Distilling Useful Clones by Contextual Differencing

Zhenchang Xing
Nanyang Technological University, Singapore
zcxing@ntu.edu.sg

Yinxing Xue, Stan Jarzabek
National University of Singapore, Singapore
tslxuey@nus.edu.sg, stan@comp.nus.edu.sg

***Abstract*—Clone detectors find similar code fragments and report large numbers of them for large systems. Textually similar clones may perform different computations, depending on the program context in which clones occur. Understanding these contextual differences is essential to distill useful clones for a specific maintenance task, such as refactoring. Manual analysis of contextual differences is time consuming and error-prone. To mitigate this problem, we present an automated approach to helping developers find and analyze contextual differences of clones. Our approach represents context of clones as program dependence graphs, and applies a graph differencing technique to identify required contextual differences of clones. We implemented a tool called CloneDifferentiator that identifies contextual differences of clones and allows developers to formulate queries to distill candidate clones that are useful for a given refactoring task. Two empirical studies show that CloneDifferentiator can reduce the efforts of post-detection analysis of clones for refactorings.**

***Index Terms*—Clone analysis, program dependence graph, graph matching.**

I. INTRODUCTION

Similar code fragments are called code clones, and techniques [3][4][9][16][21][22] have been proposed to detect code clones. Many refactorings [8] are concerned with code clones, so clone detectors are often used for such refactorings.

Fig. 1 shows a method *hasArray()* that is cloned in seven subclasses of the *Buffer* class. Looking at the cloned methods themselves it seems that they could be removed by pulling up *hasArray()* to the superclass *Buffer*. However, having examined the context of the method *hasArray()*, we see that field *hb* referred in the method is declared differently for each subclass. Thus, even if *hasArray()* is textually identical in seven sub-classes, pulling it up to superclass *Buffer* would lead to an error.

Clone detection techniques report large numbers of clones for industrial systems, only some of which can be refactored. *How can developers tell useful clones for a given refactoring task from irrelevant ones*? Clone detectors provide little or no additional information to aid developers in distilling useful clones from the rest. To ease post-detection analysis of clones, researchers have investigated using textual differencing [15][18], code metrics [1][2], visualization [26], and query-based filtering techniques [31]. However, these methods ignore the program context in which clones occur and cannot identify differences in the context of clones such as in our buffer example. As a result, cloned *hasArray()* would be reported as type-1 clone [5], i.e. they are considered identical and can be refactored. However, *hasArray()* is not a valid candidate for pull-up method refactoring. On the other hand, cloned methods in Fig. 2 would be reported as type-2 clone [5], because they have identifier differences (***b*** versus ***v***). But the current clone detectors or analyzers cannot conduct analysis on clone context, i.e. the two methods in fact are identical and can be refactored.

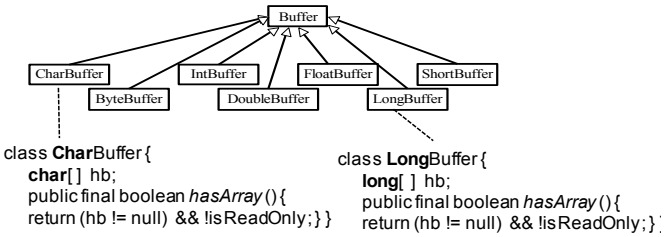

Fig. 1. Can we pull-up these cloned methods?

```
ObjectOutputStream$BlockDataOutputStream (Java IO 1.5)
1767.   public void write(int b) throws IOException {
1768.     if(this.pos >= MAX_BLOCK_SIZE)
1769.       this.drain();
1770.     this.buf[pos++]= (byte)b; }
ObjectOutputStream$BlockDataOutputStream (Java IO 1.5)
1874.   public void writeByte(int v) throws IOException {
1875.     if(this.pos >= MAX_BLOCK_SIZE)
1876.       this.drain();
1877.     this.buf[pos++]= (byte)v; }
```

Fig. 2. Cloned methods that have no contextual diffs

To distill useful clones for a given refactoring, we must raise clone analysis to a higher level of abstraction, and must examine the program context in which clones occur. The program context that affects computation of clones includes program elements referenced in cloned methods (e.g. fields being accessed, methods being invoked), associated properties of these program elements (e.g. data type of the field, return type of the method), and control and data flow surrounding cloned code fragments. Contextual differences must be identified and understood to correctly do refactorings on clones.

In this paper, we propose an approach and implement it as a tool called CloneDifferentiator [27] to help developers identify and analyze contextual differences of clones. It captures context of clones using a Program Dependence Graph (PDG) [6]. It compares PDGs of clones using a graph differencing algorithm to automatically identify contextual differences of clones. Furthermore, it offers querying and filtering support to enable developers distill useful clones of interest, and a GUI to visually inspect clones and their contextual differences.

We evaluated the effectiveness of our approach and the CloneDifferentiator tool in two empirical studies aiming at refactoring JavaIO library and Eclipse JDT-model unit-test

suites. Our studies show that our approach is able to distill a small number of useful clones for various refactoring tasks, and thus reduces the efforts of post-detection analysis of clones for refactorings. We make the following contributions in this paper:

1. We identify contextual differences of clones that must be identified and understood to correctly refactor clones.
2. We present an automated approach to help developers distill useful clones for a given refactoring task by identifying and analyzing contextual differences of clones.
3. We report two empirical studies and demonstrate the performance, accuracy and effectiveness of our approach for post-detection analysis of clones for refactorings.

The rest of this paper is structured as follows. Section II discusses contextual analysis of clones for refactoring. Section III describes our CloneDifferentiator approach. Section IV reports our empirical studies. Section V discusses related work. Section VI discusses threats to validity of our approach. Finally, we conclude with ideas for future work.

II. CONTEXTUAL ANALYSIS OF CLONES

Suppose John ponders if it might be possible to refactor Java NewIO library to remove code duplication reported in [12]. One of the specific refactorings that he is interested in is to pull up cloned methods from subclasses into superclass. For the candidate clones that he is looking for are *cloned methods that occur in sibling classes* and *appear to perform the same computation*.

