Detecting Differences across Multiple Instances of Code Clones

Yun Lin\textsuperscript{1,2}, Zhenchang Xing\textsuperscript{3}, Yinxing Xue\textsuperscript{4}, Yang Liu\textsuperscript{5}, Xin Peng\textsuperscript{1,2}, Jun Sun\textsuperscript{5}, Wenyun Zhao\textsuperscript{1,2}

\textsuperscript{1} School of Computer Science, Fudan University, China
\textsuperscript{2} Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China
\textsuperscript{3} School of Computer Engineering, Nanyang Technological University, Singapore
\textsuperscript{4} School of Computing, National University of Singapore, Singapore
\textsuperscript{5} Singapore University of Technology and Design, Singapore
\{linyun, pengxin, wyzhao\}@fudan.edu.cn, \{yczx, 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE ’14, May 31 - June 7, 2014, Hyderabad, India
Copyright 14 ACM 978-1-4503-2756-5/14/05 ...$15.00.
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. Each code fragment represents a clone instance in the clone set (see Section 4.1 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() and DrawProject.read() call domi.openElements() with different parameters (PertDiagram versus PlasmaDraw, i.e., parameterized difference). When comparing NetProject.read() and SVGProject.read() and PertProject.read() and DrawProject.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() and PertProject.read() and DrawProject.read() call domi.openElements() with different parameters (PertDiagram versus PlasmaDraw, i.e., parameterized difference). When comparing NetProject.read() and SVGProject.read() and PertProject.read() and DrawProject.read(), these clones have gapped differences.

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
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 identifiers 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., $MS = \{ t_k \ | \ t_k \in ts(ci_k) \ and \ 1 \leq k \leq n \}$. A token $t_k$ from a clone instance $ci_k$ can only be in one multiset. The token $t_k$ can be $c$, 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.

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 $\exists t_k = c$, 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, cyan 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 $\{InputStream, InputStream, InputStream, InputStream\}$. 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: $MS_2=$\{PerDiagram, NetFactory, DrawFigureFactory, SVGFigureFactory\}, $MS_3=$\{openElement, c2, openElements, c4\}, $MS_4=$\{PerDiagram, c2, PlasmaDraw, c4\}. These differential multisets identify “All Parameterized”, “Same+Gapped”, and “Same+Parameterized+Gapped” differences discussed in motivating example.

3.3 Differentiating Multiple Clone Instances

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

Require: List < TSS > tss, List < Range > ciRanges
Ensure: List < Multiset < Token >>

1: results ← ∅
2: List < int[] > lcs ← computeLCS(tss, ciRanges);
3: for each token index array tokens in lcs do
4:   multiset ← tokens at corresponding token index tokensIndices from
token sequences in tss;
5:   results.add(multiset);
6: end for
7: List < Range[] > ldiffRanges ←
determineDiffRanges(ciRanges, lcs);
8: for each diffRange in ldiffRanges do
9:   derive a new list of non-empty ranges newciRanges by removing ε
eranges in diffRange;
10: if 1 < newciRanges.size < ciRanges.size then
11:   results.addMCIDiff(tss, newciRanges);
12: else if newciRanges.size() = ciRanges.size then
13:   results.add,
   identifySimilarTokens(tss, newciRanges);
14: end if
15: for each token index i in the non-empty range in newciRanges do
16:   multiset ← ∅
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 param-
er 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 in-
stance index cinIndex 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 cinIndex[first, last].
Initially, ciRanges contains a Range for each clone instance. Range
at ciRanges[k] is initialized as cinIndex[1, tss[k].length], i.e., the whole
token sequence of clone instance cinIndex. 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 solu-
tion. 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, com-
puteLCS() 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 com-
pared. Given two token sequences, computeLCS() implements dy-
namic 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() corre-
spounding tokens, one from each toke sequence specified by ciRanges.

Take the four token sequences shown in Figure2 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 ci1, 66th token of
ci2, 88th token of ci3, and 66th token of ci4 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 subse-
quint token indices in the LCS of token sequences specified by
ciRanges.

