Detecting Differences across Multiple Instances of Code Clones

Yun Lin[1,2], Zhenchang Xing[3], Yinxing Xue[4], Yang Liu[3], Xin Peng[1,2], Jun Sun[5], Wenyun Zhao[1,2]
[1] School of Computer Science, Fudan University, China
[2] Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China
[3] School of Computer Engineering, Nanyang Technological University, Singapore
[4] School of Computing, National University of Singapore, Singapore
[5] Singapore University of Technology and Design, Singapore
{linyun, pengxin, wyzhao}@fudan.edu.cn, {zcxing, yangliu}@ntu.edu.sg,
dcsxuey@comp.nus.edu.sg, sunjun@sutd.edu.sg

ABSTRACT

Clone detectors find similar code fragments (i.e., instances of code clones) and report large numbers of them for industrial systems. To maintain or manage code clones, developers often have to investigate differences of multiple cloned code fragments. However, existing program differencing techniques compare only two code fragments at a time. Developers then have to manually combine several pairwise differencing results. In this paper, we present an approach to automatically detecting differences across multiple clone instances. We have implemented our approach as an Eclipse plugin and evaluated its accuracy with three Java software systems. Our evaluation shows that our algorithm has precision over 97.66% and recall over 95.63% in three open source Java projects. We also conducted a user study of 18 developers to evaluate the usefulness of our approach for eight clone-related refactoring tasks. Our study shows that our approach can significantly improve developers' performance in refactoring decisions, refactoring details, and task completion time on clone-related refactoring tasks. Automatically detecting differences across multiple clone instances also opens opportunities for building practical applications of code clones in software maintenance, such as auto-generation of application skeleton, intelligent simultaneous code editing.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance

General Terms

Algorithm, Human Factors

Keywords

Code clone, Program differencing, Human study

1. INTRODUCTION

Code clones are similar code fragments in software. Many techniques have been proposed to detect code clones based on token similarity (e.g., CCFinder [17], CloneMiner [1] and CloneDetective [16]), Abstract Syntax Tree (AST) similarity (e.g., CloneDR [2], Deckard [14]), or Program Dependence Graph similarity (e.g., [19, 22]). These clone detectors can detect not only textually identical clones (Type I) but also parameterized clones (Type II) and gapped clones (Type III) [3].

Textually identical clones refer to code fragments with differences only in whitespace, layout and comments. Parameterized clones refer to syntactically identical code fragments except for differences in identifers, literals and types. Gapped clones refer to copied fragments with further modifications such as changed, added or removed statements. A code clone often appears in multiple places in the system, i.e., having multiple instances. Table 1 presents a four-instances code clone in JHotDraw system. Note that the clone instances are different in 5 places. Some clone instances may be the same in one place but can be different in other places. In a specific place, one clone instance may have parameterized and/or gapped differences with other instances.

Differences in whitespace, layout and comments are not important. But detecting and analyzing differences in parameterized and gapped clones (parameterized and gapped differences thereafter) such as those shown in Table 1 is important to manage and maintain code clones, for example to identify refactoring opportunities [40], to detect bugs [15], to support change propagation in code clones [12, 24].

Existing program differencing techniques compare a pair of cloned code fragments at a time. Pairwise differencing of multiple clone instances cannot provide developers with a complete picture of how multiple clone instances are different. Developers have to manually investigate where clone instances are the same, where they are different, and if different, which instances are different and what types of differences. Unless developers take notes during investigation, differences across multiple clone instances are not readily available when developers need them in software maintenance tasks. Furthermore, manual analysis of clone differences makes it impossible to automatically use clone differencing results in software maintenance.

In this paper, we present an automatic approach to detecting differences across multiple instances of a code clone. Our approach *MCIDiff* (Multi-Clone-Instances Differencing) takes as input multiple clone instances in a clone set. It parses source code of each clone instance into a sequence of tokens enriched with relevant syn-

ICSE '14, May 31 - June 7, 2014, Hyderabad, India

Table 1: Code Clone Example in JHotDraw System

0	PertProject.java:	NetProject.java:	DrawProject.java:	SVGProject.java:
1	**public void** read(File f) {	**public void** read(File f) {	**public void** read(File f) {	**public void** read(File f) {
2	InputStream in = **null**;	InputStream in = **null**;	InputStream in = **null**;	InputStream in = **null**;
3	**try** {	**try** {	**try** {	**try** {
4	in = **new** BufferedInputStream(	in = **new** BufferedInputStream(	in = **new** BufferedInputStream(	in = **new** BufferedInputStream(
5	**new** FileInputStream(f));	**new** FileInputStream(f));	**new** FileInputStream(f));	**new** FileInputStream(f));
6	NanoXMLLiteDOMInput domi =	NanoXMLLiteDOMInput domi =	NanoXMLLiteDOMInput domi =	DOMInput domi =
7	**new** NanoXMLLiteDOMInput(	**new** NanoXMLLiteDOMInput(	**new** NanoXMLLiteDOMInput(	**new** NanoXMLDOMInput(
8	(PertFactory)view.	(NetFactory)view.	(DrawFigureFactory)view.	(SVGFigureFactory)view.
9	getDOMFactory(), in);	getDOMFactory(), in);	getDOMFactory(), in);	getDOMFactory(), in);
10	domi.openElement		domi.openElement	
11	(‘‘PertDiagram’’);		(‘‘PlasmaDraw’’);	
12	...	...	...	...
13	domi.closeElement();		domi.closeElement();	
14	}	}	}	}
15	**catch**(InterruptedException e) {	**catch**(InterruptedException e) {	**catch**(InterruptedException e) {	**catch**(InterruptedException e) {
16	...	...	...	...
17	}	}	}	}
18	}	}	}	}

tactic information (e.g., data type of a variable). *MCIDiff* computes Longest Common Subsequence (LCS) of multiple clone instances. Then, it analyzes the LCS to determine differential ranges across multiple clone instances, and identifies similar tokens in differential ranges. *MCIDiff* produces as output a list of matched and differential multisets (i.e., bags) of corresponding tokens of multiple clone instances. Differential multisets summarize differences that can be found in parameterized and gapped clones.

MCIDiff has been implemented as an Eclipse plugin with GUI that allows developers to interactively investigate code clones and their differences. Our *MCIDiff* tool can be downloaded at http://www.se.fudan.edu.cn/clonepedia/diff/. Our evaluation shows that *MCIDiff* can build a detailed and accurate summary of differences (over 97.66% precision and over 95.63% recall) across multiple instances of a code clone. It can help developers to perform clone-related refactoring tasks more easily and achieve better performance in refactoring decisions, refactoring details, and task completion time.

The rest of the paper is structured as followed. Section 2 presents a motivating example of our work. Section 3 describes our approach to detecting clone differences. Section 4 presents our *MCIDiff* tool. Section 5 discusses empirical evaluation of our approach. Section 6 discusses our on-going work that exploits *MCIDiff* in three software development tasks. Section 7 reviews related work. Finally, we conclude and discuss the ideas for future work.

2. MOTIVATING EXAMPLE

Table 1 shows four similar code fragments contained in *read*() method of four classes (i.e., *PertProject*, *NetProject*, *DrawProject*, and *SVGProject*) of JHotDraw applications. Such similar code fragments are called code clones [29]. Each code fragment represents a clone instance in the clone set (see Section 3.1 for a formal definition of clone set).

Although the four *read()* methods are very similar, they demonstrate four typical types of differences across multiple clone instances.

All Parameterized. All clone instances are different from one another. For example, at line 8, four clone instances cast return value of a method call *view.getDOMFactory*() to four different types respectively, i.e., *PertFactory*, *NetFactory*, *DrawFigureFactory*, and *SVGFigureFactory*.

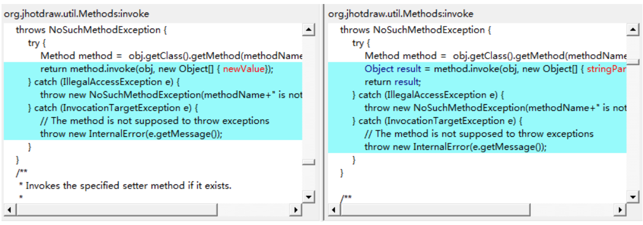
Same+Parameterized. Some clone instances have no differences, others have parameterized differences. For example, at line 6, *PertProject.read*(), *NetProject.read*() and *DrawProject.read*() have no difference (the data type of variable *domi* is the same, i.e., *NanoXMLLiteDOMInput*), while *SVGProject.read()* is different from the other three methods (the data type of variable *domi* is *DOMInput*). That is, *SVGProject.read*() and the other three methods have parameterized differences. Similar case can be found at line 7.

Same+Gapped. Some clone instances have no differences, others have gapped differences. For example, at line 13, both *PertProject.read()* and *DrawProject.read()* call *domi.closeElements()*, while *NetProject.read()* and *SVGProject.read()* do not call. When comparing *PertProject.read()/DrawProject.read()*) and *NetProject.read()/SVGProject.read()*), these clones have gapped differences.

