + p.name);
    total += p.salary;
    return total;
  }
```

4: Method call and result

```
public void calculateTotalSalary() {
    int total = 0;
    for(Person p : getAllPersonnel()) {
      total = addSalaryToTotal(total, p);
      }
    printLine("total salary: " + total);
  }

  private int addSalaryToTotal(int total, Person p) {
    printLine("name: " + p.name);
    total += p.salary;
    return total;
  }
```

Table 2.2: The micro-refactorings of an extract method refactoring

this does not mean that the automated refactorings cannot induce any errors, as programmers can decide to ignore warnings and execute an error inducing refactoring. Aside from that, several Java refactorings implementations contain bugs themselves [Hafiz and Overbey, 2015].

Refactoring has several types: (full) manual refactoring, automatic refactoring and automated refactoring.

Full manual refactoring means that the developer has no assists from smell or refactoring tools whatsoever. The advantage of this approach is that the developer is perfectly aware of every refactoring change to the code. The disadvantage is the amount of work it takes to perform every refactoring by hand and the increased chance of inducing errors [Schäfer et al., 2009].

Automatic refactoring is a program that finds the bad smells for you, it selects corresponding refactorings and executes them. The key element is that the developer has little to no interaction with the refactoring program. The downside to this approach is, that it is easy to lose track of the changes on the system.

Automated refactoring is a balance between interaction and automation. It also can perform refactorings, and depending on the tool, search for bad smells and advice refactorings. The difference is that the developer keeps the initiative. In a refactoring supportive IDE, the developer can decide when to refactor. For example, an automated refactoring can be as easy as: selecting a block of code, choosing a refactoring from a list of refactorings and providing the desired parameters.

2.5. ABSTRACT SYNTAX TREE & REWRITES

An Abstract Syntax Tree (AST[1]) is a tree representing the abstract syntactic structure of the source code of a program. A node in the AST corresponds to a certain block of code structure in the source code.

The Java Development Tools (JDT[2]) provide the structure for all Java projects in the Eclipse workspace represented as an AST. For example, the root node, represents a Java file. The child nodes contain AST representations of the package declaration, import declarations and types (classes, interfaces, etcetera). Class type AST nodes also have children, which are its method declarations. The method declarations contain the method's statements as children, and so on[1]. The nodes also have additional information, for example, a method declaration node has a property that shows the AST of the method's implementation.

An Abstract Syntax Tree Rewrite (AST-Rewrite[3]) is a change on a node of an abstract syntax tree. An AST-Rewrite describes the structural change. Changes on the source code of a program can be reflected as changes on nodes of an AST. When, for example, a refactoring is executed, nodes in the AST are either inserted, removed or replaced [Passier et al., 2016].

[1]T. Kuhn and O. Thomann, "Abstract Syntax Tree". Retrieved on 08-09-2016 from https://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html
[2]http://www.eclipse.org/jdt/
[3]http://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/rewrite/ASTRewrite.html

Before we move on, we will explain the different rewrite operations and some variants with an extra 'placeholder' value. To explain the AST-Rewrite variants, we use an example situation where we move code from one class to another. There is no 'move' AST-Rewrite, therefore, a move has to be realized as a combination of rewrites.

Table 2.3 is an example program, where the highlighted code will be moved to another class. This example shows two operations: first, the removal of the method in '`class1`', and second the insertion of the same method into '`class2`'. Thus, a move is essentially a combination of a removed rewrite and an inserted rewrite. However, this inserted rewrite is actually an insert with placeholder. The placeholder's essential function is to hold data during a move or replace, to temporarily 'store' a value until it is used. In the example of Table 2.3, it is important to keep the original code, as it still needs to be inserted elsewhere. Thus, the value is stored in a placeholder before the original is removed. Next, an insert (or replace) rewrite will put this placeholder value at the new location node in the AST. This placeholder is not a node in the AST, but a separate 'String' value.

Program code before move	

`public class class1 {`
`  public void method1(int param) {`
`    //Do stuff...`
`  }`
`}`
	`public class class2 {`
	
	`}`

Program code after move	

`public class class1 {`
 	
`}`
	`public class class2 {`
	`  public void method1(int param) {`
	`    //Do stuff...`
	`  }`
	`}`

Table 2.3: Example of a move in a simple program

2.5.1. A REFACTORING IN AST-REWRITES

The Extract method example in Table 2.4 can also be described as AST-Rewrites, based on a combination of changes.

Just as in Table 2.2, the highlighted code will be extracted. The first change is the 'Insert with placeholder' rewrite, which writes a temporary 'MISSING' node on the spot of the extracted code and keeps the body as placeholder data. The next step is the 'Insert' rewrite, to create a new method with the extracted code as its body. The last step is the 'Replace' rewrite, to replace the extracted code with a call to the new method.

AST-Rewrites and micro-refactorings can both be used to describe changes as a result of a refactoring. However, an AST-Rewrite is more abstract than a micro-refactoring. An AST-Rewrite shows the structural change in the program. It focuses on the change in nodes (rep-

Extract method refactoring in AST-Rewrites

0: Original code

```
public void calculateTotalSalary() {
    int total = 0;
    for(Person p : getAllPersonnel()) {
      printLine("name:  " + p.name);
      total += p.salary;
      }
    printLine("total salary: " + total);
  }
```

1: Insert with placeholder

Code	*Placeholder*

`public void`
`   calculateTotalSalary() {`
`  int total = 0;`
`  for(Person p :`
`     getAllPersonnel()) {`
`    MISSING();`
`  }`
`  printLine("total salary: " +`
`     total);`
`}`
	printLine("name: " + p.name);
	total += p.salary;

2: Insert

```
private int addSalaryToTotal (int total, Person p) {
    printLine("name: " + p.name);
    total += p.salary;
    return total;
  }
```

3: Replace

```
public void calculateTotalSalary() {
    int total = 0;
    for(Person p : getAllPersonnel()) {
      total = addSalaryToTotal(total, p);
      }
    printLine("total salary: " + total);
  }

  private int addSalaryToTotal(int total, Person p) {
    printLine("name: " + p.name);
    total += p.salary;
    return total;
  }
```

Table 2.4: Extract Method in AST-Rewrites

resenting blocks of code) and the corresponding operation (insertion, deletion or replacement) [Passier et al., 2016]. A micro-refactoring is a small sub step of a refactoring, which provides a better understandable and testable task. Micro-refactorings steps are changes on the code on a much lower level than AST-Rewrites. Therefore micro-refactorings show the challenges of each refactoring [Schäfer et al., 2009]. Examples of refactoring challenges are: the preservation of control and data flow, and the preservation of (name) bindings. To visualize the difference, compare Table 2.2 with Table 2.4. Challenges for the extract method example, are the determination of the parameter(s) and return value of the new method. The more abstract AST-Rewrites do not show these specific details.

2.6. JUNIT

In several agile development methods, (public) methods are accompanied by a corresponding test method in the corresponding test class. This research focuses on unit tests with the use of the JUnit framework, as this framework is commonly used in a TDD setting. It is important to provide the definition for the word ’test’ in this research context. With ’test’ we refer to a black-box JUnit unit test. Black-box means that the internal structure of the unit under test is not taken into account, hence the term. A unit test is a test that examines the behavior of a distinct unit [Tahchiev et al., 2010]. The main characteristic of a unit test, is that it is an independent task, and therefore not directly affected by the completion of other tasks. JUnit refers to a single method when mentioning a "distinct unit of work" [Tahchiev et al., 2010]. However, the definition of a unit is debatable, as a method, a class or a package can all be referred to as a unit [IEEE Standards Board, 1999]. In this research, we use the term ’unit’ to refer to a method.

The JUnit framework has features that make it easy to write and run tests. In JUnit, individual tests are represented by methods. Test classes or test cases contain these individual test [Tahchiev et al., 2010]. A test suite is a group of tests put together. If no suite is defined, a suite containing all methods in a test class will be automatically created. Lastly, the test runner is the "processor" that runs the test suites [Tahchiev et al., 2010]. In Table 2.5, we show a simple example of a unit test with corresponding code.

Code and corresponding JUnit test

Program code

```
public class MyCalculator {
  public double add(double x,
      double y){
    return x + y;
  }
}
```

Test code

```
public class MyCalculatorTest {
  @Test
  public void testAdd() {
    MyCalculator calc = new
        MyCalculator();
    double result = calc.add(5,
        -2);
    assertEquals(3,result);
  }
}
```

Table 2.5: An example of a test class

In the example of Table 2.5 there are a few thing worth noting. The ’`@Test`’ annotation in front of the test method marks the method as a unit test [Tahchiev et al., 2010]. The 'assertEquals' method is a special method within the JUnit framework to compare certain values. Instead of only one assertion, a test method can have multiple assertions. JUnit has a special test runner when testing with multiple different values is desired; parameterized tests. Table 2.6 shows an example of the parameterized test construct. The parametized class has the same basic structure as a normal test, but has an additional constructor and a create parameters method. This method creates and fills the list of parameter to run the test with, and can be recognized by the '@Parameters' annotation.

Parameterized test

```
@RunWith(value=Parameterized.class)
public class ParameterizedMyCalculatorTest {
  private double expectedValue;
  private double firstValue;
  private double secondValue;

  @Parameters
  public static Collection<Integer[]> getTestParameters() {
    return Arrays.asList(new Integer[][] {
      //{expected value, first parameter value, second parameter
         value}
      {2, 1, 1},
      {3, 2, 1},
      {4, 3, 1},
    });
  }
  public ParameterizedTest(double expectedValue
    double firstValue double secondValue){
    this.expectedValue = expectedValue;
    this.firstValue; = firstValue;
    this.secondValue; = secondValue;
  }
  @Test
  public void testAdd() {
    MyCalculator calc = new MyCalculator ();
    assertEquals(expectedValue, calc.add(firstValue, secondValue),
       0);
  }
}
```

Table 2.6: A parameterized example of a test class

Aside from parameterized tests, one can choose to use multiple assertions in a method, or use several distinct test methods to test the same method. This is however a matter of personal preference, as opinions differ on the elegance of these approaches and are sometimes even considered as 'test smells' [van Deursen et al., 2001].

2.7. Retestoring Plug-in

At this moment, a plug-in is available which is capable of detecting Refactorings via AST-Rewrites [Passier et al., 2016]. The 'Retestoring' plug-in uses the JDT, Aspect J and the Equinox Weaving feature. Aspect J[4] is a Java extension that adds Aspect-Oriented Programming (AOP). This allows modularity across units regarding design concerns. Equinox Weaving[5] makes it possible to use a tracing aspect within a JDT plug-in. Combined, these tools, extensions and features are used to trace refactorings.

The goal of the plug-in is to give advice about the state of tests after refactorings. The combination of AST-Rewrites can be used to generate a (generic) advice. For example, the extract method refactoring mentioned earlier, has as its main result a new (untested) method. The advice would therefore be to create a similar method in the corresponding test class [Passier et al., 2016]. Below, an example of a desired advice of the plug-in:

"Revise the test method or assertions for method 'public void myMethod(int a, int b)'. Create a new test method for the new method 'public void myNewMethod(int a, int b)' in the JUnit class for type 'public class MyClass' use knowledge from test method or assertions for method 'public void myMethod(int a, int b)'" [Passier et al., 2016].

There are, however, many possible refactorings (combinations of AST-Rewrites), which result in many possible advisable solutions. Previous research lists important AST-Rewrite combinations, with the corresponding generic advice on some refactorings [Passier et al., 2016]. However, this is just a first concept. Additionally, to give a more precise advice, it is important to take a closer look at the listed rewrite combinations. The main goal of this research is to create an as detailed as possible advice on how to maintain the tests, based on these AST-Rewrites (combinations).

2.8. Eclipse

This research is in particular focused on the Eclipse for Java IDE. The main reason is because the Eclipse for Java IDE build-in automated refactorings use the AST of the workspace to perform refactorings. Thus, a refactoring is essentially a set of AST-Rewrites, or in case of a small refactoring, one single AST-Rewrite.

It is possible to track AST-Rewrites as a result of a refactoring. It is not of great importance whether the refactoring is automatic or automated, as long as the changes are processed using AST-Rewrites. During a refactoring, rewrites on the AST can be traced with the use of the corresponding 'Rewrite Listener'. The rewrite event gives information about the parent node, which property changed, the type of change and the changed node itself[3].

Because these AST-Rewrites are already part of the process, it would be most logical to make use of them instead another standard.

2.9. Summary

In this chapter, we described that a bad smell can be resolved with a refactoring. Additionally, we provided example situations to elucidate when and how to refactor. We discussed that refactorings are not always directly applicable, and that a bad smell sometimes

[4]`https://eclipse.org/aspectj/`
[5]`http://www.eclipse.org/equinox/weaving/`

requires multiple (other) refactorings in order to be resolved. We have shown that a refactoring can be decomposed into a set of smaller changes, which can be presented as either micro-refactorings or AST-Rewrites. For this research we decided to use AST-Rewrite, because our target IDE already uses this technique to perform refactorings. Fortunately, the 'Retestoring' plug-in already tracks these rewrites for us, which provides us a great starting point for the advice determination. Lastly, we described that we focus on black-box unit tests for the JUnit test environment.

3
AST-Rewrite Types

In this chapter, we are going to further explain AST-Rewrites by examining their content. We provide several refactoring examples, were we focus on the corresponding AST-Rewrite output, provided by Retestoring plug-in.

Every refactoring results in a certain set of AST-Rewrites. A larger refactoring requires more rewrites. Rewrites are presented in a 'Rewrite group', which is essentially a set of AST-Rewrites. The rewrite groups are dependent on the implementation of the refactoring in Eclipse. Therefore, a rewrite group may or may not bear the name of its refactoring. It is also possible that a refactoring is divided into several separated rewrite groups. The Retestoring plug-in does show the rewrite groups, but due to the refactoring implementation inconsistencies in Eclipse, these barely have any value. Therefore, we will not include the rewrite groups in the following examples.

3.1. AST-Rewrite Attributes

Rewrites are often described as a change on a node. In a previous chapter, we mentioned that a node can represent different code elements, for example a class or a method. The node in which the rewrite took place is called the parent node. For example, in case of a rewrite describing a method insertion, the corresponding class is the parent node. An AST-Rewrite consist of several attributes, each describing a different part of the rewrite.

- **Changed node**
  The first attribute of a rewrite is the 'Changed node', which is the parent node that has been changed.

- **Property**
  The second attribute is the node's changed 'Property'. This attribute describes which sub element of the 'Changed node' changed. The two attributes together describe where the change happened. For example, a method declaration (node type) has it's body (property) changed.

- **Change**
  The 'Change' attribute reflects the type of change on the node. There are five distinct change types: insert, remove, replace, insert with placeholder and replace with placeholder.

  - *Insert* describes the insertion of a new node into the AST.
  - *Remove* describes the removal of a node in the AST.
  - *Replace* describes the overwrite of a node the AST in with another (new) node.
  - *Insert with placeholder* describes the insertion of a new node into AST, in particular a node with values derived from another node.
  - *Replace with placeholder* describes an overwrite of a node the AST in with another (new) node, in particular a node with values derived from another node.

- **Original value**
  The 'Original value' attribute represents the changed value before the actual change.

- **New value**
  The 'New value' attribute represents the value after the change.

- **Placeholder value**
  The 'Placeholder value' functions as a data hold during a move or replace, to temporarily 'store' a value (as String) until it is used elsewhere.

There are many different attribute combinations possible, which results in at least 12,500 different types of AST-Rewrites. As the goal is to give advice based on these rewrites, it is important to research the possible distinct AST-Rewrite types. It is also important to note that we are initially limited to the available automated refactorings in the Eclipse IDE.

3.2. EXAMPLE REFACTORINGS

To get a better idea of what data AST-Rewrites contain in case of a refactoring we will now provide some examples. These examples show the AST-Rewrite output of the Retestoring tool as result of a refactoring. There are, however, a few things worth noting. First, the AST-rewrite output is listed in no particular order, so in these examples we manually put the rewrites in a better readable order. Second, we manually added a description for all AST-Rewrites to make the examples easier to understand. This description is obviously not available in the plug-in output. Third, the type of a node (for example, the changed node is a method declaration), is not listed in the AST-Rewrite output, however, in the plug-in output, this type data is available.

3.2.1. EXTRACT METHOD

In this first example, the highlighted code will be extracted into the '`extM`' method.

Extract method					
Original code			*Refactored code*		

`public MyCalculator(){`
`  public int add(int x, int y){`
`    return x + y;`
`  }`
`}`
			`public MyCalculator(){`
			`  public int add(int x, int y){`
			`    return extM(x, y);`
			`  }`
			`  private int extM (int x, int`
			`      y) {`
			`    return x + y;`
			`  }`
			`}`		

AST-Rewrites					
Changed node	*Property*	*Change*	*Original value*	*New Value*	*Placeholder*
Insert a new empty method					
`public class My-Calculator {...}`	body-Declarations	Inserted		`private int extM (int x,int y){}`	
Extract the original method's body to the new method					
`private int extM (int x,int y){}`	body	Inserted (place-holder)		`return;`	return x + y;
Replace the original method's body with a call to the new method					
`public int add(int x, int y){ return x+y;}`	body	Replaced	`return x + y;`	`return extM(x,y);`	

Table 3.1: The AST-Rewrites of an 'Extract Method' refactoring

The first rewrite is a changed type declaration node, in this case a class called '`MyCalculator`'. Its body declaration has changed; the method declaration '`private int extM (int x,int y){}`' has been inserted. The second rewrite concerns a method declaration node. Here, the body of the '`extM`' method has changed; the line '`return;`' has been inserted with '`return x + y`' as placeholder. The last rewrite is also a changed method declaration node. Inside this method's body property, something is replaced. The line '`return x + y;`' is replaced with '`return extM(x,y)`'.

It is important to understand the difference between an 'Inserted' change, and an 'Inserted with placeholder' change. A regular inserted change is just a node insertion into the syntax tree. The inserted with placeholder change however, hints to moved code. Take the rewrites in this extract method refactoring example: the inserted with placeholder change is a moved line of code, namely, '`return x + y;`' is extracted from the body of the '`add`' method to the body of the new '`extM`' method.

Table 3.2 elucidates the AST changes as a result of the ’Extract Method’ refactoring in Table 3.1. A rectangle represents an AST node, an arrow points into the direction of the node’s children. Gray colored rectangles indicate the specific changes due to the refactoring. The rewrite concerning the insertion of the new method is portrayed by the new ’MethodDeclaration’ node (and its corresponding sub nodes). The second AST-Rewrite extracts a method’s body into this new method. This is portrayed by the new method’s ’Block’ node, which contains the extracted statements as children (in this case a ’ReturnStatement’). Last, the original method’s body is replaced with a call, which is shown by the changed ’Block’ statements. The ’add’ methods block statements is still a ’ReturnStatement’, but contains a method invocation to the new ’extM’ method.

Keep in mind that not all AST-Node attributes are shown in the example diagrams in 3.2 due to space limitations.

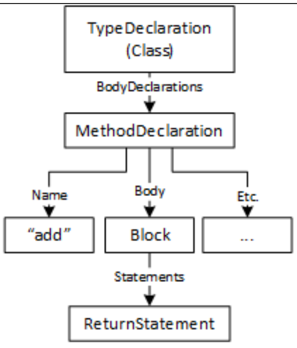
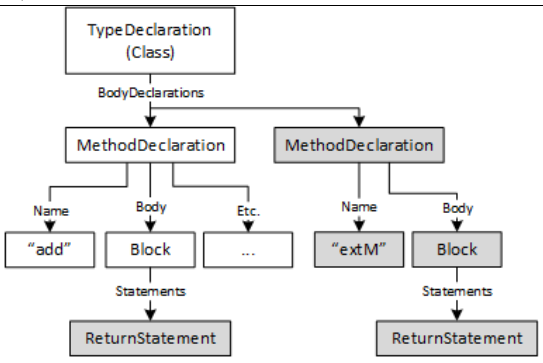
The AST changes of an Extract Method	
Before	*After*
 TypeDeclaration (Class) BodyDeclarations MethodDeclaration Name Body Etc. “add” Block ... Statements ReturnStatement	 TypeDeclaration (Class) BodyDeclarations MethodDeclaration MethodDeclaration Name Body Etc. Name Body “add” Block ... “extM” Block Statements Statements ReturnStatement ReturnStatement

Table 3.2: The AST before and after the ’Extract Method’ example

3.2.2. INLINE METHOD

The second example is the ’Inline Method’ refactoring, which is the opposite of the ’Extract Method’ refactoring. The highlighted line is the where the method call is replaced with its body.

The first rewrite is replacing ’`first`’ in the ’`multiply`’ method’s body with ’`MISSING`’ and a placeholder value called ’`number`’. This seems a little odd, however, the replacement ’`number`’ is the variable in the ’`square`’ method’s call to the multiply method. This rewrite is performed to find out that ’`first * second`’ is equal to ’`number * number`’. The second rewrite shows the characteristics of the inline operation; replacing the ’`multiply`’ method call in the ’`square`’ method’s body with the ’`multiply`’ method’s body. The last rewrite is the removal of the inline method in the class’ body declaration. As you may notice, the ’replaced with placeholder’ change is similar to the inserted variant. However, instead of only moving code, the original code is also replaced with moved code.

Inline method					
Original code			*Refactored code*		

