Exploring the Design Space of Proactive Tool Support for Copy-and-Paste Programming

Daqing Hou, Ferosh Jacob, and Patricia Jablonski

Electrical and Computer Engineering, Clarkson University, Potsdam, NY 13699
{dhou, jacobf, jablonpa}@clarkson.edu

Abstract

Programmers copy and paste code for various reasons, leaving similar code fragments (clones) in the software systems. As a result, they must spend effort and attention in tracking, understanding, and correctly adapting and evolving clones. Ideally, an integrated environment should be used to fully and seamlessly assist programmers in all these activities. Unlike clone-detection-based tools, such environments should support and manage clones proactively as clones are created and evolved. In this paper, an initial design space for such environments is sketched based on our experience with a prototype named *CnP*. The features and design decisions taken in *CnP* as well as remaining design problems are identified. To assess the potential of the developed features and identify new design input, code clones in several real-world systems are examined, and their implications on the design (and redesign) of proactive clone tools are discussed.

1 Introduction

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 the programmers on average made four non-trivial clones per hour [12]. When clones are created, dependencies are introduced between the original code and the new copy. When the complexity of such dependencies surpasses a programmer's handling capability, they start to create problems, like those observed in [14]. 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. Indeed, this has motivated the large body of work on clone detection tools and techniques (see, e.g., [16, 19]).

There are many reasons and motivations for programmers to copy and paste code, but not all of them are totally unjustified [11, 12, 15]. In addition to clone detection and removal, clones also need to be managed [15]. Appropriate tool support can potentially help maximize the benefits of cloning while mitigating or eliminating its negative effects.

As suggested by several others as well [9, 11, 12, 17], one way to managing clones 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, such proactive clone management environments (PCM) directly help programmers manage clones during active development, potentially offering the following 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 clone detection tools that work in the batch processing mode, 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 [13]. A proactive approach will be

able to capture and manage ephemeral clones.

- We have seen evidence that clones are changed in ways that obscure their otherwise obvious structural correspondences. 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 (Section 3.4).
- When understanding code clones, one must read and compare similar code multiple times to decide whether the clones are the same or what the correspondences and differences are between them. Such comparison can be fairly expensive and requires close attention to details. It can also be difficult to pinpoint the subtle differences in clone contexts. Programmers would welcome tools that help reduce such pains (e.g., [3]).

We have been working on a PCM prototype called CnP [1] for about 2.5 years. This paper is an attempt to *explore the initial design space of PCM* based on our experience with CnP so far, not its final, formal validation, which needs to be addressed separately. In Section 2, we outline the main design elements for PCM as well as key design decisions and remaining design problems. To motivate the need for, and inform the detailed design of PCM, case studies of code clones were conducted using two popular clone detection tools, CCFinder [2] and SimScan [3]. The case studies and lessons learned are reported in Section 3. A discussion of related work appears in Section 4. Finally, Section 5 concludes the paper.

2 Design Space for Proactive Clone Management Environments

An overarching design goal is that proactive clone management environments (PCM) must provide programmers with a sufficiently broad set of tools to cover all activities that may happen to a clone throughout its life cycle. Figure 1 sketches the main elements in the design space of such systems. The basis of a PCM is a clone model that captures and represents individual clones and their relationship. As shown by the four rectangles in the middle, such a proactive system will

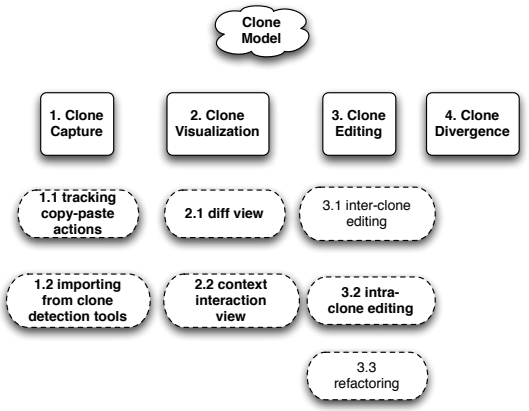


Figure 1: Clone Life Cycle and Possible Tool Support in Each Stage. Areas Where CnP Has Support Are Shown in Bold.

assist programmers in capturing clones, visualizing "important" information about clones (for example, the detailed correspondences between two clones), editing clones, and removing or refactoring clones (clone divergence).

In this section, these design elements will be discussed in general terms, with CnP as a first instance of the general design. To emphasize, important design decisions as well as design problems that require further exploration are highlighted in bold text. Our prototyping is based on Eclipse [4], showing that our designs are technically feasible.

2.1 Clone Model and Clone Capture

A proactive clone management environment must track clones before attempting to manage clones. A PCM should automatically detect the creation of new clones by intercepting copying and pasting actions in the IDE. After clones are created, PCM must accurately maintain the clone locations when code is added or deleted before or within the clone regions. PCM should also persist the clone model between IDE sessions. Furthermore, a PCM's clone model should cover the whole workspace, not just individual projects. The clone model should also be managed by version control systems so that they can be shared among team members. We assume that clones be mainly evolved within the IDE. Although there can be exceptions, we believe that this is a reasonable assumption that will cover the majority of clone evo-

[1] `www.clarkson.edu/~dhou/projects/CnP`. All URLs verified April 10, 2009

[2] `www.ccfinder.net`

[3] `www.blue-edge.bg/simscan`

[4] `www.eclipse.org`

lution scenarios.

Our experience with implementing CnP in the Eclipse IDE indicates that all of these issues can be solved with some engineering effort. However, three more interesting design decisions must be made for proactive environments, that is, coordinates for individual clones, the cloning relationship between related clones, and policies for identifying "meaningful" clones from low-level copy-and-paste actions. The first two are about clone representation, also known as a clone model.