John uses CloneMiner [3] for clone detection; CloneMiner reports 98 clone sets in NewIO library for Java 5; each clone set consists of 2 – 50 cloned methods. He then uses CloneAnalyzer [31] to inspect the detected clones. As CloneAnalyzer offers little help in identifying candidate clones for his pull-up method refactoring, John has to manually inspect all detected clones one by one; he resorts to Java Source Compare of Eclipse IDE to determine the differences between cloned methods.

```
PipedOutputStream (Java IO 1.5)
36.     private PipedInputStream sink;
101.    public void write(int b) throws IOException {
102.      if(this.sink == null)
103.        throw new IOExcetion("...");
104.      this.sink.receive(b); }
PipedWriter (Java IO 1.5)
25.     private PipedReader sink;
103.    public void write(int c) throws IOException {
104.      if(this.sink == null)
105.        throw new IOException("...");
106.      this.sink.receive(c); }
```

Fig. 3. Differential statements

```
ObjectInputStream.read(byte[] buf, int off, int len) (JavaIO 1.5)
806.  if(buf==null) {
807.    throw new NullPointerException("...");}
809.  int endoff=off+len;
810.  if(off<0||len<0||endoff>buf.length || endoff<0) {
811.    throw new IndexOutOfBoundsException();}
ObjectInputStream.readFully(byte[] buf, int off, int len) (JavaIO 1.5)
976.  int endoff=off+len;
977.  if(off<0||len<0||endoff>buf.length || endoff<0) {
978.    throw new IndexOutOfBoundsException();}
```

Fig. 4. Missing branch and statements

After inspecting 69 clone sets, John identifies a candidate clone set of seven cloned methods (see Fig. 1). Thus, John tries to remove these cloned methods using Eclipse's refactoring support. However, Eclipse reports an error that field *hb* referred in the cloned methods has a different data type in different subclasses. For example, the data type of *CharBuffer.hb* is *char[]*, while the type of *LongBuffer.hb* is *long[]*. The type difference of *hb* prevents the cloned methods *hasArray()* from being pulled up into the superclass *Buffer*. So *pulling* up *hb* into *Buffer* would lead to errors in other parts of buffer subclasses.

Note that in the source code of the subclasses of *Buffer*, the declaration of field *hb* is far away from (about 660 lines of codes in between) the declaration of method *hasArray()*. Furthermore, the textually identical appearance of the cloned methods makes John easily ignore the type differences of field *hb* in subclasses of *Buffer*, until his refactoring attempt failed.

The type difference of field *hb* of different buffer subclasses is a simple example of what we call *differential statements*, i.e. statements that appear in similar control and data flow contexts, but perform different computations. Fig. 3 presents another example of differential statements from Java IO library. The control and data flow of the cloned methods are identical, but the two methods perform different computations. The field *sink* refers to the different fields *PipedOutputStream.sink and PipedWriter.sink*, respectively. The types of the two fields are also different, *PipedInputStream* versus *PipedReader*. Thus, *sink.receive()* is different in these two methods. In fact, many clones in Java IO have such differential field-access and method-invocation statements, which result from parallel inheritance hierarchies for processing byte data and char data respectively.

Two other important types of contextual differences of clones are *missing statements* (i.e. statements that appear in some cloned methods but not others) and *missing/partially-matched branches* (i.e. missing or inconsistent branches among cloned methods). Fig. 4 shows an example. The method *read()* checks if *buf* is *null*, and creates and throws a *NullPointerException* if *buf* is *null*. *readFully()* does not have such explicit checking. This example shows a common inconsistent program style in Java IO for validating input parameters and handling exceptions.

Once contextual differences of clones are identified, they can help distill useful clones for a given refactoring task. For example, developers who are interested in pulling-up cloned methods can formulate a query searching for cloned methods that are in sibling classes and have **no** contextual differences. The cloned methods in Fig. 1 and Fig. 3 will not be returned by the query due to the existing differential statements, even if they look identical. But they are returned by the query searching for clones that can be replaced by generic method. Here is another example: after removing the differences due to inconsistent program styles in Fig. 4, programmers can extract parameter-validity-checking logic into a utility method.

Clearly, given large numbers of clones, manual contextual analysis of clones is impractical. The question then becomes: *how can we precisely capture context of clones and automatically identify contextual differences of clones*?

III. THE APPROACH

We propose an automatic approach and tool called CloneDifferentiator that help developers identify and analyze contextual differences of clones.

A. Overview

CloneDifferentiator analyzes cloned methods detected by one of the existing code clone detectors, such as CloneMiner [3]. Clone detectors usually group cloned methods to form clone sets. The cloned methods called clone instances in each clone set are pair-wise similar to each other, according to similarity metrics used by the clone detector.

CloneDifferentiator raises the level of clone analysis to Program Dependency Graph (PDG) [6]. It represents context of clones using PDG. PDG allows to precisely capture not only program elements being referenced in cloned methods and associated properties of these program elements, but also data and control flow information in cloned methods. Given PDGs of cloned methods in a clone set, our tool uses graph differencing algorithm to compare PDGs of clones. It automatically detects contextual differences of clones, in terms of PDG-based differential statements or blocks, missing statements or blocks, and missing or partially-matched branches. Because these contextual differences are identified at PDG level, they contain fine-grained static semantic information about the differences, including type of statement, program elements being referenced and its associated properties, and data/control flow discrepancies.

Our CloneDifferentiator tool [27] stores its contextual analysis results of clones in a relational database. Stored information includes PDGs of cloned methods and the instances of different types of contextual differences of these clones. Clone-Differentiator is equipped with a set of simple filters for filtering clones based on the types and number of their contextual differences. Furthermore, it allows developers to formulate task-specific queries in terms of which clones and what types of contextual differences he would like to inspect.

B. Representing Context of Clones as PDG

Let us first discuss why we adopt PDG to represent the context of clones. We then describe the PDG representation that our current CloneDifferentiator tool adopts for contextual analysis of clones.

1) Why PDG: A Program Dependence Graph (PDG) [6] is a static representation of the control and data flow through a program. CloneDifferentiator adopts PDG as the internal representation of context of clones and computes contextual differences of clones at PDG level.

Fig. 5 presents the contextual differences of a pair of cloned methods *listFiles(FilenameFilter)* and *listFiles(FileFilter)* in a *CloneDiff Compare Editor* (see [27] for more details on the introduction of the *CloneDiff Compare Editor* of our tool).

CloneDifferentiator reports that the cloned code fragments (inside light-grey box) of the two methods have a pair of *differential parameters* (highlighted in light blue background), because the two parameters declare different data types (interface *FilenameFilter* versus interface *FileFilter*). Our tool also reports that the two cloned methods have a pair of *differential method invocations* (*FilenameFilter.accept() versus FileFilter accept()*, highlighted in red background). Furthermore, the method *listFiles(Filenamefilter)* has an additional array-access statement (*ss[i]*, highlighted in yellow double underline and italic font), i.e. a *missing array-access statement* that *listFiles(FileFilter)* does not have.

CloneDifferentiator also detects the differences in the control flow of the two methods: the program control flows from the matched branch statement (*i<ss.length*) directly to the unmatched branch (*filter==null*) in *listFiles(FilenameFilter*), while in *listFile(FileFilter*) the control flows first to the instantiation of a *File* object and then to the unmatched branch (*filter==null*). CloneDifferentiator reports this difference as *partially-matched branches filter==null* (highlighted in green and accent font in the two methods respectively).

