Automatic Refactoring for Renamed Clones in Test Code

by

Jun Zhao

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

© Jun Zhao 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Unit testing plays an essential role in software development and maintenance, especially in Test-Driven Development. Conventional unit tests, which have no input parameters, often exercise similar scenarios with small variations to achieve acceptable coverage, which often results in duplicated code in test suites. Test code duplication hinders comprehension of test cases and maintenance of test suites. Test refactoring is a potential tool for developers to use to control technical debt arising due to test cloning.

In this thesis, we present a novel tool, *JTestParametrizer*, for automatically refactoring method-scope renamed clones in test suites. We propose three levels of refactoring to parameterize type, data, and behaviour differences in clone pairs. Our technique works at the Abstract Syntax Tree level by extracting a parameterized template utility method and instantiating it with appropriate parameter values.

We applied our technique to 5 open-source Java benchmark projects and conducted an empirical study on our results. Our technique examined 14,431 test methods in our benchmark projects and identified 415 renamed clone pairs as effective candidates for refactoring. On average, 65% of the effective candidates (268 clone pairs) in our test suites are refactorable using our technique. All of the refactored test methods are compilable, and 94% of them pass when executed as tests. We believe that our proposed refactorings generally improve code conciseness, reduce the amount of duplication, and make test suites easier to maintain and extend.

Acknowledgements

First, I would like to thank my supervisor, Professor Patrick Lam, for his invaluable advice and guidance. I sincerely appreciate the opportunity to learn from him and his patience throughout my master program. Without his knowledge and expertise which were always available to me, I would not have completed this thesis.

I would like to thank Professor Michael Godfrey and Professor Derek Rayside for reading my thesis and providing valuable feedback during their busy schedules.

In addition, I would like to thank my parents and my girlfriend Yuwei Jiao for their endless love and support during my studies in Waterloo. Thanks to my former colleague Yi Zhuang for his encouragement and help during my application to University of Waterloo. I would also like to thank all my friends at University of Waterloo for their willingness to help out whenever needed.

Finally, I would like to thank my colleagues Jonathan Eyolfson, Stephen Li, and Zeming Liu, for helping me improve my work.

Dedication

I dedicate this thesis to my parents, who have always supported me no matter what.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Unit testing is a widely used technique for software developers to ensure proper code behaviour. Ideally, each unit test verifies a specific function of a code unit, and the entire suite of tests works collectively to maintain the system functionality, in a regression testing application of unit tests; or can even guide development, in a Test-Driven Development application [10].

Traditionally, unit tests are designed to run independently with no input parameters. Thus, many unit tests in a suite exercise similar scenarios with small variations to achieve acceptable coverage in unit testing frameworks. Such self-contained test cases often result in test code duplication, which increases maintenance effort in test suites and error-proneness due to inconsistent updates while maintaining the suites.

Technical debt is a concept that describes the long-term effects of short-term expedients in software development [11]. Technical debt can accrue in test suites as in any other software artifact. Test refactoring can be a powerful tool to address the technical debt in test suites and improve the quality of test code. A number of techniques are available to implement refactoring in test code. Tillman and Schulte have proposed *parameterized unit tests (PUTs)* to improve the expressiveness of unit tests and to enable reusing the common logic behind similar tests [37]. In addition, Saff proposed *Theories* to generalize example-based tests as a lightweight form of specification [35]. Other techniques to refactor test clones involve using different language features such as inheritance or generics.

We believe it is valuable to retrofit existing test cases to reduce code duplication. Thummalapenta et al conducted an empirical study to generalize conventional unit tests to parameterized unit tests manually, and they concluded that retrofitting existing unit tests with parameterization is beneficial to detecting new defects and increasing branch

coverage [36]. Appropriate refactoring on existing test clones can also help make the tests easier to understand and reduce the brittleness of test suites.

Existing clone detection techniques have traditionally been applied to production code. However, our work contributes a novel application of clone detection techniques to test code. Clone detection tools usually report a large number of refactorable candidates that appear to be similar. Manually inspecting and refactoring detected candidates is both time-consuming and error-prone [12]. Automatic refactoring techniques can be a useful tool for developers to control test cloning and maintain their test suites.

Our goal is to help developers more easily refactor test clones. In this thesis, we present a technique to automatically refactor similar test clones by parameterizing clone differences and provide these refactoring suggestions to the developers. Our technique focuses on parameterizing three types of differences (section 3.2) in clone pairs: (1) Type differences; (2) Data differences; and (3) Behavioural differences. Our refactoring strategy involves extracting a parameterized template utility method and instantiating it with appropriate parameter values.

We have therefore implemented a tool, *JTestParametrizer*, which enables developers to automate the test refactoring process, control duplicated test code, and improve the quality of their test suites. We implemented our technique as an Eclipse Plugin application using the Java Development Tooling (JDT) framework. For each clone pair, our tool uses JDeodorant developed by Tsantalis et al [38] to perform AST node mapping and difference detection. It then combines the mapped nodes to construct a combined clone tree, and extracts a parameterized template to unify the type, data, and behavioural differences. Finally, our tool then writes code that passes appropriate parameter values to the extracted template method, thus instantiating specific test cases.

We conducted an empirical study on a collection of 5 Java-based open source benchmark projects. The benchmark test suites contain test code ranging from 14,000 to 55,000 lines, and 14,431 test methods in all. Our technique identified 415 renamed clone pairs in our benchmark test suites as effective candidates for refactoring. On average, our technique refactors 65% of the effective candidates (268 clone pairs). All of the refactored test methods compile, and 94% of them pass when executed as tests. We manually inspected the correctly refactored clone pairs for each benchmark project and found that, although individual tests could be more complicated to understand and specific tests could become more difficult to debug, our refactoring suggestions can make test suites more concise. Appropriate refactoring reduces the amount of duplication and makes the suites easier to maintain and extend.

Our main contributions are:

- a technique to provide refactoring suggestions for test-level clone pairs in existing test suites;
- an implementation of that technique; and
- an empirical study of five benchmark projects using our technique.

Our tool is available under the MIT License at:

https://github.com/yannickzj/JTestParametrizer

Chapter 2

Related Work

In this section, we first present the definitions of clone types that we use in our context. We then discuss refactoring activities in test code. Finally, we discuss recent research work in clone removal refactoring.

2.1 Clone Type Classification

Our technique aims to refactor method-scope renamed clones (*Type II* clones) in a test suite. Such clones may be *Type II* clones (Definition 2) as well as structural clones (Definition 5) where the syntactic bound is a method boundary. In this thesis, we use the terms *method-scope renamed clones* and *renamed clones* interchangeably.

In our context, we adopt the following definitions from Roy and Cordy's work on clone detection [34].

Definition 1 *Type I* clones are identical code fragments except for variations in whitespace (may be also variations in layout) and comments. We also refer to these as *exact clones*. Figure 2.1 presents an example of exact clones. These two code fragments differ by whitespace and comments.

```
// exact clone instance1
if (a >= b) {
  c = d + b; // Comment1
  d = d + 1;
```

```
} else
  c = d - a; // Comment2

// exact clone instance2
if (a>=b) {
  //Comment1'
  c=d+b;
  d=d+1;
} else // Comment2'
  c=d-a;
```

Figure 2.1: Exact clones differ only by whitespace

Definition 2 *Type II* clones are structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments. We also refer to these as *renamed clones*. Figure 2.2 presents an example of renamed clones. The two fragments vary in variable names, literals, and comments, but have the same syntactic structure.

```
// renamed clone instance1
if (a >= b) {
  c = d + b; // Comment1
  d = d + 1;
} else
  c = d - a; // Comment2

// renamed clone instance2: a -> m, b -> n, c -> x, d -> y
if (m >= n) {
  //Comment1'
  x = y + n;
  y = y + 2;
} else // Comment2'
  x = y - m;
```

Figure 2.2: Renamed clones differ by variations in variable names, literals, and comments

Definition 3 *Type III* clones are copied fragments with further modifications. Such clones may not only contain variations in identifiers, literals, types, layout and comments, but also differ by changed, inserted or removed statements. We also refer to these as *gapped*

clones. Figure 2.3 presents an example of gapped clones. The second clone instance differs from the first clone instance with a gap of one inserted statement.

```
// gapped clone instance1
if (a >= b) {
  c = d + b; // Comment1
} else
  c = d - a; // Comment2

// gapped clone instance2
if (a >= b) {
  c = d + b; // Comment1
  e = 5;    // This statement is added
} else
  c = d - a; // Comment2
```

Figure 2.3: Gapped clones differ by one gapped statement

Definition 4 *Type IV* clones are two or more code fragments that perform the same computation but are implemented through different syntactic variants. We also refer to these as *semantic clones*. Figure 2.4 presents an example of semantic clones. Both functions calculate the factorial of their argument `n`, but they are implemented in two different ways. The first clone instance uses iteration, while the second one uses a recursive call.

```
// semantic clone instance1
int factorial (int n) {
  if (n < 0) throw new IllegalArgumentException();
  int result = 1;
  for (int i = 1; i <= n; i++)
    result = result * i;
  return result;
}

// semantic clone instance2
int factorial (int n) {
  if (n < 0) throw new IllegalArgumentException();
  if (n == 0) return 1;
  else return n * factorial(n - 1);
}
```

Figure 2.4: Semantic clones differ by specific implementations (iteration vs recursion)

Definition 5 Structural clones are simple clones (*Type I, II, III*, or *IV*) that lie within a syntactic boundary following the syntactic structure of a particular language. These boundaries can be a function boundary, statement boundary, class boundary, etc. Figure 2.5 presents an example of method-scope structural clones. These clones have method boundaries; they are also renamed clones with variations in variable names, literals, types, and method names.

```
// method-scope structural clone instance1
public void test1() {
  TypeA t1 = new TypeA();
  assertEquals(5, t1.getSomeValue());
}

// method-scope structural clone instance2
public void test2() {
  TypeB t2 = new TypeB();
  assertEquals(10, t2.getOtherValue());
}
```

Figure 2.5: Renamed clones lie within method boundaries

2.2 Test Code Refactoring

Refactoring test code was first proposed by van Deursen et al [40]. They indicated that specific bad smells such as code duplication arose more often in test code than in production code. They also described a set of test refactorings to eliminate these bad smells. Meszaros [32] presented a detailed and systematic explanation of test patterns, including roadmaps for manually refactoring test code. However, neither of the past works proposed techniques to automate test refactoring.

Guerra and Fernandes [16] presented a framework for reasoning about test structure and proposed a tool to perform simple refactorings. They also indicated that test code refactoring should be different from application code refactoring and refactoring in test code needed to keep the same verifications of the application code behaviour. While their technique mainly focused on validating and refactoring the test structure inside a test class, our tool can provide refactoring suggestions for developers to remove duplicated code in test suites.

Similarly, Estefo [14] proposed a tool named TestSurgeon to help software engineers

restructure unit tests. Their approach profiles the execution information of unit tests and defines a metric based on execution information to measure the similarity between test methods. However, TestSurgeon does not refactor test code. Our technique automatically refactors similar test clones and directly applies refactoring changes to source code.

2.3 Clone Removal Refactoring

Clone management plays an important role in software maintenance and evolution. While some studies identified positive aspects of code clones in the software development life cycle [23, 26, 15, 27], code duplication is often overall harmful for software system, resulting in additional maintenance efforts [20, 28, 13, 9, 25, 30, 7, 19].

Extensive refactoring might not always be beneficial or practical [24]. Researchers have proposed techniques to recommend clones for refactoring. Wang and Godfrey [41] proposed a decision tree-based classifier to provide recommendations for refactoring. They collected training data from three open source projects (646 clone instances) to reflect the benefit, cost, and risk of clone refactoring and trained their model based on this refactoring history data. Mondal et al [29, 33] defined a clone change pattern, Similarity Preserving Change Pattern (SPCP), to mine software evolution history. They proposed a technique to identify clones that follow this pattern as important candidates for refactoring. While these techniques recommend opportunities for clone refactoring, they do not refactor the duplicated code. These techniques also focus on non-test code.

Recent research has also investigated automatic clone refactoring. Tsantalis et al [39] proposed a technique to investigate the applicability of lambda expressions for clone refactoring with behavioural differences. They conducted an empirical study to refactor a large dataset of *Type II* and *Type III* clones, and found that lambda expressions are a useful tool to increase the applicability of behavioural parameterization. Lambda expressions are not the only language feature that developers can use to parameterize behavioural differences between clones. Our tool introduces three parameterization techniques to parameterize type, data, and behavioural differences. Because we focus on method-scope renamed clones in test suites, our technique provides more specific refactoring suggestions than their results.

Tsantalis et al [38] propose a tool, JDeodorant, to analyze clone pairs and investigate whether the differences between clones can be safely parameterized. Their technique aims for general code; our focus on test code is novel. Their analysis involves nesting structure matching, statement mapping, and precondition examination between input pairs of code

fragments. In our implementation, we use JDeodorant as the clone analysis tool to perform statement mapping and code difference detection on test code.