Same+Parameterized+Gapped. Some clone instances have no differences, some have parameterized differences, and others have gapped differences. For example, at line 10/11, neither *NetProject.read()* nor *SVGProject.read()* call *domi.openElements()*. *PertProject.read()* and *DrawProject.read()* call *domi.openElements()* with different parameters (*PertDiagram* versus *PlasmaDraw*, i.e., parameterized difference). When comparing *NetProject.read()/SVGProject.read()* and *PertProject.read()/DrawProject.read()*, these clones have gapped difference.

Multiple clone instances may have several differences. In our example, four clone instances are different in five places (lines 6, 7, 8, 10/11, and 13). Some clone instances may be the same in one place but can be different in other places. For example, *PertProject.read*(), *NetProject.read*() and *DrawProject.read*() are the same at lines 6 and 7, but they have parameterized difference at lines 8 and 10/11 and gapped difference at line 13. Furthermore, one clone instance may have different types of differences against other clone instances in a specific place. For example, at line 11 *PertProject.read*() and *DrawProject.read*() have parameterized difference, *PertProject.read*() and *NetProject.read*() have gapped difference, while *NetProject.read*() and *SVGProject.read*() have no difference.

Clearly, it is not an easy task to identify the above differences through manual examination or pairwise differencing across multiple clone instances. Developers have to manually determine where



```
org.jhotdraw.util.Methods:invoke
    throws NoSuchMethodException {
        try {
            Method method =  obj.getClass().getMethod(methodName
            return method.invoke(obj, new Object[] { newValue});
        } catch (IllegalAccessException e) {
            throw new NoSuchMethodException(methodName+" is not
        } catch (InvocationTargetException e) {
            // The method is not supposed to throw exceptions
            throw new InternalError(e.getMessage());
        }
    }
    /**
     * Invokes the specified setter method if it exists.
     *
```

```
org.jhotdraw.util.Methods:invoke
    throws NoSuchMethodException {
        try {
            Method method =  obj.getClass().getMethod(methodName
            Object result = method.invoke(obj, new Object[] { stringPar
            return result;
        } catch (IllegalAccessException e) {
            throw new NoSuchMethodException(methodName+" is not
        } catch (InvocationTargetException e) {
            // The method is not supposed to throw exceptions
            throw new InternalError(e.getMessage());
        }
    }

    /**
```

Figure 1: Incomplete Syntactic Units of Code Clones

clone instances are the same, where they are different, and if different, which instances are different and what types of differences. Furthermore, all manually-collected difference information is not readily available when it is required for a software maintenance task. Developers have to recall the information from memory or based on notes (if any), or they may have to redo manual investigation again.

3. APPROACH

We now discuss our approach to detecting parameterized and gapped differences across multiple clone instances.

3.1 Input

Our approach takes as input a clone set as reported by a code clone detector. A *clone set CS* consists of n ($n \geq 2$) *clone instances* ci_k ($1 \leq k \leq n$). Each clone instance ci_k represents a code fragment. The code fragments of any pair of clone instances ci_k and ci_l ($k \neq l$) in a clone set CS are "similar" according to similarity metric defined by the clone detector.

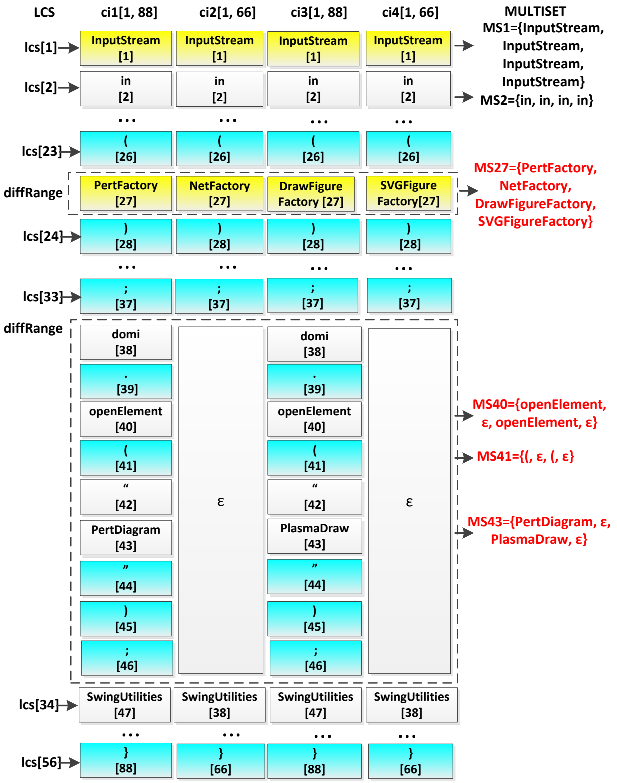
Code clones reported by a clone detector (e.g., token-based clone detector) may not be complete syntactic units. For example, in Figure 1 code fragments highlighted in green are code clones reported by *CloneDetective* in JHotDraw applications. These code fragments break syntactic boundary of try-block. We assume that source code of the software is available. *MCIDiff* builds AST of the method containing reported code clones. It then analyzes the AST to find inner most syntactic unit (i.e., least common ancestor) that contains incomplete syntactic units of cloned code fragment. The corresponding syntactically complete code fragment is considered as code fragment of clone instances for differencing.

Given a clone instance ci_k, we transform its code fragment into a *token sequence* $ts(ci_k)$. According to Java Language Specification [9], there are five types of tokens in Java program, i.e., keyword (e.g., if, while), separator (e.g., (, {), operator (e.g., +, -), literal (e.g., "abc", 10) and identifier. We further classify identifers as type, method/field/variable, and label. As such, our approach considers six categories of tokens, i.e., *Type*, *Method/Field/Variable/Literal*, *Label*, *Keyword*, *Separator*, and *Operator*.

Type, Method/Field/Variable/Literal and Label tokens have name attribute. Keyword, Separator and Operator tokens have symbol attribute. Method/Field/Variable/Literal tokens have attached data type as token attribute. Our differencing algorithm compares category and attribute of tokens to determine their correspondences. We consider Method/Field/Variable/Literal as same-category of tokens. That is, methods, fields, variables and literals are comparable as long as they share compatible data type.

3.2 Output

Given n token sequences $ts(ci_k)$, representing n clone instances ci_k of a clone set, *MCIDiff* reports a list of *multisets* (i.e., bags) of corresponding tokens. Each multiset MS consists of n corresponding tokens (one from each clone instance), i.e., $MS = \{t_k \mid t_k \in ts(ci_k)$ *and* $1 \leq k \leq n\}$. A token t_k from a clone instance ci_k can only be in one multiset. The token t_k can be ϵ, i.e., a placeholder token, which means that no real token from the given clone instance corresponds to tokens from other clone instances in the given multiset.



LCS	ci1[1, 88]	ci2[1, 66]	ci3[1, 88]	ci4[1, 66]	MULTISET
lcs[1]	InputStream [1]	InputStream [1]	InputStream [1]	InputStream [1]	MS1={InputStream, InputStream, InputStream, InputStream}
lcs[2]	in [2]	in [2]	in [2]	in [2]	MS2={in, in, in, in}
	…	…	…	…	
lcs[23]	([26]	([26]	([26]	([26]	
diffRange	PertFactory [27]	NetFactory [27]	DrawFigure Factory [27]	SVGFigure Factory[27]	MS27={PertFactory, NetFactory, DrawFigureFactory, SVGFigureFactory}
lcs[24]	) [28]	) [28]	) [28]	) [28]	
	…	…	…	…	
lcs[33]	; [37]	; [37]	; [37]	; [37]	
diffRange	domi [38]	ε	domi [38]	ε	
	. [39]		. [39]		
	openElement [40]		openElement [40]		MS40={openElement, ε, openElement, ε}
	([41]		([41]		MS41={(, ε, (, ε}
	" [42]		" [42]		
	PertDiagram [43]		PlasmaDraw [43]		MS43={PertDiagram, ε, PlasmaDraw, ε}
	" [44]		" [44]		
	) [45]		) [45]		
	; [46]		; [46]		
lcs[34]	SwingUtilities [47]	SwingUtilities [38]	SwingUtilities [47]	SwingUtilities [38]	
	…	…	…	…	
lcs[56]	} [88]	} [66]	} [88]	} [66]	

Figure 2: Results of *MCIDiff*

Given a multiset MS of corresponding tokens, if all pairs of tokens t_k and t_l ($k \neq l$) in the multiset are identical (i.e., all tokens in this multiset have exactly the same category and attribute), the multiset is a matched multiset. Otherwise, the multiset is a differential multiset. *MCIDiff* further label a differential multiset as follows. If $\exists\, t_k = t_l$, *MCIDiff* attaches label "same" to the multiset. If $\exists\, t_k \neq t_l$, *MCIDiff* attaches label "parameterized" to the multiset. If $\exists\, t_k = \varepsilon$, *MCIDiff* attaches label "gapped" to the multiset.