```
public class MyCalculator {
  public int square (int number){
    return multiply(number,number);
  }
  public int multiply ( int
      first , int second){
    return first * second;
  }
}
```
			```
			public class MyCalculator {
			  public int square (int number){
			    return number * number;
			  }
			}
			```

AST-Rewrites					
Changed node	*Property*	*Change*	*Original value*	*New Value*	*Placeholder*
Link the variable first to number					
`public int multiply(int first, int second){ return first * second; }`	body	Replaced (placeholder)	`first`	`MISSING`	number
Inline the original method's body on the location of the method call					
`public int square(int number){ return multiply(number, number); }`	body	Replaced (placeholder)	`return multiply(int first, int second);`	`return;`	return number * number;
Remove the original method					
`public class MyCalculator{...}`	body-Declarations	Removed	`public int multiply(int first, int second) {return first * second;}`		

Table 3.3: The AST-Rewrites of an 'Inline Method' refactoring

3.2.3. MOVE METHOD

In this third example, the highlighted method is moved to the 'MyNewCalc' class. The first rewrite is a type declaration; a class which body declaration changed due to a new method declaration being inserted. The second rewrite is also a type declaration; a class which body declaration changed due to a removed method declaration.

Move method					
Original code			*Refactored code*		

`public MyCalculator(){`
`  public static int add(int x, int y) {`
`    return x + y;`
`  }`
`}`
			`public MyCalculator(){`
			`}`
			
			`public class MyNewCalc {`
			`  public static int add(int x,`
			`      int y){`
			`    return x + y;`
			`  }`
			`}`		

AST-Rewrites					
Changed node	*Property*	*Change*	*Original value*	*New Value*	*Placeholder*
Move the method to the new location					
`public class MyNewCalc{}`	body-Declarations	Inserted (placeholder)		`void MISSING();`	public static int add(int x, int y){ return x+y;}
Remove the method at the original location					
`public class MyCalculator{...}`	body-Declarations	Removed	`public static int add(int x, int y){ return x+y;}`		

Table 3.4: The AST-Rewrites of a 'Move Method' refactoring

3.2.4. MOVE METHOD ALTERNATIVE

These previous examples only show a few rewrites because of the simplicity of the code. A more complex program would result in a slight variation with additional AST-Rewrites. The following example will illustrate this assumption.

Move method	
Original code	*Refactored code*

```
public class MyCalculator {
  public static int multiply(int
     x, int y)  {
    return x * y;
  }

  public int square(int x) {
    return multiply(x, x);
  }

  public int cube(int x) {
    return multiply(multiply(x,
       x), x);
  }
}

public class myClass {
  public void calculations(){
    //Static access
    int value =
       MyCalculator.multiply(1,
       2);
    //Regular access (for test
       only)
    MyCalculator calculator =
       new MyCalculator();
    value =
       calculator.multiply(1, 2);
  }
}
```

```
public class MyNewCalc {
  public static int multiply(int
     x, int y) {
    return x * y;
  }
}

public class MyCalculator {
  public int square(int x) {
    return MyNewCalc.multiply(x,
       x);
  }

  public int cube(int x) {
    return MyNewCalc.multiply(
     MyNewCalc.multiply(x, x),
       x);
  }
}

public class myClass {
  public void calculations(){
    //Static access
    int value =
       MyNewCalc.multiply(1, 2);
    //Regular access now changed
       to static access as well
    MyCalculator calculator =
       new MyCalculator();
    value =
       MyNewCalc.multiply(1, 2);
  }
}
```

AST-Rewrites

Changed node	*Property*	*Change*	*Original value*	*New Value*	*Placeholder*
Move the method to the new location					
`public class MyNewCalc{}`	body-Declaration	Inserted (Placeholder)		`void MISSING();`	public static int multiply(int x, int y) { return x * y; }
Replace the regular access method call to meet the new location					
`public void calculations() {...}`	body	Replaced	`calculator`	`MyNewCalc`	

Replace the static access method call to meet the new location					
`public void calculations() {...}`	body	Replaced	`MyCalculator`	`MyNewCalc`	
Insert the static access to the method calls to meet the new location					
`public int cube(int x) {...}`	body	Inserted		`MyNewCalc`	
`public int square(int x) {...}`	body	Inserted		`MyNewCalc`	
`public int cube(int x){...}`	body	Inserted		`MyNewCalc`	
Remove the method at the original location					
`public class MyCalculator{...}`	body-Declaration	Removed	public static int multiply(int x, int y) { return x * y; }		

Table 3.5: The AST-Rewrites of a 'Move Method' refactoring (alternative)

The example in Table 3.5 shows that a larger code example results in similar overall rewrites. Just as the example in Table 3.4, there are two rewrites concerning the moved code. Additionally we added calls to the moved method, which results in additional AST-Rewrites. In this example the method calls, '`multiply()`', '`MyCalculator.multiply()`' and '`calculator.multiply()`' all change to '`MyNewCalc.multiply()`'. We could use additional references to the changed method in all examples, but as this example already shows, this would result in multiple of the same replace or insert rewrites to repair the reference to the changed method.

3.3. MORE EXAMPLE REFACTORINGS

Now that we have seen several method refactorings, we now provide additional examples of different rewrites. We are however, limited to those refactorings available in the Eclipse IDE. Large refactorings as 'separation of the domain logic from the presentation logic' are not supported. Mainly, because these larger refactorings are difficult to automate due to the required code knowledge and understanding.

3.3.1. INTRODUCE FACTORY

The 'Introduce Factory' refactoring (Table 3.6) makes the highlighted constructor private and creates a static factory method. This refactoring is therefore also known as the 'Replace Constructor with Factory Method' refactoring [Fowler et al., 1999]. This refactoring leads to several rewrites, most of which are similar to those in previous examples. However, this refactoring also includes a not previously seen refactoring; a method declaration type with a changed 'modifiers' property. In this case, these AST-Rewrites represent the change in the constructor's access modifier from '`public`' to '`private`'.

Introduce factory

Original code

```
public class Car {
  public Car() {

  }
  public void newCar() {
    Car car = new Car();
  }
}
```

Refactored code

```
public class Car {
  public static Car createCar() {
    return new Car();
  }
  private Car(){
  }
  public void newCar() {
    Car car = createCar();
  }
}
```

AST-Rewrites

Changed node	*Property*	*Change*	*Original value*	*New Value*	*Placeholder*
Insert the factory method					
`public class Car{}`	body-Declarations	Inserted		`public static Car create-Car() { return new Car(); }`	
Replace the constructor call with a factory method call					
`public void New-Car{...}`	body	Replaced	`new Car()`	`createCar()`	
Change the constructor's access modifier from public to private					
`public Car{}`	modifiers	Inserted		`private`	
`public Car{}`	modifiers	Removed	`public`		

Table 3.6: The AST-Rewrites of an 'Introduce Factory' refactoring

3.3.2. INTRODUCE PARAMETER

The 'Introduce Parameter' refactoring is also known as the 'Add Parameter' refactoring, and is essentially nothing more than creating a parameter for a local value. This introduces a 'parameters' property change, which adds the local variable as parameter to its method. In this case the variable `name` in the '`setName`' method is changed to a method parameter called '`personName`'. See Table 3.7.

Introduce parameter

Original code

```
public class Person {
  String name;

  public void createPerson(){
    setName();
  }

  public void setName(){
    name = "John";
  }
}
```

Refactored code

```
public class Person {
  String name;

  public void createPerson(){
    setName(name);
  }

  public void setName(String
      personName){
    personName = "John";
  }
}
```

AST-Rewrites

Changed node	*Property*	*Change*	*Original value*	*New Value*	*Placeholder*
Introduce parameter to method					
`public void setName{...}`	parameters	Inserted		`String PersonName`	
Change parameter value in method call					
`public void createPerson{...}`	body	Inserted (placeholder)		`MISSING`	name
Replace variable with parameter					
`public void setName{...}`	body	Replaced	`name`	`PersonName`	

Table 3.7: The AST-Rewrites of an 'Introduce Parameter' refactoring

3.3.3. EXTRACT SUPERCLASS

This example shows an AST-Rewrite for the creation of a new class. The 'Extract Superclass' refactoring extracts methods or variables from a class and puts them in a new superclass. In this case (Table 3.8) we create a '`Fruit`' supertype for the '`Apple`' class. Its '`eat`' method and '`color`' variable will be extracted to the superclass. Most notably is the AST-Rewrite regarding the 'superclassType' property, which describes the addition of '`extends Fruit`' to the '`Apple`' class.

Extract superclass

Original code

```
public class Apple{
  String color;
  public void eat() {
    //...
  }
}
```

Refactored code

```
public class Apple extends Fruit{
}

public class Fruit {
  String color;
  public Fruit() {
    super();
  }
  public void eat() {
    //...
  }
}
```

AST-Rewrites

Changed node	*Property*	*Change*	*Original value*	*New Value*	*Placeholder*
Create new superclass					
`public class Fruit{...}`	bodyDeclarations	Inserted		`public Fruit{ return; }`	
Let the original class inherit from the superclass					
`public class Apple{...}`	superclassType	Inserted		`Fruit`	
Insert extracted code elements into the superclass					
`public class Fruit{...}`	bodyDeclarations	Inserted		`MISSING color;`	
`public class Fruit{...}`	bodyDeclarations	Inserted		`public MISSING eat(){}`	
Remove extracted code elements from the original class					
`public class Apple extends Fruit{...}`	bodyDeclarations	Removed	`String color;`		
`public class Apple extends Fruit{...}`	bodyDeclarations	Removed	`public void eat(){}`		

Table 3.8: The AST-Rewrites of an 'Extract Superclass' refactoring

3.3.4. EXTRACT CLASS

The last example is the ’Extract Class’ refactoring in Table 3.9. This refactoring divides responsibilities between classes, in this case the variable ’`wheelSize`’ is extracted to a separate class ’`Wheel`’. This rewrite introduces two properties we have not mentioned yet. Firstly, the ’types’ property indicates a new node type, in this case the creation of the new ’`Wheel`’ class. The ’fragments’ property concerns changes to fields, in this case the removal of the ’`wheelSize`’ field. The remainder of the rewrites are similar to those in previous examples.

Extract class

Original code

