Clone-Based and Interactive Recommendation for Modifying Pasted Code

Yun Lin[12], Xin Peng[12], Zhenchang Xing[3], Diwen Zheng[12], and Wenyun Zhao[12]
[1]School of Computer Science, Fudan University, China
[2]Shanghai Key Laboratory of Data Science, Fudan University, China
[3]School of Computer Engineering, Nanyang Technological University, Singapore

ABSTRACT

Developers often need to modify pasted code when programming with copy-and-paste practice. Some modifications on pasted code could involve lots of editing efforts, and any missing or wrong edit could incur bugs. In this paper, we propose a clone-based and interactive approach to recommending where and how to modify the pasted code. In our approach, we regard clones of the pasted code as the results of historical copy-and-paste operations and their differences as historical modifications on the same piece of code. Our approach first retrieves clones of the pasted code from a clone repository and detects syntactically complete differences among them. Then our approach transfers each clone difference into a modification slot on the pasted code, suggests options for each slot, and further mines modifying regulations from the clone differences. Based on the mined modifying regulations, our approach dynamically updates the suggested options and their ranking in each slot according to developer's modifications on the pasted code. We implement a proof-of-concept tool *CCDemon* based on our approach and evaluate its effectiveness based on code clones detected from five open source projects. The results show that our approach can identify 96.9% of the to-be-modified positions in pasted code and suggest 75.0% of the required modifications. Our human study further confirms that *CCDemon* can help developers to accomplish their modifications of pasted code more efficiently.

Categories and Subject Descriptors

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

General Terms

Algorithms, Human Factors

Keywords

copy and paste, code clone, recommendation, differencing, reuse

1. INTRODUCTION

Researchers have shown that cloned code takes up 5%∼20% of the code base in both open source and industrial systems [17, 28]. It indicates that copy-and-paste operation is developers' common programming practice. Although copy-and-paste practice seems to be harmful [3, 21, 22, 27], researchers have shown its legitimacy in many cases [5, 13, 15, 16, 35]. Contrary to the conceived reason of laziness, developers usually copy code out of language limitation, delayed code refactoring, or even design decision [15]. Thus, in many cases, copy-and-paste operation is regarded as an efficient and practical way of code reuse.

Developers usually need to modify pasted code to meet their needs [26]. Some modifications on pasted code could involve lots of editing efforts, and any missing or wrong edit could incur bugs. Techniques [7, 12] have been proposed to assist developers in quick-fixing syntax errors or consistently renaming variables in pasted code. However, what developers need to modify is often beyond fixing syntax errors or renaming variables. Kerr et al. [14] proposed some heuristic rules to summarize relations between copied code and its context, which can be used to suggest modifications on pasted code. Although potentially useful, the proposed heuristic rules were not systematically evaluated. Furthermore, manually enumerating and maintaining heuristic rules incurs extra efforts. Meng et al. [23] proposed a technique to learn editing scripts from code change examples for modifying similar code systematically. Their technique requires code change history recorded in version control systems, which is usually unavailable in cloning practice because developers' copy-paste-modify operations will rarely be recorded in separate revisions. To the best of our knowledge, existing techniques can only support copy-past-modify practice in a limited way.

In this paper, we propose a clone-based and interactive approach to recommending where and how to modify pasted code. In our approach, we regard existing clones of a pasted code as the results of historical copy-and-paste operations, and clone differences as historical modifications on the same piece of code. We first search clones of the pasted code in a clone repository and detect syntactically complete differences among code clones. Then, we transfer each clone difference into a *slot* on the pasted code with a list of suggested options (i.e., code modifications). We further mine two types of implicit modifying regulations from clones. First, we infer naming rules that require consistent modifications of several correlated slots and context. Second, we infer the likelihood to select an option under different conditions. Based on these regulations, the suggested options in each slot are dynamically adjusted during the process of developers' modification on the pasted code.

We implement our approach as an Eclipse plugin named *CCDemon* (Code Cloning Demon) and evaluate it based on code clones detected from five open source projects. The results show that 1) the copy-and-paste scenarios when our technique is applicable, i.e., when the pasted code has clones with differences, can happen with

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786871

Table 1: Cloned Code Samples

```
0  class JavaxDOMOutput
1    implements DOMOutput{
2    private XMLElement current;
3    ...
4    public void addAttribute(double value){
5      String str = Double.toString(value);
6      parseDoubleElement(str);
7      ((Element)current).setAttribute(str);
8
9      double v = transLog(value);
10     LoggerUtil.
11       logAddDoubleAttr(v,
12         JDOM.class);
13   }
14 }
```

```
class NanoXMLDOMOutput
  implements DOMOutput{
  private XMLElement current;
  ...
  public void addAttribute(double value){
    String str = Double.toString(value);
    parseDoubleElement(str);
    current.setAttribute(str);
    String msg = ExecHandler.check(current);
    double v = transLog(value);
    LoggerUtil.
      logAddDoubleAttr(v,
        MicroDOM.class, msg);
  }
}
```

```
class NanoXMLLiteDOMOutput
  implements DOMOutput{
  private XMLElement current;
  ...
  public void addAttribute(float value){
    String str = Float.toString(value);
    parseFloatElement(str);
    current.setAttribute(str);
    String msg = ExecHandler.check(current);
    float v = transLog(value);
    LoggerUtil.
      logAddFloatAttr(v,
        LesDOM.class, msg);
  }
}
```

considerable likelihood (the likelihood is 29.8%); and 2) our approach can identify 96.9% of the to-be-modified positions in pasted cloned code and suggest 75.0% of the required modifications. In addition, our human study with two groups of students further confirms that *CCDemon* can help developers to accomplish their modifications of pasted code more efficiently.

This paper makes the following contributions:

- We design a new clone differencing algorithm, which can detect syntactically complete differences among multiple code clones.
- We propose an accurate and scalable approach to recommending modifications on pasted cloned code.
- We develop a proof-of-concept tool, *CCDemon*, for the practical use of our recommendation approach.
- We evaluate both our approach and tool and the results show that our approach is accurate and useful in practice.

The rest of the paper is structured as follows. Section 2 presents a motivating example of our work. Section 3 describes our approach. Section 4 presents the implementation of *CCDemon*. Section 5 evaluates the effectiveness of our approach with an automatic experiment. Section 6 evaluates the usefulness of *CCDemon* with a human study. Section 7 reviews related work. Section 8 discusses related issues and concludes the paper.

2. MOTIVATING EXAMPLE

Table 2 shows a scenario where a developer needs to add a new feature by referencing existing code in a drawing application. The application can output user's drawn graph into a XML file in different ways. As shown in Table 1, different ways for outputting graph have been realized in several Java classes implementing *DOMOutput* interface. As the developer is not privileged to refactor existing code (a common constraint in industrial settings), he needs to develop a new *DOMOutput* implementation class (e.g., the *CusDOMOutput* class in Table 2), when he wants to output the graph in a new way. In such case, he can reuse existing implementations in code base. For example, he can copy the *addAttribute()* method from one of the existing *DOMOutput* classes shown in Table 1, paste it in the implementation of the *CusDOMOutput* class (e.g., line 3 of Table 2), and modify the copied code by his needs.

Table 2: A Code Cloning Scenario

```
0  class CusDOMOutput implements DOMOutput{
1    private XMLElement element;
2    ...
3    //code is to be pasted here
4  }
```

In this case, the developer is very prone to missing some important modifications or even introducing bugs. Assume that the developer copies and pastes the *addAttribute()* of the class *JavaDOMOutput* in Table 1 without being aware of the other two classes, the following scenarios are likely to happen. First, some important programming conventions may be overlooked. For example, when the developer wants to adapt the parameter type from *double* (line 4) to *float* or *long*, he needs to change the *Double* (line 5), *parseDoubleElement* (line 6), *double* (line 9) and *logAddDoubleAttr* (line 11) accordingly. Any miss of the relevant changes can cause bugs. Second, some alternative implementations could be missed. For example, the developer will be unaware of invoking the *check()* method of the *ExecHandler* class, i.e., "*String msg = ExecHandler.check(current)*" at the line 8 of the *addAttribute()* method of the *NanoXMLDOMOutput* and *NanoXMLLiteDOMOutput* classes. This is an important method call if the developer wants to check and log the status message of the *current* element. Third, some implicit application-specific rules could be missed. Table 1 indicates that the different *addAttribute()* methods should use different class literals sharing same super class, e.g., *JDOM*, *MicroDOM* and *LesDOM*, when invoking the logging method of the *LoggerUtil* class (line 12). Thus, the developer may need to modify the pasted code with an existing new different class literal sharing same super class, e.g., *CusDOM* in this case.