Figure 2 shows partial token sequences of the four clone instances shown in Table 1. We index the four clone instances from left to right as ci_1 through ci_4. Different colors illustrates different categories, yellow for Types, canyon for Separators, and gray for Method/Field/Variable/Literal. We omit datatype attribute of Method/Field/Variable/Literal tokens for clarity.

The first multiset MS_1 that *MCIDiff* reports in this clone set is a matched multiset $\{InputStream_1$, $InputStream_2$, $InputStream_3$, $InputStream_4\}$. It consists of four identical Type tokens *InputStream* from clone instances ci_1, ci_2, ci_3, and ci_4 respectively. Several reported differential multisets are: MS_{27}=$\{PertFactory_1$, $NetFactory_2$, $DrawFigureFactory_3$, $SVGFigureFactory_4\}$, MS_{40}=$\{openElement_1$, ϵ_2, $openElement_3$, $\epsilon_4\}$, MS_{43}=$\{PertDiagram_1$, ϵ_2, $PlasmaDraw_3$, $\epsilon_4\}$. These differential multisets identify "All Parameterized", "Same+Gapped", and "Same+Parameterized+Gapped" differences discussed in motivating example.

3.3 Differencing Multiple Clone Instances

Algorithm 1 presents our *MCIDiff* algorithm. The algorithm takes

Algorithm 1 MCIDiff

```
Require: $List < TS > tss, List < Range > ciRanges$
Ensure: $List < Multiset < Token >>$
 1: $results \leftarrow \varnothing$
 2: $List < int[] > lcs \leftarrow computeLCS(tss, ciRanges)$;
 3: for each token index array $tokenIndices$ in $lcs$ do
 4:     $multiset \leftarrow$ tokens at corresponding token index $tokenIndecies$ from token sequences in $tss$;
 5:     $results.add(multiset)$;
 6: end for
 7: $List < Range[] > listDiffRanges \leftarrow determineDiffRanges(ciRanges, lcs)$;
 8: for each $diffRange$ in $listDiffRanges$ do
 9:     derive a new list of non-empty ranges $newciRanges$ by removing $\epsilon$ ranges in $diffRnge$;
10:     if $1 < newciRanges.size < ciRanges.size$ then
11:         $results.add(MCIDiff(tss, newciRanges))$;
12:     else if $newciRanges.size() = ci.Ranges.size$ then
13:         $results.add(identifySimilarTokens(tss, newciRanges))$;
14:     else if $newciRanges.size = 1$ then
15:         for each token index $i$ in the non-empty $range$ in $newciRanges$ do
16:             $multiset \leftarrow \varnothing$
17:             find $range$'s corresponding token sequence $ts$ in $tss$;
18:             $multiset.add(ts[i])$;
19:             $results.add(multiset)$;
20:         end for
21:     end if
22: end for
23: return $results$;
```

two parameters. The parameter *tss* consists of *n* token sequences *TS* derived from code fragments of *n* clone instances. The parameter *ciRanges* consists of a list of to-be-compared *Ranges* of token sequences. Each *Range* specifies a range of tokens (index of *first* and *last* tokens) of a specific clone instance (identified by clone instance index ci_{index} in the clone set). This range will be compared with corresponding ranges of tokens of other clone instances. We represent a *Range* of a specific clone instance as $ci_{index}[first, last]$. Initially, *ciRanges* contains a *Range* for each clone instance. *Range* at $ciRanges[k]$ is initialized as $ci_k[1, tss[k].length]$, i.e., the whole token sequence of clone instance ci_k. The algorithms generates as output a list of multiset of corresponding tokens (can be ϵ) in *tss*.

Note that finding the global optimal alignment for *N* sequences has been shown to be an NP-complete problem [34]. *MCIDiff* adopts progressive alignment approach to compute an approximate solution. Progressive alignment approach has been widely applied to align multiple DNA sequences [10]. It has been shown to be an efficient and accurate approach when sequences are similar. In this work, we apply progressive alignment to align token sequences of code clones.

3.3.1 Computing LCS

MCIDiff first computes a Longest Common Subsequence (LCS) across input token sequences *tss* within given ranges *ciRanges* (line 2). The computation starts with two longest token sequences, and then proceeds to shorter token sequences one at a time. That is, $computerLCS()$ first computes the LCS of the two longest token sequences, and then compares the resulting LCS with the third longest token sequences, and so on till all token sequences are compared. Given two token sequences, $computeLCS()$ implements dynamic programming algorithm [13] to compute the LCS between them.

$computeLCS()$ considers only identical tokens, i.e., tokens with exactly the same category and attributes. Note that our token is enriched with relevant syntactic information as token attributes. This can improve accuracy of LCS computation. For example, if two variable tokens have the same name but different data types, $computeLCS()$ will not consider them as identical tokens.

The results of LCS computation is a list of token index arrays. The length of LCS is the number of token index arrays in the list. Each token index array contains indicies of $ciRanges.size()$ corresponding tokens, one from each toke sequence specified by *ciRanges*.

Take the four token sequences shown in Figure 2 as an example. $computeLCS()$ reports the LCS as a list of 56 token index arrays, i.e., the length of LCS across the four clone instances is 56. The first token index array (i.e., $lcs[1]$) $[1, 1, 1, 1]$ indicates that the first token *InputStream* of four clone instances correspond to each other. Similarly, the last token index array (i.e., $lcs[56]$) $[88, 66, 88, 66]$ indicates that the 88th token of clone instance ci_1, 66th token of ci_2, 88th token of ci_3, and 66th token of ci_4 correspond to each other.

3.3.2 Determining differential ranges

Next, *MCIDiff* determine differential ranges in token sequences that cannot be fully matched across all clone instances (line 7). *determineDiffRanges* essentially determines gaps between subsequent token indices in the LCS of token sequences specified by *ciRanges*.

For the four token sequences shown in Figure 2, $determineDiff-Ranges$ reports five differential ranges, two of which are shown in the figure. For example, $[27-27]$ of clone instance ci_1, ci_2, ci_3, and ci_4 is a differential range between $lci[23]$ and $lcs[24]$. We represent such a differential range as $[ci_1[27, 27]$, $ci_2[27, 27]$, $ci_3[27, 27]$, $ci_4[27, 27]]$. This range represents unmatched tokens *PertFactory*, *NetFactory*, *DrawFigureFactory*, and *SVGFigureFactory* in four clone instances respectively.

$[ci_1[38, 46], ci_2[\epsilon], ci_3[38, 46], ci_4[\epsilon]]$ is another differential range. This range represents unmatched token subsequences *[domi, ., openElement, (, “, PertDiagram, ”,), ;]*, ϵ, *[domi, ., openElement, (, “, PlasmaDraw, ”,), ;]*, and ϵ in four clone instances respectively. Note that this differential range consists of only two non-empty token subsequences from clone instances ci_1 and ci_3. Clone instances ci_2 and ci_4 contain placeholder ϵ subsequences, i.e., no real tokens from ci_2 and ci_4 in this differential range. Furthermore, token subsequences of clone instances ci_1 and ci_3 can be further matched if we compare only ci_1 and ci_3.

3.3.3 Processing differential ranges

If a differential range consisting of m $(1 < m < ciRanges.size())$ non-empty token subsequences, $MCIDiff()$ will recursively process these token subsequences by calling $MCIDiff()$ with a list of new to-be-compared ranges derived from these non-empty token subsequences (lines 9 and line 11). For example, from the differential range $[ci_1[38, 46], ci_2[\epsilon], ci_3[38, 46], ci_4[\epsilon]]$, *MCIDiff* derives a *newciRanges* consisting of two *Ranges*, i.e., $ci_1[38, 46]$ and $ci_3[38, 46]$. It then calls $MCIDiff()$ recursively with this *newciRanges*.

Such recursive calls will identify as many identical tokens as possible across clone instances before we use similarity heuristics to determine correspondences between non-identical tokens across clone instances. For example, the recursive call to $MCIDiff()$ with to-be-compared ranges $[ci_1[38, 46], ci_3[38, 46]]$ identifies the LCS ($[domi, ., openElement, (, “, ”,), ;]$) between the two token subsequences, and leaves only two identifiers *PertDiagram* in ci_1 and *PlasmaDraw* in ci_3 unmatched. This can improve efficiency and accuracy of similarity-based matching of non-identical tokens.

The number of non-empty token subsequences in a differential range can be the same as $ciRanges.size()$, for example, {*PerFactory*, *NetFactory*, *DrawFigureFactory*, *SVGFigureFactory*} in the initial call to *MCIDiff*, and {*PertDiagram*, *PlasmaDraw*} in the recursive call to *MCIDiff*. This indicates that *MCIDiff* cannot identify identical tokens across all token sequences in the range *ciRanges*. In such cases, *MCIDiff* calls *identifySimilarTokens*() (line 13) to determine correspondences between non-identical tokens based on their similarities (see Section 3.4).

