Measuring the Efficacy of Code Clone Information in a Bug Localization Task: An Empirical Study

Debarshi Chatterji[1], Jeffrey C. Carver[1], Beverly Massengill[2], Jason Oslin[1] and Nicholas A. Kraft[1]

[1]Department Of Computer Science
University of Alabama
Tuscaloosa, AL, USA
{dchatterji, oslin002}@ua.edu
{carver, nkraft}@cs.ua.edu

[2]Department of Computer Science
Tennessee Tech University
Cookeville, TN, USA
bamassengi21@tntech.edu

***Abstract*—Much recent research effort has been devoted to designing efficient code clone detection techniques and tools. However, there has been little human-based empirical study of developers as they use the outputs of those tools while performing maintenance tasks. This paper describes a study that investigates the usefulness of code clone information for performing a bug localization task. In this study 43 graduate students were observed while identifying defects in both cloned and non-cloned portions of code. The goal of the study was to understand how those developers used clone information to perform this task. The results of this study showed that participants who first identified a defect then used it to look for clones of the defect were more effective than participants who used the clone information before finding any defects. The results also show a relationship between the perceived efficacy of the clone information and effectiveness in finding defects. Finally, the results show that participants who had industrial experience were more effective in identifying defects than those without industrial experience.**

Keywords-software clones; empirical studies; clone report; bug localization; software maintenance

I. Introduction

Recent estimates indicate that software maintenance costs account for as much as 70 to 90% of total software cost [4, 9] Techniques and tools that can reduce the effort spent by developers on these activities are required to reduce maintenance costs. One characteristic of a software system that can adversely affect its comprehensibility is the presence of similar or identical segments of code, or code clones.

A common approach employed by developers while coding is to identify sections of code that can be copied, pasted and possibly edited, rather than writing completely new code [14]. Other developer activities lead to the introduction of code clones, including: language construct recurrence, pattern or paradigm adherence, and framework reuse. On average, five to twenty percent of a software system is cloned code [20]. For example, CCFinder, a popular clone detection tool [12] determined that almost 30% JDK v.1.3.0 is cloned code. CP-Miner, a tool for finding copy-paste bugs, found that over 22% of the code in version 2.6.6 of the Linux kernel was cloned [16]. In addition, one study reported over 59% cloned code in a COBOL payroll application [8].

There are conflicting views on whether code clones are problematic. Earlier work by Fowler et al. classified code clones as a 'bad smell' that would increase the difficulty of maintenance [10]. However, more recent work indicates that code clones may not be as harmful as originally believed [14, 19]. In fact, code clones may actually improve productivity [13]. Rahman et al. found little empirical evidence that clones negatively affect software maintainability but did find that cloned code may be less fault prone than non-cloned code [17]. There is agreement that a developer's awareness of code clones is critical for performing correct (and complete) software maintenance.

Researchers have recently developed both techniques for clone detection and tools to implement those techniques. However, additional data about the clones in the system are needed to assist developers who must make decisions regarding these clones during software maintenance tasks. For example, when a change is to be made to one clone fragment in a clone group (collection of clone fragments), a developer must determine whether the change should be made to all other clone fragments in that clone group. Such decisions are critical, because if the developer makes an incorrect decision, then a bug could be introduced. A developer's analysis and decision-making process regarding code clones can be complicated further by the sheer volume of clones that are present in a large software system.

Due to the lack of empirical studies in the literature, it appears to be assumed by the creators of the clone detection approaches that if a developer is provided with the report from a code clone detection tool, they will know how to use it to perform various software engineering tasks. In searching the literature, we were unable to find any human-based empirical studies that verify the usefulness of the information provided in the code clone reports that are output from the tools. Therefore, the main motivation behind this study was to observe how developers use the output from a code clone tool to perform the maintenance task of bug localization, without being given specific instructions on how to use the tool output. Often, in an industrial setting, developers use new tools without the benefit of detailed training. So, our experimental setup mirrors that reality.

To focus the study, we developed the following research questions:

- **RQ 1***: How do developers use the information from a clone report produced by a code clone detection tool while performing a bug localization task?*
    - **RQ 1.1***: Is the information in a clone report useful for finding defects?*

- **RQ 1.2:** *Does the information from the clone report lead developers to identify false-positives?*

- **RQ 2:** *Do novice and professional developers use the information from the clone report differently? If so, how?*

Our long term goal is to develop methods to improve the usefulness of code clone reports (and other clone-related artifacts). The results of this study and other follow-on studies will help guide that work.

The remainder of the paper is organized as follows: Section 2 reviews related work. Section 3 describes the study. Section 4 contains a detailed data analysis. Section 5 describes the threats to validity. Section 6 concludes the paper and describes plans for future work.

II. Related Work

There has been a considerable amount of research conducted on clone detection approaches and tools. For example, Bellon et al. [2] use eight open source software systems to quantitatively compare and evaluate six clone detectors, including CCFinder, which we used in our study. Roy et al. [18] qualitatively compare and evaluate over 40 clone detection techniques and tools using a unified conceptual framework.

Other recent empirical studies, such as those by Bettenburg et al. [3] and Thummalapenta et al. [19] investigate the genealogical traits of code clones. That is, these studies track clone groups across multiple revisions of software projects to draw conclusions about developers' knowledge of clones. For example, Bettenburg et al. reported that the majority of long-lived clones in Apache Mina and jEdit are instances of the *replicate and specialize* pattern, in which code is cloned and then customized to implement a new feature. Moreover, the authors found that errors to replicate-and-specialize clones were not introduced at a high rate by inconsistent changes. Based on this finding, the authors concluded that developers of both Mina and jEdit are aware of those long-lived clones and are able to effectively manage their independent evolution.