For the four token sequences shown in Figure2 determineDiff –
Ranges reports five differential ranges, two of which are shown in
the figure. For example, [27 – 27] of clone instance ci1, ci2, ci3, and

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 sub-
sequences (line 9 and line 11). For example, from the differential
range [ci1[38, 46], ci2[6], ci3[38, 46], ci4[6]] MCIDiff derives a
newciRanges consisting of two Ranges, i.e., ci1[38, 46] and

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 [ci1[38, 46], ci2[38, 46]] identifies the LCS
([domi, openElement, (, " objects, (,", " ```
PlasmaDraw, " ), " PlasmDraw, " ), " PlasmDraw in ci1 and
PlasmDraw in ci3 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,
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, InputStream2, InputStream3, InputStream4). 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 $c_1$ and $c_3$ in the recursive call with Ranges $[c_1[38, 46], c_2[38, 46]]$. Thus, multisets derived from such LCSs will consist only of tokens from this subset of token sequences. For the rest of token sequences that are not in this subset (for example $c_2$ and $c_4$), they essentially contribute $\epsilon$ to the multiset. For example, based on the LCS $domi_1, openElement, (, ”,”, )$ of two token subsequences in the ranges $[c_1[38, 46], c_2[38, 46]]$, 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 sequence, MCIDiff creates multisets that consist of tokens only from this subsequence (lines 15). That is, all other clone instances contribute $\epsilon$ 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 token 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 $\text{sim}(t_{cand}, \text{multiset})$ is computed as the average similarity of $t_{cand}$ and each token $t \in \text{multiset}$, i.e., $\sum_{t' \in \text{multiset}} \text{sim}(t_{cand}, t') / |\text{multiset}|$. 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 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 $\epsilon$.

Algorithm 2 identifySimilarTokens

Require: List < TS > tss, List < Range > diffRanges, h
Ensure: List < Multiset < Token >>
1: results ← ∅
2: create a new list of token subsequences bcTSs from tss and diffRanges;
3: for each token sequence $ts$ in bcTSs (from longest to shortest) do
4: candTSs ← ts.bcTSs.remove(ts);
5: for each not-yet-matched token $t$ in $ts$ do
6: multiset ← ∅; multiset.add($t$);
7: while candTSs ≠ ∅ do
8: find a not-yet-matched token $t_{cand}$ in all candidate token sequences candTSs with maximum $\text{sim}(t_{cand}, \text{multiset})$
9: if $\text{sim}(t_{cand}, \text{multiset}) < h$ 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 | multiset | ≥ 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 ($\text{sim}(t_1, t_2)$) by comparing the attributes of tokens and relative positions in token sequences.

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

identifySimilarTokens() computes position similarity ($\text{sim}_{\text{pos}}$) between two tokens $t_1$ and $t_2$ by measuring their relative positions in corresponding token sequences. Given two ranges $R_1$ with length $l_{en_1}$ and $R_2$ with length $l_{en_2}$, let first index of $R_1$ and $R_2$ be $f_{1}$ and $f_{2}$, index of $t_1$ and $t_2$ in token sequences be $p_{1}$ and $p_{2}$, position similarity $\text{sim}_{\text{pos}}(t_1, t_2)$ is computed as: $\left| \frac{p_{1} - f_{1}}{l_{en_1} - (p_{2} - f_{2}) - l_{en_2}} \right|$

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

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...
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 \cite{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.

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., $|D_{reported} \cap D_{actual}| / |D_{reported}|$, and recall is percentage of actual differential multisets reported, i.e., $|D_{reported} \cap D_{actual}| / |D_{actual}|$.

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 and gapped 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, differing results may become arguable. Figure 4 presents such an example. At line 168 and 188 of two clone instances, parameters
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, \( \epsilon \), \( \{x,y\}, \{y,x\}, \{height, width\} \), 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 \( x, \text{height} \) versus \( \text{width} \)). 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 \{\( \text{labels}[i].\text{length}() == 0 \)? " " + \( \epsilon \), \} followed by a parameterized difference \{\( \text{labels}, \text{lines} \), followed by matched multisets \{\( i \}, [i] \}. A more intuitive differencing result would be a differential multiset consists of the two parameters, because it can reveal that different clone instances initialize ILabel 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 reordering of 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 detecting clones, such as pairwise differencing and difference highlighting, AspectJ-based visualization of code clones across multiple files, keyword-based searching,

<p>| Table 3: Statistics of subject clone sets |</p>
<table>
<thead>
<tr>
<th>#Inst</th>
<th>Complexity</th>
<th>Description</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>2</td>
<td>medium</td>
<td>clones reside in two methods declared in two subclasses of the same direct superclass; have 6 parameterized differences in 40 lines of cloned code</td>
</tr>
<tr>
<td>#2</td>
<td>2</td>
<td>high</td>
<td>clones reside in two methods declared in two classes of the same direct superclass; have 2 parameterized differences in method calls</td>
</tr>
<tr>
<td>#3</td>
<td>2</td>
<td>medium</td>
<td>clones reside in two methods of two subclasses of the same direct superclass; have 3 parameterized differences in 26 lines of cloned code</td>
</tr>
<tr>
<td>#4</td>
<td>3</td>
<td>high</td>
<td>clones reside in three methods declared in three subclasses of the same far-away ancestor class; have 2 parameterized differences in method calls</td>
</tr>
<tr>
<td>#5</td>
<td>4</td>
<td>low</td>
<td>clones reside in four methods of the same class</td>
</tr>
<tr>
<td>#6</td>
<td>4</td>
<td>medium</td>
<td>clones reside in four methods declared in two subclasses of the same direct superclass; have 6 parameterized differences in 4 lines of cloned code</td>
</tr>
<tr>
<td>#7</td>
<td>5</td>
<td>high</td>
<td>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</td>
</tr>
<tr>
<td>#8</td>
<td>6</td>
<td>medium</td>
<td>clones reside in four methods of the same class; have 9 same+parameterized+gapped differences in 8 lines of cloned code</td>
</tr>
</tbody>
</table>

