Aiding Software Maintenance with Copy-and-Paste Clone-Awareness

Patricia Jablonski and Daqing Hou
School of Engineering
Clarkson University
Potsdam, NY 13699
{jablonpa, dhou}@clarkson.edu

***Abstract*—When programmers copy, paste, and then modify source code, the once-identical code fragments (code clones) can become indistinguishable as the software evolves over time. In this paper, we present three features of our software tool, a set of Eclipse plug-ins named CnP (CnP's clone visualization, CReN, and LexId), which aids the programmer during copy-and-paste programming. We believe that the clone-awareness that the tool provides can help programmers benefit from this clone information during debugging and modification tasks, develop software more efficiently, and prevent inconsistent identifier renaming within clones. We tested these hypotheses with a user study and present our results.**

Keywords-code clone; copy-and-paste programming; Eclipse integrated development environment; identifier renaming; Java.

I. INTRODUCTION

Copying and pasting source code results in multiple, identical code fragments (code clones) throughout a system, which may then be individually modified to fit their specific tasks. Though the contents of these clones may vary, some amount of similarity must be maintained between them, otherwise the programmer would not have made an exact duplicate as a base to work from. The correspondence between a pair of clones can be a useful piece of information when programmers modify or debug source code [5]. This can be helpful especially when programmers are working with someone else's source code or if it has been awhile since they have worked on their own. Maintaining the cloning relationship is thus a very important consequence of copying and pasting, and a responsibility left to the software maintainer. Without tracking clones over time, identifying and consistently changing clones can be problematic.

Due to the increase in source code maintenance as a result of cloning, there has been a variety of research with the aim to manage clones. One large area of "clone management" research focuses on clone detection and removal [17, 18]. In cases where it may not be easy or possible to remove clones [15], other research proposes ways to make consistent changes between clones [6, 22, 25]. A variety of research looks into the topic of clone-related bugs and inconsistent changes to clones [2, 8, 9, 10, 11, 12, 14, 16, 19, 20, 27]. Still other research recognizes that a common type of clone is a parameterized one [1, 4], where the programmer intends to make only small changes to the pasted code, like to the identifier names and constants.

Our contribution, the software tool CnP, is a proactive clone management environment that tracks copy-and-paste-induced clones upon creation. Based on the tracked cloning information, CnP provides support for clone-related maintenance activities [8, 9, 10]. In this paper, we study how CnP's support for copy-and-paste clone-awareness may be able to help programmers better comprehend the program source code in order to benefit from this clone information during debugging and modification tasks, develop software more efficiently, and prevent inconsistent identifier renaming within clones. We performed a user study to measure the effects of this kind of clone-aware programming on task completion speed and solution correctness. In the process, we learned about various kinds of task completion methods.

II. THE CNP TOOL

CnP proactively supports copy-and-paste programming in the Eclipse integrated development environment, detailed in [8]. In this paper, we present the results of a user study on three features of CnP – its clone visualization, its consistent identifier renaming feature (CReN), and its consistent substring renaming feature (LexId). Specifically, we tested the effect of the tool's features on the speed and accuracy of the programmer. Since CnP is an interactive software tool, a study with programmers was the best way to evaluate it.

A. CnP Clone Visualization

Clone tracking happens behind-the-scenes, keeping track of the clone and clone group relationships and their locations within the source code files. CnP automatically tracks only significant copy and pastes that contain certain program elements or number of lines of code (based on a configurable policy). The tool tracks code fragments at the granularity of a character, to the nearest contained abstract syntax tree (AST) node. The AST representation of the source code is accessible within the Eclipse framework. Further details of CnP's clone tracking can be found in [8, 9, 10].

In the latest version of CnP, we have improved the clone visualization feature to distinguish clone groups (related, similar clones that result from a series of copy and pastes) by coloring all clones within the same group with the same color of bars. We distinguish between the origin and its pastes by slightly darkening the colored bar that is next to each pasted region. For example, in Figure 1, the origin was the method "more_variables" (shown in the back), which has a regular shade of yellow for its visualization bar (since it is the original code fragment that was copied), while its pastes (the newly modified and related methods "more_arrays" and "more_functions") are shown with slightly more grayed

versions of the color yellow. These three clone instances belong to the same clone group, hence they are displayed with variations of the same color (yellow). A different code fragment that is copied and pasted (belonging to a different clone group) would be represented with shades of a different color, such as the color red.