```
public class Bicycle {
  int wheelSize;
  public void ride(){
  }
}
```

Refactored code

```
public class Bicycle {
  Wheel data = new Wheel();
  public void ride(){
  }
}

public class Wheel {
  public int wheelSize;

  public Wheel() {
  }
}
```

AST-Rewrites

Changed node	*Property*	*Change*	*Original value*	*New Value*	*Placeholder*
Extract code element into new class					
`public class Wheel{}`	types	Replaced	`class Wheel{}`	`class Wheel{ public int wheelSize; ... }`	
Remove code element from original class					
`int wheelSize;`	fragments	Removed	`wheelSize`		
`public class Bicycle{...}`	body-Declarations	Removed	`int wheelSize;`		
Insert the instantiation of the new class into the original class					
`public class Bicycle{...}`	body-Declarations	Inserted		`Wheel data = new Wheel();`	

Table 3.9: The AST-Rewrites of an ’Extract Class’ refactoring

3.4. CONCLUSION

So far, we have seen many different AST-Rewrites and the including attributes. Examples show that the property attribute together with the changed node exactly show which piece of the code changed. The property and changed nodes show the location of the change. The change attribute, containing one of five different operations, describes how that piece of code changed. The value attributes are different for every change, because these contain the actual code of the change, which is mostly situation specific. In Table 3.10, we visualized these attribute connections. Certain attribute combinations (changed node and property) are already recurring in the given examples. For instance, every method refactoring has a rewrite with a changed 'class type' node and a changed 'body-Declaration' property. These common combinations are valuable for the next step; the impact of a refactoring on a test.

AST-Rewrite attributes					
Parent node	*Property*	*Change*	*Original value*	*New value*	*Placeholder*
Context of the rewrite		Changes in the code			Moved code

Table 3.10: The AST-Rewrites attributes and their informational value

Based on the examined output of the plug-in we can now answer our first research question.

RQ1: Which different output combinations can the plug-in return?

The plug-in output as a result of a refactoring consists of a combination of at least one AST-Rewrite. An AST-Rewrite consists of the following attributes combination: the 'changed node', the 'changed property', the change (operation), the original value, new value and placeholder value. The changed node can be any of the listed ASTNode[1] types. Whereas the changed property can be any of classes listed in the AST/DOM[2] package. The change is always one of five operations (insert, replace, remove, insert with placeholder and replace with placeholder), which means other changes as move, extract or inline have to be constructed with multiple separate rewrites. The attribute values (new and original) are either empty or a value of an ASTNode type. The placeholder is always a 'String'-type, and is either empty or a string representation of an ASTNode type value.

It is unrealistic to list all different AST-Rewrite output combinations. Although the amount of different combinations is limited by the finite amount of AST-Rewrite attribute combinations and automated refactorings in Eclipse, the environment (source code situation) in which a refactoring is performed, may result in countless (slight) variations, even whilst a certain refactoring (in general) results in a similar combination of rewrites.

[1]http://help.eclipse.org/luna/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTNode.html

[2]http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/package-summary.html

4
Results on Tests

In this chapter we take a look at the process on how to determine the result of a refactoring on tests, which is an important part in determining the advice. In order to make the advice process better understandable, we divide process into four sub processes. This chapter describes the first two processes; code change analysis and test impact analysis. For each sub process we create a set of rules, which followed, should provide the next process with sufficient data.

4.1. Impact on Correctness of Tests

When the main code changes, the functionality of the tests may or may not remain correct, dependent on the change. In this case, the term 'correctness' of tests, means that main code and the corresponding tests should match. For example, this means that all public methods have a corresponding test, but also that there are no tests for private or even nonexistent methods.

To determine the result of a refactoring on a test, we should determine the result of the refactoring on the code first. The result on the tests is almost identical to the result of the initial refactoring on the code itself [van Deursen and Moonen, 2002]. If you extract part of a method's body into a separate method, you could extract that particular part of the corresponding test method into a separate test as well.

In our case however, we assume that all tests are black-box tests. This means that the test method only contains something in the lines of 1. initialization of the input, 2. a call to the corresponding method and 3. assertion of the output. Due to this test specific structure, performing a refactoring on a black-box test is no valid option.

The phrase '**results on tests**' is used to describe the impact of changes in the source code on the corresponding test code. This result creation can be divided into several steps:

- **Code Change Analysis**
  Describes the change in the source code.
  Example: *Removed a public method A from class X.*

- **Test Impact Analysis**
  Describes the effect of the change on its corresponding test.
  Example: *Unused test method ATest in test class Xtest.*

- **Advice**
  Describes the action(s) a user has to perform in order to correct the tests.
  Example: *Remove test method ATest from test class Xtest.*

- **Presentation**
  Describes how the system presents the advice to the user.
  For example: different languages or in keywords instead of full sentences.

4.2. Limitations and code assumptions

Each individual refactoring has an expected 'standard' result which holds in most cases. In case of an 'Extract Method', we can assume that we end up with a new method that requires a test. We will refer to this assumed standard (and generally applicable) result as the **global result**.

However, there are different constructs in the code and tests which can cause problems when determining the result of a refactoring. There are cases where the global result is not applicable. For example, if an extracted method has a private access type, the result is different in contrast to a public method; private methods do not require a corresponding test. We will list some examples of cautions points for rewrites regarding a class with a changed method declaration:

- **Multiple test methods**
  When a regular method has multiple test methods, the result of a refactoring should consider each test method. A naming convention for tests and test-variants could make this problem easier. The standardized approach, is to use a 'parametrized test' instead if applicable [Tahchiev et al., 2010]. However, tests with multiple assertions or test classes with multiple test methods to test a single method are common practice and are not necessarily wrong. Using this approach, one should be wary of test smells. In this particular case, potential smells are "Lazy test" or "Assertion Roulette" [van Deursen et al., 2001]. Another issue is the opposite of the above; one test method for more than one test. This is a test smell called an "Eager Test" [van Deursen et al., 2001]. The developer should extract the test code for another method in a separate method, as we do not support eager test constructions.

- **Visibility of the method**
  A private method does not have a direct corresponding test in black-box testing. Therefore, when a private method changes, this does not necessarily impact the tests. An additional check for the visibility of a changed method would solve this case. In case of a regular untested private method, a change is not a necessity.

- **No corresponding test method**
  When the change includes a visible method, and no corresponding test method is found, the result would still be an untested method, even if the method is not 'new'. Errors, for example, putting the test method in the wrong test class are the developer's responsibility. In future work, we could add a notification to address the developer of this issue.

- **Method overloading**
  When an overloaded method is changed, the result only applies to that particular

corresponding test. An additional check regarding the method's parameters should result in finding the correct corresponding test method.

- **Inheritance**
  In this case, the same problem as the overloading applies; finding the corresponding test method. For example, when a sub class' method has been removed, this does not automatically imply that all tests for that method should be removed. The global result solely applies when the subclass has a distinct test method. In most cases, no change is necessary if you test through an interface. An exception to this rule occurs when the change concerns a subclass' own additional methods.

You may notice that method declarations changes already provide several special cases, which need additional checks in order to give a more precise and correct result on tests. As we have at least twelve thousand different AST-Rewrite combinations, listing each and every special case would be impossible. However, we can summarize different issue categories, for example, most issues mentioned above concern finding the (correct) test class and test method. Other concerns are exceptional cases where none or more than one test is present, or a case where the result is that there is no change necessary.

Exceptional cases are easier detectable, or can be ignored, if we use the following assumptions:

1. Each public class, containing one or more public methods, has a corresponding test class.
2. Each public method has exactly one corresponding unit test.
3. Each test method solely concerns an existing public method.
4. Each test method is located in the correct corresponding test class.
5. Each test method has exactly one assertion.
6. Each test method solely focuses on the functionality of its corresponding method.
7. Each test class and test method uses the following naming convention: name of class or method followed by suffix 'Test'.
8. Method overloading is not supported.

Using these assumptions, we can now focus on a basic scenario, instead of exceptional cases. This is a necessity because many of these exceptions to the global result require additional knowledge of the test structure. The Retestoring plug-in is currently not capable of analyzing the structure of the test itself, therefore these exceptions are out of scope for our current research. This does however not imply that these exceptional cases are less meaningful, as solving these special cases is important future work.

Additionally, we want to point out that, code structures which are considered 'bad practice' by the 'Java Coding Convention', will obviously also not be supported. Examples are: optional parameter structures and public inner-class types, but also public fields (which is situation dependent). Future work therefore includes, detecting these 'bad practices' before refactoring.

4.3. CODE CHANGE ANALYSIS

4.3.1. CHANGE DATA COLLECTION

In order to determine which changes have been applied to the source code, we use the change data and examine it for the required information. There are two large data inputs to work with: the refactoring data and the AST-Rewrites.

AST-REWRITES

The AST-Rewrite data can be used in order to determine resulting information and thus the exact change. As a human being, reading the rewrite output of previous examples and trying to comprehend the actual change comes close to solving a puzzle. In our own attempt to read and understand AST-Rewrites, we link rewrites together, sometimes without realizing it. **A link is a matching attribute value in the AST-Rewrite data.** The main reason for this approach is because a single AST-Rewrite on itself is less valuable. We will explain this statement with an example.

Table 4.1 shows the raw AST-Rewrite output as a result of an 'Extract Method' refactoring. The rewrites are presented in the same way as we obtain them from the Retestoring plug-in; listed in no particular order, and without detailed 'before' and 'after' code. Note that the node type is available but is not shown in the table due to layout constraints.

Extract Method AST-Rewrites

#	*Changed node*	*Property*	*Change*	*Original value*	*New Value*	*Placeholder*
1	`public int add(int x, int y){ return x+y;}`	body	Replaced	`return x + y;`	`return extM(x,y);`	
2	`public int extM (int x,int y){}`	body	Inserted (placeholder)		`return;`	return x + y;
3	`public class MyCalculator{...}`	bodyDeclarations	Inserted		`public int extM (int x,int y){}`	

Table 4.1: The AST-Rewrites of an 'Extract Method' refactoring

If we look at the rewrites individually, we should determine the impact of that particular rewrite on the code. Rewrite #1 shows that a methods body has changed. Rewrite #2 also concerns a method which body has changed. Rewrite #3 is an insertion of a method into a class. So the global change is: two changed method bodies and one new method.

If we take a closer look, it seems that rewrite #2 and #3 describe a change on the same method. So the previous determined global result is incorrect and imprecise. In this case, it is important to conclude that rewrite #2 fills the new method created in rewrite #3, and therefore the results of both AST-Rewrites can be combined. This shows that using the individual results of rewrites is no viable option, as the relation between rewrites also contains valuable information. If we would consider the changed code values (new value, original value and placeholder value), we could even link rewrite #1 to the other two rewrites in order to describe the change with even more detail. The replaced code of rewrite #1 is in fact the new method's body.

We can conclude that without links between rewrites, we end up with a global and imprecise advise. We need the links between rewrites to find out if, for example, code has moved. The links provide the details to separate a move rewrite from a set of unrelated insert and remove rewrites. Additionally, in case of a move, we can use the origin of the moved (new) value for a even more detailed result conclusion.

However, it is difficult to map rewrite links. Each refactoring results in different rewrites with different links. As you may notice, links occur between different attributes and different code elements. In the 'Extract Method' example of Table 4.1, there are three links.

1. The first link is the changed node of rewrite #2 and the new value of rewrite #3.
   This link shows a change on the same method, but it does help to reveal which refactoring has been performed.

2. The second link is between the new value of rewrite #1 and the new value of rewrite #3.
   This link shows the replacement of some code with a method call, and the insertion of that particular method. This is a characteristic of the 'Extract Method' refactoring. The challenge is to compare the different code types: one value is a method call as a return type, the other a method declaration.

3. The last link is the placeholder data of rewrite #2 and the original value of rewrite #1.
   This link shows the extraction of the method's body into a separate method. This is also a characteristic of the 'Extract Method' refactoring.

Based on the second (and also the third link), we can conclude that an 'Extract Method' refactoring has been performed. Based on our knowledge of this refactoring we can conclude that:

- New method '`extM`' inserted into class '`MyCalculator`'.
- Code of method '`add`' extracted to '`extM`' method.

Unfortunately, the usage of links between rewrites to determine the result on tests comes with some challenges. Based on previous refactoring examples, we list possible challenges below.

1. The first challenge is to create an approach for finding refactorings effectively. We could create an approach with rules that specify each refactorings, for example: if we have an inserted method rewrite, we can search for a rewrite with code replaced by a method call to find out if an 'Extract Method' refactoring took place. The problem with this approach is, due to the amount of possible refactorings and resulting rewrites, we cannot specify cases for each and every rewrite combination.

2. The second challenge is its dependency on values to base links on. Values are difficult to compare. A regular line of code for example, can be changed into a return statement or a variable declaration, and is therefore never a ’100%’ match. In addition, basing links on values as ‘return’, barely have value due to common use (as it is a programming keyword). The question is where to draw the line what is considered as still useful values, as there are situations where a link can consist of a single variable name.

3. Additionally, some values have to be interpreted first. This is necessary to conclude that for example ‘return 6 * 6’ is similar to ‘return number * number’, which is of common occurrence in ’Inline Method’ refactorings. Also, a comparison with a placeholder value may require additional interpreting, as a placeholder is a ’String’.

4. The fourth challenge is the fact that not all links are (equally) valuable. Some links simply have no characteristics of refactorings. Other links are based on ’accidental’ matches, which could be the case due to common values as ‘0’ or keywords as ‘return’. In some occasions, invaluable links occur on placeholder rewrite by-products, which are the MISSING value rewrites to preserve references.

5. The last challenge is that we still need additional checks for special situations. For example, to check if a method changed visibility (access modifier), or if an ’Inline Method’ refactoring removed the original method. Having additional checks for each and every special case is in the long term unmaintainable.

REFACTORING DATA

The refactoring data consists of the refactoring parameters when performing a refactoring via the refactoring ’menu’ in Eclipse. One part is constructed with the input a user inputs in a refactoring menu (Figure 4.1) and the location of the caret in the source code. The second part is determined by the changes of a particular refactoring. The refactoring data can include the following information: the location of the refactoring (code elements), the project’s name, a description of the refactoring and/or change, and the changed values. It is important to note that all values are ’String’ representations, and not the actual objects.

Refactoring data can be collected by using the ’Refactoring Execution Listener’[1] and ’catch’ the refactoring when it is performed. The corresponding ’Refactoring Descriptor’[2] can then be queried for the refactoring data. As example, we perform the ’Inline Method’ refactoring of Table 3.3 again. Table 4.2 shows the refactoring data after the inline operation.

The refactoring data starts with a description of the change, in this case it describes the inline. The following lines in the data indicate the location of the change. The last data lines describe additional changes of that particular inline refactoring, which, in this case, is the removal of the inline method and the resolving of all references to that method.

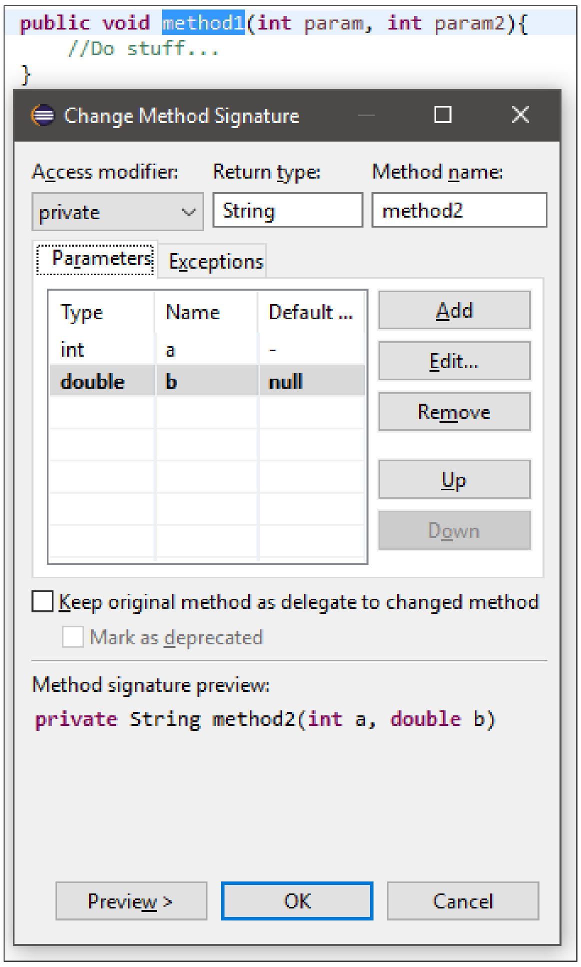
Figure 4.1 shows an example of the ’Change Method Signature’ refactoring in eclipse. This is essentially a combination of some of the ’Making Method Calls Simpler’ type refactorings [Fowler et al., 1999] presented in a single menu. Dependent on the refactoring

[1] `http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ltk/core/refactoring/history/IRefactoringExecutionListener.html`

[2] `http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ltk/core/refactoring/RefactoringDescriptor.html`

Inline Method refactoring data
`Inline method 'MyCalculator.multiply()' in 'MyCalculator'`
`Original project: 'MyProject'`
`Original element: 'MyCalculator.multiply()'`
`Remove method declaration`
`Replace all references to method with statements`

Table 4.2: The data of an 'Inline Method' refactoring