Compared with the textual differencing results of the two cloned methods (see Fig. 6), the contextual differences that our tool reports are clearly much more precise. *Java Source Compare* reports some textual differences between the code block of *listFiles(FileFilter)* and the code block of *listFiles(FilenameFilter)*. As textual differencing compares clones as lines and chars, it cannot report the subtle difference in the branch statements *filter==null*. And textual differencing also reports the two *new File()* statements are different.

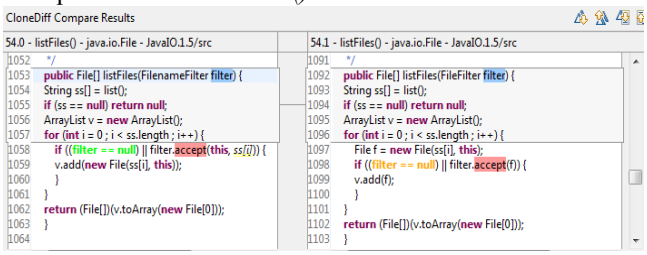

Fig. 5. Contextual differences in CloneDiff Compare Editor

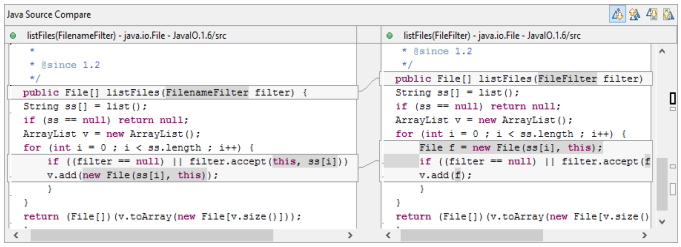

Fig. 6. Textual differences in Java Source Compare

Syntactic differencing techniques, such as change distilling [7] that compares Abstract Syntax Tree (AST), are more robust than textual differencing, e.g., they can detect type differences of the parameters of the two *listFiles* methods. However, syntactic differencing is still sensitive to arbitrary syntactic decisions a developer made while developing a program. For example, statements *File f = new File(); v.add(f)* and *v.add(new File())* result in different ASTs, but they yield the same PDG. Furthermore, AST-based differencing techniques are agnostic of control and data flow through a program, and thus cannot detect contextual differences resulted from control and data flow, such as the differences between the two *filter==null* statements in the two *listFiles* methods.

2) Wala PDG: In general, the nodes of a PDG consist of three categories of program statements constructed from the source code: simple statements, expressions, and control points. A control point represents a point at which a program branches, loops, enters or exits a procedure. The edges of a PDG encode the data and control dependencies between program statements.

Our CloneDifferentiator tool uses Wala [34] to generate PDGs of cloned methods. Wala is a static analysis library for Java. It is important to note that using Wala for PDG generation in our CloneDifferentiator tool is only an implementation choice because Wala is open source and publicly available. Furthermore, we conducted empirical studies on Java software systems. Our CloneDifferentiator approach is not limited to any specific PDG generation tools, nor is it limited to analyzing clones in Java software systems.

We use Wala-PDG to capture the context of cloned methods, including program elements referenced in cloned methods, associated properties of these program elements, and control and data flow in cloned methods. Wala-PDG supports three categories of bytecode-like program statements constructed from source code: simple statement, control point, and expression. Simple statements include field read/write *FGET/FPUT*, method invocation *INVOKE*, unary and binary operation (*negate*, *add*, *minus*, *multiply*), compare statement (>, <, !=), arrayload/store *ARRAYLOAD/ARRAYSTORE*, type checking *INSTANCEOF*, type casting *CAST*, object creation *NEW*, and exception throwing *THROW*. Control points include branching BRANCH and switching SWITCH. Expressions include parameter PARAM and constant CONST. Different types of Wala-statements consist of different sets of properties, such as identifier, data type, operator code, and operand.

Fig. 7 shows a partial Wala-PDG for the cloned method *PipedWriter.write(int):void* listed in Fig. 3. The two PDG nodes *FGET<this.sink : PipedReader>* correspond to the two *this.sink* at lines 104 and 106 respectively, i.e. access the field *sink* of *this* object, and the data type of *sink* is *PipedReader*. The node *INVOKE<virtual : PipedReader.receive(I)V : void>* corresponds to *receive()* at line 106, i.e. the invocation of the method *PipedReader.receive(I)V*, and this invocation is a *virtual* invocation and its return type is *void*. The node *NEW<IOException>* corresponds to *new IOException("...")* at line 105, i.e. the instantiation of an object *IOException*. The node *THROW<$>* corresponds to the *throw* statement at line 105. The node *BRANCH<ne : this.sink : null>* corresponds to the *if* statement at line 104; it examines whether field *this.sink* is not equal (*operator-code ne*) to *null*. The node *PARAM<c:I>* corresponds to the parameter *c* whose data type is *int*.

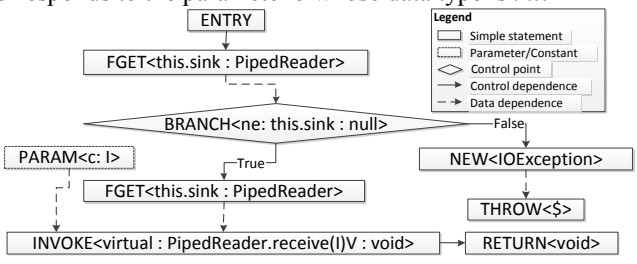

Fig. 7. Wala-PDG example: PipedWriter.write(int):void

C. Detecting Contextual Differences of Clones

Given a clone set of n cloned methods $\{m_1,\ldots,m_n\}$, let PDG_i and PDG_j be the PDGs of cloned methods m_i and m_j ($i \neq j; 1 \leq i,j \leq n$), CloneDifferentiator applies a graph differencing algorithm to pair-wisely compare PDG_i and PDG_j and detect contextual differences between cloned methods m_i and m_j. Its current implementation uses GenericDiff [28] (a configurable graph matching framework) for comparing Wala-PDGs. GenericDiff is an efficient graph matching algorithm. It can produce high-quality PDG differencing results for contextual analysis of clones. Furthermore, GenericDiff is configurable and it supports a quick development of graph comparator for various graph-based program models. This allows us to easily adapt the implementation of our tool for working with other PDG generation tools.

GenericDiff is an approximate graph matching algorithm. Given two graphs to be compared, it matches graph nodes and edges based on both property similarities of graph nodes and structural similarities between graph nodes from the two graphs. GenericDiff reports a domain independent symmetric difference between two input graphs: a set of matched graph nodes and edges that exist in both graphs, and two sets of unmatched graph nodes and edges that exist in only one of the two graphs. CloneDifferentiator interpret GenericDiff's PDG differencing results in terms of meaningful contextual differences of code clones. Interested readers are referred to our GenericDiff technical report [30] for the details about how we configure GenericDiff for comparing PDGs.