To avoid the above mistakes, the developer must be aware of the differences among the existing similar implementations of the *addAttribute()* method being copied. Table 1 highlights all the differences among the three *addAttribute()* methods. The inferred programming conventions and rules are summarized as follows.

- The parameter type of the *addAttribute()* method is strongly correlated to several statements in the method, e.g., the *double* type corresponds to the "Double" or "double" component in the *Double* in line 5, the *parseDoubleElement* in line 6, the *double* type in line 9, and the *logAddDoubleAttr* in line 11.
- When invoking the *setAttribute()* method in line 7 on the *current* element, the variable *current* can be optionally cast to the *Element* type.
- The statement "*String msg = ExecHandler.check(current)*" in line 8 can be optionally executed.
- When the optional statement in line 8 exists, the logging method (line 11) will be invoked with an additional parameter *msg* (line 12).
- The *addAttribute()* method in different *DOMOutput* class may use different class literal respectively (e.g., *JDOM*, *MicroDOM* and *LesDOM* in line 12).

To generate the above summarization of programming conventions and rules, the developer must: 1) identify all the differences among the three *addAttribute()* methods in Table 1; 2) compare all the possible associations among these differences and context; and

3) infer all the potential naming rules that summarize the correlations of different places. Clearly, manually performing the above steps is not only time-consuming but also mentally challenging.

In this paper, we propose a clone-based and interactive approach to addressing the above issues. In our approach, we use available clone information to identify the *slot*s potentially to be modified in pasted code, and infer some modifying regulations the user may consider to follow. Moreover, based on the inferred modifying regulations, our recommendations will be dynamically adjusted according to what the user edits on the pasted code.

We illustrate a typical usage scenario of our tool (*CCDemon*) as follows. After the developer pastes the *addAttribute()* method of the *JavaxDOMOutput* class to the *CusDOMOutput* class, *CCDemon* will identify and highlight the to-be-modified slots on the pasted code, e.g., the parameter type *double*, the data type *Double*. Afterwards, a list of two options (*double* and *float*) appears for the parameter type *double*. The developer may either select an option from the list or type new code such as *long*. Suppose he types a new code *long*, *CCDemon* will dynamically generate name-consistent option for *Double* (line 5), *parseDoubleElement* (line 6), *double* (line 9), and *logAddDoubleAttr* (line 11) respectively, such as *Long*, *parseLongElement*, *long*, and *logAddLongAttr*. *CCDemon* will gradually guide the developer to modify other slot such as *((Element)current)*. Finally, the developer will be suggested to modify the *JDOM* with the new suggested class literal *CusDOM*, which shares a super class with *JDOM* class based on inferred regulation.

3. APPROACH

In this section, we first present an overview of the approach and then detail its key steps. In this paper, a **clone set** is a set consisting of similar code fragments as its elements, and each of its element is called a **clone instance**.

3.1 Overview

Our approach to recommending modification on pasted code is triggered when a developer copies a code fragment from the code base and pastes it in a target location. Our assumption lies in that there exists a clone repository maintaining clone information of the code base. It can be satisfied in practice when the code base is equipped with an incremental clone detector.

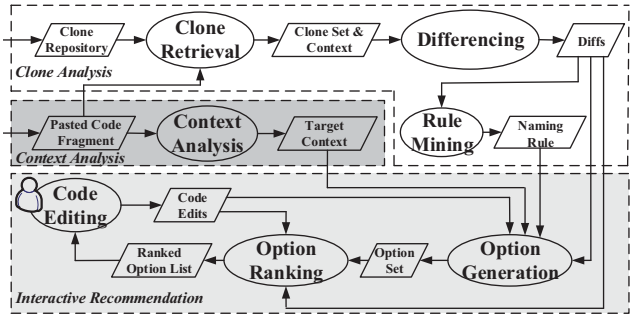

Figure 1: Approach Overview

The overview is presented in Figure 1, in which each ellipse represents a step and each parallelogram represents an input or output of a step. The whole process can be divided into three parts, i.e., clone analysis, context analysis, and interactive recommendation.

Clone analysis aims to identify the slots on the pasted code and mine naming rules. Once a copy-and-paste operation is detected, our approach retrieves a clone set with one of its clone instances overlapping the copied code. If more than one such clone sets are found, we choose the one with a clone instance most overlapping the copied code. If found, we difference the clone set and its context (i.e., class name, method name, method return type). Each difference of the clone set will be transferred into a slot on the pasted code, and the code variants for such difference will be initially used as *option*s of the slot. Afterwards, we mine naming rules from differencing results of the clone set as well as its context.

Context analysis aims to identify program elements as new options from the context of target location. These program elements include: 1) declaration elements, i.e., the class or method where the pasted code resides, and 2) accessible elements, i.e., the code that can potentially be used in slots. Declaration elements include class name, method name, and method return type, which will be used when applying naming rules for option generation. Accessible elements include the variables, fields, methods, classes and interfaces accessible from the pasted code.

Interactive recommendation recommends options for each slot in an interactive process. During the process, the developer edits the pasted code by choosing recommended options or directly modifying the code. Based on the code edits, our approach dynamically generates new options in each slot and adjusts their rank orders. This interactive process is repeated iteratively until the developer finishes editing the pasted code.

3.2 Differencing

We represent each difference of a clone set (or its context) as a multiset, which consists of *correspondent* differential code of clone instances (or their context) as its elements. For example, we represent the difference in line 4 of Table 1 as the multiset {*double*, *double*, *float*}. Note that a multiset can contain ϵ (i.e., empty code) as its element, which indicates a "gap difference" in a clone set. We call the multiset representing a difference among the code bodies of a clone set as a **clone differential multiset**, and the one representing a difference of the context of a clone set (i.e., the declaration elements such as class name, method name and method return type) as a **contextual differential multiset**.

It is straightforward to identify the contextual differential multisets of a clone set because different declaration elements (e.g., class name) of the clone instances are naturally corresponded. However, identifying the clone differential multisets of a clone set is more challenging. The natural correspondence as what the context of a clone set has is not obvious. Moreover, the differencing results needed in our recommendation approach require intuitiveness, in other words, the differencing results should be syntactically complete. For example, the clone difference in line 7 of Table 1 should be reported as a syntactically complete difference, e.g., {*((Element)current)*, *current*, *current*} instead of two isolated differences, $diff_1$={*((Element)*, ϵ, ϵ}, $diff_2$={*)*, ϵ, ϵ}. Note that the latter result often appears because the token *current* is located in the LCS (Longest Common Subsequence) of the cloned code.

To the best of our knowledge, existing clone differencing techniques [6, 8, 18, 20, 32, 33] can hardly meet the needs as they are only designed for pairwise code comparison. The closest technique is our previous work, MCIDiff [20], which is used to detect differences among multiple cloned code fragments. However, MCIDiff is mainly designed for visualizing clone differences. Its results are inappropriate to feed our recommendation approach for two reasons. First, MCIDiff reports clone differences as token-based multisets, but we need syntactically complete differences for recommendation. Second, as other token-based techniques, the results of MCIDiff may sometimes break syntactic boundary. For the example in Table 1, it is possible for MCIDiff to match the semicolon (";") in line 9 of the first fragment with the semicolon in line 8 of the second fragment because syntactic boundary information is lost

when parsing source code into token sequences. Such results will not affect human observation much, but they significantly affect the accuracy and intuitiveness of our recommendation for modifying the pasted code.