On the other hand, there is a dearth of human-based empirical studies that focus on understanding how software developers use clone detection and analysis tools to maintain cloned code. To our knowledge, only two such studies have been published. [7, 14].

de Wit et al. [7] developed CLONEBOARD, an Eclipse plug-in that tracks changes to clones to help prevent inconsistently modified clones. To evaluate CLONEBOARD, de Wit et al. conducted a user study of seven software engineers performing a programming assignment. The study results indicated that the developers saw some value in CLONEBOARD. However in practice CLONEBOARD failed to meet the expectations of the users when fixing clone related bugs. The users believed that it would need a better user interface to be more useful.

Kim et al. [14] performed an ethnographic study of copy-and-paste programming practices. They observed nine developers for about 10 hours of Java/C++/Jython programming. The researchers manually logged edit operations (copy, cut, paste, delete, undo, and redo). As these operations were performed, the developers indicated their intention for copy-and-pasting. Kim et al. also used an Eclipse plug-in to automatically track 50 hours of Java source code edits by five other developers. The researchers used this information to infer the programmers' intentions for copying-and-pasting. They then interviewed each programmer twice to confirm the accuracy of the inferences. The study conclusion was that developers typically wait until after several copy-and-paste operations before restructuring the code.

Balint et al. [1] performed a retrospective study in which they created an author-centric view of clone evolution in Apache Ant, ArgoUML, and Ptolemy II. Though the authors did not directly observe developers, they mined software repository data to track the activities of project developers over time. By analyzing visualizations of the clone evolution data, Balint et al. identified five cloning activity patterns: (1) consistent line/block cloning with unique author, (2) creation of clones by multiple authors using consistent block cloning, (3) consistent line/block cloning with multiple authors, (4) inconsistent line cloning fixed by same author, and (5) inconsistent line cloning fixed by different authors. The results indicated that the rate of detection of inconsistent changes correlated with the number of developers. Thus, the specific developers involved should be considered when analyzing code clones.

Other researchers have studied the potential impacts that code clones can have on the software development process. Krinke et al.'s [15] study of open source software indicated that half of the changes to code clone groups are inconsistent with each other. A study by Cordy et al. [6] reports that removing clones actually increases risk in large software systems. Similarly, Kapser et al. [13] claim that clones have a positive impact on maintainability. They describe several patterns of cloning and discuss their benefits for the long term evolution of software. Instead of eradicating repeated code there should be effort towards developing tools to support long term maintenance of clones. Our study takes a step in this direction by providing insight into how developers use code clone information.

We presented a preliminary version of this work at the PLATEAU 2010 workshop [5].

III. Study Overview

The goal of this study was to understand how developers use the information generated by a code clone detection tool to support a standard maintenance. task. To make the study tractable, the maintenance task was *bug localization* (i.e., identifying the source code elements that must be modified to correct a bug) rather than *bug repair*. This choice of tasks is logical because clone information is more useful in identifying cloned code that must change than it is in actually changing the code. Section III.A describes the application used in the study. Section III.B describes the clone information used in the study. Then, Section III.C gives a detailed description of the study design. Section III.D explains the data that was collected during the study. Finally,

Section III.E describes the pilot study conducted prior to performing the main study.

A. Software System: Apache Ant 1.6.5

Because the difficulty of the bug localization task is highly dependent upon the software system being examined, we determined that the software used in this study should:

- be large enough to make bug identification a nontrivial task;
- be small enough that participants could perform the bug localization task in one hour; and
- contain both a cloned bug and non-cloned bug to control for the placebo effect.

To meet these requirements we chose *Apache Ant* version 1.6.5. Apache Ant automates the software build process, similar to Make. Version 1.6.5 contained two distinct bugs that met the requirements. Below is a detailed description of the two bugs.

We retrieved the two bug reports from the Apache Bugzilla repository. The bug IDs are 38175 [1] (affected *Copy.doFileOperations* and *Copy.doResourceOperations*) and 38082 [2] (affected *Scp.parseUri*). Both bugs were reported in version 1.6.5. However, the first bug was actually introduced into the trunk after the release of version 1.6.5. Thus, revision 367315 of the trunk of the Ant Subversion repository was used in this study. This revision was the last in which both bugs existed. In particular, the bugs in *Copy.doFileOperations*, *Copy.doResourceOperations*, and *Scp.parseUri* were repaired in revisions 367316, 367342, and 417590, respectively. The Bugzilla repository only provides the reproduction rules for the Windows platform. However, because the participants in our study worked on a Unix platform, we also provided the reproduction rules for the Unix platform.

Bug 1: The *failonerror="no"* doesn't work for locked file. When we try to perform a recursive copy in a directory that contains a locked file, the copy fails before the end of the whole copy, even if I have the attribute *failonerror* set to *"no"*.

Reproduction: If Mozilla Thunderbird is open, there is a locked file in the profile directory (*parent.lock*). Fig. 1 shows the reproduction of Bug 1.

Bug 2: SCP Task password with special characters. The *scp* task does not handle password with special characters like *"@"*. Fig. 2 shows the reproduction of Bug 2.