The choice of representation for clone locations depends on the tasks intended for the clones. For example, many clone detection tools represent clones with line ranges [1]. Line ranges are probably good enough if the purpose of a tool is only to show clones to programmers. Some proactive support also proposes to use line ranges to represent clone locations, for example, Clonescape [2]. The problem with a line-based representation, however, is that it could give an imprecise clone boundary because a single line may contain multiple statements, and vice versa. This can be too coarse-grained for a proactive environment aimed at providing comprehensive support. For example, some CnP features need to detect edits and changes made to clones at the level of expression. Therefore, CnP needs a representation suitable for implementing this kind of support other than line ranges. CnP's clone coordinates consist of the character *offset* and *length* from the source code in a particular *file*. In general, we recommend the following design decision for PCM.

Design Decision 1: Clone coordinates should be at the granularity of a character.

When a pair of clones is created by copying-and-pasting, CnP keeps track of the directional "copy-and-paste" relationship between the pair. This relationship indicates the origin of a clone, which can be a useful piece of information for a programmer. On the other hand, the "similar" relation that clone detection tools commonly use to define a clone group, is a symmetric notion. Much clone-related research adopts this notion, such as "a clone group" in [5, 9, 21, 10], "a clone class", or "a region set' in' [18]. In all of these cases, a group of related clones are viewed at the same level of group membership symmetrically, and thus do not have the information about clone origin. Thus there is a conflict between the two clone models. Since a PCM should integrate results from conventional clone detection techniques (Section 2.1.1), it needs to reconcile the conflict in clone models. Hence, we recommend the following design decision.

Design Decision 2: A PCM's clone model should support the symmetric cloning relationship between a pair of clones. The clone origin can be included in the clone model as additional but optional information to a clone pair.

Note that the "cloning" relationship is not transitive by default.

A programmer's low-level copy-and-paste actions do not always result into a high-level clone that is meaningful or relevant to a programmer's tasks. For example, some copied code may be considered too trivial to track. 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) [12]. The programmer is simply copying and pasting to save typing or to remember a name's spelling. If the proactive environment 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 as much as possible, the environment should choose 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. Thus, the following general design problem can be stated for PCM.

Design Problem 1: Automatically and reliably identifying "meaningful" clones from a programmer's low-level copy-and-paste actions.

The policy that CnP currently uses for determining whether a copied code fragment is a clone suitable for tracking is that it must contain at least one of the following: (1) more than two statements, (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 remove a clone from a clone group.

Sometimes, a high-level clone that is more meaningful to a programmer is created piecemeal by multiple copy-and-paste actions and edits. Therefore, there can be a need to merge multiple smaller clones. However, it is not entirely clear how this merging process can be best supported by

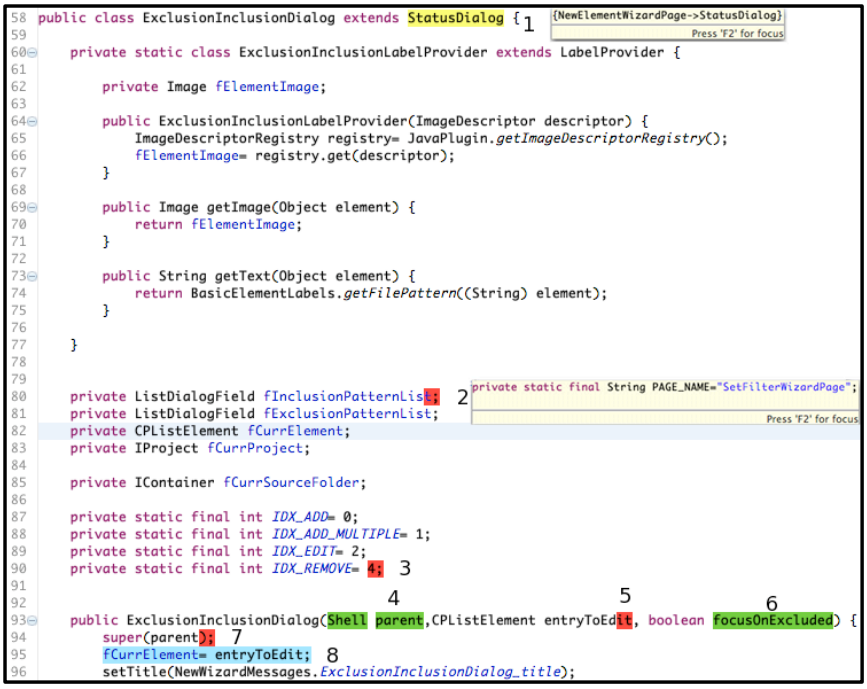


Figure 2: CSeR (Code SEgment Reuse) Incrementally and Continuously Tracks Edits Made to a Copied-and-Pasted Clones (Class in This Case) and Visualizes Code That Is *Added, Updated, Deleted, and Moved* with Different Colors inside the Editor (Figure Better Viewed Online to See the Colors).

tools. Hence,

Design Problem 2: Allowing for removing and merging clones from the clone model effectively.

2.1.1 Importing Detected Clones

In addition to tracking clones created by copy-and-paste, a PCM should also be able to integrate clones from clone detection tools. This can be useful when proactive support is applied to existing code base. CnP now supports the importing of clone data from the SimScan clone detection tool. The process of clone importing can be further improved by adding support for automatic classification and filtering as available in systems like CLICS, which will be helpful for navigating and exploring clone detection tool results [10].

2.2 Clone Visualization

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

At the system level, CnP provides a view of all tracked clones. Clones that are being tracked by CnP can include both those that were copied and pasted in the IDE (native clones) and clones imported from detection tools.