Meng et al [31] designed and implemented an automated refactoring tool for clone removal based on systematic editing (i.e., making similar changes to multiple locations). Their technique uses six refactoring operations to extract common code and parameterize differences in types, methods, variables, and expressions. Although systematic editing indicates opportunities to remove code redundancy, relying on systematic edits could limit the refactoring scope for clone removal. Our approach takes the results of clone detection as input, which could be more flexible for developers to use. In addition, their tool has limited support for parameterizing type variations, and cannot handle refactor when any parameterized type (e.g., `T`) instantiates new objects (e.g., `new T()`) or retrieves the type literal (e.g., `T.class`). Our technique can handle these cases and produce readable code.

Juillerat and Hirsbrunner [21] proposed an algorithm to refactor method-scope clones using the *Form Template Method* pattern. Their transformation was only semi-automated. Hotta et al [17] followed up with a technique to assist applying the *Form Template Method* refactoring. Their approach handles gapped clones with the program dependence graph and provides refactoring suggestions with a GUI. However, their refactoring technique relies on applying the *Form Template Method* design pattern, which requires a common abstract superclass to store the extracted template method. Our technique can provide a broader scope to refactor clones without extra levels of inheritance.

Balazinska et al [8] propose a refactoring approach to redesign clone methods using the strategy design pattern. Their transformation factorizes the commonalities of clones and parameterizes their differences. However, applying the strategy design pattern to the redesign process introduces multiple levels of indirection to mask clone differences, which significantly increases the size of the system. Moreover, their refactoring process becomes more complicated for parameterizing the differences in local variable types. In our approach, we use fewer levels of indirection to parameterize clone differences. Also, our technique can easily handle type differences by introducing generic methods.

Chapter 3

Motivating Examples

Our work proposes and implements refactoring suggestions for test pairs that are method-scope renamed clones. Such clones are exact method clones except for variations in identifiers, literals, and types [34].

In this section, we use three clone pairs from our benchmarks as examples to illustrate the typical patterns that we found in the test suites and how our technique refactors these clone pairs. Using these examples, we categorize the parameterizable differences observed in our work.

3.1 Typical Clones

Example 1

Figure 3.1 presents a pair of renamed clones found in the test suite of the Joda-Time project (version 2.10). This clone pair contains variables with different types (highlighted in pink), declaring variables of types `MutableDateTime` and `Instant` respectively. After instantiating variables with specific types, both of the methods then test the date and time fields of the variables. Therefore, both test cases follow the same logic. The main difference between the methods of this clone pair is the type difference in the variable declaration and instantiation, which can be parameterized using generic methods.

```
// test method in org.joda.time.TestMutableDateTime_Basics
public void testGet_DateTimeField() {
  MutableDateTime test = new MutableDateTime();
  assertEquals(1, test.get(ISOChronology.getInstance().era()));
  assertEquals(20, test.get(ISOChronology.getInstance().centuryOfEra()));
  assertEquals(2, test.get(ISOChronology.getInstance().yearOfCentury()));
  assertEquals(2002, test.get(ISOChronology.getInstance().yearOfEra()));
  assertEquals(2002, test.get(ISOChronology.getInstance().year()));
  assertEquals(6, test.get(ISOChronology.getInstance().monthOfYear()));
  assertEquals(9, test.get(ISOChronology.getInstance().dayOfMonth()));
  assertEquals(2002, test.get(ISOChronology.getInstance().weekyear()));
  assertEquals(23, test.get(ISOChronology.getInstance().weekOfWeekyear()));
  assertEquals(7, test.get(ISOChronology.getInstance().dayOfWeek()));
  assertEquals(160, test.get(ISOChronology.getInstance().dayOfYear()));
  assertEquals(0, test.get(ISOChronology.getInstance().halfdayOfDay()));
  assertEquals(1, test.get(ISOChronology.getInstance().hourOfHalfday()));
  assertEquals(1, test.get(ISOChronology.getInstance().clockhourOfDay()));
  assertEquals(1, test.get(ISOChronology.getInstance().clockhourOfHalfday()));
  assertEquals(1, test.get(ISOChronology.getInstance().hourOfDay()));
  assertEquals(0, test.get(ISOChronology.getInstance().minuteOfHour()));
  assertEquals(60, test.get(ISOChronology.getInstance().minuteOfDay()));
  assertEquals(0, test.get(ISOChronology.getInstance().secondOfMinute()));
  assertEquals(60 * 60, test.get(ISOChronology.getInstance().secondOfDay()));
  assertEquals(0, test.get(ISOChronology.getInstance().millisOfSecond()));
  assertEquals(60 * 60 * 1000,
      test.get(ISOChronology.getInstance().millisOfDay()));
  try {
    test.get((DateTimeField) null);
    fail();
  } catch (IllegalArgumentException ex) {}
}

// test method in org.joda.time.TestInstant_Basics
public void testGet_DateTimeField() {
  Instant test = new Instant();
  assertEquals(1, test.get(ISOChronology.getInstance().era()));
  assertEquals(20, test.get(ISOChronology.getInstance().centuryOfEra()));
  assertEquals(2, test.get(ISOChronology.getInstance().yearOfCentury()));
  assertEquals(2002, test.get(ISOChronology.getInstance().yearOfEra()));
```

```
  assertEquals(2002, test.get(ISOChronology.getInstance().year()));
  assertEquals(6, test.get(ISOChronology.getInstance().monthOfYear()));
  assertEquals(9, test.get(ISOChronology.getInstance().dayOfMonth()));
  assertEquals(2002, test.get(ISOChronology.getInstance().weekyear()));
  assertEquals(23, test.get(ISOChronology.getInstance().weekOfWeekyear()));
  assertEquals(7, test.get(ISOChronology.getInstance().dayOfWeek()));
  assertEquals(160, test.get(ISOChronology.getInstance().dayOfYear()));
  assertEquals(0, test.get(ISOChronology.getInstance().halfdayOfDay()));
  assertEquals(1, test.get(ISOChronology.getInstance().hourOfHalfday()));
  assertEquals(1, test.get(ISOChronology.getInstance().clockhourOfDay()));
  assertEquals(1, test.get(ISOChronology.getInstance().clockhourOfHalfday()));
  assertEquals(1, test.get(ISOChronology.getInstance().hourOfDay()));
  assertEquals(0, test.get(ISOChronology.getInstance().minuteOfHour()));
  assertEquals(60, test.get(ISOChronology.getInstance().minuteOfDay()));
  assertEquals(0, test.get(ISOChronology.getInstance().secondOfMinute()));
  assertEquals(60 * 60, test.get(ISOChronology.getInstance().secondOfDay()));
  assertEquals(0, test.get(ISOChronology.getInstance().millisOfSecond()));
  assertEquals(60 * 60 * 1000,
      test.get(ISOChronology.getInstance().millisOfDay()));
  try {
    test.get((DateTimeField) null);
    fail();
  } catch (IllegalArgumentException ex) {}
}
```

Figure 3.1: `MutableDateTime` and `Instant` tests differ by types of the variable `test`

Figure 3.2 shows our refactoring suggestion. To reuse the common logic behind the clone pair in Figure 3.1, we propose abstracting the types `MutableDateTime` and `Instant` as type parameters in the generic method `testBasicsTestGet_DateTimeFieldTemplate`. More specifically, the type `MutableDateTime` and `Instant` are subclasses of the type `AbstractInstant`, and we can declare generic type `TInstant` constrained to extend the type `AbstractInstant` to represent types `MutableDateTime` and `Instant`. The generic method serves as a test utility method, which can be reused in the body of test cases by passing in different concrete type parameters.

```
// parameterized template in
    org.joda.time.TestBasicsTestGet_DateTimeFieldTemplate
public static <TInstant extends AbstractInstant>
  void testBasicsTestGet_DateTimeFieldTemplate(Class<TInstant> clazzTInstant)
      throws Exception {
  TInstant test = clazzTInstant.newInstance();
  assertEquals(1, test.get(ISOChronology.getInstance().era()));
  assertEquals(20, test.get(ISOChronology.getInstance().centuryOfEra()));
  assertEquals(2, test.get(ISOChronology.getInstance().yearOfCentury()));
  assertEquals(2002, test.get(ISOChronology.getInstance().yearOfEra()));
  assertEquals(2002, test.get(ISOChronology.getInstance().year()));
  assertEquals(6, test.get(ISOChronology.getInstance().monthOfYear()));
  assertEquals(9, test.get(ISOChronology.getInstance().dayOfMonth()));
  assertEquals(2002, test.get(ISOChronology.getInstance().weekyear()));
  assertEquals(23, test.get(ISOChronology.getInstance().weekOfWeekyear()));
  assertEquals(7, test.get(ISOChronology.getInstance().dayOfWeek()));
  assertEquals(160, test.get(ISOChronology.getInstance().dayOfYear()));
  assertEquals(0, test.get(ISOChronology.getInstance().halfdayOfDay()));
  assertEquals(1, test.get(ISOChronology.getInstance().hourOfHalfday()));
  assertEquals(1, test.get(ISOChronology.getInstance().clockhourOfDay()));
  assertEquals(1, test.get(ISOChronology.getInstance().clockhourOfHalfday()));
  assertEquals(1, test.get(ISOChronology.getInstance().hourOfDay()));
  assertEquals(0, test.get(ISOChronology.getInstance().minuteOfHour()));
  assertEquals(60, test.get(ISOChronology.getInstance().minuteOfDay()));
  assertEquals(0, test.get(ISOChronology.getInstance().secondOfMinute()));
  assertEquals(60 * 60, test.get(ISOChronology.getInstance().secondOfDay()));
  assertEquals(0, test.get(ISOChronology.getInstance().millisOfSecond()));
  assertEquals(60 * 60 * 1000,
      test.get(ISOChronology.getInstance().millisOfDay()));
  try {
    test.get((DateTimeField) null);
    fail();
  } catch (IllegalArgumentException ex) {}
}
// refactored test method in org.joda.time.TestMutableDateTime_Basics
public void testGet_DateTimeField() throws Exception {
  TestBasicsTestGet_DateTimeFieldTemplate
    .testBasicsTestGet_DateTimeFieldTemplate( MutableDateTime .class);
}
```

```
// refactored test method in org.joda.time.TestInstant_Basics
public void testGet_DateTimeField() throws Exception {
  TestBasicsTestGet_DateTimeFieldTemplate
    .testBasicsTestGet_DateTimeFieldTemplate( Instant .class);
}
```

Figure 3.2: Using generic types to abstract the types `MutableDateTime` and `Instant`

Example 2

Figure 3.3 shows another duplication pattern. The only difference in this clone pair lies in the argument values passed to the `DateTickUnit` constructor (highlighted in purple), which is `1` and `10` respectively. The parameterized template method can be extracted by abstracting the different integers as a method argument.

```
// test method in org.jfree.chart.axis.junit.DateAxisTests
public void testPreviousStandardDateMillisecondA() {
  MyDateAxis axis = new MyDateAxis("Millisecond");
  Millisecond m0 = new Millisecond(458, 58, 31, 12, 1, 4, 2007);
  Millisecond m1 = new Millisecond(459, 58, 31, 12, 1, 4, 2007);

  Date d0 = new Date(m0.getFirstMillisecond());
  Date end = new Date(m1.getLastMillisecond());

  DateTickUnit unit = new DateTickUnit(DateTickUnit.MILLISECOND, 1 );
  axis.setTickUnit(unit);

  // START: check d0
  axis.setTickMarkPosition(DateTickMarkPosition.START);

  axis.setRange(d0, end);
  Date psd = axis.previousStandardDate(d0, unit);
  Date nsd = unit.addToDate(psd);
  assertTrue(psd.getTime() < d0.getTime());
  assertTrue(nsd.getTime() >= d0.getTime());

  // MIDDLE: check d0
  axis.setTickMarkPosition(DateTickMarkPosition.MIDDLE);
```

```
  axis.setRange(d0, end);
  psd = axis.previousStandardDate(d0, unit);
  nsd = unit.addToDate(psd);
  assertTrue(psd.getTime() < d0.getTime());
  assertTrue(nsd.getTime() >= d0.getTime());

  // END: check d0
  axis.setTickMarkPosition(DateTickMarkPosition.END);

  axis.setRange(d0, end);
  psd = axis.previousStandardDate(d0, unit);
  nsd = unit.addToDate(psd);
  assertTrue(psd.getTime() < d0.getTime());
  assertTrue(nsd.getTime() >= d0.getTime());
}

// test method in org.jfree.chart.axis.junit.DateAxisTests
public void testPreviousStandardDateMillisecondB() {
  MyDateAxis axis = new MyDateAxis("Millisecond");
  Millisecond m0 = new Millisecond(458, 58, 31, 12, 1, 4, 2007);
  Millisecond m1 = new Millisecond(459, 58, 31, 12, 1, 4, 2007);

  Date d0 = new Date(m0.getFirstMillisecond());
  Date end = new Date(m1.getLastMillisecond());

  DateTickUnit unit = new DateTickUnit(DateTickUnit.MILLISECOND, 10);
  axis.setTickUnit(unit);

  // START: check d0
  axis.setTickMarkPosition(DateTickMarkPosition.START);

  axis.setRange(d0, end);
  Date psd = axis.previousStandardDate(d0, unit);
  Date nsd = unit.addToDate(psd);
  assertTrue(psd.getTime() < d0.getTime());
  assertTrue(nsd.getTime() >= d0.getTime());

  // MIDDLE: check d0
  axis.setTickMarkPosition(DateTickMarkPosition.MIDDLE);
```

```
  axis.setRange(d0, end);
  psd = axis.previousStandardDate(d0, unit);
  nsd = unit.addToDate(psd);
  assertTrue(psd.getTime() < d0.getTime());
  assertTrue(nsd.getTime() >= d0.getTime());

  // END: check d0
  axis.setTickMarkPosition(DateTickMarkPosition.END);

  axis.setRange(d0, end);
  psd = axis.previousStandardDate(d0, unit);
  nsd = unit.addToDate(psd);
  assertTrue(psd.getTime() < d0.getTime());
  assertTrue(nsd.getTime() >= d0.getTime());
}
```

