CnP: Towards an Environment for the Proactive Management of Copy-and-Paste Programming

Daqing Hou, Patricia Jablonski, and Ferosh Jacob
Electrical and Computer Engineering, Clarkson University, Potsdam, NY 13699
{dhou, jablonpa, jacobf}@clarkson.edu

Abstract

Programmers copy and paste code for many reasons. Regardless of the specific reasons, similar code fragments (clones) are introduced into software systems. Like other software artifacts, clones may require attention and effort from programmers so that they can be understood, and correctly adapted and evolved. More specifically, when understanding and maintaining clones, programmers need to know where the clones are. Programmers also need to compare and contrast code clones in order to figure out how they correspond and differ. Finally, they also need to edit or remove clones. In addition to what clone detection-based tools can offer, more automated support is needed to better assist programmers in these activities. In this paper, we introduce a toolkit CnP that is aimed to support and manage clones proactively as they are created and evolved. We describe the initial features and the design decisions taken in CnP. We also discuss possible future design extensions.

1. Introduction

It is probably not too far away from reality to say that all programmers likely copy and paste code at least once in a while. Copying and pasting often results in duplicated code, or clones. For example, M. Kim et al. observed in a field study of the copy-and-paste programming practice that programmers on average made four non-trivial clones per hour [10], which appears to be consistent with our own experiences as programmers. When clones are created, a dependency is introduced between the original code and the new copy. When the complexity of such dependencies surpasses a programmer's capability of handling them, it starts to create problems, like those observed in [11]. It is even a challenge to simply know the existence and the locations of clones in a large system as it is infeasible to manually search for copied code in a large system. This has motivated a large body of work on clone detection tools and techniques [13, 16].

There are many reasons and motivations for programmers to copy and paste code, but not all of them are totally unjustified [9, 10]. We feel that clones should be viewed from a process point of view and supported by tools as such. Appropriate tool support can help maximize the benefits of cloning while mitigating or eliminating its negative effects. In addition to replacing clones with abstractions, some new techniques have been developed to help maintain clones while they are still in the source code, e.g., simultaneous editing [5, 15, 17]. However, prior research either completely relies on clone detection techniques or requires programmers to manually specify clones. Clone detection tools tend to have low precision and recall, may miss type-3 clones, or suffer from poor performance [1]. Clearly, there is still much room for improvement.

As suggested by many others as well [8, 9, 10, 14], a complementary direction is to proactively track a programmer's copy-and-paste actions in an editor or integrated development environment (IDE), and automatically identify the resulting clones. (Hereafter, the term "clones" refers to copy-and-paste-induced clones.) Once tracking down the creation of clones, other proactive tools can be provided to systematically support clone evolution. In this way, proactive support directly helps programmers manage clones in the context of active development, with the potential to offer the following important benefits.

- Once activated, proactive support will capture and support the evolution of *all* clones. Proactive support can benefit from clone detection tools, but does not rely on them to function. Unlike the batch processing mode in clone detection tools, proactive support captures clones incrementally, spreading the effort evenly over time. Moreover, some clones are ephemeral and may disappear from the code base before detection tools are applied. A proactive approach will be able to capture and manage ephemeral clones.
- When understanding code clones, one must read and compare similar code multiple times to decide whether the clones are the same or what the correspondence and differences are between them. Such comparison can be fairly expensive and requires close attention to details. It can also be difficult when there are subtle dif-

ferences in the clones or their contexts. Programmers would welcome tools that can help reduce such pains.

- We have seen evidence that clones are changed in ways that obscure their otherwise obvious structural correspondence. With proactive support, clones could have been maintained by programmers more closely, and thus are more likely to be kept consistent. As a result, it will be easier to understand or refactor the clones.

We outline the main design elements of such proactive support as well as experience of tool prototyping in the CnP project [1] (Section 2). Our prototyping is based on Eclipse [2], which has helped ensure that our design decisions are technically feasible. The design elements identified were partially inspired by a survey of related work. CnP features are contrasted with related work in Section 3. We are currently conducting user studies for selected CnP features.