CnP currently shows individual clones visually by displaying colored bars next to the clone code in the editor, shown in Figure 4a. However, when a file contains many clones, the colored bars may clutter up the editor. Thus, it may be necessary to consider other design options and to use pastel colors to be less distracting. More generally, effective interaction support is needed for inspecting clone groups that contain more than a pair of clones. For instance, given three clones, A, B, and C, where A is copied to make B, and B is copied to make C, one may want to navigate directly from A to C.

Design Problem 3: Effectively displaying the clone code and navigating around a clone group within the code editor.

Once created, a clone may be subsequently evolved, where code can be *added, updated, deleted, or moved*. Visually highlighting these four kinds of change directly in the editor can be a very useful aid to programmers in understanding clones. Another feature of CnP named CSeR

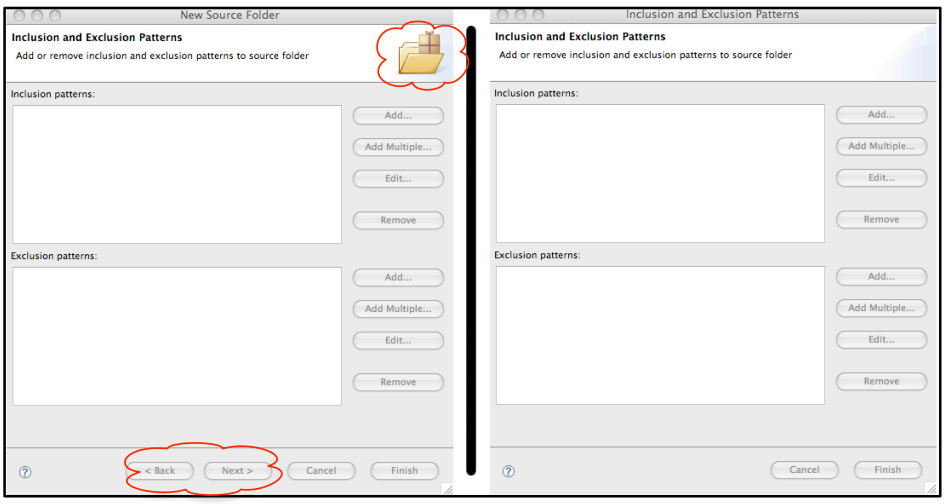


Figure 3: Classes SetFilterWizard and ExclusionInclusionDialog Produce Almost Identical Dialogs.

(Code Segment Reuse) supports precisely this (an instance of Diff View in Figure 1). When initially copied and pasted, the new code fragment is identical to the original. CSeR actively follows a programmer's editing actions, inferring and visualizing the four kinds of change in real time in the editor where the programmer is typing. Specifically, CSeR builds up the correspondence between "interesting" code elements in the original code and the clone. Changes made to tracked code elements are labeled as "update". Edits that happen between tracked code elements are labeled as "new" or "added". Edits that remove a tracked code element cause CSeR to show a deletion in the editor. Finally, identical "added" and "deleted" elements are identified as "moved".

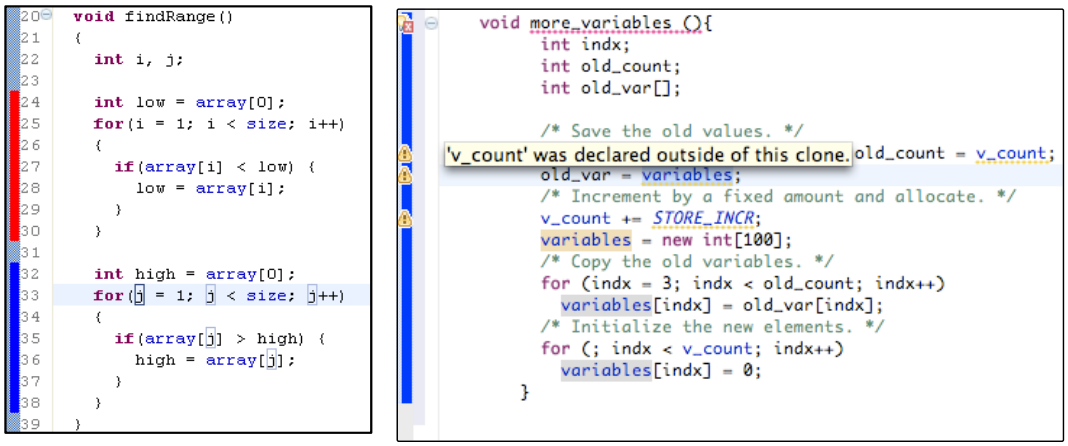
Figure 2 depicts how CSeR visualizes changes with different colors when a copied-and-pasted class is being modified in the Eclipse Java editor. The class shown is ExclusionInclusionDialog, which was copied from SetFilterWizard. Both classes are from the jdt.ui plugin in Eclipse 3.2. As shown in Figure 3, the two classes produce the dialogs which an Eclipse user can use to control the content (folders) of a project. The two dialogs are very similar despite they belong to two different type hierarchies. Their code is also very similar. For example, both classes contain a method named addMultipleEntries(), a method named getPattern(), a method named createListContents(), an inner class named ExclusionInclusionLabelProvider, and similar constructors.

Figure 2 shows only a small portion of ExclusionInclusionDialog, where updated elements are shown in yellow, deleted elements red, new elements green, and moved elements blue. Specifically, location 1 shows that the superclass was changed from NewElementWizardPage to StatusDialog. The red annotation at location 2 indicates that a static field PAGE_NAME has been deleted. Locations 4 and 6 show that the two parameters parent and onFocusExcluded are added to the constructor. Finally, the assignment at location 8 has been moved forward to its current location and thus marked blue.

Currently, CSeR is not applied to imported clones as the clones may have been evolved very differently and establishing correspondences for modified clones is not a trivial task. One possibility is to use code comparison techniques, like those of [3, 4, 6], to automatically compute an approximation of the fine-grained correspondences between detected clones. With some further manual intervention, it would be possible to convert these clones into a form that can be managed by CnP. A design problem can thus be stated as follows.