Bug 1 appeared in two clone fragments within the same clone group (and within different methods of the same class). We use the term *defect* to refer to an instance of a bug. In this case, there were two defects related to Bug 1. **Bug 2** appeared in only one code fragment (i.e., no clones). There was only one defect related to Bug 2. Using the defect identification form in Fig. 6, each participant had the opportunity to identify three defects (both instances of bug 1 and one instance of bug 2). The two bugs were actual, user-reported bugs available in the Apache Ant Bugzilla repository. We used solutions posted in Bugzilla to verify the correctness of the participants' solutions. Because the solutions were available on the web, the participants were not allowed access to the internet during the study.

```
Unix:
<copy failonerror="no" todir="test">
   <fileset dir =
            "/home/ubuntu/.thunderbird">
                   <include name="**/*" />
   </fileset>
</copy>

Windows:
<copy failonerror="no" todir="test">
   <fileset dir="C:\Documents and
        Settings\User\Application
        Data\Thunderbird\Profiles">
                   <include name="**/*" />
   </fileset>
</copy>
```

Figure 1. Bug 1

B. The clone report

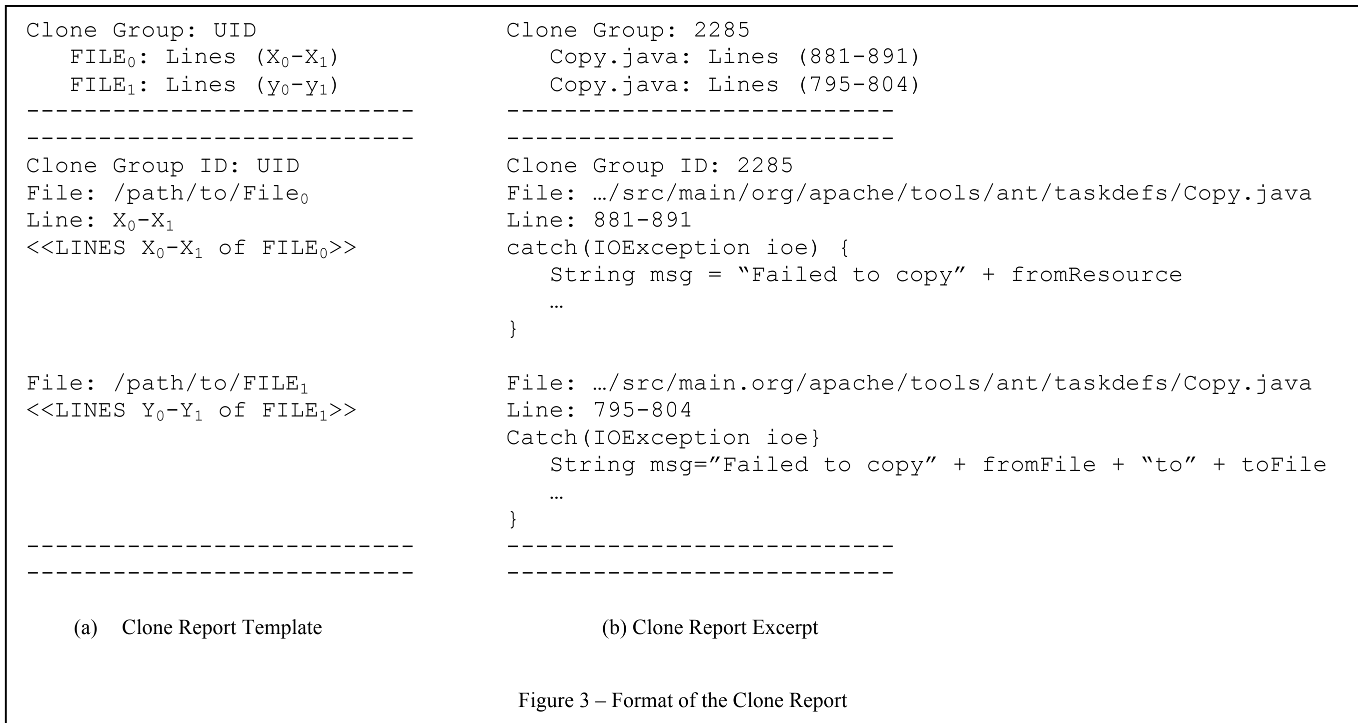
We used CCFinder version 10.2.7 to detect the clones. CCFinder records clone pairs, to which it assigns unique identifiers, in a plain text file. In addition, for each cloned code fragment, the token range is recorded in a collection of plain text files (one file for each input file). CCFinder provides an industrial-strength GUI to analyze and visualize the detected clones. However, to obviate the need for training the participants to use this complex GUI, we post-processed the plain text files and provided a simplified report, described below. The post-processing included merging related clone pairs to form clone groups and converting the token ranges to literal source code fragments.

Code Clone Report Format: The forms of clone group entries and source code are shown in Fig. 3. The format was as follows: clone group entries (separated by blank lines), a horizontal rule (60 dashes), and source code entries (separated by two horizontal rules (40 dashes).

```
<scp todir =
”user:p@ssword@cs.ua.edu:/home/ubuntu/
”>
    <fileset dir=”src_dir” />
</scp>
```

Figure 2. Bug 2

[1] https://issues.apache.org/bugzilla/show_bug.cgi?id=38175
[2] https://issues.apache.org/bugzilla/show_bug.cgi?id=38082



```
Clone Group: UID                    Clone Group: 2285
   FILE0: Lines (X0-X1)                Copy.java: Lines (881-891)
   FILE1: Lines (y0-y1)                Copy.java: Lines (795-804)
---------------------------         ---------------------------
---------------------------         ---------------------------
Clone Group ID: UID                 Clone Group ID: 2285
File: /path/to/File0                File: …/src/main/org/apache/tools/ant/taskdefs/Copy.java
Line: X0-X1                         Line: 881-891
<<LINES X0-X1 of FILE0>>            catch(IOException ioe) {
                                       String msg = "Failed to copy" + fromResource
                                       …
                                    }

File: /path/to/FILE1                File: …/src/main.org/apache/tools/ant/taskdefs/Copy.java
<<LINES Y0-Y1 of FILE1>>            Line: 795-804
                                    Catch(IOException ioe}
                                       String msg="Failed to copy" + fromFile + "to" + toFile
                                       …
                                    }
---------------------------         ---------------------------
---------------------------         ---------------------------
```

(a) Clone Report Template (b) Clone Report Excerpt

Figure 3 – Format of the Clone Report

C. Study Design

This section describes the participants, the training and the task performed.