Therefore, we develop a new clone differencing algorithm to address the above issues. The algorithm takes as input multiple cloned code fragments, and generates as output *sequence-based* multisets representing clone differences. Each element in a multiset is a token sequence representing a syntactic unit as complete as possible. In the new differencing algorithm, each cloned code fragment is parsed into a token sequence first. Then, the algorithm matches an *optimal common subsequence* from multiple token sequences with regard to code syntactic boundaries. Next, the algorithm identifies differential ranges among the token sequences against the matched optimal common subsequence. Finally, the algorithm splits and merges these differential ranges with regard to their related Abstract Syntax Tree (AST) to ensure that the differencing results are syntactically complete.

3.2.1 Matching Optimal Common Subsequence

Given a clone set, we parse each of its clone instance into a token sequence. We aim to compute an optimal common subsequence from multiple token sequences, which 1) regards code syntactic boundaries and 2) is as long as possible. In the following, we first discuss matching optimal common subsequence between two token sequences, and then proceed to matching multiple token sequences.

Given two token sequences, seq_1 = <$t_{11}, t_{12}, ..., t_{1m}$> and seq_2 = <$t_{21}, t_{22}, ..., t_{2n}$>, their common subsequence seq_{com} is constructed by sequentially *matching* one token t_{1i} ($i \leq m$) in seq_1 with a synonymous token t_{2j} in seq_2 ($j \leq n$). We represent the **match** implied by token t_k in seq_{com} as $match(t_k)$ whose value is a pair (t_{1i}, t_{2j}), t_{1i} and t_{2j} are the **supporting token**s of t_k in seq_1 and seq_2 respectively. Matching a common subsequence from two token sequences means generating one of their common subsequences as well as the matches of its tokens.

We regard the minimal AST node containing a token as its **code context**. For example, the code context of the token ";" in the line 9 of the first fragment in Table 1 is a variable declaration statement. Our rationale is that synonymous tokens with similar code context (especially for delimiters or separators such as ";" and ".") are more likely to be matched. We evaluate the fitness of a match in a common subsequence by the similarity of code context of its supporting tokens. Given a potential match $match(t_k)$ whose value is a pair of token (t_{1i}, t_{2j}), we parse the code context (i.e., AST) of its supporting tokens into two token sequences seq_{con1} and seq_{con2} and the fitness is computed as:

$$fit(match(t_k)) = sim_c(t_{1i}, t_{2j}) = \frac{2LCS(seq_{con1}, seq_{con2})}{len(seq_{con1}) + len(seq_{con2})} \quad (1)$$

$LCS(seq_{con1}, seq_{con2})$ is the LCS of seq_{con1} and seq_{con2}, and $len(seq)$ is the length of a token sequence seq. Thus, we quantify the **matching fitness** of a common subsequence seq_{com} as:

$$matching\_fitness(seq_{com}) = \sum_{t_k \in seq_{com}} fit(match(t_k)) \quad (2)$$

Thus, computing optimal common subsequence between two token sequences is to determine a common subsequence with the best matching fitness. We apply the dynamic algorithm for solving the LCS problem [11] to compute optimal common subsequence between two token sequences in $O(mn)$ time.

Similar to computing the global LCS of multiple sequences, computing a global optimal common subsequence between multiple token sequences is also an NP-complete problem [31]. Therefore, we adopt progressive alignment strategy to compute the optimal common subsequence between multiple token sequences. Progressive alignment approach has been widely applied to align multiple DNA sequences [9]. In this work, we first compute the optimal common subsequence of the two longest token sequences, and then match the resulting common subsequence with the third longest token sequence, and so on till all token sequences are considered.

As a result, the match of each token in the optimal common subsequence will be reported as a **common multiset**. For the example in Table 1, the multisets will be {*public*, *public*, *public*}, {*void*, *void*, *void*}, and so on. By this means, we compute the optimal common subsequence of multiple token sequences with regard to syntactic completeness.

3.2.2 Splitting and Merging Differential Ranges

Reported common multisets have inherent order, and any gap between two consecutive common multisets is a **differential range** of token sequences, as the dashed rectangle shown in Figure 2. As shown in the figure, those differential ranges could still be across syntactic boundary or be syntactically incomplete. Therefore, we split and merge those differential ranges to achieve intuitive differencing results for better modification recommendation.

Splitting Differential Ranges.

Figure 2 partially shows the token sequences of the three clone instances presented in line 8 and line 9 of Table 1. The differential ranges between two consecutive common multisets can be regarded as a **range multiset**. Each element of a range multiset is a **differential token sequence** such as "*String msg ... ; double*".

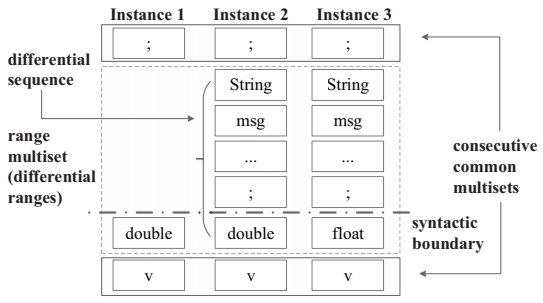

Figure 2: Example for Splitting Differential Ranges

We split each differential sequence in a range multiset into at most three subsequences with regard to syntactic boundary: one body subsequence for (consecutive) complete program statements, one head subsequence for tokens before the body subsequence, and one tail subsequence for tokens after the body subsequence. For example, the second differential sequence in Figure 2 will be split into two subsequences: "*String msg ... ;*" (body) and "*double*" (tail). In practice, any of the head, body, and tail subsequence could be missing. If the body subsequence is missing, the head and tail subsequences are split by statement boundary (e.g., ";", "}").

Then, we match the split head, body and tail subsequences in a range multiset. They are matched according to their similarity and order. As for similarity, we first apply our MCIDiff algorithm [20] to report a list of token-based multisets. Each multiset represents the correspondence between the tokens of multiple clone instances. For example, {ϵ, *String*, *String*} is a token-based multiset in the differential ranges in Figure 2. Given two differential subsequences s_1, s_2, and let the number of their shared corresponding tokens be k, their similarity is $2k/(len(s_1) + len(s_2))$. As for order, we will not match subsequences in switching order. For example, if we match the head subsequence in s_1 to the tail subsequence in s_2, we will no longer match the tail subsequence in s_1 to the head

subsequence in s_2. Instead, both the tail subsequence in s_1 and the head subsequence in s_2 will be matched to ϵ. Thus, a range multiset will be split into ordered differential *sequence-based* multisets. For the example in Figure 2, the differential ranges will be split into two differential multisets, {ϵ, "*String msg ... ;*", "*String msg ... ;*"} and {*double*, *double*, *float*}.

Merging Differential Ranges.

Figure 3 partially shows the token sequences in the line 7 of Table 1. We can see that some differential sequences in split multisets are still partially complete, e.g., "*((Element)*". In this case, merging some syntactically incomplete multisets can give rise to a better syntactically complete multiset. For example, the merged multiset {"*((Element)current)*", "*current*", "*current*"} is a more intuitive result than the two separated multisets {"*((Element)*", ϵ, ϵ} and {"*)*", ϵ, ϵ} for modification recommendation.

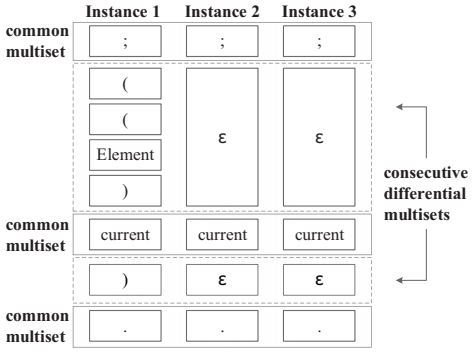

Figure 3: Example for Merging Differential Ranges

We aim to merge two consecutive differential ranges when the two *adjacent* and *syntactically incomplete* consecutive differential multisets can form a new *syntactically complete* multiset. As for adjacency, we require that the number of in-between common multisets be smaller than a user-defined threshold t_m. As for syntactic completeness, we require that 1) any token sequence in the merged multiset represent a complete syntactic unit (i.e., a complete AST subtree) and 2) any differential sequence in two consecutive differential multisets not represent a complete syntactic unit.