Design Problem 4: Effectively converting *detected clones* into a format that can be managed by proactive clone management environments.

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 exam-



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

ple is warning about the accidental capture of external identifiers (an instance of the interaction between a clone and its context shown in Figure 1). 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 4b was copied and pasted within the same class, for which CnP provides three warnings, one for each of the three fields, "v_count", "variables", and "STORE_INCR". These warnings alert the programmer that these particular identifiers within the clone (method) may need to be renamed. The programmer can then use CReN to rename the identifier instances in the pasted region, if so 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 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). The CSeR feature described previously is an editing feature. Clone editing also includes the maintenance of the clone model such as removing a clone group, and clone refactoring, which will be considered in future work.

2.3.1 Intra-Clone Editing

CnP contains a renaming utility (CReN) [8] that helps rename identifiers consistently *within* clones or any programmer-selected region, shown in Figure 4a. 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 normally be alerted of the missed instance.

As a simple example, suppose that the programmer needs to make a method to find the range of an integer array. Code has already been written to find the lowest integer in the array of integers,

shown as the first for loop in Figure 4a, so the programmer would need to write code to find the highest integer. With the existing loop and the variables "low" and "i" already present, the programmer could add in the new variable "j" and then copy and paste the declaration of "low" and the for loop together, creating an exact duplicate. CnP highlights the copied code (the origin) with a red bar and the pasted code with a blue bar. After changing "<" to ">", the programmer would then like to rename all instances of "low" to "high" and "i" to "j" in the pasted code. Manual renaming can miss an instance that is undetected by the compiler. In Figure 4a, with CReN, all instances of "i" in the pasted loop are renamed to "j" when any "i" in the loop is renamed.

Programmers may implement the example in Figure 4a differently, for example, by using a single for loop instead of two loops. The choice of implementation depends on a programmer's personal coding style. However, some may prefer the implementation outlined here, since it demonstrates a clear separation of the two concerns of finding "low" and "high". We present the solution in this way as a simple illustration that does not require much explanation. Other examples where CReN is applied to production code can be found in [8].

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 when existing refactoring support falls short. One example that demonstrates the difference is when the programmer would like to switch the "i"s and "j"s between nested for loops. The top of Figure 5 shows that it is impossible to do this with Eclipse's Rename refactoring. On the other hand, performing the same sequence of steps with CReN on the entire user-specified code fragment (including the declarations) gives the desired result, shown on the bottom of Figure 5.

As mentioned earlier, CReN initially infers that the programmer intends to rename all instances of an identifier within a code fragment. We have not found any cases where this assumption was incorrect, although it is theoretically possible. B.S. Baker confirmed that in her clone detection study, she did not find cases where such identifier instances were meant to be renamed differently. The most common example for inconsistent renaming that she found was between literals (like 0 and 1), not identifiers [5].

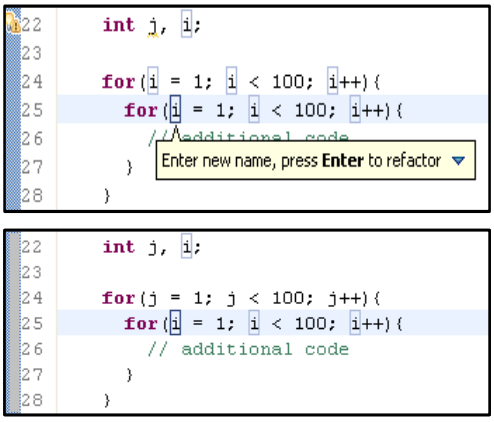


Figure 5: A Case Where Rename Refactoring Failed (top) but CReN Succeeded (bottom).

2.3.2 Inter-Clone Editing (Simultaneous Editing)

A number of editing techniques have been proposed to help consistently propagate common updates and changes across similar code fragments (linked editing [20], which was also called synchronous editing [11], and simultaneous editing [18]). But they all require that the programmer manually specify the clone code first.

Although not currently available, support for simultaneous editing between clones will be added to CnP as well. Since clones are already available in CnP and programmers do not have to manually select them any more, we anticipate that CnP's simultaneous editing feature is likely to offer a better user experience. Another important difference is that CnP relies on parsing and maintains clone correspondences continuously (as currently done with CSeR). In contrast, simultaneous editing in prior work relies on Longest Common Subsequence (LCS) algorithm as in [20], or Levenshtein Distance (LD) as in [2, 5, 21], which may produce results that are not as intuitive as that of CSeR.

2.3.3 User Interaction and Intervention

There are several scenarios in PCM where user interaction design becomes critical. Since it is impossible to have an algorithm that decides whether

[5] Personal communication. Dec. 2007

two code fragments behave in a way of interest to a programmer, PCM must allow the programmer to remove a clone from a clone group or stop tracking a whole group of clones. In this way, the programmer can have some control over the issue of which clones can be considered related. As another example, in the interest of efficiency, CReN is automatically applied when any identifier is modified. CReN also allows a programmer 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 also apply to the design of other CnP features like simultaneous editing. In general, a design goal is that a PCM should provide sufficient interaction so that programmers can still be in control, yet minimize the frequency where programmers are forced to correct mistakes created by tools. Such design decisions should be tested with user studies to understand their effectiveness.

3 Design and Requirements Exploration via Clone Case Studies

We conducted several exploratory case studies of clones in real-world code base using two public clone detection tools (CCFinder and SimScan). It is important to note that these case studies were not meant to serve as a formal validation of CnP features, which we will address separately. Rather, they represent a way of exploring the requirements for and the design space of proactive clone management. Thus, our case studies can be viewed as a formative, but not summative, evaluation.