2. Design Elements for Proactive Support

The overall design goal for a proactive copy-and-paste support system is to provide a sufficiently broad set of tools to cover all activities that may happen to a clone throughout its life cycle. Such a proactive system will assist programmers in *capturing and representing* clones, *visualizing* "important" information about clones (for example, the correspondence between two or more clones), *editing* clones, and removing cloning relationships.

2.1. Clone Model and Clone Capture

A clone model includes both coordinates for individual clones and the cloning relationship between related clones. CnP automatically detects the creation of new clones that are made via copying and pasting. Specifically, it intercepts copying and pasting actions in Eclipse, which may happen within both editors and views and in multiple ways. CnP's clone coordinates consist of the character *offset* and *length* from the source code in a particular *file*. After clones are created, CnP will accurately maintain their locations when code is added or deleted before or within the clone regions. As long as clones are evolved within the IDE, it is guaranteed that CnP always provides accurate clone locations. CnP also persists the clone model between IDE sessions in a file. [3] The model gets saved automatically whenever Eclipse quits, and loaded automatically when Eclipse starts up. CnP's clone model covers the whole workspace, not just individual projects.

Since the copy-and-pasted code may not all be syntax-valid, CnP extracts as clones only the continuous sequence of the top-most, whole AST nodes that completely fall within the range. Pasted source code that is only partially contained within an AST node will be excluded by CnP. It is also important to note that not all of the pasted code fragments are necessarily clones of interest to be tracked. Research on copy-and-paste usage patterns by M. Kim, et al., shows that programmers often copy very small pieces of code (approximately 74% less than a single line) [10]. The programmer is simply copying and pasting to save typing or to remember a name's spelling. If CnP tracks such code as clones, programmers may be forced to gratuitously remove them explicitly in order to keep their environment clean. To filter out such code, CnP chooses to track only clones that would appear to be "significant" to programmers. Unfortunately, there is not a clear definition of "significant clones" that can be automated. Therefore, proactive clone tracking needs to provide ways so that programmers can define some configurable policy to filter out uninteresting copy-and-pasted code. The policy that CnP currently uses for determining whether a copied code fragment is a clone suitable for tracking is if and only if it contains (1) more than two statements, or (2) at least one conditional statement, loop statement, or method, or (3) a type definition (class or interface). On top of this basic filtering, programmers are also given the option to take clones out of a clone group. [4]

In addition to tracking clones created by copy-and-paste, proactive support can also obtain clones from a clone detection tool. This can be useful when applying proactive support to existing code. CnP now supports the importing of clone data from the SimScan clone detection tool [5].

2.2. Clone Visualization

Clone visualization can make programmers aware of the clones in a system, and assist them in understanding and debugging clones. It can be done at the system level, between two or more related clones, or within individual clones.

At the system level, CnP provides two views for clones. One view shows the imported clones, and the other shows the clones being tracked by CnP. CnP currently shows clones visually by displaying colored bars next to them in the editor, as shown in Figure 1a.

Once created, a clone may be subsequently evolved, where code can be added, updated, or deleted. Visually highlighting these three kinds of change directly in the editor can be a very useful aid to programmers who want to understand clones. Another feature of CnP named CSeR (Code Segment Reuse) supports precisely this. When initially copied and pasted, the new code fragment is identical to the original. Therefore, for proactive support that focuses on copy-and-paste-induced clones, one way to offer this kind of service is to actively follow a programmer's editing actions, inferring and visualizing such changes in real time