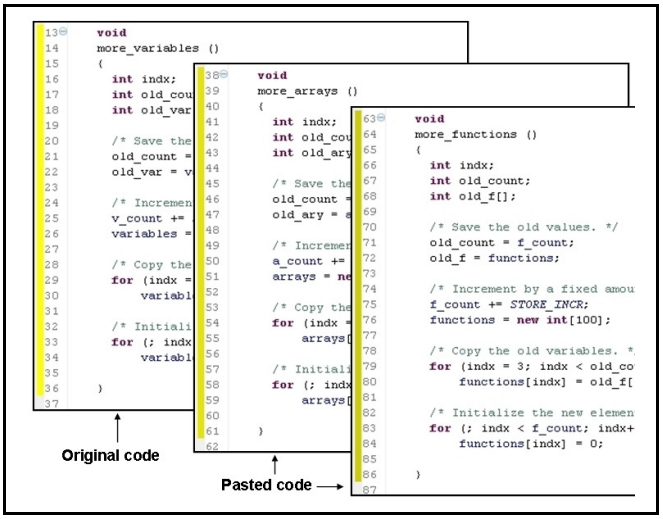

Figure 1. CnP clone visualization has distinction between clone groups and the clone origin and its subsequent pastes.

B. *CReN Identifier Renaming*

A programmer may make small modifications to pasted code, such as changing only identifier names or literal constants, in order for the new code to fit its task [1, 4]. CnP's CReN feature [9] supports the programmer in making consistent changes within these parameterized clones. CReN utilizes tracking and visualization and groups together the identifiers within the copied and the pasted code fragments. All instances of the same program element or identifier name within the code fragment are then modified consistently together when any one instance within the fragment is modified by the programmer. Examples with CReN are in our user study's Tasks 5 and 6 in Section III.C.3.

C. *LexId Substring Renaming*

LexId [10] is another part of CnP that renames parts of identifier names consistently together within code fragments. All instances of a common substring between all identifiers within a clone are renamed together as one of the substrings is renamed. LexId handles a different use case than CReN and instead focuses on inferring the lexical patterns across different identifiers, which is its own contribution. LexId can be made to have additional features, including inferring the new substring in pasted code based on the original without the programmer's editing input, inferring tokens (numbers), and inferring type and subtype patterns.

LexId currently determines substrings as those parts of an identifier that are separated by an underscore "_" or dollar sign "$" character, or by changes in character type (digits or letters) or case (uppercase or lowercase letters). The "_" and "$" are never substrings or part of a substring (they are strictly only separation characters). Figure 2 shows some examples of how our algorithm divides up an identifier into substrings. Standard Java CamelCase notation is supported. Examples with LexId are in Tasks 7 and 8 in Section III.C.4.

Identifier	Substrings
index	index
INCREMENT	INCREMENT
more_variables	more, variables
$arrays	arrays
promPhysTotal	prom, Phys, Total
CPMiner	CP, Miner
LinuxProm64Regs	Linux, Prom, 64, Regs

Figure 2. Examples of what LexId considers to be substrings.

III. USER STUDY DESIGN

The user study was designed to test three aspects of CnP that involve user interaction – CnP's clone visualization, CReN's identifier renaming, and LexId's substring renaming – with eight programming tasks across three task categories, in Table I. Based on our inferences about the tool's expected behavior, we developed a set of hypotheses for each software feature to test with human subjects in a controlled setting.

<u>CnP Clone Visualization Hypotheses:</u>

1. CnP's clone visualization makes it faster for programmers to find software bugs in copied-and-pasted code than debugging manually or with other tools, when the cloning information is not fresh in their memories.
2. CnP's clone visualization makes it faster and less error-prone for programmers to make modifications to copied-and-pasted code than modifying without visualization, when the cloning information is not fresh in their memories.