In order to build the right tools and to quickly assess their potential, it is necessary to gain at least a qualitative understanding of copy-and-paste-induced clones in real-world software systems. It is also necessary for tool builders to reflect on the process of understanding clones, e.g., the process of comparing and contrasting clones. Although some anecdotes about clones can be gleaned from empirical studies on clone detection techniques [16], these studies tend to to focus more on presenting statistical data rather than providing concrete clone examples. Our case studies were successful in helping us not only obtain concrete test cases for the development of CnP but also identify possible new features and refinements to existing features.

3.1 Case Studies Process

We have used clone detection tools to explore several open-sourced Java systems, e.g., Apache Ant, JEdit, JavaLobby, and Eclipse plugins. However, we spent the most efforts on two systems, SCL (Structural Constraint Language) [6] and jdt.ui (Eclipse 3.2). Because familiarity with the studied subject is critical to determining both the accuracy and the significance of the clone detection tool's results and whether the clones are worth tracking or can benefit from existing and additional tool support from a PCM like CnP, in our case studies, we chose systems that we are familiar with. SCL was developed by the first author. jdt.ui contains Eclipse's code for the Java tool UI as well as refactoring support. Although we did not develop the jdt.ui code, we did study portions of it previously. So we are also reasonably familiar with its code.

The version of SCL under analysis contains 22,581 lines of Java code, 10,228 lines of comments, 29 packages, and 386 classes. The jdt.ui contains 1,680 Java files. The two clone detection tools used in the case studies, CCFinder and SimScan, represent two complementary clone detection techniques (token- versus syntax-based) [1]. Only SimScan was applied to SCL. Both CCFinder and SimScan were applied to jdt.ui. CCFinder was configured to report clones that contain more than 50 tokens. SimScan was run with the settings of "medium" volume, "fairly similar" similarity for SCL and "very similar" for jdt.ui, and "fast" speed/quality. CCFinder was able to analyze the whole jdt.ui code within several minutes. Because of the large size of jdt.ui, SimScan could not analyze it as a whole, and we were forced to analyze it one package at a time. The SimScan study was conducted on a MacBook Pro with 2 GB of main memory. The CCFinder study was run on a Windows machine.

CCFinder reported 858 clone sets for the jdt.ui plugin in Eclipse 3.2 (1913 classes and interfaces in about 100 packages). The reported clones contain from 50 to 1,722 tokens. We inspected all the clones that contain between 50 to 100 tokens as well as clones containing more than 1,000 tokens. For the rest, we randomly inspect approximately 100 of them. Not all large clones are interesting, for example, those associated with AST visitor classes.

When inspecting clones, we had paid particular

[6] www.clarkson.edu/~dhou/projects/SCL

attention to cues in the source code that indicated the clones were actually copied and pasted. Such cues are often obvious for class-level clones. For example, a class may share identical but special comments (e.g., "fix for PR #5533") with another such that it is very unlikely that their similarity has been created accidentally. Code block clones are less obvious, but those within close proximity (within the same method or even the same class) are very likely to have been created by copying and pasting as well. This is true at least for SCL.

3.2 Is Proactive Clone Management Justified? Assessment of Reported Clones

To assess the need for proactive clone management in general and how it can best be designed, we conducted two case studies. For the first goal, we chose to study a small system that we are familiar enough in order to provide a reliable assessment of the amount of clones it contains. This would help us understand to what extent a clone management system is needed. SCL was used for this purpose. For the second goal, to understand how proactive management can be best designed and to assess the potential of the developed CnP features, we opted to study a large, industrial system developed by others, in this case, Eclipse JDT.

In the first case study with SCL, SimScan reported 102 clone groups in 6 minutes and 25 seconds. Because of its small size, we were able to examine all the reported clones. The reported clones from the SCL study was analyzed by the first author in two sessions, each lasting approximately 4 hours. The largest clone is 90 lines, the shortest 6 lines, with a mean of 24 lines. Of the 102 clone groups, 70 were considered intentional, useful clones, and 32 determined to be irrelevant (9 were clones in generated parser code, 4 were groups of size 1, and 19 were judged not to be clones despite their syntactic similarity). Among the 70 useful groups, 29 were the copy-and-paste of whole classes, 27 were similar methods (16 for methods in different classes and 11 for methods in the same classes), and 14 were code blocks inside methods (4 for code blocks in different methods or classes and 10 in the same methods).

For SCL, we estimated that at least a half of the method-level and block-level clones were created by copy-and-paste, and thus about 50 out of the 70 groups could have been supported by a proactive clone management tool like CnP.

In the second case study, the focus was put on discovering class-level clones since they are likely to be more semantically significant in software development. The results from the jdt.ui study was analyzed in four sessions, taking a total of about 15 hours. Due to the large number of clone candidates reported for jdt.ui, we were only able to inspect about 200 clone sets. Preliminary processing was conducted on the output of the two clone detection tools to identify candidates of potential class clones, which were subsequently examined more carefully to make a conclusion. Table 1 depicts five class hierarchies and a pair of independent classes (the two in the last row) inspected.

This assessment shows that in general, proactive clone management is useful and thus needed. Furthermore, it shows that the output of clone detection tools like SimScan and CCFinder can contain clone information useful to a programmer, verifying that proactive clone management needs the ability to import detected clone as well.

The case studies also revealed limitations of clone detection tools, in this case, CCFinder and SimScan, and the challenge in the clone understanding process, which will be elaborated on in the following sections.

3.3 State-of-the-art Clone Detection Tools Are Inadequate for Clone Management

Clone detection tools can be valuable in making the programmer aware of the existence of clones and assisting in future clone understanding and maintenance. However, after copied and pasted, clones are often modified. As a result, they are not identical but only similar to the original copy, resulting in so-called "gapped" clones. Existing clone detection tools are limited when dealing with gapped clones [1]. For example, for two classes that are modified but still obviously clones in human eyes, detection tools would report multiple smaller segments from the two classes as clones and fail to report the classes as clones as a whole.

