Real-time Code Clone Refactoring Recommendations

Travis Mandel, Todd W. Schiller
University of Washington
{tmandel,tws}@cs.washington.edu

Abstract

Code clone detection and analysis has historically been viewed as a maintenance problem. Recently, tools for managing clones during development have been introduced, however, these tools require users to maintain formal clone models. In this paper we propose a tool for (1) eliminating the introduction of code clones during development without maintaining formal clone models, and (2) leveraging code similarity to boost programmer productivity. The tool is an Eclipse plugin providing real-time clone detection and action suggestions to the developer as (s)he writes and modifies code.

To evaluate the tool, we performed a user study with two participants each performing the same development and maintenance task in Eclipse with the tool enabled. While the tool failed to identify code reuse opportunities during the development task, the tool aided one participant in locating example API uses. During the maintenance task, the tool quickly guided both users to the locations where the bug was replicated. Our results confirm that clone detection is beneficial during maintenance, but suggest that more sophisticated detection is required for development use.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments

Keywords refactoring, recommender system, code clones

1. Introduction

Numerous studies suggest that code clones impair the maintainability of software.

Yamashina et al. found in a sliding window analysis of a commercial CAD application that 79.3% of commits included modifications to files containing code clones, but that only 9.7% of such commits included modifications to files containing the other clones suggesting some clones may have erroneously not been updated (the minimum clone length considered was 50 characters). [16]. In a study of a commercial product line, Li and Ernst report that 4% of bugs were duplicated across at least one product or file; additionally, they identified 282, 44, and 33 duplicated bugs in the Linux kernel, Git, and PostgreSQL respectively [13].

Under the assumption that code clones are not maintained properly, Jeurgens et al. built a static bug detection tool based on inconsistencies in clones, and confirmed that clones were a major source of bugs in the study's subject programs [9]. Similarly, we hypothesize that when a developer (un)intentionally nearly duplicates the functionality of an existing piece of code, without referencing the original source, that the new code is more likely to contain bugs than the original as it has not been tested or used in production. Unifying code written by multiple developers has other benefits, such as improved code consistency, readability, and modularity.

Code clones have historically been viewed as a problem of software *maintenance*, as failure to revise a clone can be an error. Or, the task of identifying code clones is considered a separate and independent development task, and thus may not be performed in a manner consistent with eliminating bugs.

In addition to helping developers *maintain* clones, this work aims to help developers *develop* more effectively by facilitating actions in the presence of system clones, existing code that is a (partial) clone of the source under development. Our hypothesis is that identifying clones during development will prevent many of the problems associated with duplicated code from ever arising, reducing development time.

Actions for Duplicate Code The tool suggests two actions that eliminate code duplication:

1. INSERTCALL: Replace the code under development with a call to an existing method
2. EXTRACT: Extract all, or part, of the system clone as a method; replace the code under development with a call to the extracted method.

Additionally, two actions may be suggested to help the developer develop or maintain code with duplication:

3. JUMPTO: Open the relevant section of code to aide the developer in making analogous changes to the system clone;
4. PASTE: Copy and paste the system clone to the code under development, substituting identifiers as needed.

Unlike other recent work for managing code clones during development [4, 5], the tool does not require the devel-

oper to manage a formal model of the clone linkages; as the tool does not depend on explicitly tracked linkages, clones can be identified as they are being developed to inform developer actions, even if the developer does not perform a copy-paste action or explicitly perform a clone search query. We hypothesize many clones are written because the developer is not aware of preexisting functionality, so focusing only on copy-and-pasted clones misses cases where functionality has been inadvertently duplicated.

This paper proceeds as follows: Section 2 describes the user interface for the tool, along with the underlying clone detectors. Section 3 describes a controlled user study to evaluate the tool, with Section 4 reporting and discussing the results of the study. Section 5 discusses related work in clone detection, analysis, and refactoring. Finally, Section 6 concludes.