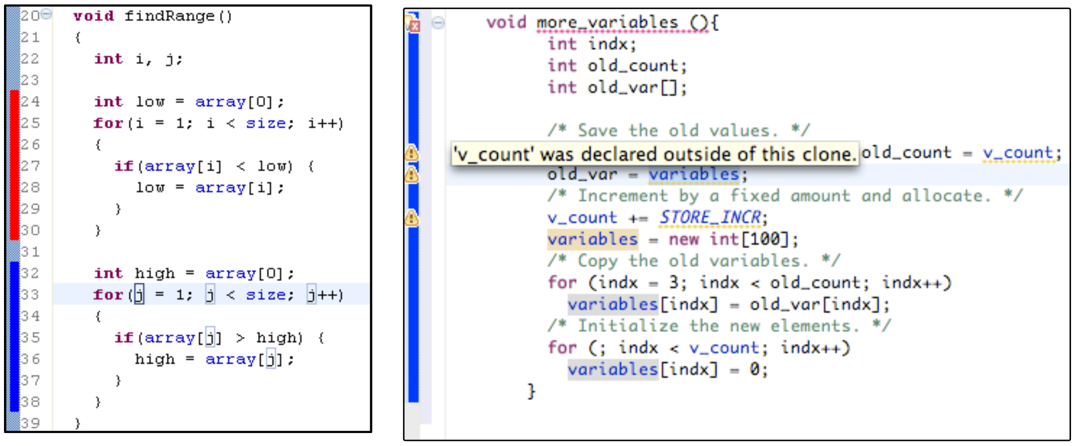
1 `www.clarkson.edu/~dhou/projects/CnP`

2 `www.eclipse.org`

3 This would imply that similar to source files, clone information can be managed by revision control systems.

4 Operations that support merging multiple clones may also be useful.

5 `www.blue-edge.bg/simscan`



```
void findRange()
{
  int i, j;

  int low = array[0];
  for(i = 1; i < size; i++)
  {
    if(array[i] < low) {
      low = array[i];
    }
  }

  int high = array[0];
  for(j = 1; j < size; j++)
  {
    if(array[j] > high) {
      high = array[j];
    }
  }
}
```

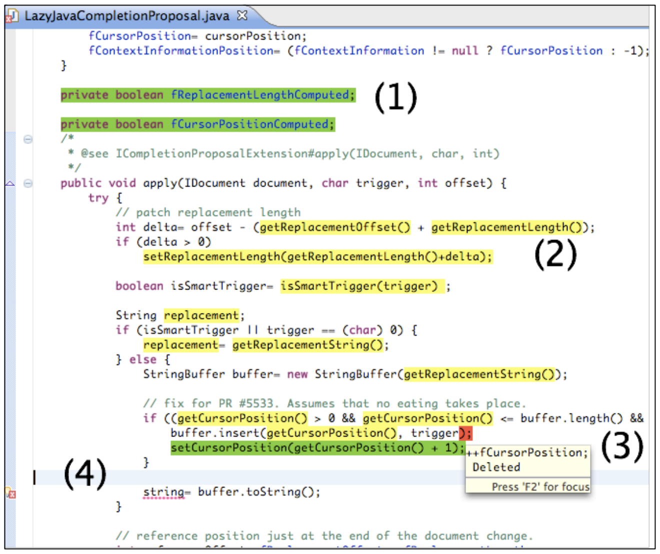
```
void more_variables (){
    int indx;
    int old_count;
    int old_var[];

    /* Save the old values. */
    old_count = v_count;
    old_var = variables;
    /* Increment by a fixed amount and allocate. */
    v_count += STORE_INCR;
    variables = new int[100];
    /* Copy the old variables. */
    for (indx = 3; indx < old_count; indx++)
      variables[indx] = old_var[indx];
    /* Initialize the new elements. */
    for (; indx < v_count; indx++)
      variables[indx] = 0;
  }
```

'v_count' was declared outside of this clone.

(a) Clone Tracking with CnP (Original Code Shown with Red Bar and Pasted Code with Blue) and Consistent Renaming with CReN.

(b) Warnings about Accidental Identifier Capture within a Clone.

Figure 1: CnP Features (Clone Tracking, Consistent Renaming, and Accidental Identifier Capture).