Figure 6 illustrates this limitation of CCFinder with two classes from Eclipse JDT, SetFilterWizardPage and ExlusionInclusionDialog, as an example. The two classes produce two almost identical dialogs which an Eclipse user can use to control the content (folders) of a project, as shown in Figure 3. As explained before in Section 2.2, their codes are

Superclass/Classes	Functionality	Clones?
JavaDocWizardPage	3 subclasses for wizard pages for exporting JavaDocs.	Not clones.
NewTypeWizardPage	4 subclasses for wizard page for creating classes, interfaces, enum, and annotations.	All four are clones.
TextInputWizardPage	4 subclasses for input dialogs for four refactorings (extract constant, extract interface, move inner to top, extract temporary).	Extract Constant and Extract Temporary are clones.
CompilationUnitContext	2 subclasses for Java context and JavaDoc context.	The two classes are clones to each other.
ReorgPolicy	3 level class hierarchies controlling whether folders, files, packages, and program elements can be reorganized (copy/move).	Policies for the same resources (e.g., move/copy folders) are clones.
SetFilterWizardPage and ExclusionInclusionDialog	2 visually identical dialogs controlling the visibility of folders in a project.	The two classes are clones to each other.

Table 1: Some Class Hierarchies Inspected to Identify Copied-and-Pasted Classes (jdt.ui in Eclipse 3.2).

also very similar.

Both CCFinder and SimScan failed to detect the two classes as clones. Instead, they were only be able to detect smaller fragments of the classes as clones. As shown in Figure 6, CCFinder completely missed the two methods that are marked as A and A', presumably due to the small modification in the middle of A'. For the same reason (the difference between C and C'), D (identical to D' but not shown) and D' are also missed by CCFinder. B and B' are also missed but we are not sure of the cause. Table 2 shows a set of cloned classes that CCFinder failed to identify as clones for the same reason.

Because proactive support tracks copy-and-paste actions, it will detect a class or a method that is copied and pasted as clones and keeps track of that relationship when the clone is further modified. As a result, proactive tools can make it easier and cheaper to present a more logical view of clones than clone detection tools.

3.4 Supporting Clone Editing and Understanding

After clones are created, they are often modified in order to fit into a new context. Many times, these modifications are small and buried in large amount of identical code and thus become hard to spot. In the second case study with Eclipse JDT, we paid particular attention to what is involved in comparing and understanding clones.

In order to gauge the amount of effort involved in comparing clones in detail, we tried to identify the exact correspondences between pairs of class clones and counted the number of changes made to convert one class to a clone class. As an example, a total of 33 individual changes need to be made to class SetFilterWizwardPage in order to convert it to ExclusionInclusionDialog (last column of Table 2). When we initially came across these two classes, we had to spend quite some effort to recognize these individual changes before concluding that the two classes were copied-and-pasted. With a tool like CSeR, much of these tedious comparisons would have been avoided and the differences between the two classes will be always visible in the code editor, as shown in Figure 2. CSeR has been applied to all the clone groups in Table 2.

While CSeR is useful, it can do more to help programmers. For example, we have also seen that many clones were evolved asynchronously. This becomes apparent in Figure 6, where for example, an inner class named ExclusionInclusionLabelProvider was moved to a different location in the right-hand side (not shown), and the method marked A was moved to a new location A'. Statements can also be moved out of order. These ed-