2. Finding Clones

To support the developer actions enumerated in Section 1, the tool searches the existing codebase for code that is similar to the region that is currently being developed or maintained. The search is performed using the clone detectors described in Section 2.1. The clone detection is implemented as an Eclipse reconciler, running in the background whenever there is a natural pause in typing.

In order to be practical in an online setting with a large codebase, the detectors must be not only be fast, but also robust to identifying clones that are more obfuscated than direct copying, such as when a programmer re-implements the same functionality.

2.1 Clone Detectors

The tool is designed to perform detection both during development and during maintenance. As such, it may not be possible to parse the source file, build an Abstract Syntax Tree (AST), or type the AST. Given this, text-based detectors are advantageous because they can be run during active development. When a program is parsable or compilable, more-sophisticated detectors that use ASTs or program dependence graphs [13] produce better results because they can use structural information when determining similarity.

The tool is currently packaged with three code clone detectors:

1. The Java Code Clone Detection (JCCD) API [2]: performs AST-based similarity detection with support for a pipeline of AST operators; requires that the source files are parsable.
2. Checkstyle [1]: performs a textual comparison on the lines of a program
3. Simian [3]: the Simian software is proprietary (though free for non-commercial use), but it appears the Simian can perform both textual and AST-based detection.

All three detectors perform detection over the entire codebase, as opposed to searching for clones for a given query.

These code clone detectors were selected because they all support Java 1.5 features (e.g., generics), are Java-based, and are freely available for at least academic use.

Currently, only a single detector can be active at a time. In the future, it may be beneficial to run the analyses simultaneously and combine results.

2.2 Displaying Suggestions

Clones *with suggested actions* are displayed as Eclipse annotations, which consist of (1) source code highlighting, (2) a marker on the left vertical bar, and (3) a colored region on the right vertical bar. When users click the left-hand marker, the corresponding clone(s) and potential refactoring options are shown, along with other Eclipse Quick Fix resolutions for errors and warnings. Multiple clones at the same location are indicated by multicolored markers in the Quick Fix dialog. Figure 1 shows the Eclipse Quick Fix interface.

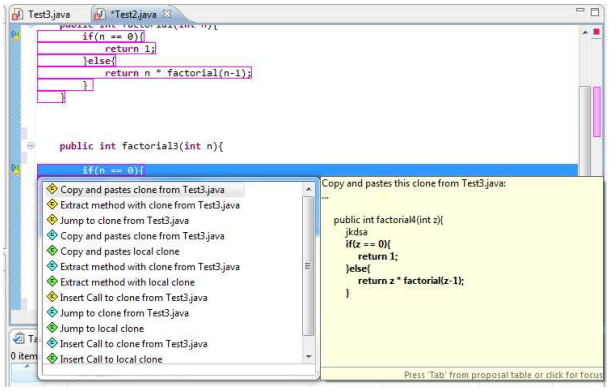

Figure 1. Eclipse Quick Fix clone suggestions. The color of the icon next to a fix differs by clone pair. The right side window shows the other side of the clone, higlighted with additional lines of context, the code to be pasted, or the region to be extracted, or the body of the method call that will be inserted.

Modes We believe that, in cases such as development, being presented with clone annotations for the entire file is prohibitively distracting. Therefore, the tool has two modes: development mode and maintenance mode. In development mode, only clones located in the active development area are highlighted (as determined by Eclipse's dirty region). In maintenance mode, all clones located in the file are displayed. The developer switches between the two modes using a toolbar button.

2.3 Determining When to Make Suggestions

Clone detectors score clone pairs based on code similarity; therefore, the results may not be suitable for certain types of downstream refactorings or other actions. Additionally, because we are presenting these clones to the user during

development, the suggestions must be conspicuous without being too obstrusive, lest a developer disable the tool. Our solution is to display conspicuous UI notifications (see section 2.2), while utilizing an adaptive scoring system to remove unhelpful UI elements based on user actions.