3.2.3 Determining Slots

Thus, we generate clone differencing results in terms of *sequence-based* differential multisets (e.g., {ϵ, "*String msg ... ;*", "*String msg ... ;*"}, {"*((Element)current)*", "*current*", "*current*"}). Note that the code developers copy may overlap with cloned code. Thus, we only consider the differential multisets involved in the overlapped code area. Given a pasted code $code_p$ and its corresponding copied code $code_c$, for each differential token sequence in $code_c$, the location of its corresponding token sequences in $code_p$ is a **slot** in $code_p$. In addition, the differential multiset for generating a slot is called as **associated differential multiset** of the slot.

3.3 Rule Mining

A naming rule specifies a possible regulation of consistent modification among slots and their context. For example, the clone differences in line 4, 5, 6, 9, and 11 in Table 1 indicate that the slots on the code pasted from any of the clone instances may need modifying "*double*" or "*float*" component consistently.

We mine naming rules from a clone set from both its clone differential multisets and contextual differential multisets. We align the elements in multiple differential multisets by clone instances (vertically) and multisets (horizontally), as shown in Figure 4. Then, we generate naming rules as follows.

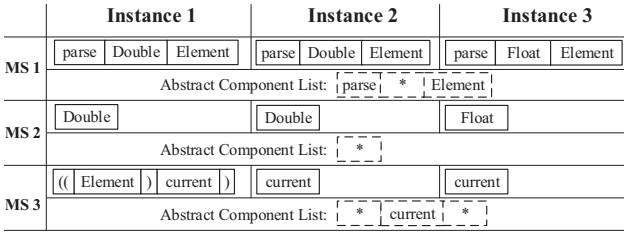

Figure 4: An Example of Naming Rules Mining

First, we split the elements (i.e., token sequence) in each differential multiset into identifiers (e.g., "*parseDoubleElement*") and delimiters such as brackets and dots. Then we further split identifiers into **component**s by code conventions such as case switching, underscores, and dashes. In addition, consecutive delimiters are also treated as components. Thus, each element in a multiset is split into a list of components. For example, in Figure 4, the component list of the first element in the third multiset is <"*((*", "*Element*", "*)*", "*current*", "*)*">.

Next, we abstract the component lists of the elements in a multiset into a pattern called **abstract component list**. For example in Figure 4, the elements of the first multiset can be abstracted into an abstract component list (dashed rectangle) <"*parse*", "*", "*Element*">. To this end, for each multiset we align its component lists by computing their LCS. The common component in LCS will be aligned first, afterwards, the adjacent differential components will be merged and further aligned against the LCS. Finally, we derive an abstract component list for each multiset by abstracting the aligned components into an **abstract component**. The aligned components of an abstract component are called its **instance**s. For example, for the third multiset in Figure 4, "*((*", "*Element*", and "*)*" in its first element will be merged together and aligned with ϵ in the second and third elements. They will be abstracted into an abstract component ("*") in the abstract component list.

Finally, we mine naming rules among different multisets by clustering their abstract components. Given two abstract components c_1 and c_2, let n be the number of all the clone instances and m be the number of clone instances where the instances of c_1 and c_2 are case-insensitively the same, their similarity is computed as m/n. For example, in Figure 4, the similarity of the second abstract component of the first multiset with the abstract component of the second multiset is $3/3 = 1$. Then given a similarity threshold, we perform hierarchical clustering on all the abstract components with mean-linkage criteria [30]. The abstract components in the same cluster are considered as **equivalent**, which derives a naming rule specifying a possible regulation of consistent modification.

As a result, we produce an abstract component list for each differential multiset and a set of naming rules among them.

3.4 Option Generation

The recommended options for each slot are generated from three sources: historical code, compatible elements, and naming rules. The options recommended from the previous two sources can be considered as static, as these options are generated before the user edits the pasted code. In contrast, the options recommended from the source of naming rules can be considered as dynamic, as they are dynamically recommended in some slots based on the user's edits on the code in other slots.

The options sourced from historical code (or historical options) of a slot are the distinct elements of its associated differential multiset. For example, the slot corresponding to the clone difference in line 4 of Table 1 has two historical options: "*double*" and "*float*".

The options sourced from compatible element (or compatible options) of a slot are the distinct program elements syntactically com-

patible with one of its historical options. For each historical option v that is a syntactically complete token sequence, we identify its compatible elements by the following criteria:

1. if v is a type (e.g., "*Double*"), its compatible elements are all the types sharing its direct super type.
2. if v is a variable/field (e.g., "*((Element) current))*"), its compatible elements are all the variables/fields (accessible from v's slot) of the same type or sharing its direct super type.
3. if v is a method (e.g., "*parseDoubleElement()*"), its compatible elements are all the methods (accessible from v's slot) with the same or compatible return types and parameters with the same number, order and type.

For example, the slot corresponding to the clone difference in line 7 of Table 1, if pasted in Table 2, has a compatible option "*element*".

The options sourced from naming rules (or rule options) of a slot are the code synthesized from the values of other slots or the context according to naming rules. In our approach, when a clone instance is pasted and modified, each *clone/contextual differential multiset* of its clone set can correspond to a piece of code in the pasted code or its context. More specifically, a clone one can correspond to new edited code in a slot, meanwhile, a contextual one can correspond to a declaration element (i.e., class name, method name, or method return type) of the pasted code. We call such corresponded code to a differential multiset as its **application value**. As mentioned in Section 3.3, each clone/contextual differential multiset has an abstract component list. Thus, after the code is pasted, for each differential multiset $diff$, we split its application value into a component list, and match it with the abstract component list of $diff$. We call the code (i.e., component) matched to an abstract component $comp_{ab}$ as the **value** of $comp_{ab}$. The values of the abstract components in clone differential multisets are initialized by the pasted code, meanwhile, the values of those in contextual differential multisets are assigned as the declaration elements of the pasted code. When the value of an abstract component in one differential multiset is updated by code c, the values of all its *equivalent component*s in other *clone differential multiset* will be updated to c accordingly. Thus, concatenating the values of abstract components will give rise to new options in the corresponding slots.

For example, if a user copies the first clone instance in Table 1 and pastes it in the place shown in Table 2, the abstract component list for clone difference in line 4 of Table 1 contains one abstract component, i.e., "<*>", and its value is initialized to "*double*". When the user modifies the "*double*" into "*long*", the value of the abstract component will be updated to "*long*", which propagates the value to its equivalent abstract component (represented by *) in the abstract component list of other clone differential multisets, e.g., *<parse*, *, *Element>*, *<log*, *Add*, *, *Attr>*. By this means, new options such as "*parseLongElement*" and "*logAddLongAttr*" are generated in other slots.

3.5 Option Ranking

Once the options of a slot are generated, we rank the options according to their possibility of being chosen by the user. As an option op in slot s can have multiple sources (i.e., historical code, compatible elements, and naming rules), we estimate op's possibility of being chosen by considering all the three sources in Equation 3.

$$S(op) = w_h \cdot S_h(op) + w_c \cdot S_c(op) + w_r \cdot S_r(op) \quad (3)$$

In Equation 3, $S_h(op)$, $S_c(op)$, $S_r(op)$ represent the estimated possibilities of op from the source of historical code, compatible elements, and naming rules; w_h, w_c, w_r represent corresponding weights, satisfying $w_h, w_c, w_r \in (0, 1)$ and $w_h + w_c + w_r = 1$. Given a slot s, we use $HisVal(s)$, $ComEle(s)$, and $EqiRul(s)$ to denote its historical, compatible and rule option sets respectively.