```
public void method1(int param, int param2){
    //Do stuff...
}
```

Change Method Signature

Access modifier: private

Return type: String

Method name: method2

Type	Name	Default ...
int	a	-
double	b	null

☐ Keep original method as delegate to changed method

☐ Mark as deprecated

Method signature preview:

```
private String method2(int a, double b)
```

Figure 4.1: Refactoring Input Parameters

Change Method Signature Refactoring data
`Change method 'public int myClass.method1(int param, int param2)' to 'private String method2(int a, double b)'`
`Original project: 'MyProject'`
`Original element: 'myClass.method1(...)'`
`New name: 'method2'`
`New visibility: 'private'`
`New return type: 'String'`
`Removed parameters: int param2`
`Changed parameters: int a`
`Added parameters: double b`

Table 4.3: Example refactoring data of a 'Change Method Signature' refactoring

Move Method Refactoring data
`Move method 'class1.add(...)' to 'class1.c2'`
`Original project: 'MyProject'`
`Original element: 'class1.add(...)'`
`Moved method: N/A`
`Target element: 'class1.c2'`
`Method name: 'add'`

Table 4.4: Example refactoring data of a 'Move Method' refactoring

and the Eclipse implementation of that particular refactoring, these refactoring parameter screens may contain valuable information. However, not all input values indicate an actual change, for example, we can choose to change the method's parameters, but keep its name intact. This also means that, if the method's name has not been changed, the refactoring data does not contain that particular section. This example is valuable because it shows that refactorings have different variants, and one refactoring menu can trigger multiple different refactorings. Performing the 'Change Method Signature' refactoring in Figure 4.1 results in the following example data of Table 4.3.

Combining AST-Rewrites and Refactoring Data

By examining the refactoring data, it becomes evident that the data itself is not always completely sufficient to determine the exact change on the code. For instance, the refactoring data in Table 4.4 describing a 'Move Method' refactoring does not provide sufficient data. The reason is the unclear target element: it refers to a class' declaration '`c2`' in '`class1`', instead of the actual target location '`class2`'. The corresponding AST-Rewrites of this refactoring in Table 4.5 show the correct target location.

Thus, refactoring data on itself is not always sufficient. However, it provides us a head start for determining the result. It shows us which refactoring took place and where in the source code the changes happened. AST-Rewrites theoretically contain sufficient information to determine the exact change. However, there are some shortcomings to overcome in regard to using matching values as linked data. Combining AST-Rewrite data with the refactoring data, may resolve some of the current problems. This approach is not free of challenges either, as the refactoring data is not ideal to work with; since it is a mere 'String'

Move method AST-Rewrites					
Changed node	*Property*	*Change*	*Original value*	*New Value*	*Placeholder*
`public class class2{}`	body-Declarations	Inserted (place-holder)		`void MISSING();`	public int add(int x, int y){ return x+y;}
`public class class1{...}`	body-Declarations	Removed	`public int add(int x, int y){ return x+y;}`		

Table 4.5: The AST-Rewrites of a 'Move Method' refactoring

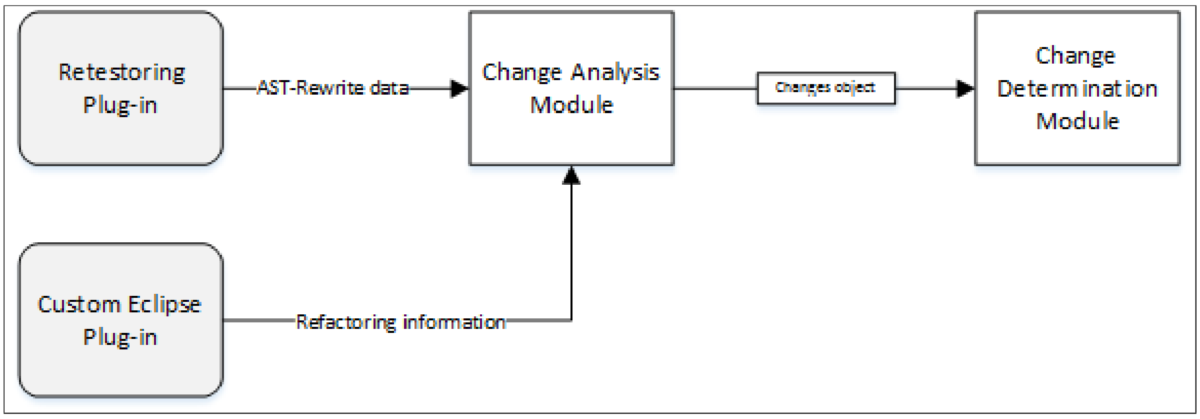


Figure 4.2: Refactoring Data Flow

representation. Nevertheless, the refactoring characteristics can still be used to narrow the search for particular additional data in AST-Rewrites.

4.3.2. DETERMINE EXACT CHANGE

In the following subsections we will provide the part of the approach for determining impact of a change on the correctness of the tests. This part consists of steps for determining the exact change on the source code. Figure 4.2 shows the refactoring data flow. The input comes from two sources: the refactoring data (gathered by a custom plug-in for the Eclipse IDE) and the AST-Rewrite Retestoring plug-in data. The 'Change Analysis Module' has several tasks with the main purpose to extract the desired change attributes from the input data and send these to the 'Change Determination Module'. The 'Changes object' describes the data structure we need in order to determine the change.

EXAMINE REFACTORING DATA

The refactoring data provides quick access to the information we need in order to describe the change. For example: the description of the refactoring tells us which refactoring has been performed and on what code element. However, this information is stored in the '`RefactoringDescriptor`' object. Therefore, we need some sort of module to extract the necessary data from this object and convert it to our own standard. Additionally, we need a function to determine if the refactoring data holds sufficient data in order to determine the exact change. There are three states for the refactoring data:

1. Refactoring data is complete, which means that the required values in our changes object can be filled with sufficient data.
2. Refactoring data is incomplete, which means that we rely on AST-Rewrites to fill in the missing data.
3. Refactoring data is empty, which means that we solely use the AST-Rewrites to fill our changes object.

EXAMINE AST-REWRITE DATA

The AST-Rewrite data can also be examined for (additional) information on the refactoring. The refactoring data should (mostly) provide us with information what to search for. We start by determining the highest priority rewrite.

In order to determine with which rewrite to start the process, some sort of prioritization is necessary. If we know the initial refactoring, we can search for that particular rewrites. We prioritize these rewrites first, as these rewrites contain the most important change. If we do not know the refactoring, or we have (possible valuable) leftover rewrites, we have to use a different prioritization. If we would encounter a problem like this ourselves, we would try to finds the most important change first. In this case, this would be the rewrite which results in the largest structural change. Something along the lines of: changes on the class are larger than changes on the method. The thought behind this approach is to find the change with the most impact on the structure first, and then find all related rewrites to that change.

With this information we can now create a prioritized list of node types and its properties:

1. **Known refactoring rewrite**
2. **Compilation Unit** (file)
   (a) **Package**: file (class) changed package.
3. **Type declaration** (classes)
   (a) **Name**: name of class changed.
   (b) **Modifiers**: visibility of class changed.
   (c) **Types**: class itself changed as a whole.
   (d) **Package**: class changed package.
   (e) **Super (interface/class)**: class changed to sub class, or former sub class is now a regular class.
   (f) **Body declaration**: class' code body changed. Possibly contains valuable data.
4. **Method declaration**
   (a) **Name**: name of method changed.

(b) **Modifiers**: visibility of method changed.

(c) **Return type**: return type of method changed.

(d) **Parameters**: parameters added or removed from method.

(e) **Body**: method's code body changed. Possibly contains valuable data.

5. **Other**

Links only occur between the data attributes of the rewrite (changed node, original value, new value and the placeholder). Based on the refactoring data information we may know which element has been changed. Therefore we prioritize the search for the operation characteristics between the rewrite on that particular element and another rewrite. For all remaining rewrites, the easiest way is to start with the changed node of the selected rewrite, and check the changed node of other rewrites first. The order of checking other rewrites does not matter, so searching from top to bottom will do. The next step is to keep the changed node value as value to check and, this time, look at the original values of other rewrites. Next, check the new values and thereafter the placeholders of the other rewrites. After all other rewrite's attributes have been checked, change the value to check to the original value of the selected rewrite, and continue the process.

Thus, links are searched in order of similar AST-Rewrite attributes. Below the priority of linking attributes and the reason:

1. **Known refactoring attribute**: known changed element.
2. **Node**: possibly part of the same or a larger structural change.
3. **Original value**: possibly shows origin of changed code.
4. **New value**: possibly shows destination of changed code.
5. **Placeholder**: possibly shows destination of changed code.

There is, however, an additional check necessary in order to find all links. In some occasions, data values are not of the same type, but do link. For example, a method declaration can be disguised as a method call, a return type or a variable declaration. This is even more difficult in case of moved lines of code. The body of a method declaration may contain lines of code that reappear in values of other rewrites. The problem is that this would require additional semantic analysis in order to determine a match. This problem occurs in an inline method situation (see table 3.3), where moved is no longer textually identical due to the use of a placeholder. Semantic analysis could however clarify that '`x * x`' and '`x * y`' are similar if both 'x' and 'y' are integer values.

This leads to the following additional checks for each step of the search approach:

1. **Same type**: data of the same type. For example: a method declaration.
2. **Other types**: data regarding the same code element, but of the different types. For example: a method declaration and a method call.
3. **Partial**: data that only partially matches. For example: code in a method's body and a placeholder value.

Most refactorings are primarily based on a certain 'operation'. The standard operations are the 'change' attributes of the AST-Rewrites: insert, replace and remove. The additional operations are those that are a combination of the standard operations: move, extract and inline. Operations generally have the same global characteristics. Therefore, we do not have to make specific rules for each different refactoring. For example, a move operation is a combination of a remove rewrite and an inserted with placeholder rewrite. Additionally there are other attributes of the AST-Rewrite combination that have to match as well in order to confirm a certain additional operation.

Based on the operation rules we can filter for valuable linked rewrites. Below, a list of the different operations and their corresponding 'rules':

Basic operations:

- Insert: new code
- Remove: code deleted
- Replace: code replaced with other code

Additional operations:

- Move: code removed at one place, and inserted at another place.
  - Only if inserted (placeholder) rewrite placeholder value ≈ removed rewrite original value, or
  - Only if inserted (placeholder) rewrite placeholder value ≈ replaced (placeholder) rewrite original value.
- Extract: code replaced with a call to the replaced code inserted elsewhere.
  - Only if inserted (placeholder) rewrite placeholder value ≈ replaced rewrite original value.
- Inline: call replaced with code on location of call, call location code removed afterward.
  - Only if replaced (placeholder) rewrite placeholder value ≈ removed rewrite original value, and
  - Only if replaced (placeholder) original value ≈ removed rewrite original value.

4.3.3. CHANGE ANALYSIS MODULE

The change analysis module is the part where the 'raw' data is filtered and converted to the specified 'Changes' object. Many tasks of this module lie beyond the goals of this research. These tasks will therefore be listed as future work. Below, a list of challenges, to give a broad idea on what issues need to be solved:

- Converting the refactoring data to the specified standard.
  - Analyzing which data is required from AST-Rewrites.
- Converting (linked) AST-Rewrite data to the specified standard.

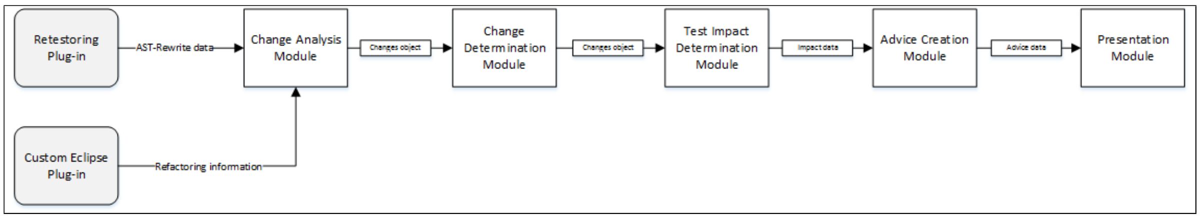


Figure 4.3: Change Data Flow

- Linking values that are not a perfect match, and/or are of different types.
- Filter rewrites that are unrelated to the refactoring or are a by-product of a refactoring.

We will however, provide the definition of the standard change object in the next section. With the use of this object, we assume the data we receive has been resolved from the previous listed challenges.

4.3.4. CHANGES OBJECT

The changes object describes the information we need in order to determine the change. It functions as a communication standard between parts of the process. The previous section describe the change data gathering process which is the first part of the process. This information is used determine the exact change and later to also determine the result on the tests and create the advice. Figure 4.3 shows the complete change data flow.

In order for the object to be of real value, we set the following requirements:

- The object should be version independent, which means that it should not matter which Eclipse version and Retestoring plug-in version are used.
- The object should be 'realistic' and therefore only rely on data that is actually proven to be obtainable.
- The object should be usable for all Eclipse's own available refactorings in the IDE.

In order to meet the requirements, we introduced the following universal specifications. First and foremost, the refactoring data provides us with the initial refactoring description. AST-Rewrites could provide the same information as well, but require additional processing in order to determine that specific information. Second, we require information on the changed code element. This could be provided partly by the refactoring data and fully by the AST-Rewrite data (after some processing). The changed element data should contain the following attributes: name, location, type (object), new value, change operation, original value. In some cases, changes on one element induce other additional changes, for example, to other code elements containing a reference to the changed element. These additional changed elements are also of value, and require the same attributes as the (regular) changed element.

Table 4.6 describes the contents of the 'Changes' object. It consists of four main objects:

- Refactoring Descriptor
  The refactorings descriptor provides a description of the performed refactoring.

- Changed Element
  The changed element that describes the 'main' (most important) change. It contains several attributes, describing the change.

- Original Changed Elements
  A list of changed element(s), as a result of the 'main' change. All items in this list are of the same type as the 'Changed Element'.

- Additional Changed Elements
  A list of changed elements unrelated to the 'main' change. All items in this list are of the same type as the 'Changed Element'.

The 'Changed Element' is always required to be filled for every refactoring situation. The 'Original Changed Elements' and 'Additional Changed Elements' lists are however not. If the operation on the 'Changed Element' is of a type that consists of multiple AST-Rewrites (move, extract, inline), the 'Original Changed Elements' list contains the fully filled 'Changed element' counterpart. The 'Refactoring Description' is only necessary for presenting the change, not for determining the advice.

4.3.5. CHANGE DETERMINATION

The 'Change Determination' module is the module behind the 'Changes' object. This module's essential task is to list all changes to the source code based on the data input. For subsequent steps in the process, the 'Changed Element' object provides sufficient information. However, it may be useful to also have a better human readable version of the change. Due to this standardized the input format, it becomes easier to define generic a 'form' for creating the change description.

CHANGE DESCRIPTION

The goal is to provide as much information on the change as possible in a human readable manner. This means a description as for instance: 'Inline method Class1.method1(), all statements replaced', provides less usable information than for example: 'Class1.method1() removed, Class1.method2() inline code'.

In order to create the change description, we use forms and try to fill these with data attributes of the 'Changes' object. A rule for describing the change, based on ChangedElement data:

- [RefactoringDescription]:[ChangedElementProperty] property of [ChangedElementType], [ChangedElementName], in [ChangedElementLocation] changed. New value: [ChangedElementNewValue] of type [ChangedElementNewValueType], [ChangedElementOperation] at old value [ChangedElementOriginalValue] of type [ChangedElementOriginalValueType].

[3]http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ltk/core/refactoring/RefactoringDescriptor.html

[4]http://help.eclipse.org/luna/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTNode.html

[5]http://help.eclipse.org/luna/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTNode.html

[6]Custom enum describing the operation (insert, extract, etcetera)

Object contents		
Variable	*Type*	*Description*
RefactoringDescription	RefactoringDescriptor[3]	Describes which refactoring took place.
ChangedElement	`ChangedElement`	Describes the most important changed element.
id	int	Unique numeral identifier for the ChangedElement.
ChangedElementName	String	Name of the new location element.
ChangedElementLocation	ASTNode[4]	Code location of the changed element.
ChangedElementType	int	Type of the new location element.
ChangedElementProperty	StructuralProperty-Descriptor[5]	The changed property of the new location element.
ChangedElement-NewValue	Object	Current value of the new location element.
ChangedElement-NewValueType	int	Value type of the new location element.