The tool determines a relevance for a clone pair and action according to the following formula, which takes into account the user's previous actions:

$$\begin{aligned}\textsc{Adj} = &\left[\textsc{Raw}_{\text{clone,action}} * \left(1 + \frac{\textsc{Pref}_{\text{action}}}{\sum \textsc{Pref}}\right)\right] \\ &* (1 - \textsc{MainDecay})^{\#\textsc{Display}_{\text{clone}}} \\ &* (1 - \textsc{DevDecay})^{\#\textsc{Dev}_{\text{clone}}}\end{aligned}$$

, where

- ADJ is the adjusted relevance of the suggestion, which the Eclipse Quick Fix mechanism uses to order the suggestions (in practice, ADJ is truncated such that $10 \leq \textsc{Adj} \leq 100$, for this reason). Additionally, if $\textsc{Adj} < \textsc{Threshold}$, a fixed threshold, our tool excludes the suggestion from the set presented by Eclipse.
- $\textsc{Raw}_{\text{clone,action}}$ is an action-specific score for the clone determined by the clone pair's similarity, and a heuristic estimate of the usefulness of the action; calculation details are provided later in this section. In order to maintain consistency across clone detectors, the current version of the tool uses the number of non-whitespace, non-Javadoc characters in the clone.
- $\textsc{Pref}_{\text{action}} > 0$ is the user's preference for the action; $\textsc{Pref}_{\text{action}} / \sum \textsc{Pref}$ is the user's relative preference for the action. The initial values are set with normative information (e.g., it is better to insert a method call than it is to paste a clone). The value is then adaptively adjusted according to the users actions, as described later in Section 2.3.1.
- $\#\textsc{Display}_{\text{clone}}$ is the number of times the clone has been displayed in a Quick Fix session.
- $\#\textsc{Dev}_{\text{clone}}$ is the number of times one or more development actions occurred between Quick Fix sessions that included the clone (or since the clone was last included in a Quick Fix session).
- $0 < \textsc{MainDecay} << \textsc{DevDecay} < 1$ are constant decay factors that reduce a suggestion's relevance when a developer does not act on the suggestion. The "development" decay factor DEVDECAY is much larger under the assumption that when a user performs development taks between clone views, they have either (1) switched to another task, or (2) have explicitly determined not to act on clone's suggestions.

2.3.1 Adapting to Developer Action Preferences

When an action is selected in a Quick Fix session, the preference for the action, $\textsc{Raw}_{action}$, is increased inversely with respect to its distance from the threshold:

$$\textsc{Pref}^{\text{new}}_{\text{action}} = \textsc{Pref}^{\text{old}}_{\text{action}} * \left[1 + \frac{100 - \textsc{Relevance}}{\textsc{Threshold}}\right]$$

, where 100 is the maximum allowable score. If the user selects an action with a score close to, or at, the threshold, it has a greater positive effect on the preference than when the user selects a "perfect" action. While a developer's preference for an action may be low, it cannot be negative. Therefore, preference alone cannot preclude a suggestion from being shown to the developer: $\textsc{Pref}_{action} > 0 \Rightarrow \textsc{Raw}_{\text{action,clone}} \left(1 + \frac{\textsc{Pref}_{\text{action}}}{\sum \textsc{Pref}}\right) > \textsc{Raw}_{\text{action,clone}}$. All developers see suggestions for which $\textsc{Raw}_{\text{action,clone}} > \textsc{Adj}$, however given the developer's action preferences (1) additional suggestions may be displayed, and (2) the ordering of the suggestions will differ.

2.3.2 Scoring INSERTCALL Actions

The score for an INSERTCALL action is determined from the (1) clone's similarity (2) the number of arguments in the resulting method call, and (3) the percent of the method being called that is covered by the clone:

$$\begin{aligned}\textsc{Raw}_{\textsc{InsertCall,clone}} = &\textsc{Similarity}_{\text{clone}} \\ &* \textsc{Coverage} \\ &* (1 - \textsc{ArgPenality})^{\#\textsc{Args}}\end{aligned}$$

, where $\textsc{Similarity}_{\text{clone}}$ is the similarity score for the clones, COVERAGE is the percent of the callee that is covered by the clone, $0 < \textsc{ArgPenality} < 1$ is a constant penalty for the number of arguments in the resulting call, and #PARAMS is the number of arguments in the resulting call.