Equation 4 estimates the possibility from the source of historical code, which considers historical occurrence and confidence. The first part of the equation measures the historical occurrence of op, in which $T(s)$ represents the size of associated multiset of s and $T(s, op)$ represents the number of times when op appears in the associated multiset of s. The second part of the equation measures the chosen confidence of op when some historical options in other slots are determined. Suppose there are k slots, $s_1, s_2, ..., s_k$, which have been determined by user, and $op_i \in HisVal(s)$ $(1 \le i \le k)$ is the chosen option in s_i, $P((op, s)|(op_i, s_i))$ is the confidence, i.e., conditional probability, of op to appear in s when op_i appears in s_i. The confidence can be induced from historical co-occurrence of op and op_i. For example, after user copies a cloned code fragment from Table 1 and pastes it in Table 2, if the slot corresponding to the clone difference at line 4 is the only slot determined by user with option "*double*", then for the slot for line 5, there are $S_h(\text{“}Double\text{”}) = \frac{1}{2} \times \frac{2}{3} + \frac{1}{2} \times 1 = \frac{5}{6}$ and $S_h(\text{“}Float\text{”}) = \frac{1}{2} \times \frac{1}{3} + \frac{1}{2} \times 0 = \frac{1}{6}$.

$$S_h(op) = \frac{1}{2} \times \frac{T(s, op)}{T(s)} + \frac{1}{2} \times \frac{\sum_{i=1}^{k} P((s, op)|(s_i, op_i))}{k} \quad (4)$$

Equation 5 estimates the possibility from the source of compatible elements. That is, if op is a compatible element of s, $S_c(op)$ is 1; otherwise, it is 0.

$$S_c(op) = \begin{cases} 1, & op \in ComEle(s) \\ 0, & elsewise \end{cases} \quad (5)$$

Equation 6 estimates the possibility from the source of naming rules, which measures the average similarity between abstract components of s and their equivalent abstract components involved in the generation of op. In Equation 6, m is the number of abstract components of s that have been updated in the generation of op; ac_k $(1 \le k \le m)$ is the kth updated abstract component; $ES(ac_k)$ is the set of equivalent abstract components of ac_k that have been involved in the generation of op; $sim(ac_k, e)$ is the similarity between ac_k and an equivalent abstract component e.

$$S_c(op) = \begin{cases} \frac{1}{m} \times \sum_{k=1}^{m} \frac{\sum_{e \in ES(ac_k)} sim(ac_k, e)}{|ES(ac_k)|}, & op \in EqiRul(s) \\ 0, & elsewise \end{cases} \quad (6)$$

We set the weights based on the following two principles. First, rule options are more likely to be chosen than historical and compatible options. Second, compatible options are more likely to be chosen than historical options if the elements of associated multiset of the slot are diversified, and vice versa. More specifically, the diversity of a clone differential multiset means that almost each historical copy-paste-modify operation of the code gives rise to a new option in the same slot, therefore, the corresponding slot is more likely to have a new different option. We evaluate the diversity of the associated multiset of a slot s with information entropy [29], as shown in Equation 7. In Equation 7, $P(op_h)$ is the frequency of op_h appearing in the associated multiset of s.

$$H(s) = -\sum_{op_h \in HisVal(s)} P(op_h) \ln P(op_h) \quad (7)$$

Therefore, we express the two principles as follows.

1. $w_h + w_c < w_r$.
2. If $H(s) \ge \frac{\ln |HisVal(s)|}{2}$, $w_c > w_h$; otherwise, $w_h > w_c$.

Note that the information entropy ranges from 0 to $\ln |HisVal(s)|$, so we take the median as the threshold to determine whether the historical options are diversified enough.

The three weights (i.e., w_h, w_c, w_r) in Equation 3 can be set based on the above two principles. For example, in our implementation (see Section 4), we use the following weight settings: $w_r = 0.6$; $w_h = 0.1$ and $w_c = 0.3$ if $H(s) \geq \frac{\ln |HisVal(s)|}{2}$, otherwise, $w_h = 0.3$ and $w_c = 0.1$.

4. TOOL SUPPORT

We implement our approach as a proof-of-concept tool named *CCDemon*, which is an Eclipse plugin. Figure 5 shows how *CCDemon* helps a developer when he copies the code fragment in the *JavaxDOMOutput* class in Table 1. After the developer pastes the code, *CCDemon* will enclose some parts of the pasted code as transparent rectangles, each of which represents a slot. The code inside slots is syntactically complete in general. When the developer is to modify the code in a slot, *CCDemon* will show a combo containing a list of ranked options. The developer can either select an option or enter new code in the slot. He can press *Enter* to choose an option; press *Tab* to move forward to the next slot; or press *Shift+Tab* to go back to the previous slot. After a slot is modified, the options and their ranking in other slots will be updated accordingly. For example, in Figure 5, the developer's edits on previous slots generate new options containing *long/Long* in other slots, e.g., "*logAddLongAttr*".

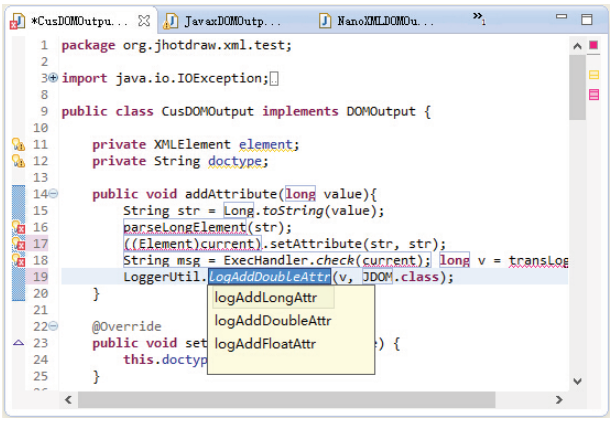

Figure 5: Screenshot of CCDemon

5. AUTOMATIC EXPERIMENT

We conduct an experimental study to answer the following research questions:

- **Q1**: How accurately can the to-be-modified positions be identified as slots?
- **Q2**: How much of the required modifications can be generated as options?
- **Q3**: How are the required modifications ranked in slots?

5.1 Design

To answer the above questions, we simulated a set of code copy, paste, and modification cases based on code clones detected from source code. We chose five open source projects (shown in Table 3) as the subject systems and used an AST-based clone detector JCCD [4] to detect code clones in each system.

For each clone set, we simulated a series of cases of code copy, paste, and modification in the following way: we take one of its clone instances cc as the copied code and another one mp as the modified pasted code; we use cc as the initial pasted code in the location of mp and use the code in the to-be-modified positions of mp as golden set; we use all its clone instances except mp as the code clones for modification recommendation; we simulate code edit for each identified slot on the pasted code in sequential order by choosing an option or directly modifying the code based on the golden set. Note that each time after a slot is simulatively edited, the options and their ranking in the other slots can be updated. We call a simulation of copy-paste-modify operation as a **case**. For a clone set with n instances, we can simulate $n \times (n-1)$ cases.

Among all the 4,209 detected clone sets that are larger than five lines of code, we filtered out two types of clone sets: Type-I clone sets, i.e., clone sets whose instances are all identical; clone sets that have only two instances. The reason is that, when simulating code copy, paste, and modification with these two types of clone sets, we have only one or several identical clone instances for modification recommendation and our approach cannot identify modification slots from past differences. So finally 1,255 out of 4,209 clone sets were used in our experimental study. This implies a rough estimation that our clone-based approach is applicable in about 29.8% of all the cases of code copy, paste, and modification.

We run all the 52,168 cases on a server with Intel Xeon CPU of 2.12GHz and 32G RAM. It took 0.48 seconds for each case on average.

Table 3: Subject Systems and Simulated Cases

Project	LOC	#Applicable Clone Set	#Case
TWE 5.0.1	90,160	157	5,104
OSWorkflow 2.8.0	47,945	71	2,544
JFreeChart 1.0.14	222,821	323	13,566
JHotDraw 7.0.6	57,049	74	2,084
JasperReport 6.0.0	496,891	630	28,870

5.2 Results

To answer **Q1**, we evaluate the precision (Q1: P) and recall (Q1: R) of slot identification in each case. For a case with cc as the copied code and mp as the modified pasted code, let IS be the set of identified slots, TP be the set of to-be-modified positions in mp, we compute the precision as $\frac{|IS \bigcap TP|}{|IS|}$ and recall as $\frac{|IS \bigcap TP|}{|TP|}$. To answer **Q2**, we compute the percentage of the to-be-modified positions that were identified as slots and had their required modifications suggested as options (Q2: RMO). To answer **Q3**, we compute the percentage of the slots where the required modifications were ranked top three (Q3: PFT). Table 4 shows the results of the cases for the five subject systems.