Figure 3.3: Two `DateAxis` tests differ by argument values in the `DateTickUnit` constructor call

Figure 3.4 presents the results of our technique for the clone pair from Figure 3.3, showing a case of parameterized unit tests [37]. Parameterizing the data difference makes the refactored test cases more concise, effectively saving 21 lines of code from the original clone pair with 54 lines. Refactoring also helps facilitate data-driven testing and reduces the technical debt of the test suite maintenance when more tests with different data are needed.

```
// parameterized template in org.jfree.chart.axis.junit.DateAxisTests
public void dateAxisTestsTestPreviousStandardDateMillisecondTemplate(int i1) {
  MyDateAxis axis = new MyDateAxis("Millisecond");
  Millisecond m0 = new Millisecond(458, 58, 31, 12, 1, 4, 2007);
  Millisecond m1 = new Millisecond(459, 58, 31, 12, 1, 4, 2007);
  Date d0 = new Date(m0.getFirstMillisecond());
  Date end = new Date(m1.getLastMillisecond());
  DateTickUnit unit = new DateTickUnit(DateTickUnit.MILLISECOND, i1);
  axis.setTickUnit(unit);
  axis.setTickMarkPosition(DateTickMarkPosition.START);
  axis.setRange(d0, end);
  Date psd = axis.previousStandardDate(d0, unit);
  Date nsd = unit.addToDate(psd);
  assertTrue(psd.getTime() < d0.getTime());
```

```
  assertTrue(nsd.getTime() >= d0.getTime());
  axis.setTickMarkPosition(DateTickMarkPosition.MIDDLE);
  axis.setRange(d0, end);
  psd = axis.previousStandardDate(d0, unit);
  nsd = unit.addToDate(psd);
  assertTrue(psd.getTime() < d0.getTime());
  assertTrue(nsd.getTime() >= d0.getTime());
  axis.setTickMarkPosition(DateTickMarkPosition.END);
  axis.setRange(d0, end);
  psd = axis.previousStandardDate(d0, unit);
  nsd = unit.addToDate(psd);
  assertTrue(psd.getTime() < d0.getTime());
  assertTrue(nsd.getTime() >= d0.getTime());
}

// refactored test method in org.jfree.chart.axis.junit.DateAxisTests
public void testPreviousStandardDateMillisecondA() {
  this.dateAxisTestsTestPreviousStandardDateMillisecondTemplate( 1 );
}

// refactored test method in org.jfree.chart.axis.junit.DateAxisTests
public void testPreviousStandardDateMillisecondB() {
  this.dateAxisTestsTestPreviousStandardDateMillisecondTemplate( 10 );
}
```

Figure 3.4: Different integer literal values are parameterized as method arguments

Example 3

Finally, Figure 3.5 not only contains type and data differences which are similar to the previous examples (highlighted in pink and purple respectively) but also introduces a new type of difference in object method invocations (highlighted in orange). This time, the clone tests still have the same structure, but they call methods with different names (i.e., `setCap` and `setDialFrame`). These different method calls can possibly be invoked on receiver objects of different types (i.e., `DialCap` and `ArcDialFrame`). We would expect methods with different names to have different behaviours. However, we can still refactor these tests as they have the same structure.

```
// test method in org.jfree.chart.plot.dial.junit.DialPlotTests
public void testCapListener() {
  DialPlot p = new DialPlot();
  DialCap c1 = new DialCap();
  p.setCap(c1);
  p.addChangeListener(this);
  this.lastEvent = null;
  c1.setFillPaint(Color.red);
  assertNotNull(this.lastEvent);
  DialCap c2 = new DialCap();
  p.setCap(c2);
  this.lastEvent = null;
  c1.setFillPaint(Color.blue);
  assertNull(this.lastEvent);
  c2.setFillPaint(Color.green);
  assertNotNull(this.lastEvent);
}
// test method in org.jfree.chart.plot.dial.junit.DialPlotTests
public void testFrameListener() {
  DialPlot p = new DialPlot();
  ArcDialFrame f1 = new ArcDialFrame();
  p.setDialFrame(f1);
  p.addChangeListener(this);
  this.lastEvent = null;
  f1.setBackgroundPaint(Color.gray);
  assertNotNull(this.lastEvent);
  ArcDialFrame f2 = new ArcDialFrame();
  p.setDialFrame(f2);
  this.lastEvent = null;
  f1.setBackgroundPaint(Color.blue);
  assertNull(this.lastEvent);
  f2.setBackgroundPaint(Color.green);
  assertNotNull(this.lastEvent);
}
```

Figure 3.5: Methods `testCapListener`, `testFrameListener` have similar structure but instantiate different methods (`DialCap` vs `ArcDialFrame`), call different methods (`setCap` vs `setDialFrame`), and use different method parameters (`Color.red` vs `Color.gray`)

Figure 3.6 shows our suggested refactoring for the code pair in Figure 3.5. The basic idea is to extract the common interface, `DialPlotTestsTestListenerAdapter`, based on the behavioural differences, then apply different implementations of the common interface to individual test cases. Similar to the parameterization of data differences, we apply the behavioural differences through the template method argument by passing specific interface implementation objects.

```
// parameterized template in org.jfree.chart.plot.dial.junit.DialPlotTests
public <TDial extends AbstractDialLayer> void dialPlotTestsTestListenerTemplate(
    DialPlotTestsTestListenerAdapter<TDial> adapter, Class<TDial> clazzTDial,
        Color color1) throws Exception {
  DialPlot p = new DialPlot();
  TDial v1 = clazzTDial.newInstance();
  adapter.set(p, v1);
  p.addChangeListener(this);
  this.lastEvent = null;
  adapter.setPaint(v1, color1);
  assertNotNull(this.lastEvent);
  TDial v2 = clazzTDial.newInstance();
  adapter.set(p, v2);
  this.lastEvent = null;
  adapter.setPaint(v1, Color.blue);
  assertNull(this.lastEvent);
  adapter.setPaint(v2, Color.green);
  assertNotNull(this.lastEvent);
}

// common behavioural interface
interface DialPlotTestsTestListenerAdapter<TDial> {
  void set(DialPlot dialPlot1, TDial tDial1);
  void setPaint(TDial tDial1, Color color1);
}

// behavioural implementation for testCapListener()
class DialPlotTestsTestCapListenerAdapterImpl implements
    DialPlotTestsTestListenerAdapter< DialCap > {
  public void set(DialPlot p, DialCap c1) {
    p. setCap (c1);
  }
```

```
  public void setPaint(DialCap v1, Color color1) {
    v1.setFillPaint(color1);
  }
}

// behavioural implementation for testFrameListener()
class DialPlotTestsTestFrameListenerAdapterImpl implements
    DialPlotTestsTestListenerAdapter<ArcDialFrame> {
  public void set(DialPlot p, ArcDialFrame f1) {
    p.setDialFrame(f1);
  }
  public void setPaint(ArcDialFrame f1, Color color1) {
    f1.setBackgroundPaint(color1);
  }
}

// refactored test method in org.jfree.chart.plot.dial.junit.DialPlotTests
public void testCapListener() throws Exception {
  this.dialPlotTestsTestListenerTemplate(new
      DialPlotTestsTestCapListenerAdapterImpl(), DialCap.class, Color.red);
}

// refactored test method in org.jfree.chart.plot.dial.junit.DialPlotTests
public void testFrameListener() throws Exception {
  this.dialPlotTestsTestListenerTemplate(new
      DialPlotTestsTestFrameListenerAdapterImpl(), ArcDialFrame.class,
      Color.gray);
}
```

Figure 3.6: Test utility method `dialPlotTestsTestListenerTemplate` parameterizes behavioural differences in method calls between methods `testCapListener` and `testFrameListener` from Figure 3.5

3.2 Parameterizable Difference Categorization

Based on the above motivating examples and our observations during refactoring, we identify three basic types of parameterizable differences:

1. Type difference

   Type differences include tests that differ in the classes that they declare or instantiate. Note that the classes of different types are not necessarily subclasses of a common superclass except for the class `Object`. We will describe our parameterization technique for type differences in section 4.2.1.

   As shown in Figure 3.1, the matching variables `test` are declared and instantiated with different types, `MutableDateTime` and `Instant`, respectively. Also, classes of different types in Figure 3.5, `DialCap` and `ArcDialFrame`, are instantiated and assigned to variables of corresponding type.

2. Data difference

   This group mainly refers to differences in literal values or variables of different data types. We explain more details about how we parameterize data difference in section 4.2.2. Figure 3.3 presents an example of data difference in integer literal values. In our refactoring work, we observed many instances of this difference type in test cases, especially in some data-intensive test suites such as the *Joda-Time* benchmark.

3. Behavioural difference

   Method calls are key actions in Object-Oriented Programming. Behavioural differences include tests that differ in the identity of the methods that they invoke. Java method invocations include a receiver expression, method name, and actual parameter values:

   ```
   <receiver expr>.<method name>(<actual parameter values>)
   ```

   For this type of difference, we consider receiver expressions and method names. Differences in actual parameters can be detected as data differences. Section 4.2.3 will present more details about how we parameterize behavioural differences.

   As discussed in Figure 3.5, the method pair of `p.setCap` and `p.setDialFrame` only has a behavioural difference in its method name identifier, while the method pair of `c1.setFillPaint` and `f1.setBackGroundPaint` contains differences in both receiver expression and method name.

Our observations show that many differences in parameterized clones fit one of the three above types. Our refactoring technique focuses on automatically parameterizing these three types of differences in clone pairs. Other difference types are out of scope for this work. Additionally, these basic parameterizable differences can appear individually or

together in a nested way. We apply the parameterization techniques in our approach in a depth-first search (DFS) way to refactor the abstract syntax tree structure of the clone pair, which we will describe in the next section.

Chapter 4

Approach

Our tool, *JTestParametrizer*, refactors renamed clones in test cases and provides these refactorings as suggestions to the developers.

In this section, we first introduce the tools and concepts that we use in our approach. Then, we present three parameterization techniques that we developed for refactoring test cases and describe more implementation details about our approach. Finally, we discuss the limitations of our refactoring technique.

4.1 Tools and Concepts

Abstract Syntax Trees (AST) are tree-based data structures widely used in compilers to represent the structure of source code in a programming language [6]. These trees represent text-based source code in a tree form. Each node of the tree denotes a syntactic element in the source code. AST nodes can represent literal values, variable declarations, expressions, statements, etc. ASTs are a convenient and powerful tool for programmatically analyzing and modifying source code [1].

In our implementation, we use the Java Development Tooling (JDT) API [4] on the Eclipse platform for parsing source code to ASTs. The JDT allows users to write, compile, test, debug, and edit programs written in the Java programming language. The Eclipse JDT API provides a large hierarchical organization of AST node types including different types of nodes such as *Expression*, *Statement*, *Type*, *Name*, *BodyDeclaration*, etc. Each type may contain dozens of concrete AST nodes. For instance, the *Statement* group has 22 kinds of statement nodes, and the *Expression* group contains up to 32 kinds of expression

nodes. To perform appropriate operations over such a complex object structure, our implementation applies the *visitor pattern*, which allows flexibly defining specific behaviour based on the AST node type.