3.3.4 Generating multisets of corresponding tokens

First, *MCIDiff* generates multisets of corresponding tokens based on the LCS (lines 3-6). It visits each token index array in the LCS list and creates a multiset of corresponding tokens accordingly. For example, for the first token index array $lcs[1]$=[1, 1, 1, 1], *MCIDiff* creates a multiset $\{InputStream_1, InputStream_2, InputStream_3, InputStream_4\}$. For the last token index array $lcs[56]$=[88, 66, 88, 66], *MCIDiff* creates a multiset $\{\}_1, \}_2, \}_3, \}_4\}$.

Note that the LCS determined in the recursive call to *MCIDiff*() consists of tokens from a subset of all token sequences, for example token sequences ci_1 and ci_3 in the recursive call with *Ranges* $[ci_1[38, 46], ci_3[38, 46]]$. Thus, multisets derived from such LCSs will consist of tokens only from this subset of token sequences. For the rest of token sequences that are not in this subset (for example ci_2 and ci_4), they essentially contribute ϵ tokens to the multiset. For example, based on the LCS [*domi*, ., *openElement*, (, “, ”,)] of two token subsequences in the ranges $[ci_1[38, 46], ci_3[38, 46]]$, *MCIDiff* creates multisets such as $\{domi_1, \epsilon_2, domi_3, \epsilon_4\}$.

Next, *identifySimilarTokens*() generates multisets of corresponding tokens as it determines correspondences of non-identical tokens in a given differential range.

Finally, if a differential range consists of only one non-empty token subsequence, *MCIDiff* creates multisets that consist of tokens only from this subsequence (lines 15). That is, all other clone instances contribute ϵ to these multisets.

3.4 Identifying Correspondences Between Similar Tokens

Given a list of token sequences (*tss*) and a list of differential ranges (*diffRanges*), *identifySimilarTokens*() (see Algorithm 2) processes toke sequences from the longest to the shortest (line 3). For each to-be-compared token sequence *ts*, *identifySimilarTokens*() scans it from the beginning to the end. For a not-yet-matched token *t* in *ts*, *identifySimilarTokens*() adds it as seed token to a multiset *multiset* (line 6). It then attempts to find similar tokens from the rest of token sequences (*candTSs*) iteratively (lines 7-14). In each iteration, *identifySimilarTokens*() finds a not-yet-matched token t_{cand} from all candidate token sequences that is most similar to tokens already in the multiset (line 8).

The similarity between a candidate token and tokens in the multiset $sim(t_{cand}, multiset)$ is computed as the average similarity of t_{cand} and each token $t \in multiset$, i.e., $\sum_{t \in multiset} sim(t_{cand}, t) / \mid multiset \mid$. If this similarity is above the threshold h, the candidate token t_{cand} is added to the multiset (line 12) and the candidate sequence containing t_{cand} is removed from candidate sequence list (line 13). This process continues until no more similar enough candidate token can be found (lines 9-11) or candidate sequence list is empty (i.e., all candidate sequences already contribute one token to the multiset) (line 7). If the resulting *multiset* contains two or more tokens, it is added to the results list (line 16).

Finally, *identifySimilarTokens*() scans each token sequence and identifies still-not-yet-matched tokens for which *identifySimilarTokens()* cannot find any similar enough tokens in other token sequences. It creates multisets for these still-not-yet-matched tokens in each sequence. Such multisets consist of only one real token and all other tokens are ϵ.

Algorithm 2 identifySimilarTokens

```
Require: $List < TS > tss, List < Range > diffRanges, h$
Ensure: $List < Multiset < Token >>$
 1: $results \leftarrow \varnothing$
 2: create a new list of token subsequences $tbcTSs$ from $tss$ and $diffRanges$;

 3: for each token sequence $ts$ in $tbcTSs$ (from longest to shortest) do
 4:     $candTSs \leftarrow tbcTSs.remove(ts)$;
 5:     for each not-yet-matched token $t$ in $ts$ do
 6:         $multiset \leftarrow \varnothing$; $multiset.add(t)$;
 7:         while $candTSs \neq \varnothing$ do
 8:             find a not-yet-matched token $t_{cand}$ in all candidate token sequences $candTSs$ with maximum $sim(t_{cand}, multiset)$
 9:             if $sim(t_{cand}, multiset) < h$ then
10:                 break;
11:             end if
12:             $multiset.add(t_{cand})$;
13:             remove the token sequence containing $t_{cand}$ from $candTSs$;
14:         end while
15:         mark all tokens in $multiset$ as matched;
16:         if $\mid multiset \mid > 2$ $results.add(multiset)$
17:     end for
18: end for
19: create multisets for all still-not-yet-matched tokens
20: return $results$;
```

identifySimilarTokens() only matches tokens with same category. That is, if two tokens are of different categories, their similarity is 0. Given two same-category tokens t_1 and t_2, *identifySimilarTokens*() computes their similarity ($sim(t_1, t_2)$) by comparing the attributes of tokens and relative positions in token sequences.

For Keyword (or Separator, Operator, Label) tokens, $sim_{attr} = 1$ if two tokens have the same symbol (or name), otherwise $sim_{attr} = 0$. For Type (or Method/Field/Variable/Literal) tokens, sim_{attr} is computed as Jaccard coefficient of common supertypes of Type tokens (or data (return) types of Method/Field/Variable/Literal tokens), i.e., $\mid SuperType(t_1) \cap SuperType(t_2) \mid / \mid SuperType(t_1) \cup SuperType(t_2) \mid$.

identifySimilarTokens() computes position similarity (sim_{pos}) between two tokens t_1 and t_2 by measuring their relative positions in corresponding token sequences. Given two ranges R_1 with length len_1 and R_2 with length len_2, let first index of R_1 and R_2 be fi_1 and fi_2, index of t_1 and t_2 in token sequences be p_1 and p_2, position similarity $sim_{pos}(t_1, t_2)$ is computed as: $1 - \mid (p_1 - fi_1)/len_1 - (p_2 - fi_2)/len_2 \mid$.

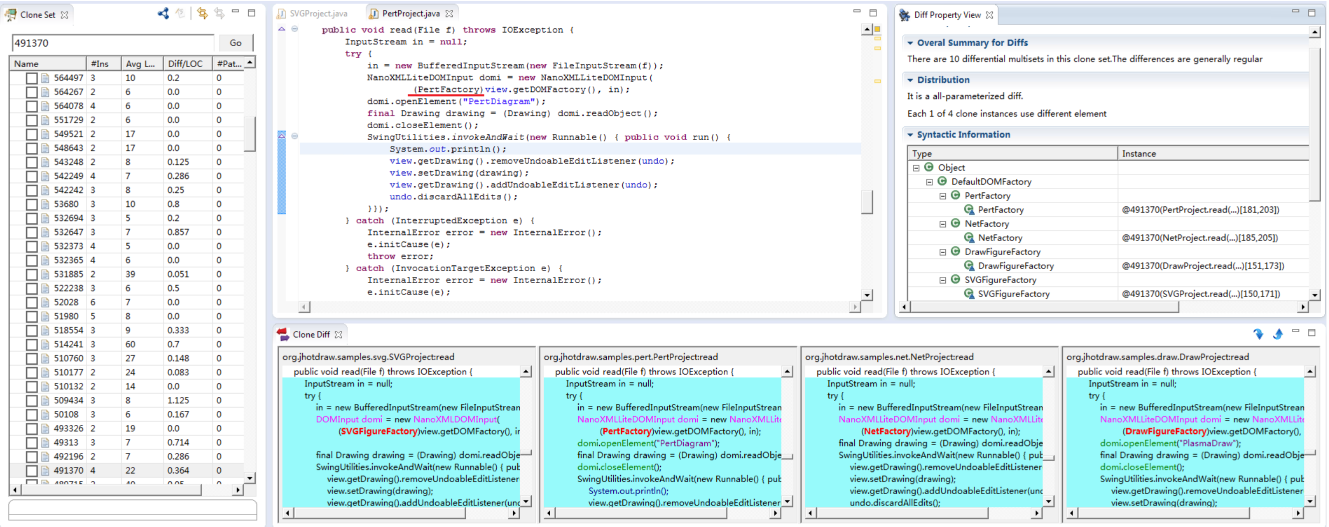
Given two same-category tokens, their overall similarity is then computed as average of their attribute similarity and position similarity, i.e., $sim(t_1, t_2) = (sim_{attr}(t_1, t_2) + sim_{pos}(t_1, t_2))/2$.

4. TOOL SUPPORT

We have implemented *MCIDiff* as an Eclipse plugin. The current implementation parses *CloneDetective*'s clone detection report as input. Note that *MCIDiff* does not make any specific assumption about clone detectors. Given a different clone detector, *MCIDiff* only needs a new parser to parse clone detection report of that clone detector.