Participants: 43 participants were drawn from graduate software courses, 13 from the University of Alabama (UA) and 30 from the University of Alabama in Huntsville (UAH). Some participants were novices (i.e. they had little or no industrial experience) while others had industrial experience.

The Training: In both courses, the course instructors gave the participants a brief lecture about code clones and their impact on software development and maintenance. These lectures helped the participants become familiar with the terms *code clone* and *code clone report*. The lectures did not give the participants specific guidance on how to use the clone information to perform various software engineering tasks. This information was not included in the training because the goal of the study was to test the efficacy of the clone report for the average developer, not one with detailed knowledge about code clones and the use of code clone information.

After the training, the participants received an overview document containing the following information needed to complete the experimental tasks.

1. A brief introduction to code clones, an overview of the Apache Ant system and definitions of important terms;
2. The directory structure of Apache Ant 1.6.5; and
3. Information on how to reproduce the two bugs.

The participants also received the Ant documentation, the j2dsk-1.4.2 documentation, and the clone report.

The Task: Based on the information provided during training, the participants were instructed to identify the specific location (file, method and line number) that should be modified to correct each bug. They were told that they should identify all portions of the code that must be modified to correct the bug. The participants were not expected to actually make the repair. All tasks were conducted in a Linux environment. The participants were allowed to use the standard Linux utilities "find" and "grep" within the terminal or were allowed to use the search dialog box provided by Ubuntu. All analysis was performed statically (i.e., the participants could not compile or execute the code). We used this approach to avoid introducing any bias due to the participants' familiarity with the development environment (e.g. the compiler or the execution environment). We wanted to ensure we focused the participants on the clone information rather than the functionalities of the development tools. A similar approach is used in the study by Fry et al. [11]. Once a participant identified a bug, he recorded his findings on the *defect identification sheet* described in the next section.

D. Data Collection

We collected two types of data: data from naturalistic observation and self-reported data.

Naturalistic Observation During the bug localization tasks, two of the authors stood behind the participants and watched over their shoulders (i.e., as passive observers). In the study

by Kim et al. [14] (which was mentioned earlier), the researchers began by using naturalistic observation. They later shifted to automated data collection when they noticed that the observers were creating an unnecessary disturbance to the participants. In our case, the observers did not interfere with or disturb the activities of the participants. In addition, the observers were able to answer the participants' questions about the conduct of the study. We considered adapting an automated data collection process similar to Kim et al., but because this approach has its own share of shortcomings, we chose not to use it.

Fig. 4 illustrates the experimental setting. The participants performed the study in groups of four. To observe all four participants simultaneously, the two observers had to observe multiple participants as follows. Each observer was responsible for two participants at a time. Every two minutes, they alternated between observing each of their two participants. After 30 minutes (one-half of the experimental session), the observers switched positions and observed the other two participants for the next 30 minutes.

For example, referring to Fig. 4, Observer 1 began by observing Participant 1 for two minutes while Observer 2 observed Participant 3. After two minutes, Observer 1 observed Participant 2, while Observer 2 observed Participant 4. After two minutes, Observer 1 and 2 returned to observing Participants 1 and 3, respectively. This pattern continued for 30 minutes. At that point, Observer 1 began observing Participants 3 and 4 while Observer 2 began observing Participants 1 and 2, switching every two minutes just as before.

The motivation behind this somewhat complicated data collection method was to help ensure the accuracy and consistency of data collected. The observation interval was set at two minutes so that it was long enough for the observers to observe what the participants were doing while being short enough for making as many iterations as possible to provide good insight into each participant's process. Also, by having the observers switch places after 30 minutes, we helped ensure that data was gathered consistently regardless of the observer. The pilot study, discussed later in this section, proved to be useful in finalizing the length of the observation interval.

In sessions with less than four participants, the observers observed the participants for two minutes, then did not observe them for two minutes to ensure that the same data was collected throughout the study. For example, if there were only 3 participants, Observer 2 would observe Participant 3 for two minutes and then observe no one for two minutes.

The observers maintained a 'fly on the wall' perspective during the experiment session unless the participants specifically asked questions. The questions asked by the participants included: "What exactly are we supposed to do here?"; "How do I read the clone report?"; "Can I execute the code?"; "Can I access the internet?"; and "I am not able to find the bug, what should I do?". The observers kept their answers as brief as possible and did not answer certain types of questions that would have unfairly aided a participant in completing the task. In this way the observers strove to not introduce any type of bias into the study.

During the naturalistic observation, the observers recorded the following information:

- the resources used (i.e. the code, the code clone report, the report form or the overview sheet);
- actions performed; and
- subjective notes about the participant's actions.

To record this information, the observers used the *Observer's Data Recording Form* shown in Fig. 5.

Self-Reported Data Participants self-reported data on two data collection forms: the *defect identification form* and the *post-observation questionnaire*. The participants used the *defect identification form* (Fig. 6) to report the identified defects. The form included places to record a time stamp, defect location, description of defect, steps taken to locate the defect and how they would fix the defect. Even though only bug one had two associated defects (the original and the clone), so as not to bias the results, the defect form provided