2.3.3 Scoring EXTRACT Actions

The set of consecutive statements to extract for a clone pair is determined by finding the longest chain of statements (as measured by the *number* of basic statements) for which some part of the system clone region overlaps every basic statements. The score for the extraction action is then determined by (1) the clone's similarity, and (2) the number of variables used in the statements that aren't declared locally in the statements — i.e., fields, method parameters, local variables declared prior to the statements — excluding static fields:

$$\begin{aligned}\textsc{Raw}_{\textsc{Extract,clone}} = &\textsc{Similarity}_{\text{clone}} \\ &* (1 - \textsc{NonLocalPenality})^{\#\textsc{NonLocal}}\end{aligned}$$

, where $\text{SIMILARITY}_{\text{clone}}$ is the similarity score for the clones, $0 < \text{NONLOCALPENALITY} < 1$ is a constant penalty for non-local variable usage, and #NONLOCAL is the number of non-static variables and fields used by the statements that are not declared within the statements.

Some chains of statements cannot be extracted because multiple local variables defined in the chain are subsequently used in the block. In this case, no EXTRACT action is generated.

2.3.4 Scoring JUMPTO Actions

JUMPTO actions aide analogous edits to system clones during maintenance and bugfixing, therefore the scores are higher when the developer is maintaining code. The score for a JUMPTO action is determined by (1) the clone's similarity, and (2) the mode (see section 2.2):

$$\text{RAW}_{\text{JUMPTO,clone}} = \text{SIMILARITY}_{\text{clone}} * (\text{ISDEVELOPING} * (1 - \text{DEVPENALTY}))$$

, where $\text{SIMILARITY}_{\text{clone}}$ is the similarity score for the clones, ISDEVELOPING is an indicator variable that is 1 when the tool is in development mode, and $0 < \text{DEVPENALITY} < 1$ is a constant penalty to be applied when the tool is in development mode, as opposed to maintenance mode.

2.3.5 Scoring PASTE Actions

The PASTE action replaces the active clone with the the system clone extended to the end of block; external identifiers are substituted where possible. The current replacement implementation naively assumes that the order in which new external identifiers is introduced is consistent between the clones. The extended system clone is used under the assumption that it is likely that the user will need to replicate the subsequent behavior during development, and that the developer effort required to delete extraneous code is small relative to the effort required to write new code.

The score for a paste action is determined by (1) the clone's similarity, and (2) the quality of the external identifier matching as determined by the number of unmatched identifiers.

$$\text{RAW}_{\text{PASTE,clone}} = \text{SIMILARITY}_{\text{clone}} * (1 - \text{IDMISMATCHPENALTY})^{\#\text{UNMATCHED}}$$

, where $\text{SIMILARITY}_{\text{clone}}$ is the similarity score for the clones, #UNMATCHED is the number of unmatched external identifiers, and $0 < \text{IDMISMATCHPENALITY} < 1$ is a constant penalty to apply for external identifier mismatches.

2.3.6 Tracking clones

The score ADJ decays when the same clone is viewed multiple times (MAINDECAY), and when the developers performs a development action instead of selecting a suggested action (DEVDECAY). The current implementation uses Eclipse markers to track clones over time, which are invariant to changes outside the cloned region. However, this approach is suboptimal in two scenarios: (1) in development mode, the marker information is lost when the user switches in which area they are developing, and (2) any change inside a clone results in it being identified as a new clone. We leave developing a more robust model of clone equality for tracking user action history to future work.

3. Experimental Design

To evaluate the tool, we performed a user study designed to emulate the process of software development and maintenance. Two study participants each performed the same development and maintenance task on a subject program. The development task is given first so as to give the participants the opportunity to become acquainted with the codebase before performing the maintenance task.