Table 4: Results of Automatic Experiment

Project	TWE	OSWorkflow	JFreeChart	JHotDraw	JasperReport	Overall
Q1: P (%)	62.76	77.77	72.1	61.07	64.49	67.92
Q1: R (%)	95.36	96.53	97.76	93.99	96.87	96.85
Q2: RMO (%)	68.33	84.13	82.94	74.86	71.33	75.04
Q3: PFT (%)	91.26	85.20	92.40	93.99	93.18	92.37

5.2.1 *Q1: Slot Identification*

Our approach identified slots with an overall precision of 67.92% and an overall recall of 96.85%. In some cases (about 32%), an identified slot did not need to be modified, i.e., its value in the golden set is the same with that in the copied code. Such false positives usually have little influence, since the developer can simply keep the default code in the slot. In very few cases (about 3%), a to-be-modified position cannot be identified as a slot. In other words, clone difference cannot help predict to-be-modified positions in these cases.

5.2.2 *Q2: Option Generation*

Our approach generated the required modifications as options for most (75.04% overall) of the to-be-modified positions in all the

cases. For the other positions (less than 25% overall), the code need to be modified by the "simulated user". The reasons for failing to recommend correct options lie in that 1) our sources for identifying options is limited in some cases and 2) some code naming is inconsistent in those projects. For the example of limited sources, when an unprecedented complicated expression appear in the pasted code, none of the sources in this work could help predicate it. We leave such more complicated issues in our future work. For an interesting example of inconsistent code naming, we found that most clone instances of a clone set in JasperReport system have two variables with naming rule as "*" and "*Drawer" (e.g., "*rectangle*" vs "*rectangleDrawer*"; "*ellipse*" vs "*ellipseDrawer*"). In a case of this clone set, our approach recommend a "*textElementDrawer*" option in one slot when we simulatively input "*textElement*" in its previous slot. However, the "correct" option according to original source code is "*textDrawer*", which lowers the RMO value to 0 in this case. In fact, our recommended modification incur no bugs and even improve clone consistency in this case.

5.2.3 Q3: Option Ranking

In most of the cases, the required modifications were ranked high in the corresponding slots. Overall, the required modifications in 92.37% slots were ranked top three in the option lists of their slots.

5.3 Further Analysis

We investigated the contributions of the three sources (i.e., historical code, compatible elements, and naming rules) to generate the required modifications as options in slots. The results show that 38.25% of the required modifications were generated from historical code, 35.50% from compatible elements, and 26.25% from naming rules. Clearly, all the three sources played an important role in generating required modifications as options in identified slots.

Table 5: Distribution of Different Case Types

Type	Q1: R	Q2: RMO	Num	Percentage (%)
Type#1	1	1	24144	46.28
Type#2	1	(0, 1)	14999	28.75
Type#3	1	0	3073	5.89
Type#4	(0, 1)	1	415	0.80
Type#5	(0, 1)	(0, 1)	769	1.47
Type#6	(0, 1)	0	451	0.86
Type#7	0	-	783	1.50
Type#8	-	-	7534	14.44

We classified the cases into eight types according to two criteria: how much of the to-be-modified positions were identified as slots (Q1: R); how much of the required modifications were generated as options (Q2: RMO). Table 5 shows the details. Type#1 cases are those having all their to-be-modified positions identified as slots and all the required modifications generated as options. For these cases, the developer can finish his code modification without any keystroke. Type#2 cases are those having all their to-be-modified positions identified as slots but some required modifications missing. For these cases, the developer needs to modify the code in some slots. These two types make up 75.03% of the cases. Type#3 cases are those having all their to-be-modified positions identified as slots but all the required modifications missing. For these cases, the developer needs to modify the code of all the slots. Type#4, Type#5, Type#6, and Type#7 cases are those having to-be-modified positions missing. These four types are only a small proportion of the cases. Type#8 cases are those having no to-be-modified positions, i.e., the copied code is identical with the golden set.

6. HUMAN STUDY

We conducted a human study on our tool, *CCDemon*, to investigate whether our technique can help developers in practice.

6.1 Study Design

In this study, we asked programmers to finish 6 programming tasks that can be accomplished by copying, pasting and modifying existing code. We aim to check the developers' efficiency to accomplish tasks when our recommendation is and is not available. We choose *MCIDiff* [20] as the baseline tool to compare with *CCDemon*. *CCDemon* can learn multiple clone instances to recommend modifications, while *MCIDiff* can help developers compare multiple clone instances by visualizing clone differences, but without automatic recommendation for code modification.

Sixteen graduate students from School of Software, Fudan University, participated in this study. We surveyed all the participants and divided them into two equivalent groups based on their programming experience. Participants were matched in pairs by their capability and each pair was randomly allocated to experimental group or control group. Experimental group used *CCDemon* and control group used *MCIDiff* to accomplish same tasks in the study. We gave tutorials of both tools three hours before the study and asked the participants to familiarize themselves with some example tasks with the respective tool.

We selected 6 cases generated from the JFreeChart and JasperReport system (one case per Type#1∼#6) as the experimental tasks assigned to the participants. In each task, we deleted the code of the target clone instance in projects and asked participants to complete the code based on the existing clones of the removed code. As showed in Table 6, the tasks varies from different complexity and details, in terms of number of clone instances to check, number of places to modify and the extra knowledge to learn for modification. We asked the participants to accomplish tasks from low complexity to high complexity.

Table 6: Task Description

Task	Type	Project	Gen. Desc	Complexity
#1	#1	JasperReport	3 clones for reference; 7 places to consistently modify;	low
#2	#4	JFree-Chart	2 clones for reference; 6 places to consistently modify	low
#3	#3	JasperReport	2 clones for reference; 3 places to customize	medium
#4	#5	JFree-Chart	3 clones for reference; 8 places to customize	medium
#5	#2	JFree-Chart	5 clones for reference; 25 places to consistently modify	high
#6	#6	JasperReport	2 clones for reference; 3 places to customize	high

Before the study, we briefly introduced each task and ask participants to accomplish them in a test-driven way. That is, we provided test cases for each tasks so that the participants can check whether their code is correct. To ensure the completeness of test cases, we manually go through all the data and control flow to design test cases for each deleted code. In addition, the participants were required to run a full-screen recorder throughout experiment session, which enables us to analyze their behaviors afterwards.

6.2 Results

In this study, all the participants in both groups successfully accomplished the tasks by passing all the test cases. Therefore, we evaluated their task completion time and the number of test case failures in the course of completing the tasks.

Table 7 and Table 8 show the performance of *CCDemon* group and *MCIDiff* group in terms of task completion time and the number of test case failures. Overall, *CCDemon* group accomplished 3 tasks (i.e., Task#1, Task#2, and Task#5) in shorter time compared with *MCIDiff* group, while the two groups had a tie in the number of test case failures.