MCIDiff consists of three views *Clone Set* view, *Clone Diff* view and *Diff Property* view for interactively inspecting code clones and their differences. Figure 3 shows a snapshot of *MCIDiff* for analyzing code clones in our motivating example.

Clone Set view lists all the clone sets reported by *CloneDetective* for a subject system. Each row corresponds to a clone set; it summarizes the number of clone instances (#Ins), average length of



491370 Go

Name	#Ins	Avg L...	Diff/LOC	#Pat...
564497	3	10	0.2	0
564267	2	6	0.0	0
564078	4	6	0.0	0
551729	2	6	0.0	0
549521	2	17	0.0	0
548643	2	17	0.0	0
543248	2	8	0.125	0
542249	4	7	0.286	0
542242	3	8	0.25	0
53680	3	10	0.8	0
532694	3	5	0.2	0
532647	3	7	0.857	0
532373	4	5	0.0	0
532365	4	6	0.0	0
531885	2	39	0.051	0
522238	3	6	0.5	0
52028	6	7	0.0	0
51980	5	8	0.0	0
518554	3	9	0.333	0
514241	3	60	0.7	0
510760	3	27	0.148	0
510177	2	24	0.083	0
510132	2	14	0.0	0
509434	3	8	1.125	0
50108	3	6	0.167	0
493326	2	19	0.0	0
49313	3	7	0.714	0
492196	2	7	0.286	0
491370	4	22	0.364	0

```
    public void read(File f) throws IOException {
        InputStream in = null;
        try {
            in = new BufferedInputStream(new FileInputStream(f));
            NanoXMLLiteDOMInput domi = new NanoXMLLiteDOMInput(
                    (PertFactory)view.getDOMFactory(), in);
            domi.openElement("PertDiagram");
            final Drawing drawing = (Drawing) domi.readObject();
            domi.closeElement();
            SwingUtilities.invokeAndWait(new Runnable() { public void run() {
                System.out.println();
                view.getDrawing().removeUndoableEditListener(undo);
                view.setDrawing(drawing);
                view.getDrawing().addUndoableEditListener(undo);
                undo.discardAllEdits();
            }});
        } catch (InterruptedException e) {
            InternalError error = new InternalError();
            e.initCause(e);
            throw error;
        } catch (InvocationTargetException e) {
            InternalError error = new InternalError();
            e.initCause(e);
```

Overal Summary for Diffs

There are 10 differential multisets in this clone set.The differences are generally regular

Distribution

It is a all-parameterized diff.

Each 1 of 4 clone instances use different element

Syntactic Information

Type	Instance
Object	
DefaultDOMFactory	
PertFactory	
PertFactory	@491370(PertProject.read(...)[181,203])
NetFactory	
NetFactory	@491370(NetProject.read(...)[185,205])
DrawFigureFactory	
DrawFigureFactory	@491370(DrawProject.read(...)[151,173])
SVGFigureFactory	
SVGFigureFactory	@491370(SVGProject.read(...)[150,171])

org.jhotdraw.samples.svg.SVGProject:read

```
    public void read(File f) throws IOException {
        InputStream in = null;
        try {
            in = new BufferedInputStream(new FileInputStrean
            DOMInput domi = new NanoXMLDOMInput(
                    (SVGFigureFactory)view.getDOMFactory(), ir

            final Drawing drawing = (Drawing) domi.readObje
            SwingUtilities.invokeAndWait(new Runnable() { pub
                view.getDrawing().removeUndoableEditListener
                view.setDrawing(drawing);
                view.getDrawing().addUndoableEditListener(unc
```

org.jhotdraw.samples.pert.PertProject:read

```
    public void read(File f) throws IOException {
        InputStream in = null;
        try {
            in = new BufferedInputStream(new FileInputStrean
            NanoXMLLiteDOMInput domi = new NanoXMLLite
                    (PertFactory)view.getDOMFactory(), in);
            domi.openElement("PertDiagram");
            final Drawing drawing = (Drawing) domi.readObje
            domi.closeElement();
            SwingUtilities.invokeAndWait(new Runnable() { pub
                System.out.println();
                view.getDrawing().removeUndoableEditListener
```

org.jhotdraw.samples.net.NetProject:read

```
    public void read(File f) throws IOException {
        InputStream in = null;
        try {
            in = new BufferedInputStream(new FileInputStrean
            NanoXMLLiteDOMInput domi = new NanoXMLLite
                    (NetFactory)view.getDOMFactory(), in);
            final Drawing drawing = (Drawing) domi.readObje
            SwingUtilities.invokeAndWait(new Runnable() { pub
                view.getDrawing().removeUndoableEditListener
                view.setDrawing(drawing);
                view.getDrawing().addUndoableEditListener(unc
                undo.discardAllEdits();
```

org.jhotdraw.samples.draw.DrawProject:read

```
    public void read(File f) throws IOException {
        InputStream in = null;
        try {
            in = new BufferedInputStream(new FileInputStrean
            NanoXMLLiteDOMInput domi = new NanoXMLLite
                    (DrawFigureFactory)view.getDOMFactory(),
            domi.openElement("PlasmaDraw");
            final Drawing drawing = (Drawing) domi.readObje
            domi.closeElement();
            SwingUtilities.invokeAndWait(new Runnable() { pub
                view.getDrawing().removeUndoableEditListener
                view.setDrawing(drawing);
```

Figure 3: Snapshot for MCIDiff

Table 2: Basic statistics of three subject systems

System	#Class	#LOC	#CloneSets	Precision	Recall
JavaNewIO1.6.0	132	46718	34	100%	100%
JHotDraw7.0.6	309	57031	334	97.66%	95.63%
JFreechart1.0.13	594	222812	463	98.82%	98.39%

code of clone instances (#LOC), the number of differential multisets (#Diff), and the differential ratio (#Diff/#LOC). Clone sets can be sorted by columns. Double clicking a clone set in *Clone Set* view opens *Diff Property* view and *Clone Diff* view.

Diff Property view summarizes basic statistics of the selected clone set. *Clone Diff* view visualizes differences across clone instances side by side. Differential multisets are highlighted with different background colors, all-parameterized with red, same+parameterized with pink, same+gapped with green, parameterized+gapped with purple, same+parameterized+gapped with yellow. Clicking *up* or *down* button on top-right corner of this view highlights previous or next differential multiset in bold font.

5. EVALUATION

In this section, we report our evaluation of *MCIDiff* approach and tool support. We evaluated the accuracy of our *MCIDiff* algorithm with three open source Java systems (JavaIO, JHotDraw, and JFreeChart). We also conducted a user study to evaluate the usefulness of *MCIDiff*'s differencing reports for eight clone-related refactoring tasks.

5.1 The Accuracy of MCIDiff

Table 2 summarizes basic statistics of the three subject systems. CloneDetective [16] reports 831 clone sets in the three systems, among which 77% are parameterized and gapped clones. These parameterized and gapped clone sets contain 7.3±5.7 (mean±stddev) differential multisets. About 30% lines of cloned code fragments contain parameterized and/or gapped differences.

To examine the accuracy of *MCIDiff*, the third author manually investigated all parameterized and gapped clones reported by *CloneDetective*. He used Eclipse Java Editor and Java Compare. He spent about 22 hours in building ground truth set D_{actual} of differences in the reported code clones. Among all 638 parameterized and gapped clone sets, 353 contain 2 instances, 235 contain 3-5 instances, while the rest 50 contain more than 5 instances. The third author spent on average less than 1 minute on a 2-instances clone set, about 2.5 minutes on a clone set consisting of 3-5 instances, and about 10 mins on a clone set consisting more than 5 instances. Although clone sets consisting of three or more instances account for only 44.7% ((235+50)/638) of all clone sets reported, the third author spent about 80% of his time on determining differences in these clone sets.

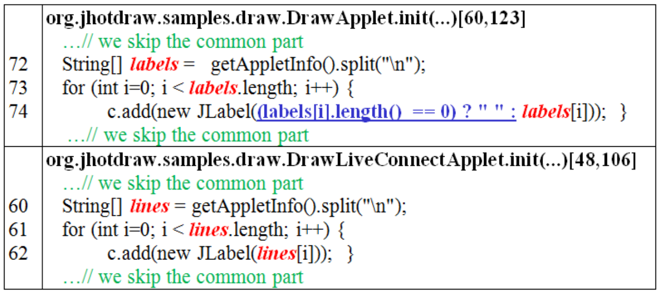
```
    org.jhotdraw.draw.action.Vertical.alignFigures(...)[163,172]
163 double y = selectionBounds.y + selectionBounds.height / 2;
164 for (Iterator i=getView().getSelectedFigures().iterator(); i.hasNext(); ) {
    .... // we skip the common part
168         tx.translate(0, y - b.y - b.height / 2);
169 .... // we skip the common part
    org.jhotdraw.draw.action.Horizontal.alignFigures(...)[183,192]
183 double x = selectionBounds.x + selectionBounds.width / 2;
184 for (Iterator i=getView().getSelectedFigures().iterator(); i.hasNext(); ) {
    .... // we skip the common part
188         tx.translate(x - b.x - b.width / 2, 0);
189 .... // we skip the common part
```

Figure 4: Reordered code elements in code clones

We used precision and recall metrics to evaluate the accuracy of *MCIDiff*'s differencing results. Let $D_{reported}$ be the set of differential multisets reported by *MCIDiff*, precision is percentage of correctly reported differential multisets, i.e., $\mid D_{reported} \cap D_{actual} \mid / \mid D_{reported} \mid$, and recall is percentage of actual differential multisets reported, i.e., $\mid D_{reported} \cap D_{actual} \mid / \mid D_{actual} \mid$.