Subject Program The subject project is a small Java image transformation libary and graphical user interface (13 files, consisting of 900 non-comment non-blank lines of source) developed for the study. To prevent the study participants from utilizing Java standard library functions, the library utilizes a custom, "high-fidelity," image format with custom RGB and Alpha scaling. The codebase consists of several complex image transformations, such as basic edge detection and smoothing. Unit tests are provided for each transformation included in the GUI.

Development Task For the development task, we asked the study participants to implement a new image transformation in a provided skeleton class. Participants were provided with both a detailed specification of the feature and corresponding unit tests. The development task was was chosen such that the new image transformation duplicates the functionality contained within two existing classes. Neither of the existing functionalites are obviously exposed in the GUI. Completing the task by extracting the first clone to a method and inserting a call to the function containing the second clone requires approximately 15 lines, whereas duplicating the functionality (e.g., via copy and paste) requires approximately 55 lines.

Maintenance Task Once the participant completed the development task, we introduced a maintenance task consisting of fixing a bug caused by a method not blending the Alpha channels during the image transformation. The bug can be fixed by replacing a line of code and adding a new line of code. We provided the participants with a bug report and the location of the buggy method in the source code.

Additionally, we suggest that the same bug might occur elsewhere in the codebase, and indicate that these bugs should also be fixed. In reality, the subject program contains two instances of the same bug: a direct, copy-and-paste clone

of the original section, and a clone which uses the buggy code as part of a different behavior. At both locations, the subject program includes a comment that mentions there may be a bug in how the Alpha channel is handled. There are no public unit tests that expose these bugs.

Tool Setup For the study, we used the Simian [3] code clone detector, as JCCD [2] and CheckStyle's [1] speed were insufficient for real-time use.

Study Participants The two study participants were a computer science PhD student (Participant 1), and a programming intern at the University of Washington (Participant 2). Both participants had at least basic experience using both Java and Eclipse.

4. Experimental Results

4.1 Development Results

During the development task, we expected that each participant would write at least two clones, as the transformation partially duplicated the behavior of two existing transformations. Additionally, for each class, we expected that the tool would detect the clones, and present the four actions to the participants, aiding their development.

Participant 1 began the development task by first exploring the codebase for existing code with similar functionality. He manually inspected several files until finding a transformation which applies a kernel filter to an image. The participant stated that he was familiar with the kernel approach, and copied the code manually. The tool detected the copied code in the new file, and alerted the participant, frustrating the participant. The participant spent a significant amount of time adapting the copied code to meet the specification. The tool continued to annotate the region under development as a clone during this process. Upon implementing the first stage of the image transformation, the participant began implementing the second stage without further consulting the codebase. Because the participant's implementation differed structurally from behavioral clone in the codebase, the detector did not detect any duplication during this phase of the task. We stopped the task as the participant was debugging the code because 40 minutes had already elapsed, and we were confident that no additional code clone information would be detected or presented.

Participant 2 began development by examining a 2-line clone that had been marked in the constructor of the provided skeleton class. He *manually* opened the system clone in an editor, keeping the editor open to the right of the main development window throughout the task. Though the class the participant opened contained the functionality required for the second stage of the clone, he did not notice this – he instead used the existing code as a reference client for calling the image API. The participant's implementation of both stages of the image transformation made use of many small helper methods, perhaps in an attempt to avoid code duplication; the clone detector did not detect any code duplication. As with the first participant, we stopped the task as the participant was debugging their code because 40 minutes had elapsed, and the code had stabilized to the point that we believed no additional code clones would be introduced.

Participant Feedback Consistent with his observed experience, Participant 1 indicated that the tool was not useful during the development task. Participant 2 reported that the tool was moderately helpful during development for pointing him to similar code, showing examples of API usage and code structure.