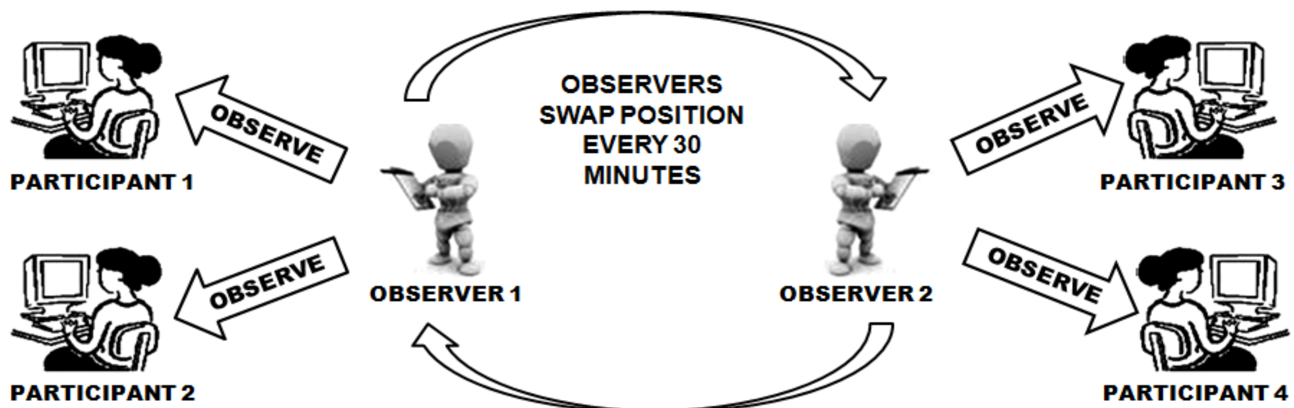


Figure 4. Study Design

Session # ______

Time: ______ Activity: ☐ familiarizing ☐ reporting Resource: ☐ clone info ☐ code ☐ documentation
Subject #: ___ Notes: ________________

Time: ______ Activity: ☐ familiarizing ☐ reporting Resource: ☐ clone info ☐ code ☐ documentation
Subject #: ___ Notes: ________________

Time: ______ Activity: ☐ familiarizing ☐ reporting Resource: ☐ clone info ☐ code ☐ documentation
Subject #: ___ Notes:

Figure 5. Observers' data recording form

space to report two defects for each bug. So the form contained four spaces for defect reports. Such a form allowed for the possibility of three correct defects and one false positive.. The information from this form helped us calculate the number of defects detected, the number of false positives, if any, and the time required to find the defects.

On the *post-observation questionnaire* the participants reported their level of experience with software development, bug localization, code clones and the CCFinder tool. The participants also could give their opinion of the usefulness of the clone report for bug localization.

E. Pilot Study

As good experimental practice, we conducted a pilot study to validate and debug the study design. Four graduate students, who were not going to participate in the main study, participated in the pilot study. The pilot study took place in a similar environment as the main study. The participants performed the same task on the same system. The results of the pilot study motivated some changes to the originally planned study design.

First, during the pilot study, the observers used a one minute observation interval. We determined that one minute was not long enough to observe and record the necessary information. As a result, we used two minute intervals in the main study. This two minute interval ensured that the observer could look closely at the work of the participants and record more accurate data.

Second, the pilot study participants suggested some other minor study design changes. These changes were mostly concerned with the overview document and the defect identification sheet to arrive at the versions used in the main study. For example, we rearranged the sections of the overview document and rephrased some questions on the defect identification sheet.

IV. Data analysis

This section is organized around the research questions presented in Section 1. We use an alpha value of 0.05 for judging statistical significance in all tests. Before presenting the analyses, we first give an overview of the data.

A. Overview of the Data

Each participant can be characterized using two variables. The first variable, *experience* came from the post-observation questionnaire. The second variable, *clone report usage strategy*, came from the observational data.

Defect Identification Sheet – Bug 1

Defect 1

Time Located: __________

Location: File ______________ Method ________________ Lines ______________

Description of Defect: ____________________________

What steps did you use to locate the defect? ____________________________

How would you fix this defect? ____________________________

Figure 6. Defect Identification Form

For the *experience* variable, on the post-observation questionnaire each participant used a five-point Likert scale to report his level of experience. Participants who had either no experience or only classroom experience were characterized as **novices (23 participants)**. Those who had industrial experience were characterized as **professionals (20 participants)**.

For the *clone report usage strategy* variable the observers noticed that each participant seemed to employ one of two strategies when using the clone report. The first strategy, called **after**, was to use the code clone information to search for cloned code only after first identifying a defect. The second strategy, called **before**, was to use the code clone information before identifying a defect. Because many participants were unfamiliar with code clone information, during the first 10 minutes a participant was allowed to familiarize himself with the clone report and still be characterized as using the **after** strategy.

Fig. 7 illustrates each participant based on the number of defects found and whether he or she reported the false positive. The vertical axis represents the participant's clone report usage strategy (**before** or **after**) and the horizontal axis represents the participant's level of experience. These two axes partition the space into four quadrants based on these two variables. Each of the concentric circles contains data points representing participants who found a specific number of defects. Within each circle, the distance from the center point is meaningless. For example, the participants represented by the two data points lying in the lower right quadrant of the outermost circle each found three defects. One of those participants also reported the false positive. The largest group of participants was those who found zero defects.

Prior to this study, most participants had little experience identifying defects in large software systems. The data showed that participants, who had some software development experience, as reported in the *post observation questionnaire*, were more confident and found more defects than those without such experience. The few participants who found all three defects had all worked on multiple industrial software projects. Another observation from this data is that in spite of having a detailed clone report; several participants were able to locate the initial defects but did not find the second defect for Bug 1 (i.e. the clone)