Our tool CloneDifferentiator reports the following three contextual differences of clones:

1) Differential Statements or Blocks: Differential statements (blocks) represent a pair of Wala-PDG statements (blocks of statements), one from each cloned method, that appear in similar control and data flow context in the two cloned methods, but perform different computations. Differential statements are because of the differences in:

- The *methods being invoked* in method invocation (*INVOKE*) statements and the *fields being accessed* in field access (*FGET*/*FPUT*) statements can be different.
- The *operator-code* of method invocation, binary operation, compare, and branch statements can be different. Furthermore, the *data type* of field-read/write, method invocation, type checking, type casting, object creation, array-load/store, and parameter statements can be different.
- The *operand* of binary-operation, compare, and branch elements or the *value* of constant elements can be different.

Fig. 8 presents an example of differential statements from our empirical study on JavaIO library. The control and data flow of the cloned methods *readArray()* and *readOrdinaryObject()* is similar. But CloneDifferentiator detects two pairs of differential statements between the two methods: the method being invoked in the two method-invocation statements is different: *Array.newInstance()* versus *ObjectStreamClass.newInstance()*; the *operator-code* of these two method invocations is different: *static* versus *virtual*; the constant *operand* of the branch statements is different: *TC_ARRAY* versus *TC_OBJECT*.

Differential blocks are similar to differential statements; the only difference is that a block consists of a sequence of statements, i.e. a subgraph of the PDG of the cloned method. Fig. 9 presents an example of the two cloned test methods from our empirical study on Eclipse JDT unit tests. Both test methods are used to test creating a member in a class. However, our tool detects that the cloned methods have a pair of differential blocks, which reveal the differences of the two test methods in retrieving and creating different types of class members. That is, *test002()* tests creating a field, while *test003()* tests creating a method.

2) Missing Statements or Blocks: Missing statements (blocks) represent statements (blocks of statements) appearing only in one of the clones, not in the other.

Fig. 10 presents an example of missing statements between the two cloned methods. CloneDifferentiator detects that one of the cloned method *read()* has two additional statements that the method *peek()* does not have: a binary operation that adds *pos* by 1 and a field-write statement that updates *pos* with the new value. These two missing statements reveal the key differences between *read()* and *peek()*, which have different computations.

Fig. 11 shows an example of missing statements and block between the two cloned methods. CloneDifferentiator detects that one of the cloned methods *SequenceInputStream.read()* has an additional block that the method *PipedOutputStream.write()* does not have, while *PipedOutputStream.write()* has some statements that *SequenceInputStream.read()* does not have. The two methods share the logic of validating input parameters so that they are reported as cloned method by CloneMiner. Note that we define a block that contains at least **6** continuous unmatched statements as an unmatched block, which spans at least 2 lines of code, and constitutes noticeable clones [16].

```
ObjectInputStream.readArray(boolean) (JavaIO 1.5)
1581.    ......
1582.    if(bin.readByte() != TC_ARAY)
1583.       throw new StreamCoruptedException()
1592.    array = Array.newInstance(ccl, len)
1593.    ......
ObjectInputStream.readOrdinaryObject(boolean) (JavaIO 1.5)
1689.    ......
1690.    if(bin.readByte() != TC_OBJECT)
1691.       throw new StreamCoruptedException()
1698.    obj = desc.isInstantiable() ? desc.newInstance() ...;
1699.    ......
```

Fig. 8. Differential statements

```
CreateMemberTests.test002() (JDT unit test)
70.  ......
71.  compilationUnit = getCompilationUnit(..., "E.java");
76.  IField sibling = type.getField("j");
77.  type.createField("int i;", sibling, true, null);
78.  ......
CreateMemberTests.test003()(JDT unit test)
89.  ......
90.  compilationUnit = getCompilationUnit(..., "Anno.java");
95.  IMethod sibling =type.getMethod("foo", new String[]{});
96.  type.createMethod("String bar()", sibling, true, null);
97.  ......
```

Fig. 9. Differential block

```
ObjectInputStream$BlockDataInputStream.peek()(JavaIO 1.5)
3960.    if(blkmode) {
3961.       return (end>=0) ? (buf[pos] & 0xFF) : -1;
3962.    } else { ... }
ObjectInputStream$BlockDataInputStream.read() (JavaIO 1.5)
3963.    if(blkmode) {
3964.       return (end>=0) ? (buf[pos++] & 0xFF) : -1;
3965.    } else { ... }
```

Fig. 10. Missing statements

```
SequenceInputStream.read (byte b[], int off, int len) (JavaIO 1.5)
181.   ..... // cloned codes
189.   else if (len == 0) {
190.     return 0;
191.   }
192.   int n = in.read(b, off, len);
193.   if (n <= 0) {
194.      nextsStream();
195.      return read(b, off, len);
196.   }
197.   return n;
PipedOutputStream.write(byte b[], int off, int len) (JavaIO 1.5)
122.   ..... //cloned codes
129.   else if (len == 0) {
130.      return 0;
131.   }
132.   sink.receive(b, off, len);
```

Fig. 11. Missing statements and block

3) Missing or Partially Matched Branches: Missing branches represent control points that appear only in one of the cloned method but not the other, while partially matched branches reveal the inconsistencies between a sequence of control points between the cloned methods.

Fig. 4 presents an example of a common inconsistent program style in JavaIO that results in a missing branch between cloned methods. Fig. 12 presents a typical example of another common inconsistent program style in Java IO that results in partially-matched branches for parameter validity checking. In this example, the cloned methods perform a sequence of similar but also different parameter validity checkings. They have two pairs of matched parameter validity checkings (*off<0* and *len<0*), but they check different expressions (*len>buf.length-off* versus *off+len>buf.length*) to ensure that the sum of *off* and *len* is less than the length of *buf*. Furthermore, *StringBufferInput-Stream.read()* has one more checking (*off+len<0*), which is actually unnecessary, since it always evaluates to be false.

```
ByteArrayInputStream.read(byte[] buf, int off, int len) (JavaIO1.5)
160.   ......
161.   } else if(off<0||len<0||len>buf.length-off) {
162.     throw new IndexOutOfBoundsException();}}
163.   ......
StringBufferInputStream.read(byte[] buf, int off, int len) (JavaIO1.5)
95.    ......
96.    } else if(off<0||len<0||
97.           off+len>buf.length||off+len<0) {
98.      throw new IndexOutOfBoundsException();}}
99.    ......
```

Fig. 12. Partially-matched branches

IV. EVALUATION

We evaluated our CloneDifferentiator approach and tool on two Java software systems: JavaIO library and Eclipse JDT-

model unit tests. JavaIO library 1.5 contains 101 classes and 1038 methods. Our tool uses CloneMiner for clone detection. CloneMiner detects 103 clone sets; each set consists of 2−15 cloned methods. JDT-model unit tests (jdt.core.tests.model) 3.6.1 contain 336 test suites and 10740 test methods. For JDT-model unit tests, CloneMiner detects 961 clone sets; each set consists of 2 – 35 cloned methods.