<u>CReN Identifier Renaming Hypotheses:</u>

3. Using CReN to rename identifier instances consistently in copied-and-pasted code is quicker than performing the same task manually or with other tools.
4. CReN prevents such inconsistent renaming errors that can happen otherwise.

<u>LexId Substring Renaming Hypotheses:</u>

5. Using LexId to rename substring instances consistently in copied-and-pasted code is quicker than performing the same task manually or with other tools.
6. LexId prevents such inconsistent renaming errors that can happen otherwise.

In order to validate these claims, we created a set of tasks in three main areas of programming – debugging (for CnP's clone visualization), modification (also for CnP's clone visualization), and renaming (for CReN and LexId). This resulted in eight tasks total, two for each task category or tool feature (see Table I).

TABLE I. DESCRIPTION OF THE TASKS IN THE USER STUDY

Task Category	Task #	Time Limit	Tool Feature Being Tested
Debugging Tasks	Task 1	10 minutes	CnP Clone Visualization
	Task 2	10 minutes	
Modification Tasks	Task 3	10 minutes	
	Task 4	10 minutes	
Renaming Tasks	Task 5	5 minutes	CReN Identifier Renaming
	Task 6	5 minutes	
	Task 7	5 minutes	LexId Substring Renaming
	Task 8	5 minutes	

A. *Subject Characteristics*

After Clarkson University Institutional Review Board (IRB) approval and the conclusion of a brief pilot study to test the experimental setup, we sent a recruitment email to all Clarkson computer science and engineering undergraduate and graduate students via their department mailing lists. We required that the participants in our user study were able to read and write simple Java programs and that they had experience with Swing (GUI programming). Familiarity with IDEs, especially Eclipse, was preferred, but not necessary.

We had fourteen subjects who participated in our user study, eight who were undergraduate students and six who were graduate students. All subjects were male. Based on the answers given on a user experience questionnaire, seven subjects had a long-time knowledge of both Java and Swing, three subjects had a long-time knowledge of Java but only recent knowledge of Swing, and the remaining four subjects reported having relatively recent knowledge of both Java and Swing (having learned them within the past two years).

Seven subjects said that they have written at least 10,000 lines of Java code since they have known the language, while the other seven gave a lower estimate when describing their Java programming experience. Some subjects mentioned that they have worked on medium to large-sized software projects, but this was not quantifiable. Ten subjects out of the fourteen said that the software that they have written in Java were for courses only, while the other four subjects had worked on software projects in industry. As a result, subjects who had worked on projects outside of the classroom considered themselves to be "very experienced", while those who were relatively new to Java and Swing or had only used them for courses claimed to be knowledgeable about Java and Swing, but admitted not being regular users of them.

B. *Study Procedure*

We invited each subject, one at a time, into a user study laboratory for a session that lasted between one and two hours. After signing an informed consent form, we presented background information to the subject about what code clones are and problems that can result from copy-and-paste programming. We then gave the subject a short introduction to our software tool's three features, explaining how they address the mentioned copy-paste-modify issues, and told them about the other tools that they could use during the experiment when they were doing a task without CnP.

Following this part of the introduction material, we showed them the software that they would be working with during the user study, which we had set up within the Eclipse IDE (Ganymede version 3.4.2 build M20090211-1700) on Windows XP. We adapted the publicly available Paint program from Carnegie Mellon University [16] for use in our user study, which is shown with its identifier names labeled in Figure 3. When we had modified the CMU Paint program, we became very familiar with its source code and found that it is a typical GUI program. We located the related clones in the source code and designed the user study's tasks based on this. The Paint program was a good choice for our user study, since GUI software often contains many common code fragments that are intentionally copied and pasted and not abstracted away into procedures. We ran the program to show them what it looks like graphically, and then we went through the major parts of the source code with them so that they would have some basic familiarity with it before starting the programming tasks. We told the subjects that all tasks in the user study would be performed in one file (PaintWindow.java, which is 264 lines of code).