Additionally, we use Deckard [18] to detect potential candidates for refactoring. Deckard is a tree-based clone detection tool which is applicable to any language with a formally specified grammar [18]. It reports clone candidates, which serve as raw inputs to our refactoring system. We then use JDeodorant [38] as a library in our implementation to perform clone analysis, which involves mapping corresponding AST nodes and identifying the differences in clone pairs. JDeodorant is an Eclipse plugin that supports identifying and resolving duplicated code smells in Java software [3]. Unlike Deckard, it does not detect clone candidates, but focuses on clone analysis and refactoring with given clone candidates. In our implementation, we use the analysis results from JDeodorant for further refactoring.

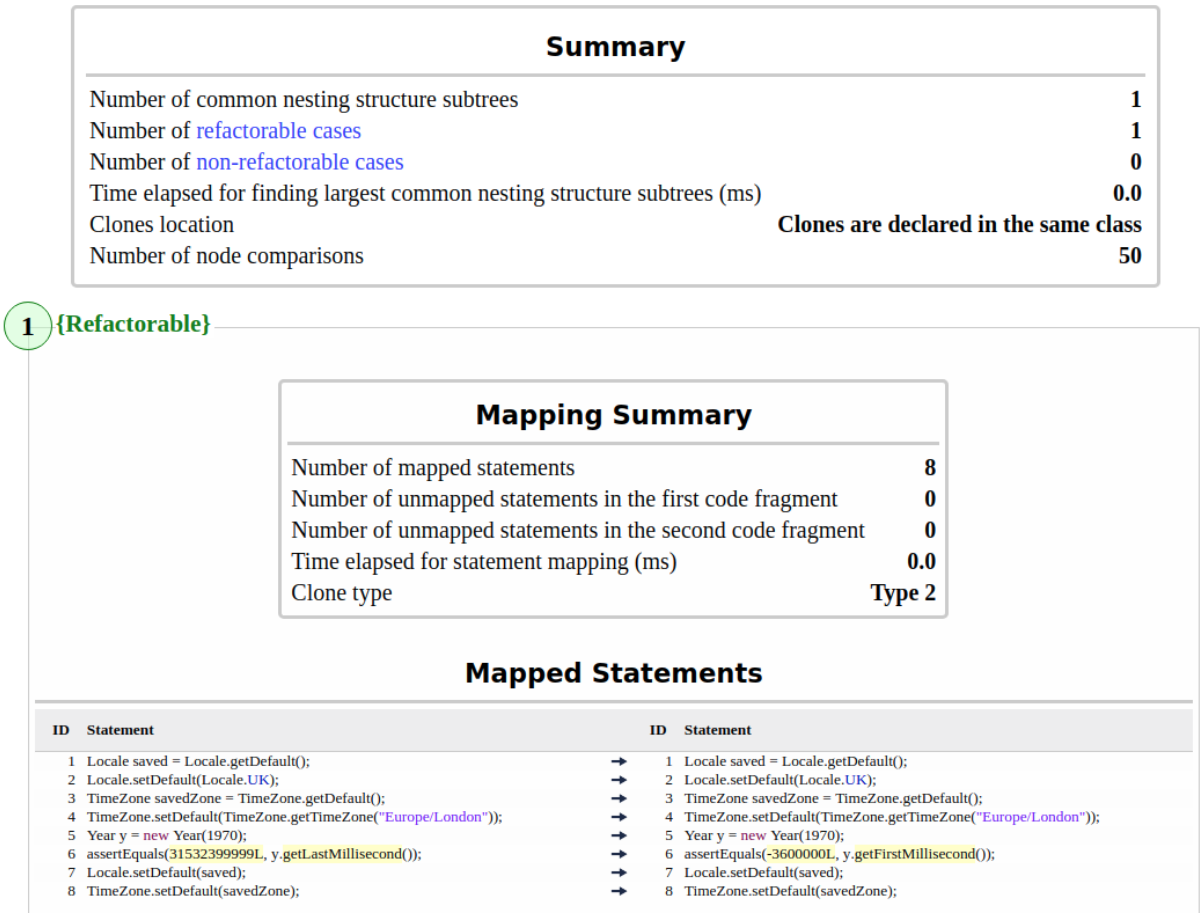
Figure 4.1 shows a pair of renamed clones detected by Deckard in the test suite of the JFreeChart project (version 1.0.10). Manual inspection confirms that a data difference (highlighted in purple) and a behaviour difference (highlighted in orange) exist in the argument lists of the `assertEquals` method calls. Figure 4.2 presents the analysis results from JDeodorant. JDeodorant can map statements between clone instances and identify the differences in the arguments of the `assertEquals` method call. Our technique uses this analysis information to provide refactoring suggestions.

```
// test method in org.jfree.data.time.junit.YearTests
public void testGetLastMillisecond() {
  Locale saved = Locale.getDefault();
  Locale.setDefault(Locale.UK);
  TimeZone savedZone = TimeZone.getDefault();
  TimeZone.setDefault(TimeZone.getTimeZone("Europe/London"));
  Year y = new Year(1970);
  assertEquals( 31532399999L , y. getLastMillisecond ());
  Locale.setDefault(saved);
  TimeZone.setDefault(savedZone);
}

// test method in org.jfree.data.time.junit.YearTests
public void testGetFirstMillisecond() {
  Locale saved = Locale.getDefault();
  Locale.setDefault(Locale.UK);
  TimeZone savedZone = TimeZone.getDefault();
  TimeZone.setDefault(TimeZone.getTimeZone("Europe/London"));
```

```
  Year y = new Year(1970);
  assertEquals( -3600000L , y. getFirstMillisecond ());
  Locale.setDefault(saved);
  TimeZone.setDefault(savedZone);
}
```

Figure 4.1: Deckard detects `testGetLastMillisecond` and `testGetFirstMillisecond` as a potential refactorable clone pair



Summary

Number of common nesting structure subtrees	**1**
Number of refactorable cases	**1**
Number of non-refactorable cases	**0**
Time elapsed for finding largest common nesting structure subtrees (ms)	**0.0**
Clones location	**Clones are declared in the same class**
Number of node comparisons	**50**

1 **{Refactorable}**

Mapping Summary

Number of mapped statements	**8**
Number of unmapped statements in the first code fragment	**0**
Number of unmapped statements in the second code fragment	**0**
Time elapsed for statement mapping (ms)	**0.0**
Clone type	**Type 2**

Mapped Statements

ID	Statement	ID	Statement
1	Locale saved = Locale.getDefault();	1	Locale saved = Locale.getDefault();
2	Locale.setDefault(Locale.UK);	2	Locale.setDefault(Locale.UK);
3	TimeZone savedZone = TimeZone.getDefault();	3	TimeZone savedZone = TimeZone.getDefault();
4	TimeZone.setDefault(TimeZone.getTimeZone("Europe/London"));	4	TimeZone.setDefault(TimeZone.getTimeZone("Europe/London"));
5	Year y = new Year(1970);	5	Year y = new Year(1970);
6	assertEquals(31532399999L, y.getLastMillisecond());	6	assertEquals(-3600000L, y.getFirstMillisecond());
7	Locale.setDefault(saved);	7	Locale.setDefault(saved);
8	TimeZone.setDefault(savedZone);	8	TimeZone.setDefault(savedZone);

Figure 4.2: JDeodorant maps statements between test methods *testGetLastMillisecond* and *testGetFirstMillisecond* and identifies the clone differences

4.2 Parameterization Techniques

Our examples in the previous section motivate three levels of refactoring:

- Type parameterization
- Data parameterization
- Behaviour parameterization

All of our refactoring techniques work at a per-method granularity. Our refactoring strategy first extracts a parameterized template method from the clones, and then instantiates it with the proper parameter values. We apply this strategy at all three levels.

4.2.1 Type Parameterization

A type-parameterizable clone includes entries with different type identifiers. Java includes generics to abstract over types. The basic idea of generics is to allow types, including basic data types such as Integer, String, and user-defined types, to be a parameter to methods, classes, and interfaces. When the classes of different types are subclasses of a common superclass except for the class `Object`, we add the common superclass as the bound of the extracted generic type.

In the abstract syntax tree (AST), differences in type identifiers mainly exist in the following two types of AST nodes:

- Variable declarations

  A variable declaration in Java contains a type identifier and a variable name. For instance, the following statement declares a variable `test` of the type `MutableDateTime`.

  ```
  MutableDateTime test;
  ```

  In renamed clones, paired variables declared with different types result in a type difference. Type differences in this type of AST node can be easily parameterized with generic methods.

  As shown in Figure 3.2, `MutableDateTime` and `Instant` are different paired type identifiers. They are abstracted as the type parameter `TInstant` in the parameterized template, which can be later instantiated with `MutableDateTime` and `Instant` in the test cases.

- Object instantiation

  Java instantiates objects using the `new` operator. The following expression creates an object of the type `MutableDateTime`.

  ```
  new MutableDateTime()
  ```

  When classes of different types are instantiated with constructor calls, there exist type differences in the class instance creation node. To parameterize this type of AST node, we use the generic class, `Class<T>` in the `java.lang` package, to unify the type differences. This generic class provides a `newInstance()` method to create a new instance of its generic type. Furthermore, in Java, writing `.class` after a class name retrieves the `Class` object which represents the given class.

  As shown in Figure 3.6, the argument `clazzTDial` of type `Class<TDial>` in the parameterized template method is used to unify the type difference in `new DialCap()` and `new ArcDialFrame()`. We invoke the `newInstance()` method of `clazzTDial` to create a new class instance of `TDial` type, which parameterizes the `DialCap` and `ArcDialFrame` classes.

4.2.2 Data Parameterization

A data-parameterizable clone has differences in literal values and variables of different types. As shown in Figure 3.4, the main refactoring idea is to replace the differing values with a parameter to unify the clone pair and pass the particular values as arguments to the extracted template method in different test cases. The key to parameterizing data differences is to determine the common compatible type for the pair of differing values. Based on the data types of the value pair to parameterize, we take into account two groups in our refactoring approach:

- Primitive values

  Primitive values include numbers, characters, boolean literals, and variables of primitive types. We use the following implicit casting order for Java number literals in our data parameterization:

  ```
  byte -> short -> int -> long
         float -> double
  ```

The left-side value can be implicitly assigned to any right-side value. When the paired values are of different primitive types, we use a best-effort approach to find the closest assignment compatible common type with implicit casting. Figure 4.3 presents a refactoring example with primitive value casting from the Joda-Time project (version 2.10). When we parameterize the primitive value pair of `int` and `long`, we use the type `long` as the argument type. This widening conversion may not be desirable for test coverage reasons. Developers need to review the test cases to further confirm the refactoring suggestions. In our benchmarks, we observed 22 clone pairs with this type of casting, accounting for 5% of our total effective refactoring candidates.

As a conversion of `int` or `long` to `float`, or of `long` to `double`, may lose precision, it would not be desirable to parameterize value pairs of integer and floating-point numbers. In our approach, we consider that integer values (`byte`, `short`, `int`, and `long`) are incompatible with floating-point values (`float` and `double`). If there is no compatible common type, the clone pair will be marked as non-refactorable.

- Non-primitive values

  Non-primitive values can be any variables of Java library types or user-defined types. If the data pair binds with different non-primitive types, our technique tries to find the nearest common superclass or interface of the data types as the parameter type. When the differing non-primitive types share no common superclass or interface except for the class `Object`, the clone pair will be marked as non-refactorable.

```
// test method1 in org.joda.time.field.TestMillisDurationField
public void test_getMillis_int() {
  assertEquals(0, MillisDurationField.INSTANCE.getMillis(0));
  assertEquals(1234, MillisDurationField.INSTANCE.getMillis(1234));
  assertEquals(-1234, MillisDurationField.INSTANCE.getMillis(-1234));
}

// test method2 in org.joda.time.field.TestMillisDurationField
public void test_getMillis_long() {
  assertEquals(0L, MillisDurationField.INSTANCE.getMillis(0L));
  assertEquals(1234L, MillisDurationField.INSTANCE.getMillis(1234L));
  assertEquals(-1234L, MillisDurationField.INSTANCE.getMillis(-1234L));
}
```

```
// parameterized template in org.joda.time.field.TestMillisDurationField
public void testMillisDurationFieldTest_getTemplate(long l1, long l2, long l3,
    long l4, long l5, long l6) {
  assertEquals(l1, MillisDurationField.INSTANCE.getMillis(l2));
  assertEquals(l3, MillisDurationField.INSTANCE.getMillis(l4));
  assertEquals(l5, MillisDurationField.INSTANCE.getMillis(l6));
}

// refactored test method1 in org.joda.time.field.TestMillisDurationField
public void test_getMillis_int() {
  this.testMillisDurationFieldTest_getTemplate(0, 0, 1234, 1234, -1234, -1234);
}

// refactored test method2 in org.joda.time.field.TestMillisDurationField
public void test_getMillis_long() {
  this.testMillisDurationFieldTest_getTemplate(0L, 0L, 1234L, 1234L, -1234L,
      -1234L);
}
```

Figure 4.3: Refactoring example with primitive value casting from `int` to `long`

4.2.3 Behaviour Parameterization

We parameterize methods with behavioural differences, that is, method invocation calls having different signatures. These method calls must still have the same length argument lists, but may have different method names, or perhaps different receiver objects. In our approach, differences in method argument lists are unified before parameterization of method invocation pairs. After parameterizing the argument lists, the argument lists may then have comparable types. Therefore, the behaviour parameterization focuses on the differences in receiver object and method name of the method invocation node.