Discussion The results suggest that the tool is not useful for development because new developers on a project are unlikely to write code that is detectable as a clone, due to differences in both style and strategy. Furthermore, false positives in clone detection can be harmful, as the developer may waste time discovering that code which is structurally similar is, in fact, behaviorally different than what is required. Future work ought to explore (1) how to detect clones that are behaviorally similar, but differ structurally, and (2) aide users in understanding the behavior of a piece of code, or the difference in behavior between two clones.

4.2 Maintenance Results

During the maintenance task, we expected the participants to first examine the code in the class and method where the bug was initially reported. The participants would then utilize the JUMPTO action proposed by the tool to view the two clones, and fix the bug in each of the clones *separately*.

Participant 1 examined the initial bug location. Prior to beginning to fix the bug, the participant viewed the tool's suggestions for the clone, which only included the PASTE and JUMPTO actions for the exact clone, and then selected the JUMPTO action. The participant then visually examined the two clones to assure himself the the clones were identical. Upon assuring himself, he manually replaced the system clone in the existing code with a call to the buggy function. Once again viewing the initial bug location, the participant selected the JUMPTO action for the partial clone. Via visual inspection, the participant correctly identified the code as merely a partial clone, and therefore decided not to extract it as a method. He then manually fixed the bug in both clones by making the expected changes. Upon fixing the bug, the participant stated that he did not trust the clone detector, and therefore manually inspected the other classes for other code containing the bug. He incorrectly identified another location, and made the corresponding change (He did not run the unit tests, and so did not notice the mistake). This mistake can be attributed to an imprecise description of the bug in the instructions.

Participant 2, upon viewing the bug location, immediately examined the tool's suggestions without first examining the buggy class and method. Instead of selecting the JUMPTO

actions for the exact and partial clones, the participant *manually* opened the clones in separated buffers, again opting for a side-by-side view. As with Participant 1, he manually replaced the system clone in the existing code with a call to the buggy method. Again, before fixing the bug in the method, he examined the partial clone closely. He then attempted to rewrite the partial clone to use the buggy method by adding additional parameters (including an interface he defined) to the buggy method. In the process of fixing compilation errors, the participant investigated a short clone for which the system clone was in an unrelated class, and tried unsuccessfully to manually refactor the clone. Eventually, the participant reread the instructions for the task, and attempted to fix the bug. He struggled to investigate the his code with the Eclipse debugger, as the code was made complicated by his code changes. We stopped the task after 30 minutes had elapsed, as we believed the code had stablized.

Participant Feedback Both participants indicated that the tool was useful for discovering clones during maintenance. Participant 1 felt that tool would be especially useful for discovering cases in which novice developers had copied and pasted a significant amount of code. However, he could not recall a situation from working with his own code in which the tool would be significantly helpful.

Discussion As expected, the tool enabled both participants to quickly identify the locations where the bug had been duplicated, regardless of whether the clone was exact or partial. Both participants attempted to fix the bugs in the other locations. While the both participants chose to refactor code by inserting a method call, the tool failed to provide INSERTCALL and EXTRACTMETHOD suggestions for those clones. We have not yet investigated whether this is because the score was too low or because Eclipse's refactoring engine could not determine how to extract the code into a method.

4.3 General Discussion

Niether participant trusted the tool: Participant 1 suspected that the tool missed clones of the buggy code in the maintenance task, and Participant 2 generally executed the actions by hand instead of selecting the actions in Eclipse's Quick Fix dialog, as he was unsure what automatic action the tool would take. This lack of trust may have been caused by a lack of understanding, probably caused by the limited tool description that were included with the participant instructions. Evidence of this lack of understanding included the fact that both participants neglected the suggestion rankings. Participant 1 reported only focusing on the top-ranked suggestion. Participant 2 reported assuming the suggestions were unordered. In subsequent studies it may be helpful to describe the tool's operation in greater detail, perhaps by demonstrating the tool on example code.