ChangedElement-Operation	ChangeEnum[6]	Change on the new location element.
ChangedElement-OriginalValue	Object	Old value of the new location element.
ChangedElement-OriginalValueType	int	Original value type of the new location element.
OriginalChangedElements	`List<Changed-Element>`	List of changed elements as a result of the most important change.
Same attributes as 'ChangedElement'		
Additional-ChangedElements	`List<Changed-Element>`	List of changed elements unrelated to the most important change.
Same attributes as 'ChangedElement'		

Table 4.6: A description of the Changes object contents

For all original and additional changed elements we use:

- [ChangedElementProperty] property of [ChangedElementType], [ChangedElementName], in [ChangedElementLocation] changed. New value: [ChangedElementNewValue] of type [ChangedElementNewValueType], [ChangedElementOperation] at old value [ChangedElementOriginalValue] of type [ChangedElementOriginalValueType].

If a certain attribute is empty, it will be obviously left out of the description.

4.4. Impact on Tests

4.4.1. Filtering changes

In the previous section we defined the change on a code element as an object called 'ChangedElement'. It is important to note that not all changes necessarily impact the tests. We can determine which changes are invaluable based on the changed element type and changed property. Due to the use of black-box unit tests, we can ignore changes that do not alter the interface. For instance, changes on a method's body do not change the interface.

Before we can create a list of valuable changes, we require additional checks. For instance, changes on a class' body can be of value, but depend on the changed element's new value type. For example, an inserted method does change the interface, whereas a removed variable declaration does not. Additionally, we require a check to determine whether it concerns a private method.

Changing a method's parameters may concern many changes: insertion or removal of parameters, but also renaming a parameter. The latter refactoring does not alter the interface and thus does not impact the tests.

Table 4.7 consists all changes which in general impact tests. Fields marked with a '[!]' sign require an additional access control (visibility) check. The 'Modifier' property requires this additional check to determine if the access modifier changed the interface. For example, a change from 'private' to 'protected', does not change the interface.

4.4.2. Impact determination

Before we present the standard rules to describe the impact, there are some concerns to address:

First, why is there still need for impact determination (and advice), as the Eclipse IDE automatically resolves broken code (including test code) as a result of a refactoring? The Eclipse IDE has a lot of built-in tools that automatically fixes code for you. For example, in case of a 'Move Method' refactoring, it tries to resolve all references to the moved method automatically. Although the test code is not incomplete, this specific situation still impacts the tests: the corresponding test method is located in the wrong test class. A similar situation occurs when performing the rename method refactoring. The IDE will resolve all references to the renamed method, but the corresponding test method may still use the old naming convention. These examples indicate that, even if a refactoring does not directly affect tests, tests can still be 'incorrect'.

Secondly, additional checks are a necessity. When an AST-Rewrite contains a change of the 'inserted' type, the impact may be to update the tests as well. If a new method is inserted, the impact would be a new and untested method. However, even if we only focus on inserted method declarations there remains a problem: in case of a private method, there

Changes Impacting Tests		
Changed Element	*Property*	*New Value Type*
Compilation Unit		
	Name	-
Type Declaration		
	Name	-
	Modifiers	Modifier [!]
	Types	-
	Package	-
	Body Declaration	Method Declaration [!]
Method Declaration		
	Name	-
	Modifiers	Modifier [!]
	Return Type	-
	Parameters	Single Variable Declaration

Table 4.7: A list of changes that impact tests.

is no need for a corresponding test method, as a private method is normally not directly accessible by test classes. Additionally, the kind of operation also needs to be taken into consideration. Due to AST-Rewrite linking we could find out a change is is part of a 'move' refactoring. This means that instead of the original 'inserted' operation, we now know this change is due to a move. For example, when a method is moved, there is no need for a new test method, as the old test method can be moved as well. This changes the impact from 'untested method' to 'method in wrong test class'.

IMPACT OF CHANGES

In Table 4.8 we added the impact on the tests for each change of Table 4.7. In some cases, the impact on tests is dependent on additional attributes. For instance, in case of a changed method in a class' body. Aside from the access modifier, we have to check the operation as well.

The additional rules for the access modifier are straightforward. Note that an element's access modifier can be empty in case of 'package private' visibility, or due to being previously nonexistent.

- Original value: '**private**'
  If new value is '*protected*'; no necessary changes.
  If new value is '*public*'; a new test method/class is required.
  If new value is '*(empty)*'; no necessary changes.

- Original value: '**protected**'
  If new value is '*private*'; no necessary changes.
  If new value is '*public*'; a new test method/class is required.
  If new value is '*(empty)*'; no necessary changes.

- Original value: '**public**'
  If new value is '*private*'; the test method/class is no longer required.

Changes Impacting Tests		
Changed Element / Property	*New Value Type*	*Impact on tests*
Compilation Unit		
Name	-	Incorrect test package name.
Type Declaration		
Name	-	Incorrect test class name.
Modifiers	Modifier	Dependent on original and new (access) modifier: untested class / unused test class.
Types	Type Declaration	Untested class.
Package	-	Test class in incorrect test package.
Body Declaration	Method Declaration	Dependent on the access modifier and operation: untested method / unused test method / test method in wrong test class.
Method Declaration		
Name	-	Incorrect test method name.
Modifiers	-	Dependent on original and new (access) modifier: untested method / unused test method.
Return Type	-	Incorrect return type for assertion in test method.
Parameters	Single Variable Declaration	Incorrect method parameters in test method.

Table 4.8: A list the impact of changes on tests.

If new value is '*protected*'; the test method/class is no longer required.
If new value is '*(empty)*'; the test method/class is no longer required.

- Original value: '(empty)'
  If new value is '*private*'; no necessary changes.
  If new value is '*protected*'; no necessary changes.
  If new value is '*public*'; a new test method/class is required.

The rules for the operation are similar:

- Operation: '**inserted**': a new test method/class is required.
- Operation: '**removed**': the test method/class is not longer required.
- Operation: '**moved**': the test method/class is located in the incorrect class/package.
- Operation: '**replaced**': a new test method/class is required.
- Operation: '**extract**': a new test method/class is required.
- Operation: '**inline**': the test method/class is not longer required.

The access modifier rules outweigh the operation rules. This means that the insertion of a private method results in 'no necessary changes' instead of ' a new test method/class is required'.

IMPACT TYPES

Based on the data of Table 4.8, we can identify a set of similar impact situations. There are multiple changes that have a similar effect on the tests, we combine these changes as a set and call them 'impact types'. The impact type is a description of impact of the changes on tests. The impact description can be constructed by using the rules of the previous section and inserting the 'ChangedElementType' and 'ChangedElementLocation' to provide the description with the correct type and location. Below, the current list of impact types:

- Incorrect name: A test [ChangedElementType] (read: method/class/package) has an incorrect name.
- Incorrect location: A test [ChangedElementType] (method/class) is located in the incorrect test [ChangedElementLocation] (class/package).
- Incorrect coverage: A [ChangedElementType] (method/class) has no corresponding test [ChangedElementType] (method/class).
- Incorrect coverage: A test [ChangedElementType] (method/class) is unused (has no existing counterpart).
- Incorrect assertion: assertion uses incorrect [ChangedElementNewValueType] (parameters or return type).

As it is possible to have multiple changes that impact the test, we create an object called 'impact data'. The impact data is essentially a list of impact types and corresponding change data, which should provide sufficient information to base an advice on.

4.5. CONCLUSION

In this chapter we have presented the advice process as four distinct sub processes. In order to leave the analysis of the test structure as future research, we provided a set of test structure assumptions.

For the process we have two data sources to work with; AST-Rewrites and the refactoring data. The main challenge with refactoring data is that it is a mere description, which means that we cannot acquire actual changed objects. The AST-Rewrites are challenging to work with, because processing them individual would occasionally result in incorrect advice. Therefore we presented basic rules to determine the correct changes on the code as a result of a refactoring. These rules focus on the search for 'links', which are matching values between rewrites. Additionally, we provided a 'Changes' object to specify which information we require to preform the impact determination.

For the test impact determination we concluded that only a select set of rewrite attribute combinations could affect the black-box unit test. Several of these combinations are dependent on the changed values, for which we provided an additional set of rules. Due to this limited set of situations, we could create a list of five different impact types, which should cover all possible changes.

We can now complement our research questions:

RQ1: Which different output combinations can the plug-in return?

In previous chapters we pointed at the challenge to work with large amounts of possible different AST-Rewrites. Based on examining links between rewrites, we showed that only in case of an extract, inline or move operation the combination of AST-Rewrites can provide additional information. Other rewrites can be processed individually by using their regular operation type. This reduces the important output combinations; extract, inline, move or no combination (individual assessment).

For the individual rewrites, we can further reduce the amount of important rewrites if we only focus on rewrites that impact the tests. Only a certain amount of attribute combinations have a chance of affecting the tests.

RQ2: Can the original refactoring be requested from a set of AST-Rewrites, and if not, can AST-Rewrites be recombined to reveal the initial refactoring?

With the use of a refactoring listener, we can catch a description of the performed refactoring. Additionally we presented the basics of an approach to reverse engineer the original refactoring from the AST-Rewrites. Thus, both approaches are a possibility, but both introduce additional challenges.

5
Creating advice

With the impact of a refactoring on the tests determined, the next step is to determine and create the advice. In this chapter we describe how we can create advice based on the previous determined impact types. We describe two advice types, distinguishable by the amount of detail.

5.1. Global advice

The amount of data required to determine the impact on the tests is already of significant size. Examples in previous sections show that certain data combinations of an AST-Rewrite result in the same global impact, but slight differences can cause a huge change in impact.

Previous research resulted in the 'Retestoring' plug-in. As mentioned before, this plug-in is capable of tracking AST-Rewrites, but does not give advice. However, this research provides a first draft for creating advice. The approach lists several of data combinations that result in the same global advice. For example: "(TypeDeclaration, bodyDeclarations, Inserted, MethodDeclaration)" would result in an advice similar to "add a new test method or add new assertions to an existing test method." [Passier et al., 2016]. In the previous section we pointed out that listing all these combinations is inefficient and would not provide us with a correct (fitting) 'result' for each and every situation.

Instead, we focus on the impact types listed in the previous subsection. This list combines changes with a similar outcome, which is also useful for the advice. We can extend these impact types with a generic advice, which can be filled with the available change data similar to how the impact type description is constructed. For example: the impact type 'incorrect name' would result in an advice similar to: 'Update name of test [ChangedElementType] for [ChangedElementType] [ChangedElementName] in [ChangedElementLocation].' Some changes provide additional data, for instance, a 'move' provides the new values as well as the original. Note: 'Original:' means this concerns the first element in the 'OriginalChangedElements' list.

Additionally, several parameters are dependent on the specific code situation, hence the '/' between the possible input attributes. The For instance, in case of a remove rewrite, we would be interested in the 'original value', whereas in case of an insert rewrite, the 'new value' is important. The 'Changed Element Property' is the important attribute in rewrites where, for example, the access modifier has been changed. This means the same rules can still be applied for each situation, but require preliminary checks to select the correct parameter values.

- Incorrect name: Update name of test [ChangedElementType] for [ChangedElementType] [ChangedElementName] in [ChangedElementLocation].
- Incorrect location: Move test [ChangedElementType/ChangedElementNewValueType/ChangedElementOriginalValueType] for [ChangedElementType/ChangedElementNewValueType/ChangedElementOriginalValueType] [ChangedElementName/ChangedElementNewValue/ChangedElementOriginalValue] from [Original:ChangedElementLocation] to [ChangedElementLocation].
- Incorrect coverage (no test): Create a test [ChangedElementType/ChangedElementNewValueType/ChangedElementOriginalValueType] for [ChangedElementType/ChangedElementNewValueType/ChangedElementOriginalValueType] [ChangedElementName/ChangedElementNewValue/ChangedElementOriginalValue] in [ChangedElementLocation].
- Incorrect coverage (no original): Remove the test [ChangedElementType/ChangedElementNewValueType/ChangedElementOriginalValueType] for [ChangedElementType/ChangedElementNewValueType/ChangedElementOriginalValueType] [ChangedElementName/ChangedElementNewValue/ChangedElementOriginalValue]] in [ChangedElementLocation].
- Incorrect assertion: Update the [ChangedElementProperty] for the assertion used in the test for [ChangedElementType] [ChangedElementName] in [ChangedElementLocation].

This provides us with a global advice on which actions to take in order keep the tests up-to-date.

5.2. DETAILED ADVICE

Global advice is the first step to making it easier to maintain test. The 'change data' and 'impact type' provide us with more than sufficient information to give the advice some extra detail. For example, in case of an 'Extract Method' refactoring, we could also provide additional advice based on the origin of the new method's body. For the completion of the new test method, we could add the following line to the global advice: '... , use knowledge from the test of [original:ChangedElementName]'. For the leftover of the original method we could advice: 'Revise test method for method [original:ChangedElementName] in the corresponding test class '[original:ChangedElementLocation]', focus less on functionality of [ChangedElementName].

In a previous section, we briefly mentioned an approach where, subsequent to a refactoring, the same refactoring can be applied on the corresponding tests. Due to the use of black-box tests, such approach cannot be used. The detailed advice presented above suffers from the same problem. The contents of black-box test methods do not match the corresponding regular methods, therefore, the extra detail has no real value.

In general, detailed advice could only be applicable in case of white-box tests, as white-box tests are based on the knowledge of the internal structure of methods. Because we use black-box tests, we list detailed advice under future work. For the current research the global advice is sufficient.

5.3. ADVICE OBJECT

In order to have different presentations of the advice, we have to create an object for the advice as well. Depending on the presentation style or language, the order of the values can change and some values have to be translated. Parts that are language dependent are indicated between the '<' and '>' signs.

- Situation: [Impact type]
- Advice: [Advice type] (Containing: the change (<Update>/<Create>/<Move>/<Remove>) and the element (<test> [ChangedElementType/ChangedElementNewValueType/ChangedElementOriginalValueType] / <the assertion used in the test>))
- Element Type: [ChangedElementType]
- Element Name: [ChangedElementName]
- Location: [ChangedElementLocation] and [Original:ChangedElementLocation].

5.4. CONCLUSION