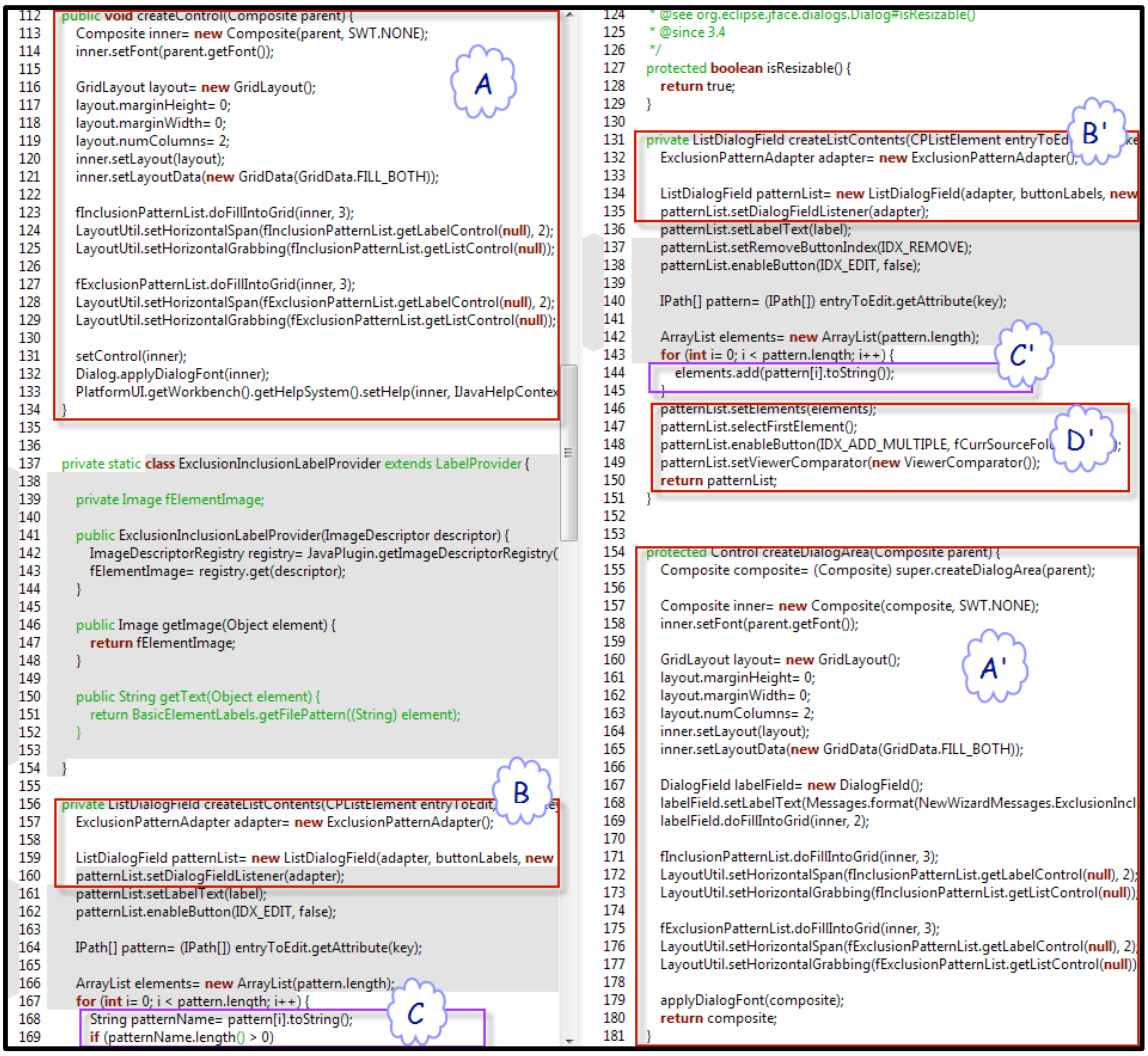


Figure 6: CCFinder Failed to Detect Cloned Methods (A-A') and Code Fragments (B-B' and D-D') between Two Classes SetFilterWizardPage (Left) and ExclusionInclusionDialog (Right).

its, if consistently applied across the related clones, would have made the clones more similar in human eyes. Our experience indicates that such asynchronous edits among clones are fairly common. Simultaneous editing can be used to improve the situation and keep the code structure in sync.

3.5 Use Cases for CReN

In our case studies, we have discovered a few cases where CReN can be used to consistently rename identifiers.

Figure 7 depicts two examples where CReN could have been used to rename identifiers for the clones. The code is part of the "move static method" refactoring in Eclipse JDT. In the example on top, isSourceAccess() and isTargetAccess(), are method clones. Note that the only difference between the two methods is the data members, fSource and fTarget that they respectively access.

The example at the bottom shows three for-loops that clone each other. These loops access three data members, fields, methods, and types, respectively. CReN could have been used to rename the data members after the clone was created.

We have discovered at least five groups of clones where CReN can be applied. Clearly, this is only a lower bound estimation of CReN's utility.

Class Name	LOC	Missed LOC	#Clones	#Changes
SetFilterWizardPage	340			
ExclusionInclusionDialog	327	54	5	33
NewClassWizardPage	292			
NewAnnotationWizardPage	142	24	2	26
NewEnumWizardPage	146	5	2	23
NewInterfaceWizardPage	133	5	2	22
NewClassCreationWizard	96			
NewAnnotationCreationWizard	96	0	2	6
NewEnumCreationWizard	96	6	2	6
NewInterfaceCreationWizard	96	6	2	5
CleanUpPreferencePage	60			
CodeFormatterPreferencePage	62	0	1	5
CodeStylePreferencePage	129			
CodeAssistPreferencePage	98	6	1	11

Table 2: Analysis Of Some Class Clones Reported by CCFinderX (jdt.ui in Eclipse 3.2). (The First Class in Each Group Is the Original. Third Column: Missed Lines of Code That Should Have Been Identified as Clone Code. Fourth Column: Number of Identified Clones between a Pair of Classes.)

3.6 Further Design Input for PCM

The case studies also provide input that can be used to inspire the design of new features and refine the existing design in proactive clone management.

First, when comparing related clones, it was particularly annoying that we had to frequently switch back and forth between multiple Java files or different locations when the clones are in the same file, each time just to conclude a small difference. We conclude that a feature that supports the side-by-side comparison of multiple clones can be a very useful aid in this scenario. We plan to add this feature as a logical extension to CSeR.

Second, we have found one instance where a larger clone contains smaller ones. Specifically, in Eclipse JDT, a method that contains 2 cloned fragments is further copied and pasted to make another method. So we have 2 bigger clones that contain 4 smaller clones. Although not frequently encountered, this instance shows that a PCM should probably support "clone overlapping".

Third, the data from SCL shows that additional tool support can be offered. Note that 29 groups are the copy-and-paste of whole classes. In SCL, these classes model similar but distinct language constructs, like the four arithmetic operations. They share a lot of similarity in code structure and vary in fixed locations in a predictable way. The number of cloned classes in each group also tends to be large, with the largest group containing 51 classes. A tool can be built to automatically infer a change script that records the common changes that have been done to previous clones of the same class. Such a script can then be used to guide the programmer in modifying a newly cloned class.

Fourth, some clones are created via "symmetrical code patterns", which may happen at class, method, and block levels. For example, the JavaEditor class contains a pair of inner classes, PreviousSubWordAction versus NextSubWordAction, and a pair of methods, getPreviousPosition() verus getNextPosition(). In the MovedMemberAnalyzer class, there are two methods named isSourceAccess() and isTargetAccess(), which differ by the fields they access (fSource and fTarget). Another class, RefactoringAvailabilityTester, contains isPullupAvailable() verus isPushdownAvailable().