We then concluded the presentation by reminding the subjects that they will complete eight programming tasks (four for clone visualization, two for CReN, and two for LexId), followed by a user experience questionnaire. We asked that they talk aloud while completing each task, so that we could better understand their thoughts and actions, and we had them announce when they were finished with the task. Each task had a time limit that the subject was told about in advance (see Table I). As an incentive to work as efficiently as possible, we told everyone that we would provide extra compensation to the four subjects who completed the eight tasks with the best accuracy and speed. Finally, the subjects were told that their session was being recorded with TechSmith's Morae software (version 2.0.1), which captured the activity on the computer screen and recorded the user with video and audio.

C. *Task Descriptions*

Before the subject was asked to start a task, we read the current task's description to him, which explained the problem that he was to solve with a screenshot of what our desired solution should look like. The subject was able to keep this instruction sheet and a sheet that contained the identifier names used in the program to look at as a reference. Subjects were also allowed to use an online Swing tutorial that we had opened in the browser and to ask any clarification questions before beginning. We ran the specific Paint program that was different for each task for the subjects to see the task's problem visually. We told them whether the current task involved CnP, and if not, reminded them of some other tools that they could use.

Each pair of tasks (Tasks 1 & 2, Tasks 3 & 4, Tasks 5 & 6, and Tasks 7 & 8) was designed to be very similar in terms of difficulty and effort, yet different enough that there would not be a learning effect. The subjects completed each of the eight tasks once, alternating task completion with and without our software tool. Although odd-numbered subjects always performed the first task in each task pair with tool support, this is not a problem with the experiment design and would not create bias because each task category was unique and unrelated to the previously done pairs.