We then use our tool to identify and analyze contextual differences of the detected clones. Our evaluation aims at gaining insights into these questions: How often is the context of clones different? Do contextual differences of clones in different systems manifest different characteristics? Can such contextual differences help distill useful clones for refactorings?

In this section, we report the results on using our tool on refactoring of clones in JavaIO and JDT-model unit tests. We also evaluate runtime performance and accuracy.

A. *Characteristics of Contextual Differences of Clones in JavaIO and JDT-model Unit Tests*

Our quantitative analysis suggests that in both systems the detected cloned methods usually have various types and instances of contextual differences. The differences are often subtle. The contextual differences of cloned methods residing in various systems may manifest various characteristics, due to the nature of the subject systems.

Table 1 reports the statistics of contextual differences of cloned methods in JavaIO. Each row in the table represents a type of contextual difference discussed in Section III.C. Column "#diff" lists the number of instances of a particular type of contextual difference; column "#cloneset(cs)" lists the number of clone sets that have at least one instance of a particular type of contextual difference; column (#diff/#cs) lists the average instance number of the different types of contextual differences per clone set.

For example, the first row of Table 1 shows that our tool identifies 329 instances of differential statements in 79 clone sets; on average one clone set has 4.2 instances of differential statements. Note that one clone set can have more than one type of contextual differences. Thus, the sum of column "#cloneset" is greater than the number of clone sets that CloneMiner reports.

CloneDifferentiator reports in total 849 (sum(#diff)) instances of different types of contextual differences in the cloned methods of JavaIO library. For a particular type of contextual difference, each clone set has on average at least one instance (#diff/#cs) of that type of difference, for example 1.2 instances of partially-matched branches per clone set. Each clone set has on average three (sum(#cs)/103) types and eight (sum(#diff)/103) instances of contextual differences.

The most common type of contextual differences of the cloned methods of JavaIO are missing statements (392 instances), followed by differential statements (329 instances). These two types of contextual differences account for about 85% of all 849 instances of differences. Missing blocks (68 instances) and differential blocks (13 instances) account for about 10% of all 849 instances. Partially-matched branches (PartialMatch Brch, 21 instances) and missing branches (26 instances) account for a very small percentage (5%) of all 849 instances. Overall, differences between cloned methods of JavaIO are usually subtle, but sometimes they can be notable.

TABLE 1. STATISTICS OF CONTEXTUAL DIFFERENCES IN JAVAIO 1.5

Type	# diff	#cloneset(cs)	#diff/#cs
Differential Statemt	329	79	4.2
Differential Block	13	10	1.3
Missing Statement	392	80	4.9
Missing Block	68	44	1.6
Missing Branch	26	18	1.5
PartialMatch Brch	21	17	1.2

TABLE 2. STATISTICS OF CONTEXTUAL DIFFERENCES IN JDT-MODEL TESTS

Type	#diff	#cloneset(cs)	#diff/#cs
Differential Statemt	7900	931	8.5
Differential Block	101	90	1.1
Missing Statement	6761	666	10.2
Missing Block	1217	460	2.64
Missing Branch	512	203	2.5
PartialMatch Brch	13	12	1.1

Table 2 presents the statistics of contextual differences of cloned methods in JDT-model unit tests. The cloned methods of JDT-model unit tests have much more (sum(#diff)=16504) instances of contextual differences. This is not surprising because JDT-model-unit-tests is a much bigger project and it has nine times more clone sets than JavaIO. However, the percentages of different types of contextual differences in the cloned methods of JDT-model-unit-tests are roughly similar to those of JavaIO. Furthermore, the percentages of clone sets that have a particular type of contextual differences are also roughly similar to those of JavaIO.

One important difference is that the cloned methods of JDT-model unit tests have on average more types (sum(#cs)/961) and instances (sum(#diff)/961, #diff/#cs) of contextual differences than the cloned methods of JavaIO. This is mainly because the JDT-model unit test methods are usually longer than the methods of JavaIO.

The other difference is that almost all clone sets of JDT-model unit tests have differential statements (931/961, 96.8%). This is due to the existence of a large amount of differential constant statements. In fact, this reflects a common practice in writing unit tests in which similar tests are developed to test different input values (see examples in Section IV.C).

B. *Refactoring JavaIO Library*

In this study, we are interested in identifying clones that can be refactored using *Folwer's refactorings* (e.g. *extract method, pull up method*) or *Java generics*, thus reducing code duplication.

1) Refactoring Clones Using Folwer's Refactorings: Many of Folwer's refactorings are concerned with code duplication [8]. Folwer's refactorings usually target at identical or almost identical cloned methods, which can be removed by refactorings such as extract method, pull up method.

To identify candidate clones for Folwer's refactorings, we formulate the following two queries searching for:

1. The cloned methods that have no contextual differences, i.e., the PDGs of such cloned methods are perfectly matched;

2. One of the cloned methods is "part of" the other cloned methods, i.e., the PDG of one cloned methods is the subgraph of the PDG of the other. So only one of the clone methods has missing statements, blocks, and/or branches.

The first query for cloned methods without contextual differences returns 3 pairs of cloned methods. Fig. 2 presents one pair of these cloned methods. The two methods perform identical computation: they write eight lower-order bits of the input argument to the output stream and ignore the 24 high-order bits. These cloned methods can be refactored by replacing the body of one method with a call to the other method. Note that CloneDifferentiator does not report that the cloned methods in Fig. 2 have differential parameters (*b* versus *v* highlighted in italic font), because the two parameters declare the same data type (*int*), and the simple identifier difference does not affect the computation performed by the cloned methods.

The second query returns 1 pair of cloned methods, *PipedInputStream.checkStateForReceive()* and *PipedReader.-receive(int)*. Both *PipedInputStream* and *PipedReader* need to perform the same checking of pipe state in several places before starting receiving data. The developer of *PipedInputStream* recognized the repetition of this state checking and extracted the state checking logic into the method *PipedInputStream.check-StateForReceive()*. In contrast, the developer of *PipedReader* did not extract the state checking logic from *PipedReader.receive(int)* into a separate method. As a result, CloneDifferentiator detects that the state checking method *PipedInputStream.checkState-ForReceive()* is "part of" the method *PipedReader.receive(int)*. Identifying this "part of" relation between the cloned methods suggests the opportunity to extract method.

Overall, only very few cloned methods (4/103) in JavaIO library represent identical or almost identical code clones that can be removed by Folwer's refactorings.

2) Relaxed Queries for Folwer's Refactorings: To identify more candidate clones for Folwer's refactorings, we relaxed the two queries given in the last section, by allowing the cloned methods to have a small number of contextual differences. In particular, relaxed queries allow the cloned methods to contain a maximum of six instances of differential statements, missing statements, missing branches, and/or partially-matched branches.