To unify different method invocations, our parameterization carries out the following steps:

1. We create a common interface to collect all unified behavioural methods with compatible signatures.
2. We add an argument, `adapter`, to the parameterized template method with the common interface type. This argument serves as the receiver object for the refactored operations.

3. For each pair of different method invocations in the clone pair, we extract an interface method declaration into the common interface. More specifically, we abstract the receiver object as the first parameter in the interface method declaration, then add all the parameterized arguments to the interface method argument list. Note that we can skip extracting the receiver object if the method invocations do not contain a receiver object (static or implicit `this`).

Figure 4.4 revisits the example from Figure 3.6 and shows how we extract a common behavioural interface and its instantiations. We extract a common generic interface, `DialPlotTestsTestListenerAdapter<TDial>`, to parameterize the behavioural differences in the clone pair. This common interface contains two method declarations. The method declaration `set(...)` unifies `setCap(...)` and `setDialFrame(...)`, while the method declaration `setPaint(...)` unifies `setFillPaint(...)` and `setBackgroundPaint(...)`. For naming the unified method declarations, we use a best-effort approach to extract the longest common parts of the pair of original method names. If the original method names have no common parts, we assign unique names of the form *actionN* where *N* is a counter that increments with each generated name.

```
// common behavioural interface
interface DialPlotTestsTestListenerAdapter<TDial> {
  void set(DialPlot dialPlot1, TDial tDial1);
  void setPaint(TDial tDial1, Color color1);
}

// behavioural implementation for testCapListener()
class DialPlotTestsTestCapListenerAdapterImpl implements
    DialPlotTestsTestListenerAdapter< DialCap > {
  public void set(DialPlot p, DialCap c1) {
    p. setCap (c1);
  }
  public void setPaint( DialCap v1, Color color1) {
    v1. setFillPaint (color1);
  }
}
```

```
// behavioural implementation for testFrameListener()
class DialPlotTestsTestFrameListenerAdapterImpl implements
    DialPlotTestsTestListenerAdapter< ArcDialFrame > {
  public void set(DialPlot p, ArcDialFrame f1) {
    p. setDialFrame (f1);
  }
  public void setPaint( ArcDialFrame f1, Color color1) {
    f1. setBackgroundPaint (color1);
  }
}
```

Figure 4.4: Common behavioural interface `DialPlotTestsTestListenerAdapter<TDial>` and its implementations in Figure 3.6

After unification of different behaviour, we define specific behavioural operations by implementing each declared method in the common interface. Finally, we pass different behavioural implementation variables of the common interface as arguments in the parameterized template method to perform appropriate behavioural operations. As shown in Figure 4.4, both `DialPlotTestsTestCapListenerAdapterImpl` and `DialPlotTestsTestFrameListenerAdapterImpl` implement the same interface, `DialPlotTestsTestListenerAdapter<TDial>`, with a set of different behavioural definitions. For instance, in the implementation of the interface method, `void set(DialPlot, TDial)`, the class `DialPlotTestsTestCapListenerAdapterImpl` invokes the method `setCap`, while the class `DialPlotTestsTestFrameListenerAdapterImpl` calls the method `setDialFrame`.

4.3 Refactoring Roadmap

Our refactoring tool, `JTestParametrizer`, is implemented as an Eclipse Plugin application, which batch processes all clone pairs in the input file and directly applies refactoring changes to the source files.

4.3.1 Input and Output

Our refactoring tool is designed to process a pair of test methods which are considered to be duplicates. We expect these method-scope clones to be reported by a clone detection tool like Deckard. The input of our refactoring technique is therefore the output of clone

detection along with a program to refactor. The user can also specify custom clone pairs in the input file.

Each clone pair has two records about the clone instances. Each clone instance record includes the following information:

- Clone group ID, an integer indicating the ID of the clone group to which the current clone instance belongs;
- Source folder, indicating the root path of the source code;
- Package name (e.g. `com.google.gson`);
- Class name;
- Method signature;
- Start and end line location.

Currently, we only refactor clone pairs in the same package. Clone pairs located in different packages are not in scope for our refactoring work. Conceptually, it is not difficult to apply our technique to clone pairs across different packages. However, we believe clone pairs in different packages are less related to each other than those in the same package. We thus choose to focus on refactoring clone pairs in the same package.

Our tool directly applies the refactoring changes to the targeted source files. If the clone pair is located in the same test class, our tool adds a parameterized template method, the common behavioural interface, and its implementation classes to the same class. When the pair of test methods are in different test classes, we create a utility class to store the parameterized template method and the common behavioural interface, while we put the implementation classes of the common interface as inner classes in their corresponding test classes. Additionally, all targeted test methods are modified in their source file.

4.3.2 Workflow

Figure 4.5 presents the basic workflow of our technique. From a high-level perspective, the workflow consists of three major processes, and our refactoring technique fits in the *Clone Refactoring* step.

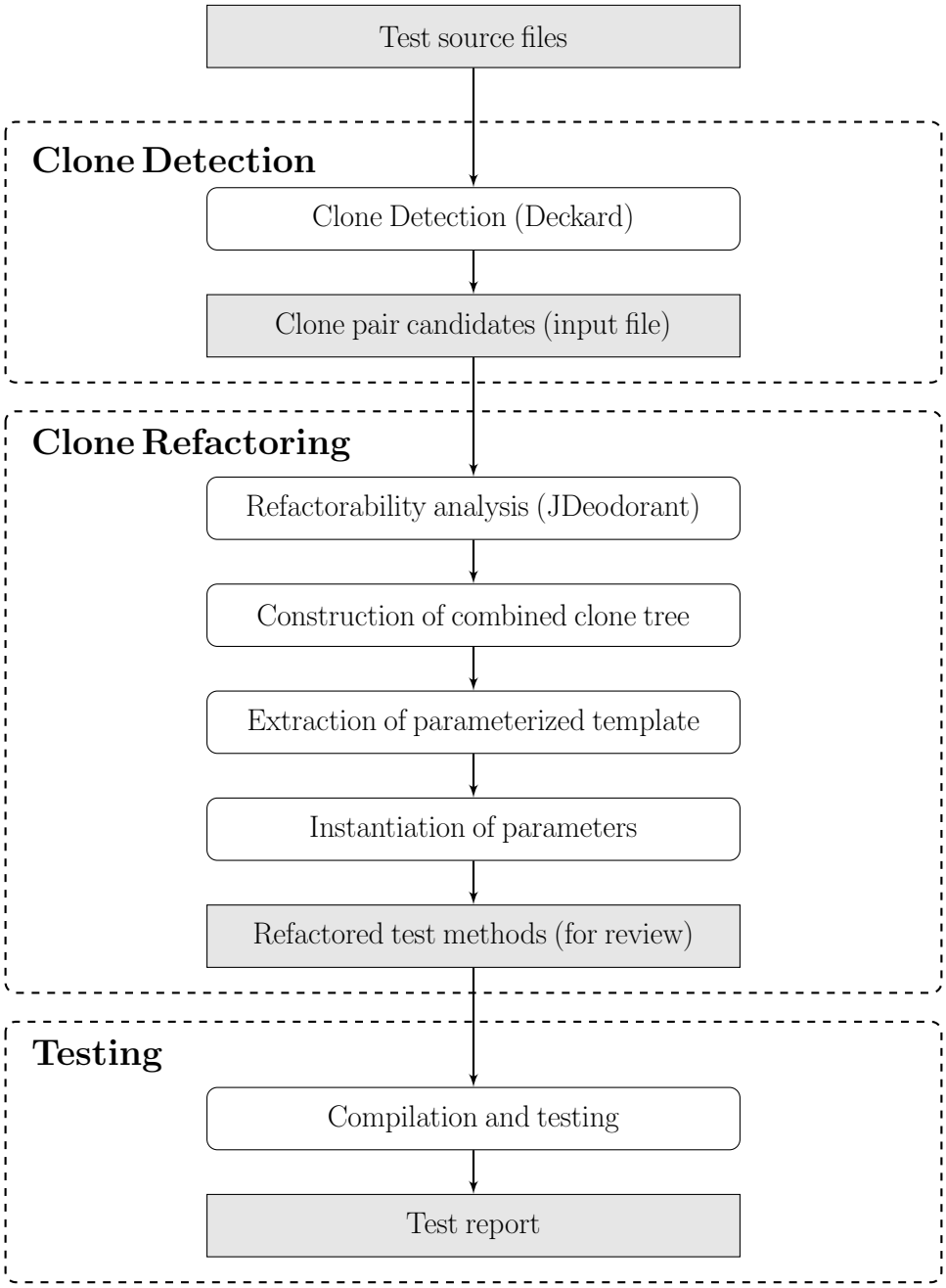


Figure 4.5: JTestParametrizer's overall workflow

1. Clone Detection (Preprocessing)

   First, we use popular clone detection tools to select refactoring candidates. In our preprocessing work, we have tried some popular tools such as CloneDR [9], Deckard [18], and CCFinder [22]. Finally we decided to use Deckard as our clone detection tool because it is free software that easily works under Linux. It is also a tree-based, accurate, and scalable code clone detection tool, which is easy to install and use.

   After detecting the clone candidates with the clone detection tool, we use a simple script to extract the clone pairs in the appropriate format from the clone detection report. We save these results in the input file for the following processing. Also, the user is free to modify the input file with a custom selection of clone pairs. Note that our technique can work with other clone detection tools. Any clone detection tool can be used as long as it generates output in the appropriate format.

2. Clone Refactoring

   At this stage, our refactoring tool reads all the clone pairs from the input file and starts to analyze and refactor the method-scope clones. Using the analysis results from JDeodorant, our tool manipulates the abstract syntax tree (AST) structure of the clone pair and directly applies changes to the source files.

   After our tool finishes refactoring all the clone pairs, developers need to review the modified test cases. They can continue to modify the refactored code, such as refactoring variable names or adding comments, to further improve the refactoring quality. Also, local changes to a specific file or multiple files can be removed if the refactoring changes are not desirable. It is easy to undo changes with the use of version control tools like *Git*.

3. Testing

   After applying appropriate refactoring, we build the modified test code under the project build system, and look for compilation errors. If the project cannot be built due to compilation errors, developers must make additional manual modifications to erroneous refactored test code to eliminate the errors. Once the project can be successfully compiled, we execute the test cases within the project test framework. By inspecting the refactored code as well as the test results after refactoring and those before refactoring, developers can determine the final refactoring quality and decide whether to make further modifications or merge changes to the project master branch.

4.3.3 Implementation

Following the workflow in Figure 4.5, our technique for refactoring renamed clone pairs comes in the *Clone Refactoring* process. Our refactoring analysis is based on the AST structure, which can represent a source code construct, such as a name, type, expression, statement, or declaration.

More specifically, the roadmap of our refactoring technique consists of the following major steps:

1. Refactorability analysis

   First, our tool needs to ensure that a clone pair is actually refactorable. That is, the pair should have isomorphic AST structure. We perform our analysis at AST node level, which includes statement mapping, difference marking, and parameterization analysis.

   With the recent research in clone refactoring, we choose to use JDeodorant [38] as the clone analysis tool to map the AST nodes and detect the differences in clone pairs. Based on the analysis results from JDeodorant, we select the renamed clone pairs (i.e, structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments [34]) as effective candidates for further refactoring.

2. Construction of combined clone tree

   With the results from the previous step, we have all the information about AST node mapping and the differences in a clone pair. At this stage, we combine the mapped nodes to construct a combined clone tree, which serves as the primary data structure to unify the clone pair and extract the parameterized template.

   Figure 4.6 illustrates how we construct a combined clone tree. The clone pair instances have the same AST structure. That is, each node in one clone instance can be mapped to its corresponding node in the other clone instance. From the previous step, we collect all the mapped nodes with differences (highlighted in colors other than green). Using this difference mapping information, we combine the nodes containing differences in a bottom-up approach. For each pair of differing nodes, we calculate the paths from the differing nodes to their root node (highlighted in red) in the clone instances. We combine each pair of mapped nodes in these paths. Thus, the combined node pairs are either differing nodes or nodes containing differences in their child elements, while the nodes without combining in the combined clone tree do not contain any differences.