LazyJavaCompletionProposal.java

```
        fCursorPosition= cursorPosition;
        fContextInformationPosition= (fContextInformation != null ? fCursorPosition : -1);
    }

    private boolean fReplacementLengthComputed;

    private boolean fCursorPositionComputed;
    /*
     * @see ICompletionProposalExtension#apply(IDocument, char, int)
     */
    public void apply(IDocument document, char trigger, int offset) {
        try {
            // patch replacement length
            int delta= offset - (getReplacementOffset() + getReplacementLength());
            if (delta > 0)
                setReplacementLength(getReplacementLength()+delta);

            boolean isSmartTrigger= isSmartTrigger(trigger) ;

            String replacement;
            if (isSmartTrigger || trigger == (char) 0) {
                replacement= getReplacementString();
            } else {
                StringBuffer buffer= new StringBuffer(getReplacementString());

                // fix for PR #5533. Assumes that no eating takes place.
                if ((getCursorPosition() > 0 && getCursorPosition() <= buffer.length() &&
                    buffer.insert(getCursorPosition(), trigger);
                    setCursorPosition(getCursorPosition() + 1);
                }

                string= buffer.toString();
            }

            // reference position just at the end of the document change.
```

++fCursorPosition;
Deleted
Press 'F2' for focus

Figure 2: CSeR (Code SEgment Reuse) Incrementally Tracks Edits Made to a Copied-and-Pasted Class and Visualizes Code That Is Added, Updated, and Deleted with Different Colors inside the Editor (Figure Better Viewed Online to See the Colors).

in the editor where the programmer is working. [6] Specifically, CSeR builds up the correspondence between "interesting" code elements in the original code and the clone. Edits that happen within these code elements are labeled as "update". Edits that happen between "interesting" code elements are labeled as "new" or "added". Of course, edits that remove an "interesting" code element cause CSeR to show a deletion in the editor.

Figure 2 depicts how CSeR visualizes changes with different colors when a copied-and-pasted class is being modified in the Eclipse Java editor. The class shown is LazyJavaCompletionProposal, which was copied from JavaCompletionProposal in jdt.ui 3.2. Figure 2 shows only a small portion of its code. Specifically, location (1) contains two fields that are newly added, colored in green. The multiple code elements colored in yellow around location (2) are the results of replacing direct access to fields by getters and setters and long boolean expressions by predicates. At location (3), a red marker shows that there are some codes deleted after it, and the tooltip shows the code that was deleted. Finally, near location (4), there is a compilation error on the identifier string. This is because the name of this variable has been changed to replacement a few lines above. Currently, CSeR can visualize the differences between a pair of clones directly inside the code editor. But its internal design can be readily generalized to support the comparison of multiple clones side by side at the same time. We are also improving the basic design of CSeR so that it can work reliably in the presence of arbitrary editing practice such as those presented in [12].

Currently, no correspondence data are being tracked for SimScan-reported clones, since it is less obvious how to automatically determine the correspondences among clones that have already evolved very differently, unlike native clones that are identical when initially copied and pasted. It should be possible to convert these clones into a form that can be managed by CnP by combining code comparison techniques, like those of [3, 4, 6], which automatically compute the fine-grained correspondence between detected clones, with user input when necessary.

6 Note that CSeR relies on parsing rather than diff-like techniques.

Visualizing clone information may also help prevent or detect clone-related errors. CnP contains features that may help prevent errors (like CReN). It also contains features that detect potential errors (warnings) in the tracked clones. One example is warning about the accidental capture of external identifiers, which is an instance of the interaction between a clone and its context. CnP issues a warning if an identifier in the pasted code binds to a declaration in the enclosing context. For example, the method "more_variables" in Figure 1b was copied and pasted within the same class. Once pasted, Eclipse shows an error for the method name "more_variables" because a method of the same name already exists in the class. On top of Eclipse, CnP also provides three warnings, one for each of the three variables "v_count", "variables", and "STORE_INCR", which are fields in the class. These warnings alert the programmer that these particular identifier instances within the clone (method) may need to be renamed. This is useful, since it is common for programmers to copy and paste a code fragment that contains references to external identifiers that are intended only in the original fragment. The programmer can then use CReN to rename the identifier instances in the pasted region, if desired.

More features similar to the accidental capture of external identifiers can be added to a proactive system. For example, proactive support may include features that infer commonalities between and within clones. These can then be used to either assist in coding, e.g., common lexical patterns ("leftWnd"/"rightWnd"), or alert about "unusual" facts or relationships, e.g., inconsistent data and control flow between clones and their contexts.