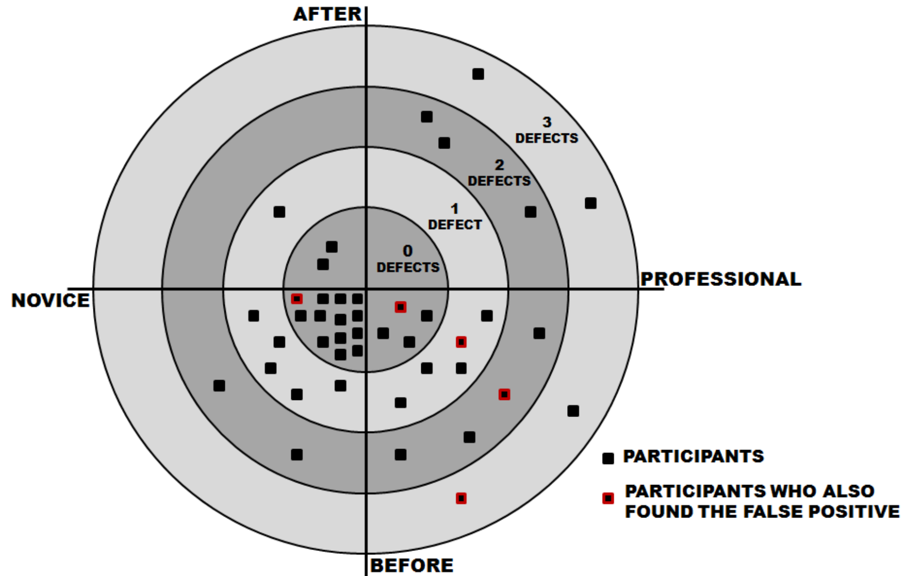


Figure 7. Overview of Data

B. RQ 1.1 –Is the information in a clone report useful for finding defects?

On the *post observation questionnaire* the participants used a 5-point Likert scale (1-not effective at all to 5-highly effective) to report the efficacy of the clone report for the bug localization task. The scatterplot in Fig. 8 shows the relationship between this efficacy value and the number of defects found by each participant. Because neither variable is normally distributed (based on the Shapiro-Wilk test) we conducted a conducted a Kendall's tau_b correlation. The correlation was .196 and was not significant (p=.13).

We also split the participants into two groups: those that had a positive impression of the clone report (4 and 5 on the scale) and those that had a negative impression (3 or lower). We then performed a t-test to compare the average number of defects found by the participants in each group. The participants who had a positive impression of the clone report were more effective than those with a neutral or negative impression of the clone report (1.45 defects vs. .78 defects on average). This difference was not significant (t_{41} = 1.946; p = .059).

We suspected that the analysis may be skewed by the large number of participants who found no defects. Therefore, to further analyze these results, the 19 participants who found no defects were excluded and the data reanalyzed. Fig. 9 shows a scatter plot of the data. In this case the Kendall's tau_b coefficient increased to .468 and was significant (p=.009). In addition, the t-test showed that

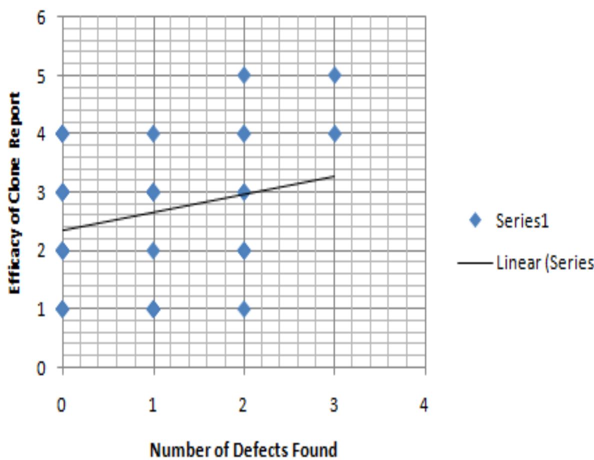


Figure 8. Positive relation between number of clones found and the clone efficacy

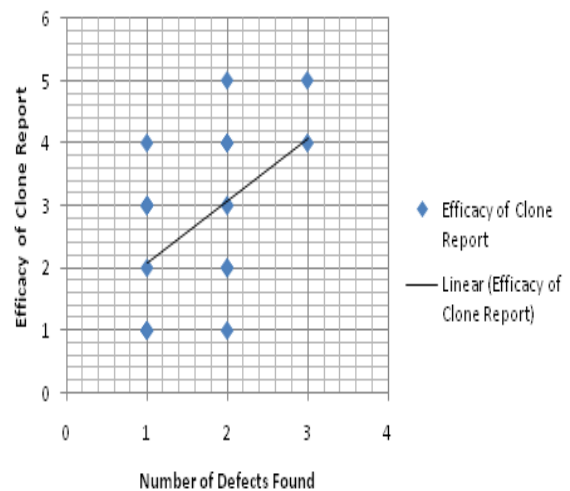


Figure 9. Positive relation between number of clones found and the clone efficacy excluding the data points for which number of bugs found was zero.

those who found the clone report useful were significantly more effective (2.29 defects vs. 1.47 defects) (t_{22} = 2.739, p = .012) than those who did not.

The data does not allow us to draw any conclusions about causality. We are not sure whether 1) finding the clone report useful caused the increase in defects found, 2) finding more defects resulted in a more positive view of the clone report, or 3) some other variable influenced the results. Further study is required.

As a second analysis to address this question, we investigated whether the *clone report usage strategy* impacted effectiveness. If a participant fully understood how to use the clone report, he would realize that the clone report was only useful after identifying a defect to help search for clones of the problematic code. Because the clone report is merely a list of cloned code, it is not useful for identifying the original defect. Fig. 10 and 11 illustrate the overall trend that participants who used the clone report after finding a defect tended to identify more real defects and fewer false positives than those who used it before finding a defect. The qualitative observations indicated that many of the thirty-five participants who used the clone information before finding a defect appeared to be attempting to use the clone information to locate the initial defects. This observation suggests that developers need at least a small amount of training on how to use clone information in order to use it effectively.