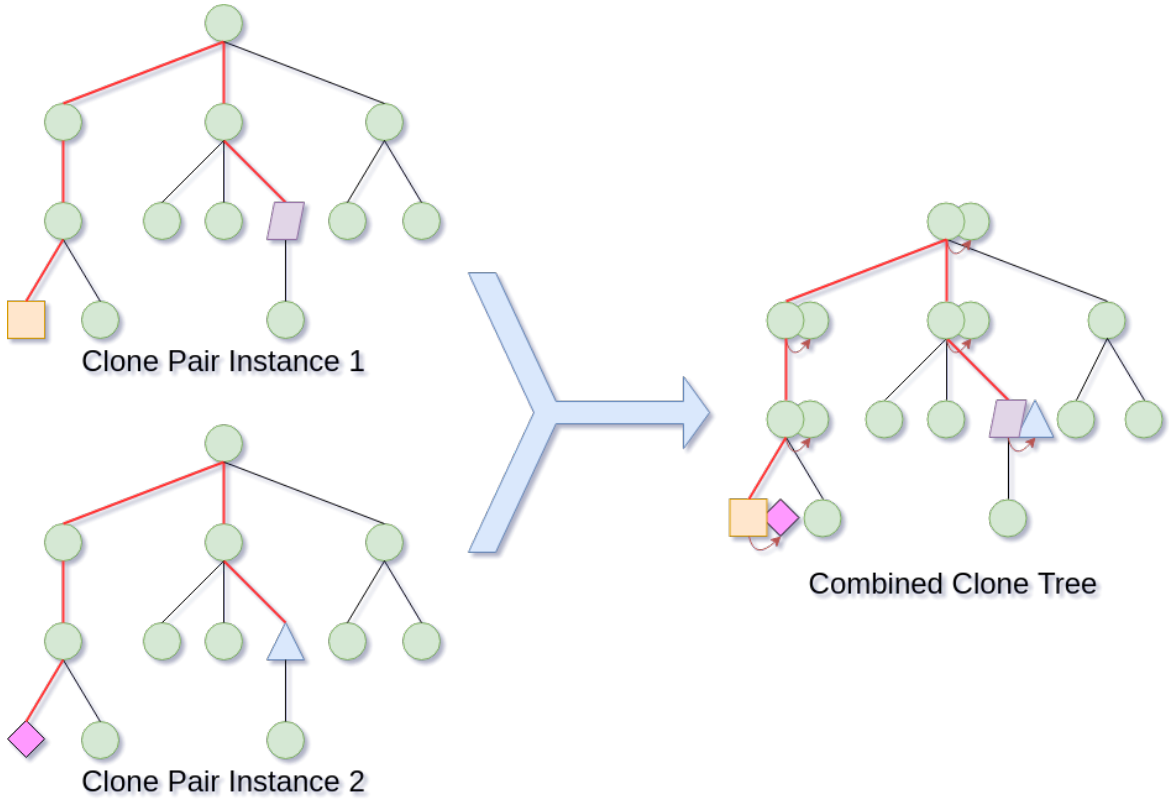


Figure 4.6: Clone pair instances have the same AST structure and we combine each pair of mapped nodes in the paths from the differing nodes to their root node

3. Extraction of parameterized template

   After constructing the combined clone tree, we continue to extract the template to unify the type, data, and behavioural differences with parameters. The parameterization technique described in section 4.2 is implemented by manipulating AST nodes in the combined clone tree. We apply the *visitor pattern* in our implementation to define appropriate operations based on the specific AST node type that we are refactoring.

   Overall, we follow a top-down approach to extract the parameterized template. Our implementation starts from the root node in the combined clone tree and examines every node in a depth-first search (DFS) manner. Using the characteristics of the combined clone tree, we can immediately figure out whether or not the current combined node contains a difference itself or in its descendent nodes. If the current

node does not contain any difference, we can stop diving deeper to improve refactoring efficiency. When there exists a difference inside the current node, we apply our parameterization techniques based on the difference categorization to unify the combined nodes along the whole paired path. Finally, the template we extract is represented in the form of the AST structure, which can be easily transformed to the source code. In addition, we create the common interface at this stage if the combined clone tree contains behavioural differences.

4. Instantiation of parameters

   Finally, we pass the clone differences as parameters to the extracted template method to implement different behaviour with specific data and types in the test cases. Based on the type of clone difference, the generic template method extracted from the previous step accepts parameters in two different ways:

   - Type parameters

     The generic types can be replaced with specific concrete types when the template method is invoked in the body of test methods.

   - Argument values

     By making use of the common behavioural interface, different implementations of the interface can be passed as arguments to the template, which enables different behavioural operations with the compatible interface in the template method execution. Similarly, the template method arguments are also able to accept different data values to apply the data differences.

4.4 Limitations

As described in section 4.3.3, our technique uses JDeodorant [38] as the clone analysis tool to map nodes at the AST structure level and detect the difference in the clone pair. The approach we propose inherits JDeodorant's limitations. JDeodorant does not support the analysis of more than two clone fragments. Additionally, our technique is based on the parameterization of clone differences. This parameterization refactoring approach works less well when clone pairs cannot be fully parameterized. We discuss our limitations in detail below.

Clone Multiplicity

Our refactoring tool only supports the analysis and refactoring of clone pairs, that is, clone groups containing two method-scope clones. When it comes to clone groups with more than two clone instances, our technique cannot be applied directly. The main difficulty lies in determining and refactoring the differences across multiple clone instances.

The current technique could be extended to support multi-clone refactoring. We could first select a clone member among the clone group as the master instance, which serves as the reference to identify the differences in the clone group. By applying our mapping approach to every combination of clone pairs between the master instance and the other members in the clone group, the difference information across clone instances could be collected with the same reference (master instance). This difference information could be further summarized to determine the global differences across all clone instances. With the same parameterization techniques described in section 4.2, the parameterized template could be extracted and multiple test methods refactored in a similar way to the proposed approach.

In principle, there is no problem with applying our technique to more than two clone instances. It is a straightforward extension that we have not explored yet. Our technique can conveniently refactor any two members of a N-member clone set.

Access Issue

Another limitation of our approach is the access issue to private fields or methods when the clone pair instances located in different classes need access to their private class members. For clone pairs across different classes, our refactoring technique extracts a parameterized template method in a new utility class located in the same package. Thus, the template method has no access to the private members of the original class of the clone pair, which leads to an access issue if the template method needs to access private class members (fields or methods). In our benchmarks, we observed 48 clone pairs with this access issue, accounting for 12% of the effective refactoring candidates.

Currently, we simply mark the clone pairs with this issue as non-refactorable clone groups. Access modifiers of the related clone classes can be refined according to developers' needs. There are other ways to access private members, through reflection, but that would be detrimental to code quality. Developers should manually inspect this type of clone candidates to determine whether refactoring is appropriate.

Consecutive method calls

When refactoring consecutive method calls in the fluent pattern, our technique may lead to undesirable refactoring results, which can make the tests more complicated to understand.

Figure 4.7 presents a motivating example. This example illustrates a pair of naive test clones for the *builder pattern*. Using a fluent interface pattern, the variable `builder` invokes consecutive method calls to perform a series of object mutations. When transformed to the AST representation, the behavioural differences exist in multiple levels of a nested method invocation node. Applying our technique to these nested behavioural differences might decrease refactoring quality, especially when the number of nested levels increases.

```
// clone pair1
public void testBuilder1() {
  Builder builder = new Builder()
          .setA1()
          .setB1()
          .setC1()
          .build();
  // other duplicated statements
  ...
}

// clone pair2
public void testBuilder2() {
  Builder builder = new Builder()
          .setA2()
          .setB2()
          .setC2()
          .build();
  // other duplicated statements
  ...
}
```

Figure 4.7: Clone pair with nested behavioural differences

Figure 4.8 shows the parameterized template method and the common behavioural interface refactored by our approach. With our parameterization techniques, the consecutive behavioural differences are parameterized in a nested way, which generates complicated adaptive method calls in the template method. This nesting effect could be more severe if the clone pair contains more consecutive differences in the nested method invocation node.

Therefore, developers must use code review to avoid this side effect and confirm refactoring results.

```
// parameterized template
public void testBuilderTemplate(BuilderTestAdapter adapter) {
  Builder builder =
          adapter.setC(adapter.setB(adapter.setA(new Builder())))
          .build();
  // other duplicated statements
  ...
}

// common behavioural interface
interface BuilderTestAdapter {
  Builder setA(Builder builder);
  Builder setB(Builder builder);
  Builder setC(Builder builder);
}
```

Figure 4.8: Parameterization of nested behavioural differences

JDeodorant Issues

Our refactoring tool exposed the following issues in JDeodorant which cause our refactoring to fail:

- Statement mismatch

  JDeodorant applies an optimal mapping approach which is intended to minimize the number of differences between the members of a clone pair [38]. However, JDeodorant sometimes mismatches statements, which can cause test failures in the refactored test methods or unpredictable exceptions in the refactoring execution.

- Missing of difference mapping in the arguments of constructor calls

  We also observe that JDeodorant misses differences in the arguments of constructor calls when creating class instances with the `new` operator. Thus, the clone pair cannot be appropriately refactored with our technique if there exist differences in the argument list of the class instance creation node.

Chapter 5

Evaluation

To evaluate our refactoring tool, we performed an empirical study and applied our technique to a collection of 5 Java-based open source benchmarks, shown in Table 5.1. We selected these benchmarks for evaluation because they are all popular and active open source projects across different application domains. Overall, our selected benchmarks totally have 157,205 lines of production code (Main LoC) and 182,364 lines of test code (Test LoC).[1] The benchmark test suites contain test code ranging from 14,137 to 55,491 lines, and in total 14,431 test methods.

Table 5.1: Selected benchmark projects

Benchmark	Description	Version	Main LoC	Test LoC
JFreeChart	Java chart library	1.0.10	83,039	44,892
Gson	JSON representation conversion	2.8.5	8,131	14,137
Apache Commons Lang	Java API helper utilities	3.7	27,428	48,530
Apache Commons IO	IO development library	2.5	9,836	19,314
Joda-Time	Standard date and time library	2.10	28,771	55,491
Total			157,205	182,364

[1] LoC data generated using CLOC [2].

With the above benchmarks, we evaluate our technique along the following dimensions:

- Applicability

  This dimension measures how often our refactoring technique can apply to the tests in the benchmarks.

- Correctness

  In the scope of our work, a correct refactoring means that the pair of refactored test methods can be built without compilation errors, and all tests of the project test suite pass after the application of refactorings. Note that all test cases in our benchmarks declare passing results before our refactorings.

- Refactoring quality

  Assessing the quality of a refactoring appears to be an open question in practice, and for code in general, not just test cases. In the context of test case clone refactoring, we chose to evaluate the quality of refactoring results by evaluating the refactored tests from perspectives including conciseness, repetition, extensibility, maintenance cost, understandability, and debuggability.

Overall, our technique refactors 65% of detected renamed clone pairs (268 clone pairs) in our benchmark test suites. All of the refactored test methods compile, and 94% of them execute as unit tests and declare passing results. While our refactoring could potentially make individual tests more complicated to understand and more difficult to debug, we believe that our proposed refactorings generally improve code quality regarding conciseness, repetition, extensibility, and maintenance cost. Finally, we individually discuss results for each benchmark project.

5.1 Applicability

Using our proposed technique, we refactored the detected clone candidates for each benchmark. Table 5.2 summarizes the applicability of our technique on the selected benchmarks. The *Total tests* column denotes the number of test methods in the benchmark. The *Detected clone pairs* column lists the total number of potentially refactorable clone pairs reported by the Deckard clone detection tool. Detected clone pairs include refactoring candidates of renamed (*Type II*) and gapped (*Type III*) clones. The *Effective candidates* column presents the number of renamed clone (*Type II*) pairs identified by *JTestParametrizer*,

while the *Refactorable* column counts the number of renamed clone (*Type II*) pairs that our technique can actually refactor. Finally, the *Refactoring rate* column reports the percentage of refactorable clones as a fraction of the effective candidates.

Table 5.2: 65% of the effective candidates (268 clone pairs) in our benchmark projects are refactorable using our technique

Benchmark	Total tests	Detected clone pairs	Effective candidates	Refactorable	Refactoring rate
JFreeChart	3934	107	65	49	75%
Gson	1050	31	14	10	71%
Commons Lang	4068	144	93	65	70%
Commons IO	1157	78	56	38	68%
Joda-Time	4222	249	187	106	57%
Total	14431	609	415	268	65%

Based on our refactoring results, our technique can parameterize a significant fraction of effective candidates, ranging from 57% for the *Joda-Time* project to 75% for the *JFreeChart* project. Our technique examined 14,431 test methods across five benchmark projects and identified 415 of those as effective candidates for refactoring. On average, our empirical analysis found that 65% of the effective candidates (268 clone pairs) are refactorable using our technique.

5.2 Correctness

To assess the correctness of our technique, we run entire test suite after the application of all possible refactorings for each benchmark. We believe that using the project testing frameworks is an effective and efficient way to evaluate the refactoring correctness. We will also discuss the threats to validity of this assessment in section 5.5.