We concluded that we could not use detailed advice because it would not provide additional value in contrast to the global advice. Therefore we use the global advice creation rules based on previous determined impact types. This approach requires additional checks to determine which attribute values we require as parameter, which is not ideal. However, in this way, the advice creation rules can be kept to a select set.

Based on our advice creation rules, we can now answer our fourth research question:

RQ4: Can the refactored code itself contribute to the advice?

The refactored code itself is used in several advice cases. The code is included in several of the AST-Rewrite attributes, which are also included in the 'Changes' object. For example, in case of an inserted method rewrite, the 'ChangedElementNewValue' attribute contains the entire method declaration code.

Currently, we only extract the name of code elements from these values, but if we would implement the linking of AST-Rewrite, the values would be indispensable. Thus, the refactored code itself certainly has value for the advice determination and creation.

6 Presentation of advice

The created advice has to be presented to the user at some point. This chapter describes possible ways of presenting the determined advice. We take a look at possible advice presentation standards and styles.

6.1. Advice as a refactoring

Ideally we would like to advice a ’fixture’ refactoring for the tests as a result of a refactoring. This refactoring could be automated and the issues with the corresponding tests would be solved in no-time. Theoretically, advising refactoring should be possible. If we, for example, extract code from method A to method B with an ’Extract Method’ refactoring, we could assume that we could also extract the corresponding test code from test method A to test method B with an ’Extract Method’ refactoring.

Finding the corresponding test code would already be quite the challenge. However, we discussed the main issue before: the JUnit tests are black-box tests. This approach could only work if tests have similar code as its corresponding methods, which is not the case. A test consists of an initialization, providing input, receive the output and the final assertion of the received and expected value.

Therefore it does not make sense to present advice as a refactoring in all cases. You could perform the refactoring for the overall ’meaning’ of the code but it is not literally applicable for the code itself. However, we notice that in case of an ’Incorrect name’ or ’Incorrect location’, the resulting advice is similar to the initial ’move’ and ’rename’ refactorings. Because this reasoning does not apply to the majority of the advice cases, we do not elaborate this in more detail.

6.1.1. Advice in AST-Rewrites

An interesting point could be to reconvert advice into a set of resulting AST-Rewrites. Every change to the code can be presented as an AST-Rewrite, thus this should apply for the advice as well. The advantage is the possibility to automate the advice by performing the advised AST-Rewrites. The downside is that AST-Rewrites are not meant to be presented to users and therefore lack readability. Additionally, it is a separate challenge to determine the content of new test elements when inserted.

For current research, AST-Rewrites are not a fitting standard either. However, for future work, the automation aspects could be useful.

6.2. TEXTUAL ADVICE

Textual advice is a string representation of the advice created in the 'Advice module'. The textual advice depends on two 'parameters': presentation style and language.

- **Language**
  Language of keywords and advice constructs.

- **Presentation style**
  Way of presenting the advice; full sentences, keywords, etcetera.

For this research, we will provide a description of one presentation style and one language.

6.2.1. LANGUAGE

The language of the advice specified by the language dictionary, which contains a list of keywords/sentences and the corresponding translation. We decided not to translate programming keywords, and to keep the keyword name identical to the definition in programming language. To accomplish this, we use a 'Resource Bundle[1]'. This bundle contains locale-specific (language) objects. In our case, we require string resources. For demonstration purposes we only implement one single language. Fortunately, the 'Resource Bundle' is designed to be extended with resources and/or additional languages support.

The following keywords and phrases are language dependent:

Keywords: Update, Create, Move, Remove, Assertion, Coverage, Location, Test.
Phrases and non-keywords: Name, Incorrect, For, From, To, In, Name of, For the assertion used in.

6.2.2. PRESENTATION STYLE

The most straightforward approach is to present the advice as is; a simple string representation of the advice object. We present an example form which can be applied as a simple presentation style:

- Standard form:
  [Advice type] <for> [ChangedElementType] [ChangedElementName] <in> [ChangedElementLocation].

- In case of a move (if [Original:ChangedElementLocation] is not empty):
  [Advice type] <for> [ChangedElementType] [ChangedElementName] <from> [Original:ChangedElementLocation] <to> [ChangedElementLocation].

Another example style, would be to provide the user with additional information on the situation, by presenting the impact type or a description of the change.

6.3. CONCLUSION

Presenting advice as a refactoring is not always possible due to the use of black-box tests. An AST-Rewrite is not applicable as an advice standard, because of the lack of readability.

[1]http://docs.oracle.com/javase/6/docs/api/java/util/ResourceBundle.html

Therefore we provided a straightforward textual standard presentation approach, as well as some corresponding styles as an example, which should be sufficient for a proof of concept.

With these insights we can now complement our research questions:

RQ3: Is it possible to group AST-rewrites into a larger standardized advised change, and if so, which standards are applicable to present the advice in?

We have concluded that advising a larger standardized change, for example a refactoring, is not applicable in most cases due to the use of black-box tests. Currently, only a few pieces of advice have similarities with available refactorings. Therefore, we decided that global advice is sufficient. In the future, advice could be converted into AST-Rewrites, which should be applicable for all advice situations. However, AST-Rewrites are not readable and therefore would require a separate standard for the presentation purposes.

RQ5: Is it possible to automate the advised changes with available refactorings in the Eclipse IDE or JDT?

In RQ3 we discusses that it should be possible in certain cases to advice a refactoring. Because advising a refactoring is currently not applicable in all cases, we see no value in further researching 'advice refactorings'. Therefore we will add this research question to future work.

7 Plug-in

In the context of this research we mentioned the 'Retestoring' plug-in, to track the modifications as a result of a refactoring. Throughout this research, we described its output and how we intend to effectively use this data.

The initial goal is to create our own plug-in as an addition to the whole process (from refactoring to advice). As a second step, we intend to complement the Retestoring plug-in with the additional functionality.

7.1. Study

7.1.1. Custom Plug-in

The 'Custom plug-in' is our own approach to obtain 'global' refactoring information from the Eclipse IDE after a refactoring. We intended to keep this plug-in simple, as its goal is to provide a proof of concept that it is possible to track the refactoring data available in Eclipse.

Our Java plug-in consists of two parts: an activator and a handler. The activator[1] controls the life-cycle of the plug-in; it can be used to define start-up and shutdown functions. Other functions are defined in the handler. The handler defines the actions that need to be performed when certain menu commands are executed.

In our case, we have one single action; track refactoring events. This action starts a refactoring listener, which is triggered by the execution of a refactoring event. When the refactoring is 'done' (all properties set and fully executed), we retrieve a description of the refactoring from the event. This 'RefactoringDescriptor' is what we call the 'refactoring data'.

7.1.2. Existing Retestoring Plug-in

The Retestoring plug-in is more sophisticated than our custom plug-in. Aside from the detailed user interface, its tracking functions are complex due to the additional required techniques. For example, AspectJ and Equinox/Weaving are a necessity in order to trace AST-Rewrites within the plug-in [Passier et al., 2016].

The plug-in uses an aspect (AspectJ) to record changes to the AST as a result of a refac-

[1]http://help.eclipse.org/luna/index.jsp?topic=/org.eclipse.pde.doc.user/guide/tools/project_wizards/plugin_content.htm

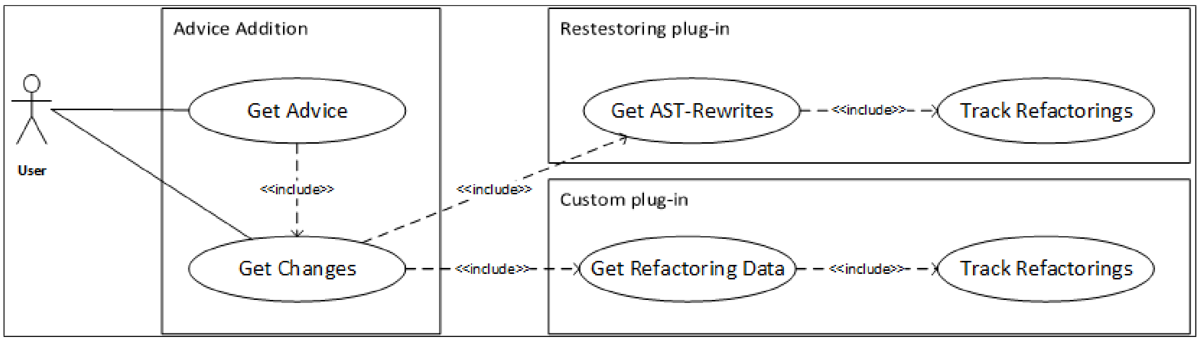


Figure 7.1: Advice Addition UML Use-case diagram

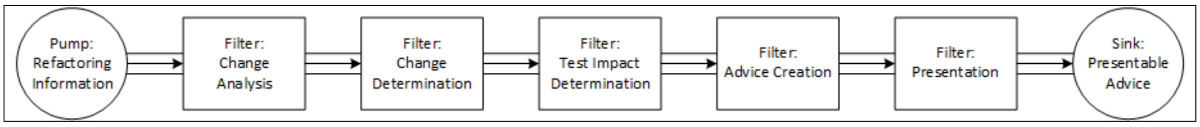

Pump: Refactoring Information
Filter: Change Analysis
Filter: Change Determination
Filter: Test Impact Determination
Filter: Advice Creation
Filter: Presentation
Sink: Presentable Advice

Figure 7.2: Pipe Filter Advice Architecture

toring. This technique makes it possible to inject behavior into another plug-in. This allows the plug-in to intercept the refactoring control flow in the JDT, and thus obtain the AST-Rewrites. The rewrites are then stored and processed separately.

The Retestoring plug-in uses an extension point[2] to provide others with AST information. The extension point provides modularity; it allows the plug-in to be extended by another plug-in. In this case, the plug-in sends AST-Rewrite information to subscribers of the 'NewRewrite' extension point. As a refactoring may consist of several rewrites, and each rewrite is processed separately, the result is not a single 'NewRewrite' event, but a series of new rewrites.

7.1.3. THE ADVICE ADDITION

Our addition to the current situation, is to create and present advice based on the data of both previous mentioned plug-ins. The main goal is to provide a user with advice. Additionally, we could provide information on the change as well. The commonality between both tasks, is that they require refactoring information. Currently, refactoring information is tracked by both plug-ins; respectively to gather the AST-Rewrite data and refactoring data. This situation is summarized in the UML use-case diagram in Figure 7.1.

The advice addition includes all modules required to create and present advice based on the plug-in input data. Each module has a separate task, with a predefined interface and specified output. The output is actually the interface of the next module in the process chain. Thus, in order to present the advice, data has to flow through all modules in the correct order. As the input data essentially is based on a continues stream of refactorings, we conclude that the situation is very similar to the pipe/filter architecture.

The plug-in input data can be seen as the 'pump'. The data flow between the modules are the 'pipe'. The modules themselves are essentially a 'filter'. The data output at the end of the module chain is the so called 'sink', which, in this case, is equal to the presentable advice. This architecture is visualized in figure 7.2.

Although all modules can perform their task independently, modules have a specific

[2] https://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points%3F

executing order. Aside from filtering data, modules also add information and convert data to make it suitable for the next module. This behavior may not be ideal for a filter, as it becomes difficult to separate the global task into threads. However, parallelization is not an obligation. Additionally, we cannot predict if threading is a necessity at all, aside from an additional thread to preserve the user interface functionality.

7.2. ADVICE CREATION PLUG-IN EXTENSION

We have created a proof of concept plug-in extension for the Retestoring plug-in. We use the aforementioned pipe-filter design as a basis for the internal architecture. Due to the fact that all modules have different input and output, we use a filter with generic input and output. The pipe contains a list of AST-Rewrites with a description of the refactoring. The sink eventually receives a list of presentable textual advice.

The plug-in starts when the ’`NewRewrite`’ extension listener receives AST-Rewrite data from the Retestoring plug-in. As a refactoring may be split in multiple separate ’`NewRewrite`’ events, we added a button to start the advice creation process manually. When the advice creation is started, the refactoring data flows through each filter and eventually result in the presentation of the pieces of advice to the user. If a refactoring requires no changes, there will not be any advice.

For this proof of concept version we have not fully implemented every filter. We have implemented the filters to such an extent, that we can receive raw advice for predetermined refactorings. We left the ’Change Determination Module’ and ’Presentation Module’ unimplemented, due to not having additional value for this concept. As converting the refactoring data to the defined ’Changes’ object is a task on itself, we implemented the ’Change Analysis’ module to fill the ’Changes’ object without rules. Therefore, we only support refactorings for which the individual rewrite advice applies. For instance, according to our test impact rules, in case of a simple ’extract method’ refactoring, the insertion of the new method is the only rewrite that affects the tests state. Because we currently only create global advice, the individual rewrite assessment results in the same advice. As an example, we implemented the rules to recognize a ’move’ refactoring. Because, in case of a ’Move Method’ refactoring the individual rewrite result would be incorrect. Similarly, we implemented rules for refactorings that changes the access type.

7.2.1. VERSIONS

Current results may deviate from future results due to the different software versions. The advice addition plug-in project is created in Eclipse Neon version 4.6.2. It is an extension to the Retestoring plug-in version 1.0.0. We worked with the Java Development Tools version 3.12.2, Asptect-J version 2.2.4, and lastly, Weaving version 2.2.4. In order to obtain similar results, use these particular software versions.

7.3. VALIDATION

In order to determine the usefulness of the advice addition plug-in, we test it with an example refactoring situation. Additionally, we also provide the manual advice creation result in order to validate the output.

7.3.1. TEST CASE

In this first example, we created a '`Calculator`' class and a '`Figure`' class. The '`Figure`' class uses several calculations of the '`Calculator`' class. The method '`multiply`' is too simple, so we decide to inline its functionality. In Table 7.1 we show the initial code and resulting code of this refactoring example. To validate the output, we will manually determine the advice based on all available refactoring information. The 'Inline Method' refactoring results in the following list of AST-Rewrites, see Table 7.2. Note that, due to space constraints, certain body and parameter values are presented as three dots. Table 7.3 shows the corresponding refactoring data.

MANUAL ADVICE DETERMINATION

For the manual approach, we loosely follow the steps described in the 'Results on Tests' and 'Creating Advice' chapters.

The refactoring data provides us the element of interests, and the name of the refactoring. The most important rewrite is the removal of the '`multiply`' method in the '`Calculator`' class (rewrite #9). Because we know the initial refactoring, it becomes easier to find the fitting linked rewrites, which in this case are rewrite #2, #6, #7 and #8. Thus, the 'Changed Element' is the data of rewrite #9. Rewrite #2, #6, #7 and #8 are the 'Original Changed Elements', and rewrite #1, #3, #4 and #5 are the 'Additional Changed Elements'.

The next step is to determine which rewrites could impact tests, starting with the changed element (Rewrite #9). This element impacts the tests, because it removes a public method from a class. The 'Impact Type' for this element therefore is 'Incorrect coverage' (unused test). Next up, are the original changed element. None of these elements have an impact on tests, because they all concern the body of a method. The same goes for the additional changed elements. This leaves us with one changed element to determine an advice for.

The last step is to determine the advice. Based on the impact type, we can use an advice template to create the advice. This results in the following piece of advice: '*Remove the test method for method multiply in Calculator.*'.

ADVICE PLUG-IN OUTPUT

The plug-in outputs the following line of advice: '*Remove the test method for method public int multiply(int first,int second) in public class Calculator.*'. This advice is almost identical to the manual advice determination as intended.

7.3.2. ADDITIONAL TEST CASES

For the additional test cases, we use the example refactorings of Chapter 3. Instead of a comprehensive stepwise explanation on how to determine the advice in each situation, we provide our predetermined advice right away and check corresponding the plug-in's output advice. These test cases are shown in Table 7.4.

In general, the advice seems to match the expectation. There are, however, some noteworthy cases, which we discuss below:

- Test case #1 has no advice because the extraction introduces a private method.
- The reason why test #7 fails is due to the particular AST-Rewrite output of the refactoring. For example, one would expect a new class rewrite, but instead receive a rewrite of the class filling its own body. Additionally, several rewrites are duplicated

Inline method result

Original code