Table 2 presents precision and recall of *MCIDiff* in the three subject systems. Overall, *MCIDiff* is able to detect parameterized and gapped differences in code clones of JavaNewIO, JHotDraw, and JFreeChart with a good combination of precision and recall. *MCIDiff* is accurate in identifying parameterized differences and added/removed statements such as those shown in our motivating example. But the accuracy of *MCIDiff* will most likely suffer when clone instances have reordered code fragments and/or complex expressions.

When code fragments are reordered across clone instances, differencing results may become arguable. Figure 4 presents such an example. At line 168 and 188 of two clone instances, parameters



```
   org.jhotdraw.samples.draw.DrawApplet.init(...)[60,123]
     …// we skip the common part
72   String[] labels =   getAppletInfo().split("\n");
73   for (int i=0; i < labels.length; i++) {
74           c.add(new JLabel((labels[i].length()  == 0) ? " " : labels[i]));   }
     …// we skip the common part
   org.jhotdraw.samples.draw.DrawLiveConnectApplet.init(...)[48,106]
     …// we skip the common part
60   String[] lines = getAppletInfo().split("\n");
61   for (int i=0; i < lines.length; i++) {
62           c.add(new JLabel(lines[i]));   }
     …// we skip the common part
```

Figure 5: Complex expressions in code clones

for calling *tx.translate*() are swapped and also different, $(0,\ y - b.y - b.height/2)$ versus $(x - b.x - b.width/2,\ 0)$. *MCIDiff* reports five differential multisets, $\{0, \epsilon\}$, $\{y, x\}$, $\{y, x\}$, $\{height, width\}$, and $\{\epsilon, 0\}$. That is, *MCIDiff* reports the second parameter of method call at line 168 and the first parameter of method call at line 188 as parameterized, and reports the first parameter of method call at line 168 and the second parameter of method call at line 188 as gapped. This differencing result is arguably correct. First, these differential multisets reveal important differences across clone instances (e.g., y versus x, *height* versus *width*). Second, differential multisets $\{0, \epsilon\}$ and $\{\epsilon, 0\}$ can still remind developers reordering of parameters.

Complex expressions pose another challenge to *MCIDiff*. For example, at line 74 and 62 of the two clone instances shown in Figure 5, *MCIDIFF* reports gapped differences {*(labels[i].length()* == 0) ? " " :, ϵ}, followed by a parameterized difference {*labels*, *lines*}, followed by matched multisets $\{[i],\ [i]\}$. A more intuitive differencing result would be a differential multiset consists of the two parameters, because it can revel that different clone instances initialize *JLabel* object in different way. This is a general limitation of token-based differencing approach, because it does not consider syntactic structure of the program. *MCIDiff* favors efficiency and simplicity of token-based approach over robustness of syntactic differencing. In the context of differencing code clones (i.e., similar code fragments), our results show that token-based approach can produce highly accurate results, except for about less than 5% differential multisets involving reorder code fragments and/or complex expressions.

5.2 The Usefulness of MCIDiff

Having evaluated the accuracy of *MCIDiff*, the next question we would like to address is "what is *MCIDiff*'s differencing results good for?". We conducted a user study to investigate whether *MCIDiff* can help developers achieve better performance in clone-related refactoring tasks. We acknowledge that refactoring (or removing) code clones usually involves many factors other than just checking the differences of multiple clone instances [38]. However, detecting and understanding differences across multiple clone instances is one of the key factors to make informed decisions on clone refactorings. Therefore, we deem that clone refactoring is a valid means to evaluate the usefulness of MCIDiff.

5.2.1 Study design

In this study, we chose *CloneDetective* [16] as baseline clone analysis tool to compare with our *MCIDiff*. *CloneDetective* is also implemented as an Eclipse plugin, which allows us to make a fair comparison between *MCIDiff* and *CloneDetective* in the same development environment. Furthermore, *CloneDetective* not only detects code clones, it also provides rich features for analyzing detected clones, such as pairwise differencing and difference highlighting, ApsectJ-based visualization of code clones across multiple files, keyword-based searching.

Table 3: Statistics of subject clone sets

	#Inst	Complexity	Description	Refactoring
#1	2	medium	clones reside in two methods declared in two subclasses of the same direct superclass	pull up methods
#2	2	high	clones reside in two methods declared in two classes of the same far-away ancestor class	do not suggest refactoring
#3	2	medium	clones reside in two methods of two subclasses of the same direct superclass; have 3 parameterized differences in 26 lines of cloned code	extract parameters; pull up methods
#4	3	high	clones reside in three methods declared in three subclasses of the same far-away ancestor classes; have 2 parameterized differences in method calls	cannot be refactored
#5	4	low	clones reside in four methods of the same class	replace method body with method call
#6	4	medium	clones reside in four methods declared in two subclasses of the same direct superclass; have 6 parameterized differences in 40 lines of cloned code	extract parameters; pull up methods
#7	5	high	clones reside in five methods of different anonymous classes contained in three different classes; the three different classes have grandparent-parent-child relations; have 6 parameterized+gapped differences in 16 lines of cloned code	extract parameters; extract a new method in grandparent class
#8	6	medium	clones reside in four methods of the same class; have 9 same+parameterized+gapped differences in 8 lines of cloned code	extract parameters; extract method

We recruited 18 graduate students from the School of Software, Fudan University. Before the experiment, we surveyed capability of participants, including years of programming experience, familiarity with Java and Eclipse, and familiarity with code clones and refactoring. We used matched-participant two groups design to allocate participants to study groups. Participants are matched in pairs based on their capability. Each pair is then randomly allocated to experimental group or control group. Experimental group G_1 using *MCIDiff* to perform refactoring tasks, while control group G_2 using *CloneDetective* to perform the same set of tasks. We gave a tutorial of *MCIDiff* and *CloneDetective* tool three hours before the experiment and asked the participants to familiarize themselves with important concepts and features of the relevant tool they will use.

Both groups (G_1 and G_2) were given 8 clone sets in JHotDraw (see Table 3). The eight clone sets used in this study differ in the number of clone instances and their complexity. The complexity is determined based on whether refactoring decisions and details require mainly localized information within clones or more global information beyond clones (e.g, type hierarchy information).

Participants were asked to answer two questions for each clone set: 1) *refactoring decision*, whether these clone sets can be refactored, what refactoring can be applied and why; 2) *refactoring details*, what has to be done (e.g., reconcile naming inconsistencies, extract parameters) before clones can be refactored, and what are expected outcomes of refactorings (e.g., where to place cloned code

Table 4: Scoring Criteria for Refactoring Decisions and Details

Score	*Criteria*
0	Irrelevant, wrong or non-sensible reason
1	General or vague reason
2	Partly specific and right reason
3	Specific and right reason

fragments). Participants were required to run a full-screen recorder throughout experiment sessions. The recorded task videos enable us to analyze behaviors of each participant after the experiment.

We evaluated participants' performance in terms of correctness and quality of their refactoring decisions and details. We invited two experts: one expert developed and maintained an in-house JHotDraw application for more than one year, and the other expert is a senior software architect from Alcatel who has nine years industrial experience and is an expert on software clones and related software maintenance issues. The two experts were asked to work together to provide "ground-truth" answers for the eight clone sets. They reached consensus that six clone sets can be refactored (see Table 3). After the experiment, we asked the two experts to grade participants' refactoring answers. A score ranging from 0 to 3 (see Table 4) was given to quantify correctness and quality of participants' refactoring decisions and details. To avoid experimenter expectancy effects, the two experts did not know which study group a participant belongs to when when they graded the participant's answers.

5.2.2 Results: Improvement on performance

To evaluate performance improvement of *MCIDiff*, we compared scores of refactoring decisions, scores of refactoring details and task completion time of experimental group (G_1) and control group (G_2).

Hypotheses.

We introduced the following null and alternative hypotheses to evaluate how different performance of experimental and control groups is.

- **H0:** The primary null hypothesis is that there is no significant difference between the performance of both groups.
- **H1:** An alternative hypothesis to H0 is that there is significant difference between the performance of both groups.

Results of individual participants.