2.3. Clone Editing

Clone editing is broadly construed as any feature that supports the modification of clone code. Clone code editing techniques can be further distinguished between those that make consistent edits *within a clone* (e.g., consistent renaming) and those *between clones* (e.g., simultaneous editing). Clone editing also includes the maintenance of the clone model like removing a clone group. The CSeR described above could also be considered an editing feature. Clone refactoring can also be included for completeness.

2.3.1. Intra-Clone Editing (Consistent Renaming) CnP contains a renaming utility (CReN) [7] that helps rename identifiers consistently *within* clones or any programmer-selected region, as shown in the lower part of Figure 1a. CReN uses a heuristic that identifiers referring to the same program element within a code fragment should be renamed together consistently. To know which identifier instances in a code fragment are to be renamed consistently together, CReN groups together instances that bind to the same program element. If bindings are not available, CReN will simply rename identifier instances within the clone that have the same name. In this way, CReN helps speed up coding efficiency. Perhaps more importantly, CReN also helps prevent inconsistent renaming errors because manual renaming can miss an instance that was intended to be renamed. When the declaration is outside of the fragment, the missed name can still be okay according to the compiler (since it is still in scope), so programmers would not be alerted of the missed instance.

The most common use case for CReN is an *open* code fragment that contains free identifiers. *free* means that the definition of the identifier does not appear in the code fragment itself. In such cases, CReN can be beneficial in situations where existing refactoring support falls short. Examples for CReN can be found in [7].

2.3.2. Inter-Clone Editing (Simultaneous Editing) A number of editing techniques have been proposed to help maintain common updates and changes between similar code fragments (linked editing [17], which was also called synchronous editing [9], and simultaneous editing [15]). However, they all require the programmer to manually specify the clones in order to work.

Although not currently available, support for simultaneous editing between clones will be added to CnP as well. CnP's simultaneous editing feature is likely to offer a better user experience since clones are already available in CnP and programmers do not have to manually select them any more. Another important difference is that CnP relies on parsing and maintains clone correspondence (as currently done with CSeR). In contrast, simultaneous editing in prior work relies on Longest Common Subsequence (LCS) algorithm as in [17], or Levenshtein Distance (LD) as in [2, 5, 18]. Both LCS and LD have practical problems.

2.3.3. User Interaction and Intervention A design goal is that proactive support should provide sufficient interaction so that programmers can still be in control and yet minimize user intervention when programmers have to correct mistakes created by tools. Since it is impossible to have an algorithm that decides whether two code fragments behave in a way of interest to a programmer, CnP allows the programmer to remove a clone from a clone group or stop the tracking the whole group. In this way, the programmer can control which clones can be considered related. In the interest of efficiency, CReN is automatically applied when any identifier is modified. CReN also allows to selectively undo the automatic renaming on a particular identifier. An alternative design would be to ask a programmer to explicitly switch to a "CReN mode". Similar considerations should also apply to the design of other CnP features like simultaneous editing. These design decisions should be tested with user studies to understand their effectiveness.

3. Related Work

Mainstream research on clones is based on clone detection. Surveys of clone-related research and clone detection techniques can be found in [13] and [16]. CnP differs by tracking copied-and-pasted clones directly rather than relying on clone detection tools. As a result, CnP may support those clones that clone detection tools fail to capture. Moreover, clone detection tools tend to have low precision and recall [1], which can be expensive for the programmer to sort through in batch processing. Proactive support can potentially ease this problem by distributing the effort over time.

Clonescape [2] and CPC [18] are recent projects aimed to develop proactive clone support. CnP differs from Clonescape and CPC in providing features like the accidental capture of external identifiers, consistent renaming (CReN), and clone diff view (CSeR).

CloneTracker [5] proposes a novel representation for clone locations that is independent of physical properties like character offsets or line ranges, and also supports a form of simultaneous editing by using Levenshtein Distances to locate similar lines between clones. But CloneTracker relies on clone detection. CnP may also perform more accurately in cases when Levenshtein Distances fail.