```
public class Calculator {

  public int multiply(int first,
      int second){
    return first * second;
  }

  public int square(int number){
    return
        multiply(number,number);
  }

  public int cube(int number){
    int side =
        multiply(number,number);
    return multiply(number,
        side);
  }
}

public class Figures {

  Calculator calc;
  public Figures(){
    calc = new Calculator();
  }

  public int
      calculateCubeVolume(int
      length){
    return calc.cube(length);
  }

  public int
      calculateCubeArea(int
      length){
    int side =
        calc.square(length);
    return calc.multiply(6,
        side);
  }
}
```

Refactored code

```
public class Calculator {

  public int square(int number){
    return number * number;
  }

  public int cube(int number){
    int side = number * number;
    return number * side;
  }
}

public class Figures {

  Calculator calc;
  public Figures(){
    calc = new Calculator();
  }

  public int
      calculateCubeVolume(int
      length){
    return calc.cube(length);
  }

  public int
      calculateCubeArea(int
      length){
    int side =
        calc.square(length);
    return 6 * side;
  }
}
```

Table 7.1: Code before and after the 'Inline Method' refactoring

Inline method AST-Rewrites						
#	*Changed node*	*Property*	*Change*	*Original value*	*New Value*	*Placeholder*
1	`public int multiply(...){...}`	body	Replaced (placeholder)	`first`	`0`	`6`
2	`public int calculateCubeArea (int length){...}`	body	Replaced (placeholder)	`return calc.multiply(6, side);`	`return;`	`return 6 * side;`
3	`public int multiply(...){...}`	body	Replaced (placeholder)	`first`	`MISSING`	`number`
4	`public int multiply(...){...}`	body	Replaced (placeholder)	`first`	`MISSING`	`number`
5	`public int multiply(...){...}`	body	Replaced (placeholder)	`first`	`MISSING`	`number`
6	`public int cube(int number){...}`	body	Replaced (placeholder)	`multiply (number, number);`	`MISSING()`	`number * number`
7	`public int cube(int number){...}`	body	Replaced (placeholder)	`return multiply(number, side);`	`return;`	`return number * side;`
8	`public int square(int number){...}`	body	Replaced (placeholder)	`return multiply(number, number);`	`return;`	`return number * number;`
9	`public class Calculator{...}`	bodyDeclarations	Removed	`public int multiply(int first, int second){...}`		

Table 7.2: The AST-Rewrites of the 'Inline Method' refactoring example

Inline method refactoring data
`Inline method 'Calculator.multiply()' in 'Calculator'`
`Original project: 'MyProject'`
`Original element: 'Calculator.multiply()'`
`Remove method declaration`
`Replace all references to method with statements`

Table 7.3: The data of an 'Inline Method' refactoring

and also contain '`MISSING`' references. To support this refactoring, we would require a specific method to correctly fill the 'changes' object for this situation.

- For test #8 one would expect an advice to also create a test for the constructor of the class. However, the AST-Rewrite output of this refactoring does not explicitly shows the creation of methods for an extracted class. Therefore, we would also require a special method to handle this refactoring situation.

7.3.3. CONCLUSION

We have created a proof of concept plug-in which is capable to determine the advice for each AST-Rewrite individual. Because we only determine the global advice, this approach already results in the correct advice determination for several refactoring situations. As an example, we added support for the 'move' refactoring, which always requires a link between AST-Rewrites in order to determine the correct advice.

Based on the results of the test cases, we can conclude that we indeed determine the correct advice for several refactoring situations. However, there are still unsupported refactorings, which lead to an incorrect result. An important reason for this problem, is the fact that the implementation of refactorings in Eclipse are inconsistent. The approach for determine the correct results is therefore slightly different for every situation. In most of the cases, these refactorings could be supported by creating an approach for filling the 'Changes' object in case of that particular refactoring.

Test cases				
#	*Case*	*Expected output*	*Plug-in output*	*Match*
1	Extract Method, Table 3.1	No advice	”	Yes
2	Inline Method, Table 3.3	Remove the test method for method multiply in Calculator.	’Remove the test method for method public int multiply(int first,int second) in public class Calculator.’	Yes
3	Move Method, Table 3.4	Move the test method for method add from MyCalculator to MyNewCalc.	’Move test method for method public static int add(int x, int y) from public class MyCalculator to public class MyNewCalc.’	Yes
4	Move Method Alternative, Table 3.5	Move the test method for method add from MyCalculator to MyNewCalc.	’Move test method for method public static int multiply(int x, int y) from public class MyCalculator to public class MyNewCalc.’	Yes
5	Introduce factory, Table 3.6	Create a test method for method createCar in public class Car. Remove the test method for method Car.	’Create a test method for method public static Car createCar() in public class Car. Remove the test method for method public Car().’	Yes
6	Introduce parameter, Table 3.7	Update the parameters for the assertion used in the test for method setName.	’Update the parameters for the assertion used in the test for method public void setName().’	Yes
7	Extract superclass, Table 3.8	Create a test class for class Fruit. Move the test method for method eat from Apple to Fruit. Create a test method for method Fruit in Fruit.	’Create a test method for method public Fruit() in public class Fruit. Remove the test method for method public void eat() in public class Apple extends Fruit. Create a test method for method public MISSING eat() in public class Fruit. Remove the test method for method public void eat() in public class Apple extends Fruit. Create a test method for method public MISSING eat() in public class Fruit.’	No
8	Extract class, Table 3.9	Create a test class for class Wheel. Create a test for method Wheel in Wheel.	’Create a test class for class public class Wheel.’	No

Table 7.4: Test cases for the advice addition plug-in

8
Related Work

Test maintainability is a common topic in TDD environments. We will address some important related work below, and describe how it relates to our research.

8.1. XUnit Test Patterns & Test Smells

The 'xUnit Test Patterns' book describes an approach to improve the structure of test code in order to also improve maintainability. It introduces strategies and patterns that aid the programmer to write tests that are better understandable, maintainable, and to reduce bad smells in the test code [Meszaros, 2007].

Bad smells for test code specific are called 'Test smells', and are used to identify test code flaws [van Deursen et al., 2001]. An example of a test smell is the so called "*copy, paste, modify*" style of creating tests, which is a common practice [Li and Thompson, 2009]. This style of creating tests uses a code copy of an existing test as a base for a new test, which obviously introduces a certain amount of code duplication. Code duplication is therefore also a test smell. The test smells come with corresponding test code refactorings to make test code more understandable and maintainable [van Deursen et al., 2001]. As with ordinary code smells, the search for test smells and its corresponding refactorings can also be automated [Greiler et al., 2013].

Similarly to our own research, the aforementioned test maintenance approaches mostly fall into the category of preventive maintenance. However, in contradiction to our research, the focus lies on improving the structure of the test suite itself in order to achieve maintainability. Whereas our research focuses on maintaining the correctness of tests while improving the structure of the source code. Additionally, our research differentiates from aforementioned related work, because we solely focus on black-box tests. As a black-box test does not take the implementation of the method under test into consideration, it is difficult, or sometimes not even possible, to meaningfully apply (smell fixture) refactorings on the test.

8.2. Software Evolution

The book 'Software Evolution' devotes a chapter to describe that refactorings can indeed invalidate tests [Moonen et al., 2008]. In this chapter, they divide refactorings [Fowler et al., 1999] into categories based on how they change the interface. They list five refactoring categories:

- Composite: large refactorings that consist of multiple smaller refactorings.
- Compatible: a refactoring that does not change the interface, for example, the ’Split Temporary Variable’ refactoring.
- Backwards compatible: a refactoring that only adds to the interface, for example, an ’Extract Method’ refactoring.
- Make backwards compatible: a refactoring that changes the interface, but can be made backwards compatible by using a wrapper to retain the old interface. For example, the ’Move Method’ refactoring.
- Incompatible: a refactoring that changes the interface, and cannot be made backwards compatible. For example, the ’Inline Method’ refactoring.

For the compatible categories they provide examples which roughly describe a manual approach on how one could update the tests.

The approach in this book comes closest to our research, as it describes how a refactoring on the source code can impact the tests. The refactoring categories are different from our ’impact types’, because their categorization differentiates refactoring changes that can and cannot be solved with a refactoring or pattern. The main reason is probably due to their use of white box tests, which makes more refactorings applicable to have a possible impact the tests.

Another important difference is the additional focus on validation of the behavior of the source code while updating the tests after a refactoring [Moonen et al., 2008]. For example, in case of a ’Move Method’ refactoring, their first focus lies on fixing the test with a wrapper to check the refactoring. Moving the test method to another location comes after the validation. The authors even state that "*we do not want to change the tests together with a refactoring since that will make them less trustworthy for validating correct behavior afterwards.*" [Moonen et al., 2008]. We agree that, if you want to maintain the behavior of the source code after a refactoring, changing tests at the same time disrupts the validation of the refactoring. However, our research is aimed at maintaining the correctness of tests. Maintaining the correctness of the source code is a side aspect our research is not focused on.

It is difficult to further compare this approach in more depth, as it only describes the actual fixture part with a few examples and lacks concrete steps or rules on how to update the tests.

8.3. CONCLUSION

There is sufficient research available on the subject of test maintainability. Most related work focuses on the aspect of improving maintainability by improving the structure of the tests. There is however, little related work based on the aspect that source code refactorings can invalidate tests. We conclude that determining the effect of refactorings on tests is a distinct and still relative unexamined aspect of test maintenance.

9 CONCLUSION

We have proven that it is indeed possible to create an universal approach for determining the advice on the test situation after a refactoring. Although we still require additional checks or rules in order to be able to create the correct advice for every refactoring, we already created the basics to support a handful of refactorings.

The reason why we cannot support all available refactorings immediately, is the fact that the AST-Rewrite output depends on the refactoring implementation for a particular version of the Eclipse IDE. Additionally, a single refactoring can have many variations dependent on structure of the source code, which makes it even more complex. Refactorings that result in a set of AST-Rewrites which can not be processed individually are most difficult to support, as these require specific approaches for determining the correct required change on the test set. This is the main reason why we did not (fully) implement the 'Change Analysis' module, as creating a smart approach to tackle this challenge is a separate task on its own.

We have divided the advice determination process into several parts: change analysis, change determination, test impact determination, advice creation and finally advice presentation. This provides a future steady structure, in which modules can be swapped with newer versions when required. The partially implemented modules are therefore not a problem.

In order to deal with the vast amount of different AST-Rewrites, we concluded that we can decrease the amount of important rewrites by only focusing on rewrites that can actually have an impact on the tests. This resulted in a select group of AST-Rewrites. For these particular rewrites, the impact of the rewrite on the test can be summarized by one of five impact types. This makes the advice creation manageable, because we only have to provide advice for five distinct (global) situations.

The following section contains the conclusion for each of our research questions. In the remainder of this chapter we discuss our contribution to this research subject and evaluate the outcome.

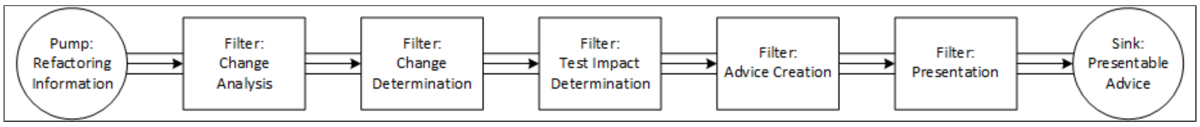

Pump: Refactoring Information
Filter: Change Analysis
Filter: Change Determination
Filter: Test Impact Determination
Filter: Advice Creation
Filter: Presentation
Sink: Presentable Advice

Figure 9.1: Pipe Filter Advice Architecture

9.1. CONCLUSION - RESEARCH QUESTIONS

RQ1: Which AST-Rewrite plug-in output includes a change that can impact tests?

The output of the Retestoring plug-in is a set of one or more AST-Rewrites, each containing a combination of attributes. Theoretically, because of the large amount of possible AST-Rewrites, we could have countless rewrite combinations.

Therefore, we conclude that we have to shift our focus to determine which combinations are valuable. We experience that certain rewrites as a result of an extract, inline or move refactoring only provide additional information when combined. Rewrites of other combinations, that do not provide additional useful information when combined, can be processed individually.

Due to the use of black-box unit tests, we conclude that only a select amount of rewrites can actually have an impact on the tests. This further reduces the amount of important rewrites. For a full list of rewrites that can impact black-box tests, see Table 4.7.

In more detail: the reason why providing a list of all possible output combinations is nearly impossible, is because we have an impractical amount of combinations, even if we would only focus on the AST-Rewrite output of the available refactorings in the Eclipse IDE. This is mainly because each refactoring can be performed in multiple different situations. For example, a 'move' can be performed on one or more (static) methods, (static) fields, types, compilation units, packages, source folders and projects[1]. Combined with the amount of possible different existing code references to this moved code element, which have to be changed subsequently, there are innumerable combinations.

As a result, we focus on more abstract combinations. These abstract combinations describe correlated attribute combinations between AST-Rewrites. These combinations ensure that we are able to recognize and distinguish additional operations (extract, inline and move), from the regular individual operations (insert, replace, remove) in a set of AST-Rewrites. The additional operations, in general, consist of multiple rewrites with certain correlated attributes. These correlated attributes are what we call 'links'. With the use of these links, we would be able to find correlated rewrites, which only provide additional information when linked.

This rewrite linking approach provides a manageable set of attribute combinations rules to check for, which are listed at the end of subsection 4.3.2. For the leftover (non-linked) rewrites, we conclude that we do not need a special approach, as processing these individually leads to the correct result.

Because black-box tests do not take the implementation of the method into consideration, we conclude that only a select set of rewrite attribute combinations could actually impact the tests. In this case, only changes to the interface can impact the tests.

RQ2: Can the original refactoring be requested from a set of AST-Rewrites, and if not, can AST-Rewrites be recombined to reveal the initial refactoring?

Yes, the original refactoring can be recombined from a set of AST-Rewrites, or requested from the refactoring data.

In more detail: based on our description of the two sources of data to work with; the AST-Rewrites and the refactoring data, we have shown that either of the data sources (or a combination of both) could provide information on the original refactoring. Although, both approaches each come with their own set of challenges; the refactoring data is a mere de-

[1] http://help.eclipse.org/luna/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm

scription and does not provide actual objects, whereas the AST-Rewrite output is sometimes inconsistent and related rewrites are difficult to link. See subsection 4.3.1 for more information.

RQ3: Is it possible to summarize the result of the analysis of a refactoring as a larger standardized advised change, and if so, which standards are applicable to present the advice in?

Currently, there are no applicable standards to present the advice in.
In more detail: with the creation of the universal advice creation rules, we conclude that a refactoring, is not applicable in most cases. This is due to the fact that the code of black-box tests does not reflect the actual method's code. Some advised changes can therefore not be portrayed by a refactoring.

The reason we are mainly interested in refactorings as a standard, is because of the possibility to automate the advice in the future. Another standard, for instance an AST-Rewrite, can also be used to automate the advice, but is not suited for presentation, since it lacks readability. See section 6.1 for more information.