Following on this discussion of *clone report usage strategy*, we conducted two analyses. First, we tested whether either strategy made the participants more effective. Those who used the clone report **after** finding a defect found an average of 1.62 defects compared with 0.8 defects (out of a possible 3) for those who used the clone information **before** finding a defect. An independent samples t-test showed that this difference was significant (t_{41} = -2.146; p=.038). Second, we tested whether the participants who used the clone information after finding a defect were more effective in finding the cloned defect related to Bug 1. Twenty-five percent of the participants who used the clone

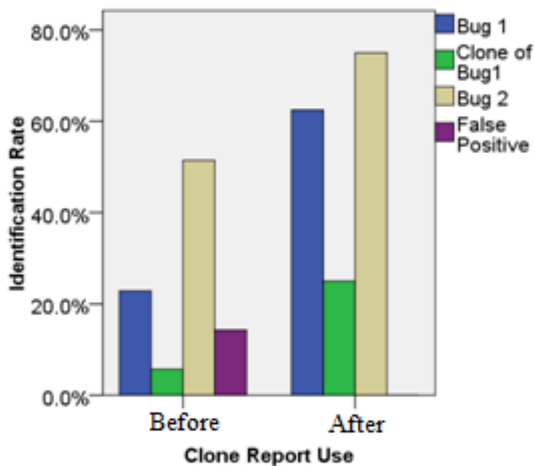


Figure 10. Clone report usage

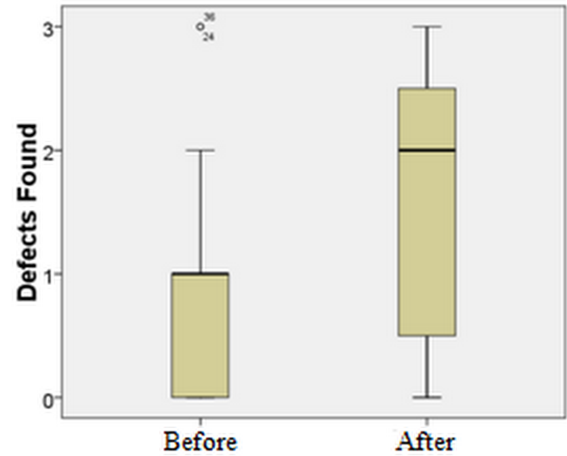


Figure 11. Correct vs. Incorrect Use of Clone Report

information after first finding a defect did find the cloned defect whereas only 5.71% of the participants who used the clone information before finding a defect were successful in finding the cloned defect. However, this difference was not significant (t_{41} = -1.713; p = .094).

Additionally, we computed the percentage of time each participant spent using each of the three resources provided to them (i.e., the clone report, the code base and the documentation). During each observation, the observer noted which resource(s) the participant used. If a participant used more than one resource, the time was divided accordingly. For example if during an observation a participant used only the clone report, 100% of the time was assigned to the clone report. If a participant used both the clone report and the documentation then each resource was assigned 50% of the time. Similarly, if the participant used all three resources, each resource was assigned 33% of the time. We made an assumption that the distribution of time between the resources would be the same for the time that each participant was not being observed.

Based on the number of defects found, we divided the sample into three groups: those who found no defects, those who found at least one defect, but not the cloned defect, and those who found all three defects, including the cloned defect. Table 1 shows the average percentage of time the participants from each group used the resources. The participants who found no defects used the clone information the most, followed by those who actually found the cloned defect. It is not surprising that the participants who found the cloned defect used the clone information more than those who did not find it. It is surprising to see that the participants who found no defects used the clone report more than anyone else.

A chi-square test indicated that the distributions among the three groups was significantly different (X^2_4 = 13.727; p = .02). At this point we cannot conclusively explain this result. It seems to indicate that those who found no defects were unsure of how to use the clone report and therefore were using it incorrectly. Logically, if a participant did not

TABLE I. PERCENTAGE USE OF RESOURCES

		Means of percentage use of resources		
		Clone Report	Code Base	Documentation
Defects found by the participants	No defects found (19)	27.95%	54.16%	17.89%
	One or two defects found (20)	14.5%	74.5%	10.7%
	All three defects found (4)	22.75%	71%	6.25%

find any defects, they should not have had little use for the clone information, as it is most useful in finding cloned defects

C. RQ 1.2 – Does the information from the clone report lead developers to identify false-positives?

This research question evaluated whether the clone report misleads a developer, resulting in the reporting of false positives while looking for the clone of a defect. None of the eight participants who used the **before** strategy reported a false positive. All five participants who did report a false positive used the **after** strategy. While this result was not significant (likely due to small number of participants who used the before strategy), the fact that no one who used the **before** process reported a false positive suggests that there may be an important phenomenon here.

D. RQ 2 – Novice vs. Professionals

To determine the effects of the participants' previous programming experience, we compared the effectiveness of novices and professionals. On average, the professionals identified more defects than the novices (1.53 vs. .43 respectively). This difference was significant (t_{41} = -4.222; p < .001). In addition, all four participants who correctly identified the cloned defect were professionals.

The professionals also tended to employ the **after** strategy of using the clone report more often than the novices did. Table 2 is the contingency table showing the distributions of these two variables. Even though a chi-square test did not show a significant result (X^2 = 1.010; p= .315), there is a positive trend.

V. THREATS TO VALIDITY

Construct Validity: We made the assumption that using the **after** strategy would be more effective based on theory and on participant observation. It is possible that this assumption is incorrect or that the method we used to partition the participants into the before group and the after group was biased. Either of these problems could introduce a threat to construct validity.

Internal Validity: There was potentially a subject expectancy threat that could have arisen from the fact that we gave the participants a clone report. The participants could have assumed that the clone report was supposed to be used for the bug localization task without fully understanding how to properly use it. We chose a system written in Java because we assumed most students would be familiar with that language. If someone was not familiar with Java, it could bias the results. Ideally the participants should have had an option to select the programming language with which they were most familiar. The participants performed the task in a Linux environment which could be a validity threat if they were not familiar with Linux and the search features it provides. Finally, to prevent a threat to validity of the participants being able to locate the solutions on the web, we did not provide internet access. Similarly, the participants were not allowed to execute the code. This approach reduced a threat to internal validity that the results may be caused by participants' familiarity with the development and execution tools.. **External Validity:** The participant population consisted of both novices and professionals, reducing a threat to external validity. Conversely, the task was performed in isolation rather than as part of a complete maintenance process, which may introduce a threat to external validity. The task was a realistic task (i.e. bug localization), but it was not complete (i.e. the participants did not fix the bug). Also, because we increased internal validity by preventing the participants from compiling or executing the code, the way a developer would typically work, there is a threat to external validity. Finally, the participants were not trained in using the clone report. Although this lack of training could introduce a threat to validity, one of our study goals was to see how people would use the report without training.