Table 5 and Table 6 present participants' performance, in terms of each participant's average score of refactoring decisions, average score of refactoring details, and average task completion time (minutes) on eight clone sets. Overall, *MCIDiff* users achieved higher scores in refactoring decisions (2.152 ± 0.577 versus 1.653 ± 0.595) and refactoring details (2.083 ± 0.430 versus 1.681 ± 0.537) than *CloneDetective* users. *MCIDiff* users also completed the tasks in shorter time ($2m53s\pm44s$ versus $3m31s\pm47s$) than *CloneDetective* users.

Results of hypotheses testing.

We performed Shapiro-Wilk test [30] on refactoring decision scores, refactoring details scores, and task completion time of both groups. The analysis shows that the data conforms to normal distribution. Therefore, we used paired sample t-tests to evaluate the null hypothesis H0 in terms of refactoring-decision score, refactoring-details score, and task completion time. We evaluate the hypotheses at a 0.05 level of significance. The results of these three tests are shown in Table 7. Based on the results we reject null hypothesis H0 for all the measures of refactoring-decision score, refactoring-details score, and task completion time. Therefore, we accept the alternative hypothesis H1, i.e., there is significant difference between the performance of participants who use *MCIDiff* and *CloneDetective* respectively.

Table 5: Performance of *MCIDiff* Group

Participant	Decision Score	Details Score	Time(mins)
P1	2.625	2.750	1m53s
P2	2.750	2.500	1m46s
P3	2.125	2.250	3m18s
P4	2.875	2.250	3m51s
P5	1.875	2.250	2m10s
P6	2.375	2.125	2m40s
P7	2.250	1.625	3m27s
P8	1.500	1.625	3m54s
P9	1.000	1.375	3m24s
Average	2.152	2.083	2m53s
Std.Dev.	0.577	0.430	44s

Table 6: Performance of *CloneDetective* Group

Participant	Decision Score	Details Score	Time(mins)
P10	2.500	2.375	2m03s
P11	2.000	2.000	2m48s
P12	1.750	2.000	3m59s
P13	2.250	2.125	3m27s
P14	1.500	1.500	3m48s
P15	2.000	2.125	3m15s
P16	1.375	1.250	3m09s
P17	1.000	0.875	4m09s
P18	0.500	0.875	4m23s
Average	1.653	1.681	3m31s
Std.Dev.	0.595	0.537	47s

Table 5 and Table 6 show that *MCIDiff* users achieved higher average scores in refactoring decisions and details and they completed the tasks faster. Thus, we conclude that *MCIDiff* group perform significantly better than *CloneDetective* group in clone-related refactoring tasks.

5.2.3 Discussion

Our qualitative analysis of two study groups' task completion time, refactoring decisions and refactoring details suggests that task complete time of *CloneDetective* group prolongs and quality of their refactoring decisions and details degrades as the number of clone instances in a clone set or the complexity of clone instances increases. In contrast, performance of *MCIDiff* group remains stable. Our analysis of task videos and interviews with participants suggests that this is because *MCIDiff*'s differencing results provide a consistent and readily available means to investigate differences across multiple clone instances.

The performance of two study groups on 2-instances clone sets (i.e., clone sets #1, #2 and #3) were comparable. For those clone sets, *MCIDiff*'s differencing results have no fundamental differences from those of *CloneDetective*. However, for clone sets consisting of multiple clone instances, *CloneDetective* users start facing two main challenges.

First, *CloneDetective* users have to remember which pairs of clone instances have been analyzed. This is not an easy task because N clone instances requires $N(N-1)/2$ pairwise comparison. For clone sets with four or more clone instances, screen size often does not allow developers to see all $N(N-1)/2$ Eclipse Java Compare Editors. Some participants tried to split Eclipse editor area into several regions to accommodate as many Compare Editors as possible. However, such arrangement often makes each editor too small to read code effectively. Some participants gave up pairwise

Table 7: Results of T-Tests of Hypotheses, for the variable Refactoring-Decision score, Refactoring-Details Score and Task Completion Time. Measurements are reported in the following columns: minimum value, maximum value, median, means (μ), variance (σ^2), Degrees of freedom (*DF*), Pearson correlation coefficient (*PC*), statistical significance (*p*), T_{crit}, and *T* statistics.

H	Var	Approach	Samples	Min	Max	Median	μ	σ^2	*DF*	*PC*	*T*	T_{crit}	*p*	Decision
H0	Refactoring Decision	MCIDiff	9	1.00	2.88	2.25	2.15	0.374	8	0.935	6.656	2.306	1.6E-4	Reject
		CloneDetective	9	0.50	2.50	1.75	1.65	0.398	8					
	Refactoring Details	MCIDiff	9	1.38	2.75	2.25	2.08	0.203	8	0.900	4.720	2.306	0.002	Reject
		CloneDetective	9	0.88	2.38	2.00	1.68	0.325	8					
	Completion Time	MCIDiff	9	1m46s	3m54s	3m18s	2m53s	2203.188	8	0.698	-3.062	2.306	0.016	Reject
		CloneDetective	9	2m03s	4m23s	3m27s	3m31s	2518.305	8					

differencing when clone sets have 4 or more clone instances. They opened cloned code fragments in multiple Eclipse Java Editor side by side to visually search for differences. Such visual searching is very demanding for clone sets with complex differences. For example, clone set #7 consists of five clone instances scattered in five methods of three different classes. These five clone instances are different in 6 places, each of which contains 2-5 parameterized and/or gapped differences.

Second, *CloneDetective* users have to remember which clone instances are different from other instances, where and how. As shown in our motivating example, some clone instances may be the same in one place but different in other places. Furthermore, one clone instance may have different types of differences against other clone instances in a specific place. Identifying these differences is challenging, remembering them is even more challenging, because the amount of information exceeds the capacity of human working memory [26]. Cognitive studies suggest that human working memory has a capacity of about 7 ± 2 chunks. However, our subject clone sets have on average 3.8 differential multisets, each of which contains 2-6 parameterized and/or gapped differences. Our video analysis shows that *CloneDetective* users frequently switch or revisit code editors. They explained in our post-experiment interviews that they were trying to recall differences across multiple clone instances. The key challenge lies in the fact that these differences are not readily available for use, unless they took note to keep track of all necessary information.

Our *MCIDiff* addresses these two challenges in a systematic way. Differences across multiple clone instances are automatically identified with high accuracy. Differences are then visualized side by side so that developers can know where clone instances are different and how by a quick glance. As such, *MCIDiff* users spent more time on further investigating differences and relevant program information. For example, do parameterized differences share common syntactic structure? In our motivating example, the four class *PertFactory*, *NetFactory*, *DrawFigureFactory*, and *SVGFigureFactory* are all subclases of *DefaultDOMFactory*. Such further investigation resulted in better refactoring decisions and more specific refactoring details. In contrast, *CloneDetective* users spent much more time on identifying and recalling differences, especially when clone sets consist of multiple clone instances with complex differences. As such, their refactoring decisions and details were usually vague and uncertain.

5.3 Runtime Performance of MCIDiff

Our *MCIDiff* algorithm adopts a progressive alignment strategy to apply classic LCS algorithm [13] to multiple cloned code fragments. It have time complexity O(MND) where M is the number of cloned code fragments, N is the sum of the lengths of two compared token sequences, and D is the number of differences in clones. LCS algorithm performs well when differences are small (i.e., sequences are similr). *MCIDiff* is consequently fast because it compares cloned code fragments that are similar to each other.

We used our *MCIDiff* tool to compare all 638 parameterized and gapped clone sets on a PC with a CoreI7 CPU of 2.7GHz, 4G RAM, and Windows 7. It took *MCIDiff* 67 seconds to generate differencing results of these 638 clone sets. For the largest clone set that contains 19 clone instances and 14 LOC on average, *MCIDiff* took about 2 seconds to detect differences in this clone set. The differencing result of this clone set consists of 9 differential multisets (i.e., differential ratio of this clone set is 0.64 (9/14)).

5.4 Threats to Validity

There are mainly three threats in our evaluation. First, we only studied three small-to-medium sized Java systems. These subject systems may not contain all representative code clones and clone-related refactoring tasks. Furthermore, our study involved only a limited number of developers. Their capabilities and experience may not be representative. Further studies are required to generalize our findings in large-scale industrial systems and with more professional developers.

Second, differences in capabilities of the two groups of participants may threaten "equivalence" between experimental group and control group. To address this threat, we had tried our best to allocate participants with comparative capabilities into different groups based on our pre-study survey and our evaluation of participants' capabilities.

Third, grading of eight refactoring tasks cannot be completely objective because expert opinions may be biased. In the experiment, we asked the two experts to spend as much time as they need on providing ground-truth answers as well as on grading participants' answers. This allows the experts to fully explore the tasks and answers. To avoid experimenter expectancy effects, we hid the participants' group information from the experts. We hope this helps to reduce subjectiveness to the minimum.