4.4 Threats to Validity

The evaluation performed is a preliminary study of the efficacy of the tool, and was not intended to be conclusive. That being said, this study, and potentially larger studies of the same design have the following potential threats to validity:

- The study participants do not have to maintain the code in the future, and are therefore may be more likely to perform a short-sighted action. Participant 1 reported not performing method extraction due to time constraints.
- The results may not generalize, as in real software there may be many de facto or de jure restrictions (e.g., a certain module cannot be changed) restricting the set of actions a user can make or coding styles they must use.
- This tool seems especially helpful if it suggests that a developer is cloning code which that developer personally wrote in the past. Such code would likely be structurally and stylistically similar to the original code. Additionally, the developer may be more likely to refactor or modify the existing code. In contrast, if the tool is able to detect a section of code written by another developer in production code, it may be less beneficial for the developer to refactor or otherwise modify it.
- As the developer is performing relatively few tasks, the adaptive ranking system does not have the opportunity to adapt to the developer's preferences.

5. Related Work

The code clone literature can be divided into two areas: (1) *post*-hoc code clone detection and (2) *development-time* clone management. By post-hoc we mean the scenario in which detection is manually invoked to find all clones across an entire codebase, typically with the assumption that the codebase compiles and has already passed some level of testing. In contrast, we use development-time to refer to the scenario in which detection automatically occurs as the user modifies code, typically without requirements about the compilability of the code. The techniques used in the post-hoc detection can vary greatly based on the authors' definition of what level of similarity defines a "clone", but typically utilize an ad-hoc model of clones. In contrast, the development-time literature focuses on the creation and maintenance of formal code clone models during the development process.

Roy et al provide a survey of post-hoc code clone detection techniques [15], and compares them by testing each on clones with varying levels of similarity. We do not wish to repeat the survey here, so we refer the reader to [15]. The rest of this section surveys the related work on clone maintenance and refactoring, recommender systems, development-time clone management, and *development-time* code clone detection.

Post-hoc Clone Maintenance and Refactoring Several tools are similar to our work in that they attempt to not only detect clones, but also provide refactoring techniques to eliminate the clones. However, they are post-hoc, so in contrast to our technique the detection and refactoring proposals are only presented once the programmer manually runs detection on a compiling codebase.

For example, Fanta and Rajlich propose a number of potential refactorings for clones including function insertion, function encapsulation, and method extraction [6]. They present a case study for a C++ project which demonstrates that code refactoring is an important addition to clone detection. However, they do describe any way to rank or score the refactorings, the programmer is required to choose based on code knowledge

Higo et al. present Aries, a tool which organizes various clone information and presents it to the user [7]. The tool displays the cloned blocks of code and presents refactoring options such as method extraction; it augments the options with various metrics, including position in the class hierarchy, and number of external variables. These metrics are intended to help the programmer decide which refactoring, if any, is appropriate. They do not, however, consider how to automatically propose the most helpful refactoring(s) based on these metrics, in contrast to our adaptive scoring system.

Kawaguchis et al. present a Microsoft Visual Studio interface for displaying code clones in real-time to support software *maintenance* tasks [10, 16]. Their SHINOBI system uses the CCFinderX's preprocessor and the Suffix Array technique for indexing clones. Displayed clones are ranked via the sum of the ratio files committed at the same time and the ratio of files opened or edited at the same period in Visual Studio. Note that, unlike our tool, this system only *displays* detected clones in real-time, detection is still a manual post-processing step.

Development-time Clone Management From a user-interface stand-point, perhaps the work most similar to ours is de Wit et al.'s CLONEBOARD Eclipse plugin that tracks clone created by copy-paste operations [4]. Inspired by [14], the plugin registers code from copy-paste operations as clones and prompts the developer with a set of actions when the clone is modified: parameterize clone, unmark clone's tail, unmark clone's head, postpone resolution, unmark clone, apply changes to all clones, ignore changes. Inconsistent clones are identified via a red marker on the left-column of the editor. Unlike our tool, only clones arising from copy-paste operations are tracked, and the developer explicitly manages the clone linkages.