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 \( (G_1 \text{ and } G_2) \) 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 refactoring (e.g., where to place cloned code.
Table 4: Scoring Criteria for Refactoring Decisions and Details

<table>
<thead>
<tr>
<th>Score</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Irrelevant, wrong or non-sensible reason</td>
</tr>
<tr>
<td>1</td>
<td>General or vague reason</td>
</tr>
<tr>
<td>2</td>
<td>Partly specific and right reason</td>
</tr>
<tr>
<td>3</td>
<td>Specific and right reason</td>
</tr>
</tbody>
</table>

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 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 (G1) and control group (G2).

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 ± 44s versus 3m31s ± 47s) 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 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 related refactoring tasks.

### Table 5: Performance of MCIDiff Group

<table>
<thead>
<tr>
<th>Participant</th>
<th>Decision Score</th>
<th>Details Score</th>
<th>Time(mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2.750</td>
<td>2.250</td>
<td>1m31s</td>
</tr>
<tr>
<td>P2</td>
<td>2.750</td>
<td>2.250</td>
<td>1m46s</td>
</tr>
<tr>
<td>P3</td>
<td>2.125</td>
<td>2.250</td>
<td>3m18s</td>
</tr>
<tr>
<td>P4</td>
<td>2.875</td>
<td>2.250</td>
<td>3m51s</td>
</tr>
<tr>
<td>P5</td>
<td>1.875</td>
<td>2.250</td>
<td>2m10s</td>
</tr>
<tr>
<td>P6</td>
<td>2.375</td>
<td>2.125</td>
<td>2m40s</td>
</tr>
<tr>
<td>P7</td>
<td>2.250</td>
<td>1.625</td>
<td>3m27s</td>
</tr>
<tr>
<td>P8</td>
<td>1.500</td>
<td>1.625</td>
<td>3m54s</td>
</tr>
<tr>
<td>P9</td>
<td>1.000</td>
<td>1.375</td>
<td>2m34s</td>
</tr>
</tbody>
</table>

### Table 6: Performance of CloneDetective Group

<table>
<thead>
<tr>
<th>Participant</th>
<th>Decision Score</th>
<th>Details Score</th>
<th>Time(mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P10</td>
<td>2.500</td>
<td>2.375</td>
<td>2m03s</td>
</tr>
<tr>
<td>P11</td>
<td>2.000</td>
<td>2.000</td>
<td>2m48s</td>
</tr>
<tr>
<td>P12</td>
<td>1.750</td>
<td>2.000</td>
<td>3m59s</td>
</tr>
<tr>
<td>P13</td>
<td>2.350</td>
<td>2.125</td>
<td>3m27s</td>
</tr>
<tr>
<td>P14</td>
<td>1.500</td>
<td>1.500</td>
<td>3m48s</td>
</tr>
<tr>
<td>P15</td>
<td>2.000</td>
<td>2.125</td>
<td>3m15s</td>
</tr>
<tr>
<td>P16</td>
<td>1.375</td>
<td>1.250</td>
<td>3m40s</td>
</tr>
<tr>
<td>P17</td>
<td>1.000</td>
<td>0.875</td>
<td>4m09s</td>
</tr>
<tr>
<td>P18</td>
<td>0.500</td>
<td>0.875</td>
<td>4m23s</td>
</tr>
</tbody>
</table>

Average 1.653 1.681 3m31s

Std.Dev. 0.595 0.537 47s
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 [25]. 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 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 has time complexity O(MNMD) 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 similar). 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 differing 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 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 version 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, AST, and Program Dependence Graph. Roy and Cordy and Koschke 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, Gemini uses a scatter plot to visualize code clones detected by CCFinder, 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. CP-Miner finds bugs based on inconsistent identifiers between clones. Kapser and Godfrey classify code clones through syntactic analysis of locality of clones. These approaches analyze clone instances pairwise, and thus cannot systematically identify and summarize differences across multiple clone instances.

Program differencing techniques have long been used in software maintenance tasks. Existing program differencing techniques compare two programs at a time. For example, Cottrell et al. 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 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. 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.
10. REFERENCES