Finally, some clones differ only in a pair of types (TextChangeManager and TextEditBasedChangeManager) or a pair of expressions (JavaRefactoringContribution's 12 subclasses trivially differ by a literal string that represent the name of a refactoring.) These clones could have been avoided using Java generics, but the original developers may have good reasons not to do so, and then it is valuable to make the clone group visible.

4 Related Work

Mainstream research on clones is based on clone detection. Surveys of clone-related research and

```
class MovedMemberAnalyzer {
    // fSource and fTarget are data members...
       private boolean isSourceAccess(IBinding binding) {
         if (binding instanceof IMethodBinding) {
                  IMethodBinding method= (IMethodBinding)binding;
                   return Modifier.isStatic(method.getModifiers()) &&
                              Bindings.equals(fSource, method.getDeclaringClass());
           } else if (binding instanceof ITypeBinding) {
                   ITypeBinding type= (ITypeBinding)binding;
                   return Modifier.isStatic(type.getModifiers()) &&
                              Bindings.equals(fSource, type.getDeclaringClass());
           } else if (binding instanceof IVariableBinding) {
                   IVariableBinding field= (IVariableBinding)binding;
                   return field.isField() &&
                              Modifier.isStatic(field.getModifiers()) &&
                              Bindings.equals(fSource, field.getDeclaringClass());
           }
           return false;
        }

        private boolean isTargetAccess(IBinding binding) {
          if (binding instanceof IMethodBinding) {
                   IMethodBinding method= (IMethodBinding)binding;
                   return Modifier.isStatic(method.getModifiers()) &&
                             Bindings.equals(fTarget, method.getDeclaringClass());
          } else if (binding instanceof ITypeBinding) {
                   ITypeBinding type= (ITypeBinding)binding;
                   return Modifier.isStatic(type.getModifiers()) &&
                             Bindings.equals(fTarget, type.getDeclaringClass());
          } else if (binding instanceof IVariableBinding) {
                   IVariableBinding field= (IVariableBinding)binding;
                   return field.isField() &&
                              Modifier.isStatic(field.getModifiers()) &&
                              Bindings.equals(fTarget, field.getDeclaringClass());
         }
          return false;
        }
...
}
                              ------------ Group 1 --------------

class MemberVisibilityAdjuster {
// fields, methods, and types are data members ...
    private void adjustIncomingVisibility(final IType[] types, final IMethod[] methods,
              final IField[] fields,  final IProgressMonitor monitor)
              throws JavaModelException {

                  ... code elided ...

                   IField field= null;
                   for (int index= 0; index < fields.length; index++) {
                           field= fields[index];
                           if (!field.isBinary() && !field.isReadOnly())
                              adjustIncomingVisibility(field, new SubProgressMonitor(monitor, 1));
                   }

                  IMethod method= null;
                  for (int index= 0; index < methods.length; index++) {
                          method= methods[index];
                          if (!method.isBinary() && !method.isReadOnly() && !method.isMainMethod())
                              adjustIncomingVisibility(method, new SubProgressMonitor(monitor, 1));
                  }

                  IType type= null;
                  for (int index= 0; index < types.length; index++) {
                          type= types[index];
                          if (!type.isBinary() && !type.isReadOnly())
                              adjustIncomingVisibility(type, new SubProgressMonitor(monitor, 1));
                 }
                   ...
    }
    ...
}
                              ------------ Group 2 --------------
```

Figure 7: Two Groups of Code Clones in Eclipse JDT that Could Have Been Renamed Using CReN.

clone detection techniques can be found in [16] and [19]. 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. Proactive support can potentially ease this problem by distributing the effort over time.

Clonescape [2] and CPC [21] 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 [20] 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 to the related clones.

An overview of CnP features appears in [7]. However, this paper differs by exploring the design of proactive clone management environments in general rather than presenting the details of CnP features. This paper also presents case studies that motivate the need for as well as provide additional design input for PCM.

5 Conclusions and Future Work

Automated tool support for managing copy-and-paste-induced code clones can potentially better support clone evolution. However, they are not currently available in the mainstream editor or integrated development environment (IDE). In this paper, we explored the main design elements for proactive clone management in the form of recommended design decisions and remaining design problems, which other researchers may choose to work on. We described our initial experience with prototyping several features that partially implemented the outlined design, including a consistent renaming utility CReN, a clone differencing utility CSeR, and several other small features. As a preliminary validation and a means for requirements and design exploration, several clone case studies were used to motivate the need for proactive clone management and to identify and refine design requirements for individual features.

As ongoing work, we are currently analyzing the result of a lab-based user study to assess the effectiveness of a subset of these features in helping programmers work with clones, in particular, during debugging and modification tasks. Despite our attempt in outlining a complete PCM design space, we have not really addressed issues related to clone divergence and refactoring. Finally, the question of whether the presented design space is adequate can only be answered by actually using CnP in practice.

Acknowledgments

The authors thank Toshihiro Kamiya for providing a Python script for postprocessing CCFinder-reported clones, and the three anonymous CASCON 2009 reviewers for their constructive feedback.

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] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an Environment for the Proactive Management of Copy-and-Paste Programming. In *ICPC'09*, 2009.

[8] 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.

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

[10] C. Kapser and M. Godfrey. Improved Tool Support for the Investigation of Duplication in Software. In *Proceedings of ICSM'05*, 2005.

[11] C. J. Kapser and M. W. Godfrey. 'Cloning Considered Harmful' Considered Harmful: Patterns of Cloning in Software. *Empirical Softw. Engg.*, 13(6):645–692, 2008.

[12] 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.

[13] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An Empirical Study of Code Clone Genealogies. In *Proceedings of FSE'05*, 2005.

[14] 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.

[15] K. Kontogiannis. Managing Known Clones: Issues and Open Questions. In *Duplication, Redundancy, and Similarity in Software*, 2006.

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

[17] 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.

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

[19] 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.

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

[21] 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.