Table 5.3 reports the testing results of the refactored test methods on our benchmarks. The *Refactored tests* column shows the number of refactored clone pairs. The *Compilability* column reports the ratio of successfully compiled test methods, while the *Test failures* column reports the number of failures in the refactored test cases. Finally, we calculate the percentage of correctly refactored clone pairs, which can be run and tested without failures in the *JUnit* framework, as the correct refactoring rate in the last column.

Table 5.3: 94% of the 268 refactored test method pairs execute without failures across five benchmarks

Benchmark	Refactored tests	Compilability	Test failures	Correct refactoring rate
JFreeChart	49	100%	0	100%
Gson	10	100%	0	100%
Commons Lang	65	100%	4	94%
Commons IO	38	100%	3	92%
Joda-Time	106	100%	10	91%
Total	268	100%	17	94%

Our technique enjoys a high correct refactoring rate, reaching over 90% of the refactored test methods in all of the selected benchmark projects. Overall, we refactor 268 test method pairs across five benchmarks, where all of the refactored tests are compilable, and 94% of them execute without failures. Manual inspection of the test failures confirms that the failures are caused by JDeodorant issues. As discussed in section 4.4, we might not appropriately refactor a clone pair due to occasional statement mismatching between the clone pair instances or a missing difference mapping in the arguments of constructor calls. Thus, developers still need to review refactorings to eliminate JDeodorant issues and confirm the final refactoring results.

5.3 Refactoring Quality

In practice, evaluating the effect of refactoring on software quality remains an open question. In this work, we qualitatively evaluate the quality of the refactoring results that our tool proposes. We compare test methods before and after refactoring, manually inspecting and analyzing the correctly refactored clone pairs for each project. We consider the following perspectives: conciseness, repetition, extensibility, maintenance cost, understandability, and debuggability. Our manual evaluation suggests the following conclusions:

Conciseness Because our refactoring extracts the common logic in the template method, the refactored tests are more concise, especially when the clone instances only contain a small number of differences in relatively large method bodies.

Repetition With our proposed parameterization technique, we remove much repeated code and effectively reduce the duplication code smell.

Extensibility We consider the ease with which developers may add additional test cases using the refactored utility methods. By replacing the arguments in the parameterized template method with different values, developers can reuse the same testing logic with different input data, which enables extending the test suite easily. This perspective brings to mind clone groups with more than two elements; we believe that our technique can be reasonably extended to handle such groups. Additionally, we will also consider integrating *JUnit Theories* [5] with our tool in the future to further improve the extensibility of test suites.

Maintenance cost Normal test case maintenance often involves work on similar test cases, which is error-prone and tedious. In particular, maintaining clones in the test suite needs multiple changes across the test clones, resulting in a heavier technical cost for software maintenance. Our technique can effectively reduce this maintenance cost. Since the duplicated code fragments are extracted in one place, developers can easily maintain the corresponding tests without multiple modifications.

Understandability This perspective is a measure of whether the refactored test methods are easier to understand for developers. By introducing the generic type and additional common behavioral interface, our technique could make individual tests more complicated to understand because they include a new layer of abstraction. However, we believe that our arguments show that the suite of refactored tests is collectively easier to understand and maintain.

Debuggability This perspective describes how easy or convenient it is for developers to find what is wrong with the failed tests and the related main source code. While, after refactoring, test failures could be more difficult to diagnose (due to the extra layer of abstraction), we argue that it does not require costly debugging efforts and it is a reasonable trade-off to improve the other aspects of code quality.

5.4 Case Study

We next present specific refactoring examples for each of our benchmarks in more detail.

5.4.1 JFreeChart

JFreeChart is a Java library that facilitates the display of professional quality charts in applications. The JFreeChart test suite (version 1.0.10) consists of 367 test classes in 24 packages. In particular, this benchmark contains similar test methods which test related types. Figure 5.1 shows that the test methods named `testFindRangeBounds()` in classes `HighLowRendererTests` and `CandlestickRendererTests` are identical except for variations in the declared type of the variable `renderer` and its initializer. The classes `HighLowRenderer` and `CandlestickRenderer` are related types to each other; they are both subclasses of the class `AbstractXYItemRenderer`. Using the taxonomy in section 3.2, we identify these variations as type differences. Our technique can provide high-quality refactoring suggestions for these similar clones.

```
// test method1 in org.jfree.chart.renderer.xy.junit.HighLowRendererTests
public void testFindRangeBounds() {
  HighLowRenderer renderer = new HighLowRenderer();
  OHLCDataItem item1 = new OHLCDataItem(new Date(1L), 2.0, 4.0, 1.0, 3.0, 100);
  // other identical statements
  ...
}

// test method2 in org.jfree.chart.renderer.xy.junit.CandlestickRendererTests
public void testFindRangeBounds() {
  CandlestickRenderer renderer = new CandlestickRenderer();
  OHLCDataItem item1 = new OHLCDataItem(new Date(1L), 2.0, 4.0, 1.0, 3.0, 100);
  // other identical statements
  ...
}

// parameterized template in
    org.jfree.chart.renderer.xy.junit.RendererTestsTestFindRangeBoundsTemplate
public static <TRenderer extends AbstractXYItemRenderer>
    void rendererTestsTestFindRangeBoundsTemplate(
    Class<TRenderer> clazzTRenderer) throws Exception {
  TRenderer renderer = clazzTRenderer.newInstance();
  OHLCDataItem item1 = new OHLCDataItem(new Date(1L), 2.0, 4.0, 1.0, 3.0, 100);
  // other identical statements
  ...
}
```

```
// refactored test method1 in
    org.jfree.chart.renderer.xy.junit.HighLowRendererTests
public void testFindRangeBounds() {
  RendererTestsTestFindRangeBoundsTemplate
    .rendererTestsTestFindRangeBoundsTemplate( HighLowRenderer .class);
}

// refactored test method2 in
    org.jfree.chart.renderer.xy.junit.CandlestickRendererTests
public void testFindRangeBounds() {
  RendererTestsTestFindRangeBoundsTemplate
    .rendererTestsTestFindRangeBoundsTemplate( CandlestickRenderer .class);
}
```

Figure 5.1: JFreeChart contains similar `testFindRangeBounds()` methods which test related types

5.4.2 Gson

Gson is a Java serialization/deserialization library which enables conversion between Java Objects and JSON. The Gson test suite (version 2.8.5) includes 101 test classes in 10 packages. In particular, this benchmark contains method clones which test classes with type parameters. Figure 5.2 shows a typical refactoring example with differing generic classes `LinkedTreeMap` and `LinkedHashTreeMap`. Unlike in Figure 5.1, the differing types in this clone pair, identified as type difference, contain type parameters (e.g., `LinkedTreeMap <String, String>`). We can still handle this case. Our technique parameterizes the differing generic types by adding bounds to the extracted generic type.

```
// test method1 in com.google.gson.internal.LinkedTreeMapTest
public void testLargeSetOfRandomKeys() throws Exception {
  Random random = new Random(1367593214724L);
  LinkedTreeMap<String, String> map = new LinkedTreeMap<String, String>();
  String[] keys = new String[1000];
  // other identical statements
  ...
}
```

```
// test method2 in com.google.gson.internal.LinkedHashTreeMapTest
public void testForceDoublingAndRehash() throws Exception {
  Random random = new Random(1367593214724L);
  LinkedHashTreeMap<String, String> map =
      new LinkedHashTreeMap<String, String> ();
  String[] keys = new String[1000];
  // other identical statements
  ...
}

// parameterized template in
    com.google.gson.internal.LinkedTreeMapTestTestTemplate
public class LinkedTreeMapTestTestTemplate {
  public static <TLinkedTreeMapStringString extends AbstractMap<String, String>>
    void linkedTreeMapTestTestTemplate(Class<TLinkedTreeMapStringString>
        clazzTLinkedTreeMapStringString) throws Exception {
    Random random = new Random(1367593214724L);
    TLinkedTreeMapStringString map =
        clazzTLinkedTreeMapStringString.newInstance();
    String[] keys = new String[1000];

    // other identical statements
    ...
  }
}

// refactored test method1 in com.google.gson.internal.LinkedTreeMapTest
public void testLargeSetOfRandomKeys() throws Exception {
  LinkedTreeMapTestTestTemplate
    .linkedTreeMapTestTestTemplate( LinkedTreeMap .class);
}

// refactored test method2 in com.google.gson.internal.LinkedHashTreeMapTest
public void testEqualsAndHashCode() throws Exception {
  LinkedTreeMapTestTestEqualsAndHashCodeTemplate
    .linkedTreeMapTestTestEqualsAndHashCodeTemplate( LinkedHashTreeMap .class);
}
```

Figure 5.2: Gson test suite contains similar test methods with differences in generic classes

5.4.3 Apache Commons Lang and Commons IO

Apache Commons Lang and Apache Commons IO are projects of the Apache Software Foundation. Commons Lang is a Java utility library providing extra methods for manipulating Java core classes, while Commons IO is a Java utility library to assist with IO functionality. The Apache Commons Lang test suite (version 3.7) consists of 172 test classes in 14 packages, and the Apache Commons IO test suite (version 2.5) contains 112 test classes in 9 packages.

These two benchmarks include a number of refactorable tests. While refactoring the Apache Commons benchmarks, we noticed that some clone pairs located in the same classes access private fields of their original class. Figure 5.3 and Figure 5.4 present two refactorings of clone pairs with differences in private fields that we found in the test classes `StringUtilsSubstringTest` and `DeferredFileOutputStreamTest`. Our technique can still handle these cases. Since the pair of clone instances are in the same class, we extract the parameterized template method as a class member. Thus, the template method can access the private members of the original class of the clone pair.

```
// private fields in org.apache.commons.lang3.StringUtilsSubstringTest
private static final String FOO = "foo";
private static final String BAR = "bar";
private static final String FOOBAR = "foobar";
```

```
// test method1 in org.apache.commons.lang3.StringUtilsSubstringTest
public void testLeft_String() {
  assertSame(null, StringUtils.left(null, -1));
  assertSame(null, StringUtils.left(null, 0));
  assertSame(null, StringUtils.left(null, 2));

  assertEquals("", StringUtils.left("", -1));
  assertEquals("", StringUtils.left("", 0));
  assertEquals("", StringUtils.left("", 2));

  assertEquals("", StringUtils.left(FOOBAR, -1));
  assertEquals("", StringUtils.left(FOOBAR, 0));
  assertEquals(FOO, StringUtils.left(FOOBAR, 3));
  assertSame(FOOBAR, StringUtils.left(FOOBAR, 80));
}
```

```
// test method2 in org.apache.commons.lang3.StringUtilsSubstringTest
public void testRight_String() {
  assertSame(null, StringUtils.right(null, -1));
  assertSame(null, StringUtils.right(null, 0));
  assertSame(null, StringUtils.right(null, 2));

  assertEquals("", StringUtils.right("", -1));
  assertEquals("", StringUtils.right("", 0));
  assertEquals("", StringUtils.right("", 2));

  assertEquals("", StringUtils.right(FOOBAR, -1));
  assertEquals("", StringUtils.right(FOOBAR, 0));
  assertEquals(BAR, StringUtils.right(FOOBAR, 3));
  assertSame(FOOBAR, StringUtils.right(FOOBAR, 80));
}

// parameterized template in org.apache.commons.lang3.StringUtilsSubstringTest
public void stringUtilsSubstringTestTestStringTemplate(
    StringUtilsSubstringTestTestStringAdapter adapter, String string1) {
  assertSame(null, adapter.action1(null, -1));
  assertSame(null, adapter.action1(null, 0));
  assertSame(null, adapter.action1(null, 2));

  assertEquals("", adapter.action1("", -1));
  assertEquals("", adapter.action1("", 0));
  assertEquals("", adapter.action1("", 2));

  assertEquals("", adapter.action1(FOOBAR, -1));
  assertEquals("", adapter.action1(FOOBAR, 0));
  assertEquals(string1, adapter.action1(FOOBAR, 3));
  assertSame(FOOBAR, adapter.action1(FOOBAR, 80));
}

// common behavioural interface
interface StringUtilsSubstringTestTestStringAdapter {
  String action1(String string1, int i1);
}
```

```
// behavioural implementation for testLeft_String()
class StringUtilsSubstringTestTestLeft_StringAdapterImpl implements
    StringUtilsSubstringTestTestStringAdapter {
  public String action1(String string1, int i1) {
    return StringUtils.left(string1, i1);
  }
}

// behavioural implementation for testRight_String()
class StringUtilsSubstringTestTestRight_StringAdapterImpl implements
    StringUtilsSubstringTestTestStringAdapter {
  public String action1(String string1, int i1) {
    return StringUtils.right(string1, i1);
  }
}

// refactored test method1 in org.apache.commons.lang3.StringUtilsSubstringTest
public void testLeft_String() {
  this.stringUtilsSubstringTestTestStringTemplate(
    new StringUtilsSubstringTestTestLeft_StringAdapterImpl(), FOO);
}

// refactored test method1 in org.apache.commons.lang3.StringUtilsSubstringTest
public void testRight_String() {
  this.stringUtilsSubstringTestTestStringTemplate(
    new StringUtilsSubstringTestTestRight_StringAdapterImpl(), BAR);
}
```