Hypotheses: We introduced the following null and alternative

Table 7: Performance of *CCDemon* Group

Par	Time (s)						Test Case Failures					
	#1	#2	#3	#4	#5	#6	#1	#2	#3	#4	#5	#6
P1	74	41	485	265	156	219	0	0	0	1	0	2
P2	81	125	159	203	77	262	0	0	0	0	0	0
P3	36	38	408	288	99	316	0	0	3	0	0	2
P4	80	71	581	190	33	518	0	0	0	0	0	8
P5	31	180	204	223	197	295	0	0	1	1	0	1
P6	61	90	236	200	170	330	0	0	0	0	0	3
P7	51	110	175	750	298	340	0	0	0	0	3	0
P8	100	78	873	183	105	495	0	0	5	0	0	2
Avg	64.25	91.63	390.13	287.75	141.88	346.88	0.00	0.00	1.13	0.25	0.38	2.25
Dev	22.34	43.75	233.72	178.11	77.19	99.22	0.00	0.00	1.76	0.43	0.99	2.38

Table 8: Performance of *MCIDiff* Group

Par	Time(s)						Test Case Failures					
	#1	#2	#3	#4	#5	#6	#1	#2	#3	#4	#5	#6
P9	75	60	255	130	190	325	0	0	1	0	0	0
P10	190	145	1410	285	460	390	0	0	13	0	0	1
P11	130	135	560	170	200	520	0	0	1	0	0	0
P12	340	230	460	375	315	215	0	1	1	4	0	0
P13	120	150	255	180	210	680	0	0	0	0	0	0
P14	65	250	215	290	170	455	0	0	0	0	0	3
P15	130	280	305	430	315	330	0	0	1	1	0	1
P16	130	200	245	220	370	325	0	0	0	0	0	3
Avg	147.50	181.25	463.13	260.00	278.75	405.00	0.00	0.13	2.13	0.63	0.00	1.00
Dev	81.09	67.40	375.29	97.88	96.46	135.14	0.00	0.33	4.14	1.32	0.00	1.22

hypotheses to evaluate how different the performance of experimental and control group 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.

We used Wilcoxon's matched-pairs signed-ranked tests to evaluate the null hypothesis H0 in terms of the task completion time and the number of test case failures on each task at a 0.05 level of significance. Table 9 shows the results. Based on the results, we reject the null hypothesis for the task completion time of the Task#1, Task#2, and Task#5, and accept the null hypothesis for the task completion time of the Task#3, Task#4, and Task#6; We accept all the null hypothesis for the number of test case failures of all the tasks. Therefore, there is a significant difference between two groups in the task completion time of the Task#1, Task#2, and Task#5. Table 9 shows that *CCDemon* group complete those tasks in shorter average time. Hence, we conclude the results as follows:

- *CCDemon* group accomplish the Task#1, Task#2, and Task#5 in significantly shorter time, while there is no significant difference between *CCDemon* group and *MCIDiff* group in the task completion time of the Task#3, Task#4, and Task#6.
- There is no significant difference between both goups in the number of test case failures in all the tasks.

6.3 Results Analysis

We analyzed the participants' task videos and interviewed the participants to uncover the underlying reasons of the above results.

When CCDemon group completed with shorter time?.

We observed that the tasks in which *CCDemon* group completed with shorter time (i.e., Task#1, Task#2, and Task#5) shares common features. As described in Table 6, in the Task#1, Task#2 and Task#5, there are implicit naming rules crosscutting most of to-be-modified parts in the pasted code and their context. In addition, historical code and program context can assist in providing some reasonable options. *CCDemon* can summarize those modifying regulations, and propose accurate recommendations in these three tasks. After the participants in *CCDemon* group pasted a code fragment in these tasks, they can almost complete the correct modification within several keystrokes with the help of *CCDemon*. In contrast, the participants in *MCIDiff* group had to spend time on 1) manually summarizing those regulations from clone differences, 2) locating plausible places to be modified on the pasted code, and 3) modifying the code slot one by another. As showed in Table 9, the mean task completion time used by *CCDemon* group was only half of that used by *MCIDiff* group. In addition, the more to-be-modified parts on the pasted code are (7 for Task#1, 6 for Task#2, and 25 for Task#5), the larger the time gap of the two groups is. Noteworthy, Table 5 shows that such cases account for the majority of copy-paste-modify operations of the cloned code (75.0%).

On the other hand, when the modifying regulations are not obvious in the tasks (i.e., Task#3, Task#4, and Task#6), the advantage of *CCDemon* group diminished. Based on the task videos, we found that when clone differencing cannot indicated any regulation, both groups began to gain other hints by reading code comments and checking relevant code. However, both *CCDemon* and *MCIDiff* are not designed for program comprehension. Thus, the task completion time in the Task#3, Task#4 and Task#6 relied mainly on personal programming capability. As participants are divided into equivalent groups, there is no significant difference in these tasks.

The tie in test case failure times.

In this experiment, even if we introduced the tasks before the study, the participants were still unfamiliar with subjective systems. In addition, the participants accomplished the tasks in a test-driven way. We observed that all the participants adopted an error-and-trial strategy in the study. For the tasks with modifying regulations (Task#1, Task#2, and Task#5), successful summarizing these regulations can help both groups accomplish tasks. As Table 9 shows, the mean number of test case failures in both groups are almost 0. As the task videos indicated, although *MCIDiff* group took more time, most of the *MCIDiff* users can have an accurate summarization due to *MCIDiff*'s intuitive visualizing facility for comparing clone instances. In contrast, for the tasks without modifying regulations (Task#3, Task#4, and Task#6), the failures happened when the participants run test cases before fully understanding the code. Therefore, both groups shared similar test-failure performance as they all suffered from unfamiliarity with the subjective system.

We concluded that *CCDemon* can improve the efficiency of modifying a pasted cloned code when the implicit modifying regulations exist, while its capability is limited when the regulations disappear. When there are no obvious modifying regulation, project knowledge is a dominant factor to accomplish the tasks. We deem that *CCDemon* is useful in general because the former case account for the majority of copy-paste-modify operations on the cloned code (see Table 5). Furthermore, although *MCIDiff* can help developers summarize rules, our post-survey on the *MCIDiff* participants reveals their unwillingness to choose *MCIDiff* when pasting the code. Their feedbacks suggest that they will be distracted when switching from the working code to comparing clone differences in a different viewer. In contrast, *CCDemon* had more positive feedbacks for its light-weighted usage manner. Therefore, we deem that it is more practical to run a "demon" in IDE to make recommendations to developers when they are modifying the pasted code.

6.4 Threats to Validity

There are mainly three threats in our human study. First, we used six tasks in this study, which may not be representative for all the cases. In order to mitigate this threat, we selected tasks based on different types of cases to make these tasks as representative as possible. Second, our recruited participants are graduate students who

Table 9: Results of Wilcoxon's test hypotheses, for the variable completion time (time) and test case failure time (fails). Measurements are reported in the following columns: mean, standard deviation (Std. Dev), Z statistics (Z), statistical significance (p-value).

	T1-time		T2-time		T3-time		T4-time		T5-time		T6-time		T1-fails		T2-fails		T3-fails		T4-fails		T5-fails		T6-fails	
Group	EG	CG	EG	CG	EG	CG	EG	CG	EG	CG	EG	CG	EG	CG	EG	CG	EG	CG	EG	CG	EG	CG	EG	CG
Samples	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
Mean	64.25	147.50	91.63	181.25	390.13	463.13	287.75	260.00	141.88	278.75	346.88	405.00	0.00	0.00	0.00	0.13	1.13	2.13	0.25	0.63	0.38	0.00	2.25	1.00
Std.Dev	22.34	81.09	43.75	67.40	233.72	375.29	178.11	97.88	77.19	96.46	99.22	135.14	0.00	0.00	0.00	0.33	1.76	4.14	0.43	1.32	0.99	0.00	2.38	1.22
Z	-2.52		-2.10		-0.14		-0.42		-2.34		-0.70		0.00		-1.00		-0.09		-0.38		-1.00		-1.12	
p-value	0.01		0.04		0.89		0.67		0.02		0.48		1.00		0.32		0.93		0.71		0.32		0.26	
Decision	***Reject***		***Reject***		*Accept*		*Accept*		***Reject***		*Accept*		*Accept*		*Accept*		*Accept*		*Accept*		*Accept*		*Accept*	

were not very familiar with the subject systems. Further studies are required to generalize our findings in large-scale industrial systems and with developers familiar with the subject systems. Third, we assume that the experimental group is equivalent with the control group in their capability and experience, which may be threatened by the actual differences between the two groups. To mitigate this threat, we allocated participants with comparable capability and experience into different groups based on our pre-study survey.