The relaxed queries return 21 more pairs of cloned methods. Four pairs of these cloned methods have *only differential operator-code and/or operand statements*. For example, the cloned methods *LineNumberInputStream.read(byte,int,int)* and *InputStream.read-(byte,int,int)* are all the same but a pair of *differential operator-code* method invocations (*special for LineNumberInputStream.-read()* versus *virtual for InputStream.read()*). The class *InputStream* declares a template method [10] *read(byte,int,int)* that defines the skeleton of reading bytes from the input stream. *InputStream.read(byte,int,int)* calls the abstract method *InputStream.read()*, and the subclasses of *InputStream* (e.g., *LineNumberInputStream*) must implement the abstract method *InputStream.read()* to read the next byte of data from a specific type of input stream. However, the subclass *LineNumberInput-Stream* duplicates the template method *read(byte,int,int)* in itself, which deviates from the intent of Template Method [10]. So this duplicated *LineNumberInputStream.read(byte,int,int)* should be removed.

There are also 17 pairs of cloned methods that reveal two types of inconsistent program styles in JavaIO library. These inconsistent programming styles result in a certain amount of missing statements, missing branches and/or partially-matched branches in the cloned methods. Fig. 4 and Fig. 12 present two examples of these two types of inconsistent program styles, i.e., different ways to validate input parameters and handle exceptions. Investigating the cloned methods that have such inconsistent programming styles suggests that after we reconcile inconsistencies among these cloned methods, these cloned methods could also be refactored, for example, by extracting validity checking of input parameters into a utility method.

3) Refactoring Clones Using Java Generics: Java generics support developing common data structures and algorithms differing only in the types on which they operate.

To identify candidate clones that can be replaced with Java generic methods or classes, we formulate the following two queries based on the two characteristics of JavaIO library:

1. JavaIO supports reading and writing data of different primitive data types (e.g., short, char, int, long, double). Thus, we formulate a query to identify cloned methods that have *only differential typecasting statements*;
2. JavaIO supports reading and writing both byte (8-bit) data and char (16-bit) data. Thus, we formulate a query to identify cloned methods that have *only differential field-access and method-invocation statements*.

```
Bits.getChar(byte[] b, int off) (JavaIO 1.5)
26.  return (char)((b[off+1]&0xFF)<<0) +
27.                          (b[off+1]&0xFF)<<8));
Bits.getShort(byte[] b, int off) (JavaIO 1.5)
31.  return (short)((b[off+1]&0xFF)<<0) +
32.                          (b[off+1]&0xFF)<<8));
```

Fig. 13. Differential typecast statements

It is surprising that the first query returns only 1 pair of cloned methods *Bits.getChar()* and *Bits.getShort()*, as shown in Fig. 13. Our inspection of JavaIO library reveals that this is because JavaIO mainly relies on bitwise shift and logic operations instead of explicit typecasting for processing data of different primitive data types.

The second query returns 26 pairs of cloned methods, including the cloned methods *PipedOutputStream.write(int)* and *PipedWriter.write(int)* listed in Fig. 3. Although the two methods are textually identical, they actually have three instances of differential statements. Similar types of differential statements also exist in other cloned methods returned by our query, such as methods *connect()*, *flush()*, *close()* of *PipedOutputStream* and *PipedWriter*. These differential statements reveal that the overall data and control flows are similar in many methods of two types of output classes (*PipedOutputStream* versus *PipedReader*), but the specific data operations are different.

In fact, these differential statements are resulted from parallel inheritance hierarchies in Java IO for processing byte data (input/output streams) and char data (readers/writers) respectively. JavaIO initially supported only byte data. To support char data, a separate hierarchy of classes was later developed. The two parallel hierarchies share many similar data structures and processing steps. They can be restructured into one hierarchy using Java generic classes and methods.

C. Refactoring Eclipse JDT-model Unit Tests

Unit tests typically contain groups of test methods that form variations for a common testing purpose and therefore are similar to each other. In this study, we are interested in identifying clones that can be refactored using seed values, state machine, or assume/assert invariants testing patterns [32], thus reducing code duplication among test methods and improving test-case reuse.

1) Refactoring Clones Using Seed Values: A traditional unit test method tests a unit with fixed input value. It is necessary to develop several tests with variant input values to achieve a good coverage of the unit under test. These tests are often similar but also different in the input values that are actually used for testing.

We would like to refactor such duplicated unit tests into parameterized unit tests, using *seed-values* [32] (a pattern for parameterized unit testing [25]) to provide concrete input values. To that end, we formulate a query searching for cloned test methods that have *only differential-operand and/or differential-constant-value statements*.

```
JavaSearchTests.testEnum06()
3672.  method = getMethod("setRole", new String[] {"Z"});
3673.  search(method, REFERENCES, ...);
JavaSearchTests.testVarargs03()
3702.  method = getMethod("vargs", new String[] {"QSt"});
3703.  search(method, ALL_REFERENCES, ...);
```

Fig. 14. Seed values

Our query returns 173 pairs of cloned test methods in Eclipse JDT-model unit tests. Fig. 14 presents one of them. The two methods test the Java search API with different search entities (*setRole* versus *vargs*, and *Z* versus *QSt*) and search options (*REFERENCES* versus *ALL_REFERENCES*).

Investigating these 173 cloned test methods returned by our query suggests that cloned test methods for testing searching and formatting features usually have differential-operand and/or differential-constant-value statements. Such cloned test methods can be parameterized, using their differential operands and constant values as seed values, so that parameterized unit tests can verify the unit under test for a set of input values.

2) Refactoring Clones Using State Machine: Eclipse JDT-model provides APIs for programmatically rewriting Java programs, such as creating a member (e.g. field or method) in a class. The corresponding unit tests for these APIs often share similar control and data flows but also differ in the program-rewriting APIs under tests.

We would like to refactor such cloned test methods into parameterized unit tests to enforce the flow of testing logics, using *state machine* [32] (another pattern for parameterized unit testing) to encapsulate program-rewriting APIs under test. To that end, we formulate a query searching for cloned methods that have differential method-invocation statements and/or differential blocks of program-rewriting APIs.

```
ASTRewritingStatementsTest.testSwitchStatement7()
3956.  ListRewrite listRewrite = rewrite.getListRewrite(...)
3957.  listRewrite.replace(assignment, switchCase, null);
3959.  String preview = evaluateRewrite(cu, rewrite);
ASTRewritingStatementsTest.testSwitchStatement9()
4098.  ListRewrite listRewrite = rewrite.getListRewrite(...)
4099.  listRewrite.remove(assignment, null);
4100.  listRewrite.insertAfter(switchCase,assignment,null):
4102.  String preview = evaluateRewrite(cu, rewrite);
```

Fig. 15. State machine

Our query returns 153 pairs of cloned test methods. Fig. 9 presents an example of these cloned methods testing different ways to create a class member (field versus method). Fig. 15 presents another example testing different ways (*replace* versus. *remove* and *insertAfter*) to rewrite switch statements in an AST.