TABLE II. NOVICE VS. PROFESSIONALS

	Use of clone report		
	Before	After	Total
Novice	20	3	23
Professional	15	5	20
Total	35	8	43

VI. CONCLUSION AND FUTURE WORK

There have been few human-based empirical studies focused on code clones and the use of information from clone reports. This exploratory study provides insights into how developers use the information from a clone report. Some concrete conclusions from the study are:

- Initial evidence shows that, without training, most participants employed the **before** strategy of clone

report usage, which appears to be less useful than the **after** strategy. Some researchers argue that clone detection tools are needed to support the maintenance process because those tasks are often assigned to entry level developers. However, if those developers are not able to effectively use the clone report, such clone detection tools are of little use to them.

- There is a relationship between effectiveness and employing the **after** strategy of clone report use, although we have yet to establish a causal relationship.
- In a large software system the clone report might not help developers locate the initial defect, but it will help them locate clones of that defect.
- Use of the clone report may also reduce reporting of false positives.

As this was an exploratory study, there are a number of replications that we plan to conduct. Because the lack of training on how to use clone information was a threat to validity, a replication will be done to control for clone detection training (i.e. by training half of the participants how to use the clone information and not training the other half). Another replication will be performed in an environment where the participants could execute the code and repair the defects to see whether the results differ. Another replication will look at defects involving more than two instances of cloned code. For example, assume there are eight clones, but only three cause defects.

We will also investigate the false positives in more detail. We will look at the effects of the clone report on false positive identifications For example, if someone incorrectly identifies a defect and then, using the clone report, do they identify additional false positives?

Finally, this study serves as a starting point for a series of studies in which we will evaluate the effectiveness of various types of clone detection tools as well as various methods for presenting clone information to developers.

VII. Acknowledgements

We thank Dr. Letha Etzkorn for allowing us to run the study in her class at UAH. We also thank the participants for their time to complete this study. We acknowledge support from NSF grants CCF-0915559 and CCF-0851824.

References

[1] Balint, M., Girba, T. and Marinescu, R. "How developers copy." In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC).* 2006. pp. 56-68.

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

[3] Bettenburg, N., Weyi, S., Ibrahim, W., Adams, B., Ying, Z. and Hassan, A. E. "An empirical study on inconsistent changes to code clones at release level." In *Proceedings of the Reverse Engineering, 2009. WCRE '09. 16th Working Conference on.* 2009. pp. 85-94.

[4] Boehm, B. and Basili, V. R., "Software Defect Reduction Top 10 List," *IEEE Computer,* 34(1): 135-137. 2001.

[5] Chatterji, D., Massengill, B., Oslin, J., Carver, J. and Kraft, N. "Measuring the efficacy of code clone information: An empirical study." In *Proceedings of the Evaluation and Usability of Programming Languages and Tools (PLATEAU) [Held during SPLASH].* Oct. 18, 2010.

[6] Cordy, J. R., Dean, T. R. and Synytskyy, N. "Practical language-independent detection of near-miss clones." In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON).* 2004.

[7] de Wit, M., Zaidman, A. and van Deursen, A. "Managing code clones using dynamic change tracking and resolution." In *Proceedings of the Software Maintenance, 2009. ICSM 2009. IEEE International Conference on.* 2009. pp. 169-178.

[8] Ducasse, S., Rieger, M. and Demeyer, S. "A language independent approach for detecting duplicated code." In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM).* 1999. pp. 109-118.

[9] Erlikh, L., "Leveraging Legacy System Dollars for E-Business," *IEEE IT Professional,* 2(3): 17-23. 2000.

[10] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D., *Refactoring: Improving the Design of Existing Code.* ,1st ed. Addison-Wesley, 1999.

[11] Fry, Z. P. and Weimer, W. "A human study of fault localization accuracy." In *Proceedings of the Software Maintenance (ICSM), 2010 IEEE International Conference on.* 2010. pp. 1-10.

[12] Kamiya, T., Kusumoto, S. and Inoue, K., "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering,* 28(7): 654-670. 2002.

[13] Kapser, C. and Godfrey, M. ""Cloning considered harmful" considered harmful." In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE).* 2006. pp. 19-28.

[14] Kim, M., Bergman, L., Lau, T. and Notkin, D. "An ethnographic study of copy and paste programming practices in OOPL." In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE).* 2004. pp. 83-92.

[15] Krinke, J. "A study of consistent and inconsistent changes to code clones." In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE).* 2007. pp. 170-178.

[16] Li, Z., Lu, S., Myagmar, S. and Zhou, Y., "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering,* 32(3): 176-192. 2006.

[17] Rahman, F., Bird, C. and Devanbu, P. "What is that smell?" In *Proceedings of the 7th Working Conference on Mining Software Repositories (MSR).* 2010.

[18] Roy, C. K., Cordy, J. R. and Koschke, R., "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming,* 74(7): 470-495. 5/1. 2009.

[19] Thummalapenta, S., Cerulo, L., Aversano, L. and Di Penta, M., "An empirical study on the maintenance of source code clones," *Empirical Software Engineering,* 15(1): 1-34. 2010.

[20] Uchida, S., Monden, A., Ohsugi, N., Kamiya, T., Matsumoto, K. and Kudo, H., "Software Analysis by Code Clones in Open Source Software," *The Journal of Computer Information Systems,* XLV(3): 1-11. April. 2005.