7. RELATED WORK

Program Differencing: Program differencing techniques [6, 8, 18, 20, 32, 33] are widely used in software maintenance tasks. Most of the existing differencing techniques can only differentiate two programs. For example, Cottrell et al. [6] developed a pairwise differencing technique to detect correspondences between two pieces of code for code generalization. Our previous technique MCIDiff [20] is designed to detect differences among multiple clone instances. However, its results does not fit for this recommendation technique. Thus, we developed a new clone differencing technique as a part of our recommendation technique to address the issues. Its main technical difference with MCIDiff lies in two folders: 1) this work defines and uses optimal common subsequence of multiple token sequences regarding program syntax, instead of the traditional LCS used in MCIDiff, to identify common part of clones; and 2) this work merges and splits differential ranges to report sequence-based multisets, in contrast, MCIDiff iteratively computes the LCS across differential ranges and heuristically match tokens in differential ranges to report token-based multisets.

Code Completion: Nguyen et al. [24] proposed a graph-based code completion technique by matching half-developed code with the mined API usage pattern. Zhang et al. [34] proposed a technique to recommend method parameters by learning from existing parameter usage examples in source code. Asaduzzaman et al. [2] improved API method call completion by analyzing the associated relation between contextual information and method call. Apart from the above traditional completion techniques, Hindle et al. [10] shows that statistical language model is suitable to model code for code completion. Based on their findings, Allamanis et al. [1] developed a framework *NATRUALIZE* based on statistical language model to suggest natural identifier names and formatting conventions for developers. Our work is different from these existing work. What we recommend is the intended modification on existing pasted code, in contrast, what the above techniques recommend is either the intended unwritten code or the ranking adjustment of API method call completion.

Code Modification: Kim et al. [15] conducted an ethnographic study to understand programmers' copy-paste programming practices and summarize a number of taxonomy of copy-paste patterns. Doerner et al. [7] developed a tool called *EUKLAS* to help quick-fix syntax errors of the pasted Javascript code. Kerr et al. [14] further proposed a set of heuristic rules about building the syntactic relation between copied code and its context in order to suggest modifications on the pasted code with the rules. Jablonski et al. [12] developed a tool to consistently modify synonymous variables on the pasted code. Meng et al. [23]'s *LASE* tool learns editing script from changes of several code examples to change the similar code systematically. Our approach does not require change history. In addition, we provide recommendations incrementally and interactively for developer to modify the pasted code.

8. DISCUSSION AND CONCLUSION

Kim et al. [15] found that developers usually employ their copy-and-paste memory when changing code, but such memory is instant, inaccurate, and difficult to transfer from person to person. Our study not only statistically supports their findings, but also makes a further conclusion that, when a copied code is involved in code clones, the clone differences can generally help to recommend the modification on the pasted code. The interactive modification recommendation can not only improve the efficiency of copy-and-paste based code reuse, but also improve code quality by reducing bugs and inconsistent naming conventions caused by incomplete or inconsistent code modifications.

The practical use of our approach involves the integration with existing clone management techniques such as incremental clone detection [25] and quick clone searching [19]. It is also possible to integrate our approach with Internet code search engines. Some code search engines allow a developer to search code from Internet by writing a query and get returned code snippets inside the IDE. Instead of returning one most relevant code snippet, the code search engine can return multiple similar code snippets for interactive modification recommendation. Then the developer can edit the code with the support of interactive modification recommendation based on returned similar code snippets.

In summary, we have proposed a clone-based and interactive recommendation approach to modifying pasted code. Once a code copy-and-paste operation is detected, our approach retrieves from an existing clone repository a related clone set and detects syntactically complete differences among clone instances. The approach transfers each clone difference into a slot on the pasted code and generates an initial set of options for the slot based on historical code, compatible contextual elements and mined naming rules. Then the approach supports the developer to modify the pasted code by interactively updating recommended options and their ranking along with developers' code editing. We have implemented our approach in a proof-of-concept tool and evaluated the effectiveness and usefulness of our approach and tool with an automatic experiment and a human study.

9. ACKNOWLEDGE

This work was supported by the National Natural Science Foundation of China under Grant No. 61370079, National High Technology Development 863 Program of China under Grant No. 2013 AA01A605, Shanghai Science and Technology Development Funds under Grant No. 13dz2260200, 13511504300.

10. REFERENCES

[1] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton. Learning natural coding convention. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

[2] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. CSCC: Simple, efficient, context sensitive code completion. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 71–80, 2014.

[3] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 24–33, 2007.

[4] B. Biegel and S. Diehl. JCCD: a flexible and extensible api for implementing custom code clone detectors. In *Proceedings of the International Conference on Automated Software Engineering*, 2010.

[5] D. Cai and M. Kim. An empirical study of long-lived code clones. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, pages 432–446. 2011.

[6] R. Cottrell, J. J. Chang, R. J. Walker, and J. Denzinger. Determining detailed structural correspondence for generalization tasks. In *Proceedings of the the joint meeting of the European software engineering conference and the symposium on the foundations of software engineering*, pages 165–174, 2007.

[7] C. Doerner, A. R. Faulring, and B. A. Myers. EUKLAS: Supporting copy-and-paste strategies for integrating example code. In *Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 13–20, 2014.

[8] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Transaction on Software Engineering*, 33(11):725–743, 2007.

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

[10] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the International Conference on Software Engineering*, pages 837–847, 2012.

[11] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[12] P. Jablonski and D. Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *Proceedings of the OOPSLA workshop on eclipse technology eXchange*, pages 16–20. ACM, 2007.

[13] C. J. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[14] R. Kerr and W. Stuerzlinger. Context-sensitive cut, copy, and paste. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, pages 159–166, 2008.

[15] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 83–92, 2004.

[16] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the joint meeting of the European software engineering conference and the symposium on the foundations of software engineering*, pages 187–196, 2005.

[17] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*, 2006.

[18] G. P. Krishnan and N. Tsantalis. Unification and refactoring of clones. In *Proceedings of the Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, pages 104–113, 2014.

[19] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim. Instant code clone search. In *Proceedings of the international symposium on Foundations of software engineering*, pages 167–176, 2010.

[20] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao. Detecting differences across multiple instances of code clones. In *Proceedings of the International Conference on Software Engineering*, pages 164–174, 2014.

[21] H. Liu, Z. Ma, W. Shao, and Z. Niu. Schedule of bad smell detection and resolution: A new way to save effort. *Transaction on Software Engineering*, 38(1):220–235, 2012.

[22] M. Mantyla, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance*, pages 381–384, 2003.

[23] N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *Proceedings of the International Conference on Software Engineering*, pages 502–511, 2013.

[24] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the International Conference on Software Engineering*, pages 69–79, 2012.

[25] T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, and T. N. Nguyen. Scalable and incremental clone detection for evolving software. In *Proceedings of the International Conference on Software Maintenance*, pages 491–494. IEEE, 2009.

[26] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the International Conference on Software Engineering*, pages 315–324, 2010.

[27] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? *Empirical Software Engineering*, 17(4-5):503–530, 2012.

[28] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 2007-541, School of Computing, Queen's University, Canada, 2007.

[29] C. E. Shannon. A mathematical theory of communication. *Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[30] G. J. Szekely and M. L. Rizzo. Hierarchical clustering via joint between-within distances: Extending ward's minimum variance method. *Journal of classification*, 22(2):151–183, 2005.

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

[32] Z. Xing. Model comparison with genericdiff. In *Proceedings of the International Conference on Automated Software Engineering*, pages 135–138, 2010.

[33] Z. Xing and E. Stroulia. Differencing logical uml models. *Proceedings of the International Conference on Automated Software Engineering*, 14(2):215–259, 2007.

[34] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical API usage. In *Proceedings of the International Conference on Software Engineering*, pages 826–836, 2012.

[35] G. Zhang, X. Peng, Z. Xing, and W. Zhao. Cloning practices: Why developers clone and what can be changed. In *Proceedings of the International Conference on Software Maintenance*, pages 285–294, 2012.