On second thoughts, we ask ourselves, even if a refactoring is applicable as advice in a certain situation, is it actually suitable as an advice standard? We essentially intend to use the refactoring (advice) to fix something that is 'incorrect', which is no longer related to preventive maintenance.

RQ4: Is it necessary to collect the refactored code itself to contribute to the advice?

The refactored code itself is already part of the AST-Rewrite output. Several attributes of an AST-Rewrite contain the concerning changed code elements. Therefore, these values are used (multiple times) in the process of determining the advice. Thus, the refactored code does not have to be separately collected, because it already contributes to the advice as part of the AST-Rewrites.

RQ5: Is it possible to automate the advised changes with available refactorings in the Eclipse IDE or JDT?

No, it is currently not possible to automate the advise, as the output of the advice addition plug-in is a 'String' representation of advised changes. This output is solely for test purposes and is not suited for automation.
In more detail: in RQ3, we conclude that a piece of advice can not always be covered by a refactoring. Therefore we require a different solution for advice automation. This research question will be listed as future work, but will focus on advice automation in general instead of refactorings.

9.2. CONTRIBUTIONS

The result of the research is the advice addition plug-in and the corresponding new insights in the general research subject, which we will discuss below.

9.2.1. SOFTWARE

The advice addition plug-in is an extension of the Retestoring plug-in, and is meant to be a proof of concept to show how to give advice based on refactoring information. The current version of the plug-in is capable of determining the advice for, among others, the 'Extract/-Move/Inline Method' refactorings, and the 'Change Method Signature' set of refactorings.

The reason why we support a handful of refactorings, is because the majority of the currently supported refactorings are refactorings where individual rewrite processing yields the correct global advice. Currently, the global advice is identical to the detailed advice. This is because the additional details are not applicable for the black-box tests, and therefore not of any value for the advice. See chapter 5 for more details.

In order to support more refactorings we have to fully implement the 'Change Analysis' module. Especially for refactorings that result in two or more AST-Rewrites that describe the same exact part of the change. The 'Move Method' refactoring is an example to this.

For this proof of concept, we implemented the rules to support the 'move' refactoring and several of the 'Change Method Signature' refactorings, as an example to specifically test refactorings that with individual rewrite assessment would lead to incorrect results.

9.2.2. INSIGHTS & DISCUSSION

Due to the large amount of possible AST-Rewrite attribute combinations, creating a rule for every combination is not an option. We conclude that, in order to support all refactorings, we require to link AST-Rewrites. These links are essentially specific values matches between rewrites that help to determine if an certain change has been executed. This is a necessity, because several larger changes, for example, a 'move' refactoring, can not be portrayed as a single rewrite, but results in multiple rewrites. Linked rewrites provide required information about the initial refactoring and refactoring parameters among others.

We provide an approach to extract the necessary information from the AST-Rewrites and/or refactoring data to correctly determine the impact of a rewrite on the tests. The code itself is also used in this process; namely as several of the AST-Rewrite attributes' value. Nonetheless, the description and reasoning provide the basics and give insight on the challenges to overcome in future work.

A noteworthy discovery, is the fact that the Eclipse implementation of refactorings is not consistent. Aside from making the use of AST-Rewrites more complex, this aspect introduces an uncertainty. It means that two similar refactorings can result in different rewrite sets. In this process, we also found some bugs in the Retestoring plug-in.

The creation of the 'Changes' object provides the desired standard (data object) to determine the advice from. Although the plug-in is far from supporting every refactoring situation, the 'Changes' object provides a starting point for future additions.

Furthermore, we provide a set of rules in order to determine the impact of a change on the tests, which is based on the 'Changes' object. Due to the use of black-box tests, we experienced that it is invaluable to determine the impact of changes to a method's body (amongst others), because a method's body structure is nonidentical to its corresponding test method's body. This way, we conclude that only a select set of changes can impact tests. With this insight, we defined a set of impact types (see subsection 4.4.2).

We shaped an additional set of rules to process an impact type into an actual advice. We conclude that we cannot present the advice as a refactoring, and therefore instead provide a simple example presentation style. We intentionally left the presentation module simple, as the presentation layer does not provide additional value to this proof of concept.

ARCHITECTURE & IMPLEMENTATION

The decision to use the pipe-filter architecture for the internal design of the plug-in fits the process well. Dividing the advice process into several filters works as intended, as each filter performs a distinct task. A Filter can be replaced, as long as the replacement preserves the same input and output objects. Since the process uses a predetermined and fixed filter execution order, which requires all filters, we decided that each filter has fixed input and output objects.

We already expected that the creation of a universal rule set would be quite the challenge, but in particular, the amount of small checks to distinguish different situations from each other make the code more complex. For example, in the 'Advice Creation' processing methods, we would require different variables for the actual advice creation, dependent on the situation. For instance, in case of a remove rewrite, we would be interested in the 'original value', but in case of a insert rewrite, we require the 'new value'. These preliminary checks are deliberately not included in this research document, because these are implementation specific and do not contribute to the general design.

9.3. EVALUATION

All in all, we are satisfied with the results of our research. We reached our goal to further develop the advice determination and advice creation process.

The creation of the basic rules, in order to create an advice based on raw refactoring information, should provide ideas and starting points for future work. The test cases have shown that the rule set is sufficient for at least a handful of refactorings. However, it also shows that we still need to adapt and complement the rules to support more refactoring situations. These rules and checks are a necessity due to the difference rewrite output for each refactoring.

We did not implement the 'Change Analysis' as most of the challenges it introduced are out of scope for this research (see subsection 4.3.3). Due to not implementing this module, we ended up with additional checks divided over the other modules in the advice creation chain. We expect that most of these preliminary checks could be moved to the 'Change Analysis' without any issues. However, this would require fabricating a 'Changes' object by combining multiple rewrites. In this way, we can eliminate most of the checks in the impact and advice modules. The leftover checks would then only focus on selecting the correct values for the advice and impact creation. Note that this approach does not eliminate the possible need for additional rules and checks in case of a different/unsupported refactoring. This possibility of increasing the amount of required additional rules and checks is an important future attention point.

The advice addition plug-in fulfills its role as proof of concept by showing it is indeed possible to create advice based on a set of AST-Rewrites. Although we did not find an applicable standard to present the advice in, this does not affect the usefulness of the output. Due to the use of the pipe-filter architecture, one only has to replace the presentation filter to change the final output.

Regarding the presentation standard; finding a readable standard which can be automated as well is perhaps not possible. Because the presentation standard is not of added value for this proof of concept, we have not further researched this aspect, and just presented the advice as is. However, for future advice automation goals, separating advice automation from advice presentation, thus creating two distinct modules, is a possible solution. In this way, we can use a standard for the presentation which focuses on readability, while we use another standard to create automatable advice. We have listed this idea as future work.

The outcome of this research takes us one step closer to reaching the ultimate goal: automatically maintain test coverage after a refactoring by only using AST-Rewrites.

10
Future work

Throughout this research, we hint to future work on multiple occasions. This chapter provides a complete list of future work.

10.1. Test suite analysis

Currently, we use a list of test code status assumptions, containing, for example the assumption: 'every public method has a corresponding test'. Therefore, we do not check the actual test suite structure to see if a test method is present. The most straightforward approach, to obtain the status of the test structure itself, is to analyse the test structure and to identify if the required tests methods and classes are present. The opposite is also of importance; to check if code elements which do not need tests, are indeed not tested. We listed several of these possible situations, see section 4.2.

In order for the test suite analysis to work, we require some kind of approach to find the correct corresponding test method for a regular method. This could be simplified by using custom annotations, or by simply letting the user select the correct class.

Additionally, this test analysis can be extended by also checking for faulty code structures. For example, by following the Java coding convention guidelines. On top of that, checks could be added to test for available data which is not part of the interface.

10.2. Advice presentation

The current advice presentation consists of only an example implementation. Aside from different languages, one could research the semantics of messages. For instance, study how Eclipse itself determines/creates its messages for compiler, error or GUI messages. The context of the advice message can possibly also contribute to the quality. If applicable, one could use an advice standard, which solely focuses on presentation, and is not suitable for future advice automation.

10.3. Automatic advice execution

Ideally, the presented advice should be applicable with a single button press. We could not find a standard which is suitable for both advice presentation as well as automation. Therefore, creating a separate module for advice automation is a possible solution. In this way, a standard can be used which does not require to be readable. An example solution could be to convert pieces of advice back into AST-Rewrites, and perform the AST-Rewrites on the tests. This does however requires a thoroughly analysis of the tests structure.

10.4. ADVICE INTERFACE

Currently, we use the 'Changes' object as a starting point for the advice creation process. In the future, it would be interesting to research if we could create an interface/object for the whole process, which also functions as a required data standard for Eclipse refactorings. In the long run, this interface could help to eliminate the inconsistencies between Eclipse refactorings, and could reduce required additional checks in the advice creation process.

10.5. WHITE-BOX TESTS

In current research we only provide advice for black-box tests. In the future, we aim at also supporting white-box tests. With the use of white-box tests, we can provide a more detailed advice on how to update tests. Because white-box tests take the implementation of a method into consideration, we would be able to use some advice we currently ignore (for example, changes to a method's body). We can also complement the advice by looking at 'Test Smells' and 'Path coverage'.

In order to achieve coverage, one could check if all execution paths are covered. The same goes for test coverage, in which one checks if a set of tests cover all paths of its corresponding test method. As test path coverage increases the quality of the tests, it would therefore make a good addition to the advice.

10.6. MINOR TASKS

A list of smaller tasks which still have to be done.

- **Reconverting placeholders to AST-Nodes.**
  The AST-Rewrite placeholder value is currently a string representation. This makes comparing the placeholder with other AST-Rewrite attributes more difficult. Interpreting the placeholder and converting it to an AST-Node would help to correctly identify links that are not a (perfect) textual match.

- **Fully implement the 'Change Analysis' module.**
  Currently, the 'Change Analysis' is only a partial implementation of the rules described in subsection 4.3.2.

- **Fully implement the 'Presentation' module.**
  The 'Presentation' module currently lacks the styles and language templates described in chapter 6.

- **Extract the custom plug-in functionally into a separate plug-in.**
  For test purposes this functionality is included in the advice plug-in, but actually the functionality is separate from the advice plug-in.

- **Create a reusable approach to gather the correct values for each impact/advice rule.**

Each ’Changes’ object contains important values, but these values are located in different attributes, dependent on the type of change. Currently, we have an additional check to determine the type of the change before we can select the required values.

Bibliography

Beck, K. (2002). *Test Driven Development By Example.* Pearson Education.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing co.

Greiler, M., van Deursen, A., and Storey, M.-A. (2013). Automated detection of test fixture strategies and smells. In *IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*, pages 322–331, Luxembourg City, Luxembourg.

Hafiz, M. and Overbey, J. (2015). Refactoring myths. *IEEE Software*, 32(6):39–44.

IEEE Standards Board (1999). IEEE standard for software unit testing: An American national standard, ANSI/IEEE Std 1008-1987. In *IEEE Standards: Software Engineering*, volume vol. 2: Process Standards. The Institute of Electrical and Electronics Engineers, Inc.

International Standards Organisation (ISO) (1999). Iso/iec 14764. In *Standard for Software Engineering – Software Maintenance.* ISO/IEC.

Li, H. and Thompson, S. (2009). Testing-framework-aware refactoring. In *the Third ACM Workshop on Refactoring Tools*, pages 182–196, Orlando, USA.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

Meszaros, G. (2007). *XUnit Test Patterns: Refactoring Test Code.* Addison-Wesley.

Moonen, L., van Deursen, A., Zaidman, A., and Bruntink, M. (2008). On the interplay between software testing and evolution and its effect on program comprehension. In *Software Evolution*, pages 173–202. Springer Berlin Heidelberg.

Oates, B. (2006). *Researching Information Systems and Computing.* Sage Publications Ltd.

Passier, H., Bijlsma, L., and Bockisch, C. (2016). Maintaining unit tests during refactoring. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*, Lugano, Switzerland.

Schäfer, M., Verbaere, M., Ekman, T., and de Moor, O. (2009). Stepping stones over the refactoring rubicon. In *ECOOP 2009–Object-Oriented Programming*, pages 369–393. Springer-Verlag.

Tahchiev, P., Leme, F., Massol, V., and Gregory, G. (2010). *JUnit in Action.* Manning Publications.

van Deursen, A. and Moonen, L. (2002). The video store revisited–thoughts on refactoring and testing. In *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 71–76, Alghero, Italy.

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

Zhang, M., Hall, T., and Baddoo, N. (2011). Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(3):179–202.

Acronyms

AOP	Aspect Oriented Programming.
AST	Abstract Syntax Tree.
IDE	Integrated Development Environment.
JDT	Java Development Tools.
RQ	Research Question.
TDD	Test Driven Development.
UML	Unified Modeling Language.
XP	eXtreme Programming.

GLOSSARY

Abstract Syntax Tree (AST) is a tree representing the abstract syntactic structure of the source code of a program.

Abstract Syntax Tree Node (AST-Node) is a node in the AST representing a certain block of code structure in the source code.

Abstract Syntax Tree Rewrite (AST-Rewrite) or rewrite for short, is a change on the node of an abstract syntax tree, and is a result of a structural change to the source code of a program.

Bad smell or code smell, is a certain source code indication that indicates a possible design flaw or a structural problem.

Black box unit test is a test which does not take the internal structure of the unit under test into account.

Impact type is, in this research context, a categorized set of changes on the source code that have a similar effect on the tests.

JUnit is a unit testing framework for the Java programming language.

Link is, in this research context, a certain attribute value match between two AST-Rewrites.

Micro-Refactoring is a sub step of a refactoring.

Placeholder is, in the AST-Rewrite context, a 'String' type value that holds data during a move or replace.

Refactoring is the process of improving a program's internal structure without changing its external behavior.

Test is, in this research context, a unit that examines the behavior of another distinct unit.

Unit is the smallest testable part of an application. In this research context, a unit refers to a single method in the Java programming language.