Investigating these 153 cloned test methods reveals that cloned test methods for testing program-rewriting APIs often invoke different program-rewriting APIs (e.g., *createField*, *createMethod*, *remove*, *insert*, *copy*, *move*, *replace*), or invoke some program-rewriting APIs in different orders. The invocations of these program-rewriting APIs can be encapsulated into state machines [32] that can programmatically rewrite Java programs. Then, given state machines of a set of program rewriting APIs and a parameterized unit test, one can use testing framework, such as Pex [33], to instantiate sequences of state transitions to test the relevant program-rewriting APIs.

3) Refactoring Clones Using Assume and Assert Invariants: JDT-model unit tests often assert similar sets of properties that a unit under test should hold before and after exercising the unit under test, for example whether the parent AST node is not null or the class contains a specific member.

We would like to extract these similar assertions before and after exercising the unit under test into assume/assert invariants. To that end, we formulate a query searching for cloned test methods that have *at least two matched method-invocations of assertxxx() methods*. Note that Eclipse JDT-model unit tests name assertion methods in the form of *assertxxx()*.

```
ASTTest.testArrayCreation()
7994.  final ArrayCreation x = this.ast.newArrayCreation();
7995.  assertTrue(this.ast.modificationCount > previousCount);
7997.  assertTrue(x.getAST() == this.ast);
7998.  assertTrue(x.getParent ()== null);
ASTTest.testSwitchStatement()
5891.  final SwitchStatemnt x = this.ast.newSwitchStatement();
5892.  assertTrue(this.ast.modificationCount > previousCount);
5894.  assertTrue(x.getAST() == this.ast);
5895.  assertTrue(x.getParent() == null);
```

Fig. 16. Assume invariant

Our query returns 137 pairs of cloned test methods. Fig. 16 presents one example, in which the matched same method-invocation statements are highlighted. The two methods test the APIs of two different types of AST nodes, array creation versus switch statement. However, they share the same set of

assertions about the AST under test, i.e. *modificationCount* > *prevCount*, *x.getAST()==this.ast*, and *x.getParent()==null*.

Checking these 137 cloned test methods reveals that cloned test methods for JDT AST/DOM APIs often contain similar sets of assertions before and/or after exercising the AST/DOM API under test. These similar assertions can be extracted as assume and/or assert invariants. These invariants can not only remove duplicate assertions across test methods, but ensure consistent verification of test assumptions and results.

D. Runtime Performance and Accuracy

Our evaluation has been performed on a machine with a Core I5 CPU of 2.6GHz, 4G RAM, and Windows 7. For the largest subject system Eclipse JDT plugins, detecting clones took about 8 minutes; generating the PDGs of clones took about 30 minutes; computing the PDG differences of 14520 clone pairs in 961 clone sets took about 220 minutes; automatically summarizing the PDG differences of clones took about 70 minutes.

By analyzing the clones detected by CloneMiner [3], the accuracy of our tool is good. We manually inspected the PDG comparison results of randomly-selected 10% of all the analyzed clone pairs. The precision (i.e., the percentage of the correctly reported matches) and the recall (i.e., the percentage of matched reported) of our tool is around 94% and 96%.

```
ObjectInputStream$BlockDataInputStream (Java IO 1.5)
2717.    readByte() {
2718.      int v = read();
2719.      if(v<0) throw new EOFException();
2720.      return (byte)v;}
ObjectInputStream$BlockDataInputStream (Java IO 1.5)
2549.    peekByte() {
2550.      int v = peek();
2551.      if(v<0) throw new EOFException();
2552.      return (byte)v;}
```

Fig. 17. An erroneous match: read() vs. peek()

The false positives mainly consist of the erroneous matches of the same-type parameters, field accesses and method invocations. Fig. 17 presents a typical example, in which the PDGs of the two cloned methods differ only in the method signature of the two *INVOKE* statements (*peek()* vs. *read()*). As both method invocations return *int* and they have the identical neighboring statements in the two PDGs, the PDG differencing erroneously matches *peek()* and *read()* invocations. However, *peek()* and *read()* perform different computations (See Fig. 10). The false negatives (i.e., missed matches) are due to many matches that prevent statements matching to *real* counterparts.

V. Related Work

Researchers have presented many techniques to detect code clones based on token [3][15][16][22], AST [4][13], and PDG [9][11][19][21]. Roy and Cordy [24], and Koschke [20] provide comprehensive surveys of existing clone detection techniques. The difference between clone detection techniques and our CloneDifferentiator is that clone detectors report which parts of the system are similar, while our tool identifies how these similar parts are different. However, using differencing techniques for clone detection is impractical, because it requires a pair-wise differencing of any two code fragments, which results in a combinational explosion of differencing operation. On the other hand, clone detectors often use reduced representation of program (such as encoding PDG in a vector space [9]) to scale up to large systems. Such reduced representation makes it impossible to compute differences of clones during clone detection process. Our tool complements clone detectors by helping developers identify and analyze contextual differences of clones in post-detection analysis.

Clone detectors report a large number of clones in large systems, while it is common that only a small number of them is actually useful for specific maintenance tasks, such as refactorings. The effectiveness of clone detection techniques has usually been evaluated in terms of precision and recall metrics of the detected clones, such as in the quantitative evaluation of clone detectors reported in Roy et al. [23], and Bellon et al. [5].

Researchers proposed clone analysis approaches to aiding the interpretation and management of software clones. For example, Genimi [26] uses a scatter plot to visualize code clones detected by CCFinder [16], and it also computes several code metrics of clones to aid clone analysis. Balazinska et al. [1][2] define a clone classification based on the differences between the token sequences forming the clones. This clone classification helps to measure the reengineering opportunities of clones. CP-Miner [22] finds bugs based on inconsistent identifiers between clones. One major limitation of these approaches is that they examine only the information of clones, ignoring the program context in which clones occur.

Other approaches perform simple syntactic analysis of clones to aid the understanding of clones. For example, Kapser and Godfrey [17] classify code clones through the syntactic analysis of locality of clones. Jiang et al. [14] consider the inner most syntactic constructs that enclose clones as contexts and identify three types of contextual inconsistencies in clones. In contrast, our CloneDifferentiator raises contextual analysis of code clones to PDG, which captures much more contextual information than existing work. Besides, CloneDifferentiator exploits efficient graph differencing algorithm to systematically detect contextual differences of clones.

Query-based approaches have been proposed for supporting program understanding and maintenance. Xing and Stroulia [29] proposed to detect and analyze change patterns in software evolution by querying elementary design changes reported by UMLDiff. Zhang et al. [31] present the CloneAnalyzer tool that supports query-based filtering of code clones. However, CloneAnalyzer does not support contextual analysis and differencing of clones as CloneDifferentiator does.