Figure 5.3: Refactorable test methods `testLeft_String` and `testRight_String` both access private fields of test class `StringUtilsSubstringTest` and contain data and behavioural differences

```
// private fields in org.apache.commons.io.output.DeferredFileOutputStreamTest
private final String testString = "0123456789";
private final byte[] testBytes = testString.getBytes();

// test method1 in org.apache.commons.io.output.DeferredFileOutputStreamTest
public void testBelowThreshold() {
  final DeferredFileOutputStream dfos =
    new DeferredFileOutputStream(testBytes.length + 42, null);
```

```
  try {
    dfos.write(testBytes, 0, testBytes.length);
    dfos.close();
  } catch (final IOException e) {
    fail("Unexpected IOException");
  }
  assertTrue(dfos.isInMemory());

  final byte[] resultBytes = dfos.getData();
  assertEquals(testBytes.length, resultBytes.length);
  assertTrue(Arrays.equals(resultBytes, testBytes));
}

// test method2 in org.apache.commons.io.output.DeferredFileOutputStreamTest
public void testAtThreshold() {
  final DeferredFileOutputStream dfos =
    new DeferredFileOutputStream( testBytes.length , null);
  try {
    dfos.write(testBytes, 0, testBytes.length);
    dfos.close();
  } catch (final IOException e) {
    fail("Unexpected IOException");
  }
  assertTrue(dfos.isInMemory());

  final byte[] resultBytes = dfos.getData();
  assertEquals(testBytes.length, resultBytes.length);
  assertTrue(Arrays.equals(resultBytes, testBytes));
}

// parameterized template in
    org.apache.commons.io.output.DeferredFileOutputStreamTest
public void deferredFileOutputStreamTestTestThresholdTemplate(int i1) {
  final DeferredFileOutputStream dfos = new DeferredFileOutputStream(i1, null);
  try {
    dfos.write(testBytes, 0, testBytes.length);
    dfos.close();
  } catch (final IOException e) {
    fail("Unexpected IOException");
  }
```

```
  assertTrue(dfos.isInMemory());
  final byte[] resultBytes = dfos.getData();
  assertEquals(testBytes.length, resultBytes.length);
  assertTrue(Arrays.equals(resultBytes, testBytes));
}

// refactored test method1 in
    org.apache.commons.io.output.DeferredFileOutputStreamTest
public void testBelowThreshold() {
  this.deferredFileOutputStreamTestTestThresholdTemplate( testBytes.length + 42 );
}

// refactored test method2 in
    org.apache.commons.io.output.DeferredFileOutputStreamTest
public void testAtThreshold() {
  this.deferredFileOutputStreamTestTestThresholdTemplate( testBytes.length );
}
```

Figure 5.4: Refactorable test methods `testBelowThreshold` and `testAtThreshold` both access private fields of test class `DeferredFileOutputStreamTest` and contain data differences.

5.4.4 Joda-Time

Joda-Time is a widely used library providing a quality replacement for the Java date and time classes. The Joda-Time test suite (version 2.10) consists of 159 test classes in 7 packages. Based on our analysis, Joda-Time has the most refactorable test clones (106 clone pairs) in the selected benchmarks. We also found that many test clone pairs are in the same classes with a small number of differences. These clones can be refactored effectively using our technique. Figure 5.5 shows two test methods located in the same class. The clone instances have similar test method names, `testRemoveBadIndex1` and `testRemoveBadIndex2`. The only difference lies in the bad index values passed to the argument list of the method call `set.remove(...)`, which fits in the type of data differences using the taxonomy in section 3.2. We can easily extract a template method in the same class by parameterizing this data differences.

```
// test method1 in org.joda.time.convert.TestConverterSet
public void testRemoveBadIndex1() {
  Converter[] array = new Converter[] {c1, c2, c3, c4};
  ConverterSet set = new ConverterSet(array);
  try {
    set.remove( 200 , null);
    ...// other identical statements
  }
  ...
}

// test method2 in org.joda.time.convert.TestConverterSet
public void testRemoveBadIndex2() {
  Converter[] array = new Converter[] {c1, c2, c3, c4};
  ConverterSet set = new ConverterSet(array);
  try {
    set.remove( -1 , null);
    ...// other identical statements
  }
  ...
}

// parameterized template in org.joda.time.convert.TestConverterSet
public void testConverterSetTestRemoveBadTemplate(int i1) {
  Converter[] array = new Converter[] {c1, c2, c3, c4};
  ConverterSet set = new ConverterSet(array);
  try {
    set.remove(i1, null);
    // other identical statements
    ...
  }
  ...
}

// refactored test method1 in org.joda.time.convert.TestConverterSet
public void testRemoveBadIndex1() {
  this.testConverterSetTestRemoveBadTemplate( 200 );
}
```

```
// refactored test method2 in org.joda.time.convert.TestConverterSet
public void testRemoveBadIndex2() {
  this.testConverterSetTestRemoveBadTemplate( -1 );
}
```

Figure 5.5: Differing values in `testRemoveBadIndex1` and `testRemoveBadIndex2` can be parameterized locally

5.4.5 Summary

This section discussed specific refactoring results for each of our selected benchmark projects. Overall, our technique can provide useful refactoring suggestions to remove duplicated code smell. Our proposed refactorings work well for the clone pairs with a small number of differences (e.g., test clones in Joda-Time).

Also, our technique can handle some edge cases. When the differing types in clone pairs contain same type parameters, our technique can parameterize this type difference by adding bounds to the extracted generic type. For clone instances located in the same class, our technique can parameterize differences that access private fields of the original class.

5.5 Threats to Validity

The selection of 5 benchmark projects from different application domains (IO development, chart display, date and time, etc.) aims to reduce threats to external validity. However, no benchmark can capture all possible code styles in test suites. Furthermore, the overall size of our selected benchmark test suites does not exceed 190 kLoC. Larger benchmark test suites may present different results from those in our empirical study.

Our tool is designed to refactor JUnit tests. Additional work would be required to extend our results to non-JUnit or non-Java benchmarks.

Our technique is based on the three levels of refactoring to parameterize type, data, and behaviour differences in clone pairs. Our refactoring results do not apply to clone pairs with other types of differences.

In our work, we consider a refactoring correct if the refactored test methods are compilable and all tests can pass after the application of the refactoring. This assessment might

not guarantee that the test code behaviour before and after refactoring remains unchanged, although we expect such cases to be rare.

Another threat to construct validity is that we manually evaluate the quality of our refactorings. Our consideration from the perspectives of conciseness, repetition, extensibility, maintenance cost, understandability, and debuggability aims to mitigate this threat. However, evaluating from other perspectives could lead to different conclusions.

We believe that we have reasonably mitigated the threats to external validity through benchmark selection and have adequately reduced evaluation biases through consideration from a relatively large number of perspectives. Our empirical study presents an accurate evaluation of our technique on JUnit test suites.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis presents a technique to automatically refactor method-scope renamed clone pairs in JUnit test suites. We implemented a tool, *JTestParametrizer*, which helps developers easily refactor test clones.

In our approach, we introduce three levels of refactoring to parameterize type, data, and behaviour differences in clone pairs. Our technique works by extracting a parameterized template utility method from clone instances. It then creates code that passes appropriate parameter values to the extracted template method for instantiating specific test cases.

We performed an empirical study by applying our technique to 5 open-source Java benchmarks. On average, 65% of the renamed clone candidates (268 clone pairs) in our test suites are refactorable using our technique. All of the refactored test methods are compilable, and 94% of them declare passing results when executed as tests. Manual inspection suggests that our proposed refactorings generally improve the code quality regarding conciseness, repetition, extensibility, and maintenance cost. We believe that our work enables the broader use of automatic techniques for test clone refactoring, leading to more maintainable and more robust test suites.

6.2 Future Work

In this thesis, we present a technique for refactoring similar tests in test suites. In the future, there are multiple directions that we can explore to enhance our tool and improve

refactoring quality:

- Java 8 introduces lambda expressions as a new feature to support functional programming. When refactoring clone pairs with a small number of behavioural differences, we can use lambda expressions as specific implementations of common interface methods to make the refactoring suggestions more succinct. Instead of creating new inner classes, lambda expressions can be used as syntactically shorter alternatives to implement concrete behaviour, especially when there is only one behavioural difference between the clone instances.

- We would like to introduce *JUnit Theories* [5] in our tool to parameterize data differences. In practice, developers are likely to use a set of data points for testing a specific functionality. *Theories* can be a clean and powerful tool to perform data parameterization. We can combine this JUnit feature with our current system to provide more flexible refactoring suggestions for developers.

- Finally, we would also like to extend our system to support multi-clone parameterization. In our detected refactorable candidates, we observe that clone groups usually contain two or more instances. To automatically refactor multiple similar tests, we would need to determine and combine differences across multiple clone instances. Section 4.4 presents one possible solution using the current technique. The present work is a useful first step towards multi-clone test refactoring.

References

[1] Abstract syntax tree — Eclipse. https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/. Accessed: 2018-10-15.

[2] Count Lines of Code. https://github.com/AlDanial/cloc. Accessed: 2018-10-15.

[3] JDeodorant — Github. https://github.com/tsantalis/JDeodorant. Accessed: 2018-10-15.

[4] JDT programmer's guide. https://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.isv/guide/jdt_int.htm. Accessed: 2018-10-01.

[5] Theories (JUnit API). https://junit.org/junit4/javadoc/4.12/org/junit/experimental/theories/Theories.html. Accessed: 2018-09-30.

[6] V Aho Alfred, Sethi Ravi, and D Ullman Jeffrey. Compilers: principles, techniques, and tools. *Reading: Addison Wesley Publishing Company*, 1986.

[7] Brenda S Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995.

[8] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 326–336. IEEE, 1999.

[9] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.

[10] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[11] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

[12] Dustin Campbell and Mark Miller. Designing refactoring tools for developers. In *Proceedings of the 2nd Workshop on Refactoring Tools*, page 9. ACM, 2008.

[13] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118. IEEE, 1999.

[14] Pablo Estefó. Restructuring unit tests with Testsurgeon. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1632–1634. IEEE Press, 2012.

[15] Nils Gode and Jan Harder. Clone stability. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 65–74. IEEE, 2011.

[16] Eduardo Martins Guerra and Clovis Torres Fernandes. Refactoring test code safely. In *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, pages 44–44. IEEE, 2007.

[17] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 53–62. IEEE, 2012.

[18] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[19] J Howard Johnson. Substring matching for clone detection and change tracking. In *ICSM*, volume 94, pages 120–126, 1994.

[20] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 485–495. IEEE, 2009.

[21] Nicolas Juillerat and Beat Hirsbrunner. Toward an implementation of the "form template method" refactoring. In *Source Code Analysis and Manipulation, 2007.*

SCAM 2007. Seventh IEEE International Working Conference on, pages 81–90. IEEE, 2007.

[22] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[23] Cory Kapser and Michael W Godfrey. "cloning considered harmful" considered harmful. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 19–28. Citeseer, 2006.

[24] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM, 2005.

[25] Kostas A Kontogiannis, Renato DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1-2):77–108, 1996.

[26] Jens Krinke. Is cloned code more stable than non-cloned code? In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 57–66. IEEE, 2008.

[27] Jens Krinke. Is cloned code older than non-cloned code? In *Proceedings of the 5th International Workshop on Software Clones*, pages 28–33. ACM, 2011.

[28] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. 2008.

[29] Manishankar Mandal, Chanchal K Roy, and Kevin A Schneider. Automatic ranking of clones for refactoring through mining association rules. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 114–123. IEEE, 2014.

[30] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, volume 96, page 244, 1996.

[31] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 392–402. IEEE Press, 2015.

[32] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[33] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Automatic identification of important clones for refactoring and tracking. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 11–20. IEEE, 2014.

[34] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queens School of Computing TR*, 541(115):64–68, 2007.

[35] David Saff, Marat Boshernitsan, and Michael D Ernst. Theories in practice: Easy-to-write specifications that catch bugs. 2008.

[36] Suresh Thummalapenta, Madhuri R Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Retrofitting unit tests for parameterized unit testing. In *International Conference on Fundamental Approaches to Software Engineering*, pages 294–309. Springer, 2011.

[37] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 253–262. ACM, 2005.

[38] Nikolaos Tsantalis, Davood Mazinanian, and Giri P. Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090, November 2015.

[39] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 60–70, Piscataway, NJ, USA, 2017. IEEE Press.

[40] Arie van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, 2001.

[41] Wei Wang and Michael W Godfrey. Recommending clones for refactoring using design, context, and history. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 331–340. IEEE, 2014.