Codelink [17] supports both clone diff views and simultaneous editing. It uses colors to indicate the commonalities between linked clones in blue and differences in yellow and elision to hide the identical parts of the clones from view. However, unlike CSeR, Codelink does not distinguish between "new" and "updated" code. Codelink uses the longest-common subsequence (LCS) algorithm (like the one implemented by the Unix "diff" utility) to determine the commonalities and differences of clones within a clone group. Toomim et al. report two main shortcomings of the LCS algorithm: its potentially long running time and lack of intuitive results. CnP's approach in differencing clones can potentially resolve these problems.

CReN and Rename Refactoring mainly differ in that CReN works in any user-specified region while Rename works only in pre-defined scopes like blocks and classes.

The Breakaway and Jigsaw tools automatically determine the detailed structural correspondences between two classes and two methods, respectively [3, 4]. The input to Breakaway and Jigsaw are classes or methods that may contain different code. The input to CSeR, on the other hand, is always identical. CSeR *incrementally* tracks changes as they are made to the related clones.

Acknowledgments

The authors thank the three anonymous reviewers for their constructive comments, suggestions, and questions.

References

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[2] A. Chiu and D. Hirtle. Beyond Clone Detection. Course Project, CS846 Spring 2007, University of Waterloo. `http://www.cs.uwaterloo.ca/~dhirtle/publications/beyond_clone_detection.pdf`. Accessed Jan. 12, 2009.

[3] R. Cottrell, J. Chang, R. Walker, and J. Denzinger. Determining Detailed Structural Correspondence for Generalization Tasks. In *Proceedings of ESEC/FSE'07*, 2007.

[4] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating Small-Scale Source Code Reuse via Structural Correspondence. In *Proceedings of FSE'08*, pages 214–225, 2008.

[5] E. Duala-Ekoko and M. Robillard. Tracking Code Clones in Evolving Software. In *Proceedings of ICSE'07*, 2007.

[6] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[7] P. Jablonski and D. Hou. CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE. In *Eclipse Technology Exchange Workshop at OOPSLA (ETX)*, 2007.

[8] L. Jiang, Z. Su, and E. Chiu. Context-Based Detection of Clone-Related Bugs. In *Proceedings of ESEC/FSE'07*, 2007.

[9] C. Kapser and M. Godfrey. 'Cloning Considered Harmful' Considered Harmful. In *Proceedings of WCRE'06*, 2006.

[10] M. Kim, L. Bergman, T. Lau, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of ISESE'04*, 2004.

[11] A. J. Ko, H. Aung, and B. A. Myers. Eliciting Design Requirements for Maintenance-Oriented IDEs: a Detailed Study of Corrective and Perfective Maintenance Tasks. In *Proceedings of ICSE'07*, pages 126–135, 2005.

[12] A. J. Ko, H. H. Aung, and B. A. Myers. Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing. In *Proceedings of CHI'05*, pages 1557–1560, 2005.

[13] R. Koschke. Survey of Research on Software Clones. In *Dagstuhl Seminar Proceedings*, 2006.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of OSDI'04*, 2004.

[15] R. Miller and B. Myers. Interactive Simultaneous Editing of Multiple Text Regions. In *USENIX Annual Technical Conference*, 2001.

[16] C. Roy and J. Cordy. A Survey on Software Clone Detection Research. Technical Report 2007-541, Queen's University, 2007. `http://www.cs.queensu.ca/TechReports/Reports/2007-541.pdf`. Accessed Jan. 12, 2009.

[17] M. Toomim, A. Begel, and S. Graham. Managing Duplicated Code with Linked Editing. In *Proceedings of VL/HCC'04*, 2004.

[18] V. Weckerle. CPC: An Eclipse Framework for Automated Clone Life Cycle Tracking and Update Anomaly Detection. Master's thesis, Free University of Berlin, 2008. `http://cpc.anetwork.de/thesis/thesis.pdf`. Accessed Jan. 12, 2009.