Our recent work [28] presents design and key concepts of GenericDiff framework. CloneDifferentiator is a new application of GenericDiff for comparing PDGs of clones to detect their contextual differences. It performs automatic contextual analysis of code clones based on PDG differencing results of GenericDiff. We present our tool in [27] about the details of implementation challenges and visualization features of the tool, while this paper describes fundamental concepts of our approach, discusses in detail contextual differences of clones, and also reports two empirical studies.

VI. THREATS TO VALIDITY

Now our tool CloneDifferentiator uses CloneMiner [3] to detect cloned methods. The surveys [20][24] on clone detection tools suggest that clones reported by different techniques may vary due to the diverse nature of detection techniques and similarity metrics. Further studies are required to evaluate our approach with respect to different clone detection techniques.

CloneDifferentiator now compares intra-method PDGs of cloned methods, excerpted from Wala-SDG of the system. It does not consider inter-method PDGs because it assumes that two different methods being invoked in cloned methods would perform different computation. This assumption holds in most cases and allows scalable and efficient contextual analysis of clones. CloneDifferentiator can be easily adapted to analyze inter-procedure PDGs around cloned methods, because inter-procedure PDGs are available in Wala-SDG.

In this paper, we showed that contextual differences of clones are useful for distilling useful clones for refactorings. Cloning information has also been used for other types of software maintenance tasks, such as bug detection [14][15][22]. Further studies are required to investigate the usefulness of our approach for other types of maintenance tasks.

VII. CONCLUSION AND FUTURE WORK

We cannot understand code clones without understanding their differences precisely. In this paper, we proposed and implemented an automated approach to help developers identify and analyse contextual differences of clones. Our evaluation shows that our tool can reduce the effort of post-detection analysis of clones for refactorings by supporting developers to distill useful clones of interest based on contextual differences of clones.

In the future, we plan to conduct more empirical studies to enrich our taxonomy of contextual differences of clones. We believe this can open new opportunities to refine existing token-based clone definitions from a new perspective (i.e., contextual differences of clones). This can enhance the usefulness of cloning information in many software maintenance tasks.

REFERENCES

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring" *WCRE* 2000, pp. 98-107, 2000.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities". *METRICS 1999*, pp. 292-303, 1999.

[3] H.A. Basit, and S. Jarzabek, "A data mining approach for detecting higher-level clones in software". *IEEE Trans. Soft. Eng*., vol. 35(4):497-514, 2009.

[4] I.D. Baxter, A. Yahin, L. Marcelo, M. SantAnna, and L. Bier, "Clone detection using abstract syntax trees", *ICSM 1998,* pp. 368-377, 1998.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools". *IEEE Trans. Soft. Eng*. vol. 33(9): 577-591, 2007.

[6] J. Ferrante, J.K. Ottenstein, and J.D. Warren, "The program dependence graph and its use in optimization". *ACM Trans. Program. Lang. Syst.* vol. 9(3): 319-349, 1987.

[7] B. Fluri, M. Wuersch, M. Plnzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction". *IEEE Trans. Soft. Eng*, vol. 33:712-743, 2007.

[8] M. Fowler, *Refactorin*g: *Improving the Design of Existing Code*, Addison-Wesley, 1999.

[9] M. Gabel, L. Jiang, and Z. Su, Scalable detection of semantic clones. *ICSE 2008*, pp. 321-330, 2008.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[11] Y. Higo, and S. Kusumoto, "Enhancing quality of code clone detection with program dependency graph". *WCRE 2009,* pp. 315-316, 2009.

[12] S. Jarzabek, and S. Li, "Eliminating redundancies with a composition with adaptation meta-programming technique". *ESEC/FSE 2003*, pp. 237-246, 2003

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones", *ICSE 2007*, pp. 96-105, 2007

[14] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs". *ESEC/FSE 2007*, pp. 55-64, 2007.

[15] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?", *ICSE 2009*, pp. 485-495, 2009.

[16] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code". *IEEE Trans. Soft. Eng.*, vol. 28(7):654-670, 2002

[17] C. Kapser, and M.W. Godfrey, "Aiding comprehension of cloning through categorization". *IWPSE 2004*, pp. 85-94, 2004.

[18] E. Kodhai, A. Perumal, and S. Kanmani, "Clone detection using textual and metric analysis to figure out all types of clones", *International Journal of Computer Communication and Information System (IJCCIS)*, vol. 2(1): 99-103, 2010.

[19] R. Komondoor, and S. Horwitz, "Using slicing to identify duplication in source code", *SAS* 2001, pp. 40-56, 2001.

[20] R. Koschke, "Survey of research on software clones", *duplication, redundancy, and similarity in Software*, 2006.

[21] J. Krinke, "Identifying similar code with program dependence graphs", *WCRE 2001*, pp. 301-310, 2001.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code". *IEEE Trans. Soft. Eng.*, vol. 32(3):176-192, 2006.

[23] C.K. Roy and J.R. Cordy, "Scenario-based comparison of clone detection techniques", *ICPC 2008*, pp. 153-162, 2008.

[24] C.K. Roy and J.R. Cordy, *A survey on software clone detection research*. Technical Report 2007-541, Queen's University, 2007.

[25] N. Tillmannn, and W. Schulte, "Parameterized unit tests". *ESEC/FSE 2005*, pp. 253-262, 2005.

[26] Y. Ueda, S. Kamiya, S. Kusumoto, and K. Inoue, Gemini: Maintenance support environment based on code clone analysis, *IEEE METRICS 2002*, pp. 67-76, 2002.

[27] Z. Xing, Y. Xue, and S. Jarzabek, "CloneDifferentiator: Analyzing clones by differentiation", *ASE 2011*, pp. 576-579, 2011.

[28] Z. Xing, "GenericDiff: Model comparison with GenericDiff". *ASE 2010*, pp. 135-138, 2010.

[29] Z. Xing, and E. Stroulia, "Refactoring detection based on UMLDiff change-facts queries". *WCRE 2006*, pp. 263-274, 2006.

[30] Z. Xing, *GenericDiff: A general framework for model comparison*. Technical report, National University of Singapore, 2011, http://www.comp.nus.edu.sg/~pat/publications/gendiff.pdf.

[31] Y. Zhang, B.A. Basit, S. Jarzabek, D. Anh, and M. Low, "Query-based filtering and graphical view generation for clone analysis". *ICSM 2008*, pp. 376-385, 2008.

[32] Parameterized Test Patterns for Microsoft Pex, http://research.microsoft.com/en-us/projects/pex/patterns.pdf, 2012.

[33] Parameterized Unit Test with Microsoft Pex: http://research.microsoft.com/enus/projects/pex/pextutorial.pdf, 2012.

[34] WALA: http://wala.sourceforge.net/wiki/index.php/Main_Page, 2012.