Duala-Ekoko et al. present CLONETRACKER, an Eclipse plugin for managing code clones that abstracts groups of clones via clone region descriptors (CRDs) to track clones across software versions [5], a stark contrast toour ad-hoc model. The tool requires users to explicitly create tracked clone groups by "documenting" a group of results from the SimScan clone detection tool. CLONETRACKER additionally supports simultaneously editing clones. However, in the author's trials, the success rate of this feature to correctly modify the clones was 80%.

Recommender Systems Holmes and Murphy built the Strathcona tool for Eclipse which displays relevant API usage examples when the developer performs a query by selecting a region of code in the IDE; the search is based on the structural content of the query line(s) [8]. This is a similar interface to searching for clones in real-time, but a much simpler process because only the API call must be matched. Related systems also exist, but require the user to perform a formal query, or to write special comments in the code.

Real-time Code Clone Search Applying code clone analysis during development places speed demands on the detection algorithms. However, the need to only perform clone detection in a single direction provides many opportunities for speedup compared to traditional methods, which must search over all pairs of potential clones.

Keivanloo et al. describe SeClone, a system for Internet code clone search that performs clone pair clustering based on a ontology base on features such as similarity [11]. Similar to CCFinder, it preprocesses files by generating the AST and abstracting the tokens. The code patterns are used to quickly perform search, false positives are limited by a retained set of type information. Results are clustered via file-level type information.

Lee et al. introduce a method for instant structural code clone search over large repositories by utilizing an R*tree indexing structure over the characteristic vectors [12].

Both [11] and [12] only address finding the clones quickly, and do not address locating possibly partial clones during development, or presenting actions to the developer based on the results.

6. Conclusion

In the past, code clone detection and analysis has been viewed as a post-hoc maintenance problem. We believe informing developers about clones as they are modifying code can both improve code organization and reduce development time. We have described and implemented a tool which realizes this belief by identifying clones and suggesting refactorings in real-time during development and maintenance.

Additionally, we have performed a user study which indicates that, while the tool is ineffective at aiding development by leveraging existing code, the tool is beneficial for guiding users to relevant clones during code maintenance. More sophisticated clone detection techniques are required to identify clones that arise during development, as they are likely to structurally differ from existing code with the same behavior.

References

[1] Checkstyle - duplicate code. `http://checkstyle.sourceforge.net/config_duplicates.html`, Oct 2011.

[2] Jccd - a flexible java code clone detector api. `http://jccd.sourceforge.net/`, Oct 2011.

[3] Simian - duplicate code detection for the enterprise. `http://www.harukizaemon.com/simian/`, Oct 2011.

[4] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 169 –178, Sept 2009.

[5] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[6] R. Fanta and V. Rajlich. Removing clones from the code, 1999.

[7] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *J. Softw. Maint. Evol.*, 20:435–461, November 2008.

[8] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[9] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[10] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. Shinobi: A tool for automatic code clone detection in the ide. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 313 –314, Oct 2009.

[11] I. Keivanloo, J. Rilling, and P. Charland. Seclone - a hybrid approach to internet-scale real-time code clone search. *International Conference on Program Comprehension*, 0:223–224, 2011.

[12] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim. Instant code clone search. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 167–176, New York, NY, USA, 2010. ACM.

[13] J. Li and M. D. Ernst. CBCD: Cloned Buggy Code Detector. Technical Report UW-CSE-11-05-02, U. Wash. Dept. of Comp. Sci. & Eng., Seattle, WA, USA, May 2, 2011. Revised Oct. 2011.

[14] Z. Mann. Three public enemies: cut, copy, and paste. *Computer*, 39(7):31 –35, July 2006.

[15] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495, 2009.

[16] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, and H. Iada. Shinobi: A real-time code clone detection tool for software maintenance. Technical Report NAIST-IS-TR2007011, Graduate School of Information Science, Nara Institute of Science and Technology, 2008.