6. APPLICATION

We now discuss three applications (currently under development in our group) that exploit *MCIDiff*'s differencing results for software development and maintenance.

6.1 Auto Generation of Application Skeleton

Framework-based software development becomes increasingly common and important. Applications built on a framework must adhere to design structure and coding convention dictated by the framework. Complex frameworks often support generating application skeleton based on predefined code templates. However, these predefined code templates usually generate only bare bones of an application with little or no real features. Alternatively, frameworks are often shipped with code examples to demonstrate framework usage. However, such code examples usually cover only some typical usage scenarios of the framework.

We are now investigating clone analysis, programming differencing, and data abstraction techniques to transform crowdsourced code examples of building applications on a specific framework into reusable code templates. Socio-professional medias (e.g., Eclipse Marketplace, github) archives fast-growing body of crowdsourced code examples. The mined code templates will record not only commonality but also variations across similar code examples. Code clone detection is used to detect similar code examples. *MCIDiff* is used to identify differences across similar code examples. The differences will be represented as optional or alternative variation points in code templates. Such code templates can be easily customized to generate application skeleton with rich features. As such, application developers can concentrate on the specifics of their applications.

6.2 Simultaneous Code Editing

Code clones must be made explicit so that they can be consistently maintenance and managed. Several approaches [28, 31, 5, 31] have been proposed to support tracking and consistent evolution of code clones, for example during copy-paste-modify. These approaches can propagate changes made to one clone instance to others and thus ensure that common parts of code clones will be consistently modified.

Our work shows that code clones can be different in various way. A change made to one instance may not be necessary in the other instance. In our motivating example, *PertProject.read()* and *DrawProject.read()* call *domi.openElement*(*...*) and *domi.closeElement*(), while *NetProject.read()* and *SVGProject.read()* do not. These gapped differences represent optional method calls for reading a project file. Furthermore, a change in one instance may have to be adapted when propagating to the other instance. In our motivating example, suppose the developers declare a new variable of type *PertFactory* in *PertProject.read*(), this change cannot be simply propagated to the other three *read*() methods, because the other three methods should use different factory types *NetFactory*, *DrawFactory*, and *SVGFactory*. These parameterized differences represent adaptation that have to be made during change propagation.

Our results suggest that blindly propagating changes across clone instances may not work. We are investigating simultaneous editing support for code clones that can intelligently infer where to propagate the change and how based on *MCIDiff*'s differencing results.

6.3 Revision Control

Although *MCIDiff* was originally designed for analyzing code clone differences, it can be applied in other context where several pieces of similar code need to be compared, for example revision control. An original piece of source code may be modified by several developers in parallel. This can result in similar but also different codes. Existing reversion control supports only pairwise differencing and merging. Given multiple revisions, developers may have to perform cascading pairwise differencing and merging. This cascading process may produce less optimal output because revisions are analyzed in pairs without considering other revisions globally. *MCIDiff* can be applied to identify the differences across multiple revisions, based on which multi-way merging of several revisions can be supported.

7. RELATED WORK

Researchers have proposed many techniques to detect code clones based on token [1, 16, 17, 25], AST [2, 14], and Program Dependence Graph [8, 11, 20, 23]. Roy and Cordy [29] and Koschke [21] provide comprehensive surveys of existing clone detection techniques. To support scalable analysis in large systems, clone detection techniques usually rely on easy-to-compute similarity metrics to determine similarity between code fragments. Detecting detailed differences while identifying code clones is computational expensive and thus impractical. Our approach proposes to use clone detectors to identify which parts of the system are similar (i.e. where clones are) first, and then use *MCIDiff* identifies how these clones are different efficiently.

Researchers proposed clone analysis approaches to aiding the interpretation and management of software clones. For example, Genimi [32] uses a scatter plot to visualize code clones detected by CCFinder [17], it also computes several code metrics of clones to aid clone analysis. These clone analysis approaches examine only information of clones, but not differences between code clones. However, one cannot interpret code clones (i.e. similarities) without understanding their differences precisely [37]. CP-Miner [25] finds bugs based on inconsistent identifiers between clones. Kapser and Godfrey [18] classify code clones through syntactic analysis of locality of clones. These approaches analyze clone instances pairwisely, and thus cannot systematically identify and summarize differences across multiple clone instances.

Program differencing techniques [4, 33, 6, 36, 35] have long been used in software maintenance tasks. Existing program differencing techniques compare two programs at a time. For example, Cottrell et al. [4] developed a pairwise differencing technique to detect correspondences between two pieces of codes for the purpose of generalization. However, simply applying existing program differencing techniques to multiple programs will result in combinatorial explosion of pairwise differencing operations. Our *MCIDiff* performs token-based differencing. Structure differencing algorithms such as [7] may produce more accurate differencing results, but they are more computational expensive than token-based differencing. In our work, we favour efficiency rather than marginal accuracy in the context of differencing code clones.

Techniques for multiple sequence alignments have been studied in the area of bioinformatics for the purpose of aligning DNA sequences [39, 27]. Our *MCIDiff* is applied to software programs that have completely different characteristics than DNA sequences. Furthermore, our approach not only identifies longest common parts across multiple clone instances, it also zooms into differential parts to detect detailed differences across multiple clone instances.

8. CONCLUSION AND FUTURE WORK

In this paper, we have presented *MCIDiff*, an automatic approach to detecting differences across multiple clone instances. Our evaluation has shown that the accuracy of *MCIDiff* is good in practice and it is robust to analyze different types of code clones. Our user study has demonstrated the usefulness of *MCIDiff*'s differencing results for clone-related refactoring tasks.

MCIDiff is a key component in our clone analysis framework that aims to enable practical use of code clones in software maintenance by summarizing syntactic, semantic and differential patterns in code clones. Within this framework, *MCIDiff* automatically builds a detailed and accurate report of differences across multiple instances of code clones. This open up many opportunities for practical applications of code clones in software maintenance, such as code-generation, simultaneous code editing, revision control.

9. ACKNOWLEDGEMENT

This work is supported by National High Technology Development 863 Program of China under Grant No.2012AA011202, National Natural Science Foundation of China under Grant No.61370079 and NTU Startup Grant M4081029.020.500000.

10. REFERENCES

[1] H. A. Basit and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *ESEC/SIGSOFT FSE*, pages 513–516, 2007.

[2] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

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

[4] R. Cottrell, J. J. C. Chang, R. J. Walker, and J. Denzinger. Determining detailed structural correspondence for generalization tasks. In *ESEC-FSE '07*, pages 165–174, 2007.

[5] E. D. Ekoko and M. P. Robillard. Clonetracker: tool support for code clone management. In *ICSE'08*, pages 843–846, 2008.

[6] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[7] B. Fluri, M. Wursch, M. Pinzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *TSE'07*, 33(11):725–743, 2007.

[8] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE*, pages 321–330, 2008.

[9] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification: Java Se 7 Ed*. Java Series. Prentice Hall PTR, 2013.

[10] D. G. Higgins and P. M. Sharp. CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1):237–244, 1988.

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

[12] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous modification support based on code clone analysis. In *APSEC '07*, pages 262–269, 2007.

[13] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343.

[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[15] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE '07*, pages 55–64, 2007.

[16] E. Jürgens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE*, pages 485–495, 2009.

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

[18] C. Kapser and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *IWPSE*, pages 85–94, 2004.

[19] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *In Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, 2001.

[20] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.

[21] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*.

[22] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE'01*, pages 301–310, 2001.

[23] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, pages 301–309, 2001.

[24] J. Krinke. A study of consistent and inconsistent changes to code clones. In *WCRE '07*, pages 170–178, 2007.

[25] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.

[26] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, 1956.

[27] L. A. Newberg. Significance of gapped sequence alignments. *J. Comput Biol.*, 15(9):1187–1194, 2008.

[28] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *TSE*, 38(5):1008–1026, 2012.

[29] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Queen's Technical Report:541*, pages 0–115, 2007.

[30] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, Dec. 1965.

[31] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VLHCC '04*, pages 173–180, 2004.

[32] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *IEEE METRICS*, pages 67–76, 2002.

[33] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[34] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.

[35] Z. Xing. Model comparison with genericdiff. In *ASE*, pages 135–138, 2010.

[36] Z. Xing and E. Stroulia. Differencing logical uml models. *Autom. Softw. Eng.*, 14(2):215–259, 2007.

[37] Z. Xing, Y. Xue, and S. Jarzabek. Clonedifferentiator: Analyzing clones by differentiation. In *ASE*, pages 576–579, 2011.

[38] G. Zhang, X. Peng, Z. Xing, and W. Zhao. Cloning practices: Why developers clone and what can be changed. In *ICSM'12*, pages 285–294, 2012.

[39] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning dna sequences. *IEEE Trans. Software Eng.*, 7(1):203–214, 2000.

[40] M. F. Zibran and C. K. Roy. Towards flexible code clone detection, management, and refactoring in ide. In *IWSC '11*, pages 75–76, 2011.