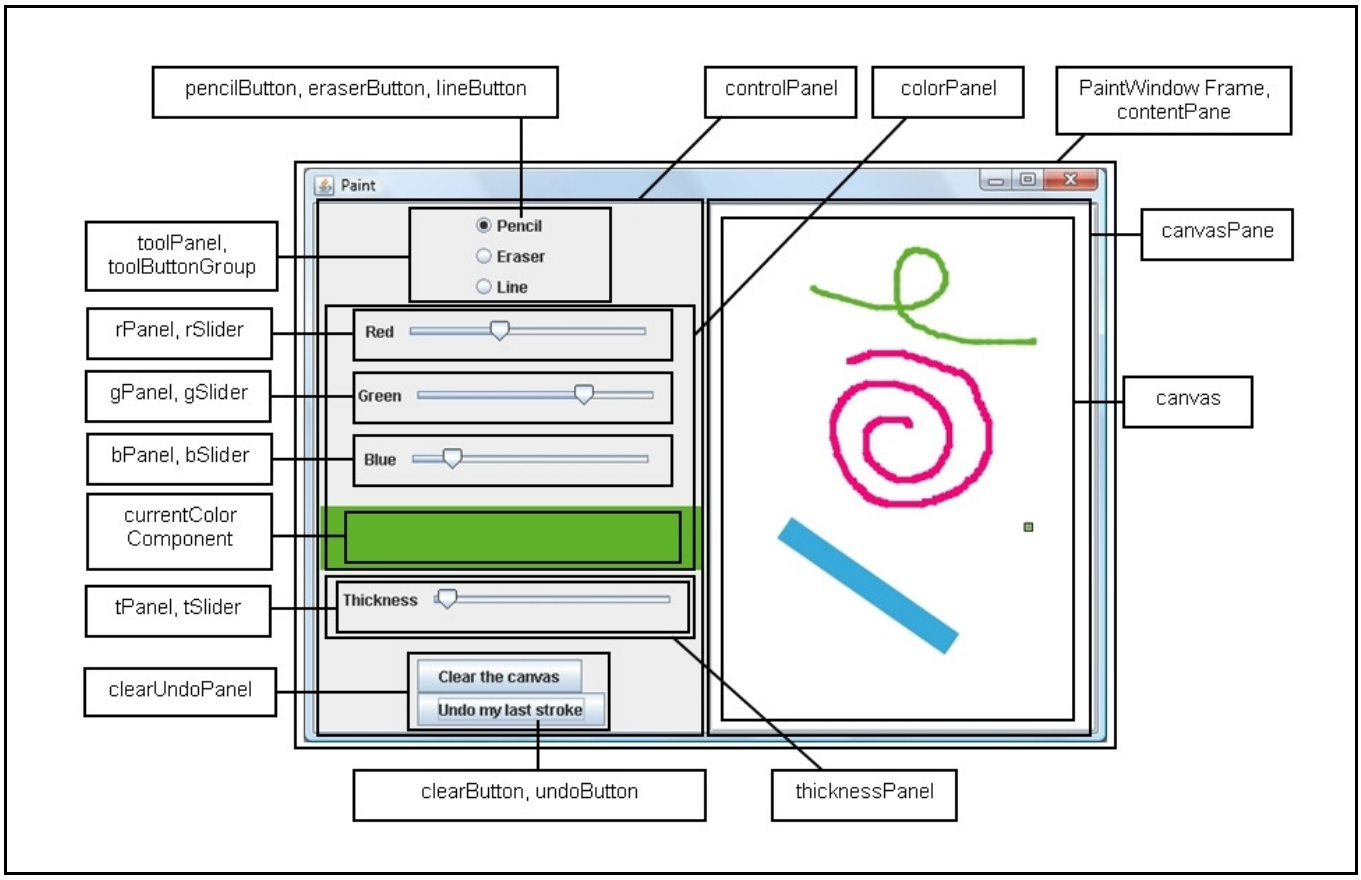

Figure 3. The modified version of the CMU Paint program that was used in our user study with widgets annotated by corresponding instance variables.

Of the fourteen subjects who participated in the user study, seven completed a certain task with our tool, while the other seven participants completed that same task without our tool present. (Note: Our subject sample size was fourteen, not seven. We paired similar tasks together rather than pairing subjects together for comparison to avoid introducing this variable into the study. The same subject's performance on one of the tasks in a pair with the tool present was compared with his own performance on the other task in the pair without the tool available, hence we did not need to take into account the subjects' expertise and experience when establishing the pairs).

1) Debugging Tasks: The Task 1 and Task 2 pair were debugging tasks to test Hypothesis 1 (listed in the beginning of Section III). The subjects were told that the single bug in the program for each task was in cloned (copied and pasted) code. We told the subjects this because otherwise they would have to examine the whole program to find the bugs, making it difficult, if not impossible, to measure their debugging performance. Odd numbered subjects performed Task 1 with CnP support and Task 2 without it, vice versa for even numbered subjects.

For the subjects who had CnP support for a task, the clone groups were already highlighted with different colors in the PaintWindow.java file. We explained to them that those colored code fragments were copied and pasted before. We specified the following five clone groups, which we consider as copy and pastes, in PaintWindow (see Figure 3):

Group 1. The red (r) slider/panel, the green (g) slider/panel, the blue (b) slider/panel, and the thickness (t) slider/panel.

Group 2. The color panel and the thickness panel.

Group 3. The tool panel and the clear/undo panel.

Group 4. The UI constraints for each panel: the tool panel, the color panel, the thickness panel, and the clear/undo panel.

Group 5. The declaration of the thickness change listener and the declaration of the color change listener.

We gave subjects who did not have CnP's clone visualization for a task the option to use the clone detection tool CCFinderX (version 10.2.7.1). Using a clone detection tool or manual searching are traditional ways to find the clones without CnP. We chose CCFinderX because its algorithm is token-based, which is known to have high recall. CCFinderX also has a graphical interface, is available for free download, and is easy to install and use. We pre-installed it, showed the subjects how to use it, and provided written instructions that they could refer back to.

The problem that we presented to the subjects in Task 1 was that "moving the blue slider does not change the pixel color". We ran the program for them to show them that this was the case, and showed them that the other color sliders

(red and green sliders) however did work correctly. The subjects were then told to find the bug in the source code and fix it so that the blue slider changes the color correctly.

The bug was that there was an instance of the identifier rSlider that appeared in the blue slider/panel clone that was supposed to be bSlider (on line 120, shown in Figure 4). The blue slider clone could have likely been copied and pasted from the existing red slider clone and then modified. Besides this bug in the program's functionality, there were no syntax errors reported by the compiler, since the slider and panel variables were all defined at the class level as instance variables. This kind of error can happen in practice and it would take extra time to search the code, fix it, and then test.

```
115   bPanel = new JPanel(new FlowLayout());
116   bPanel.setOpaque(false);
117   bPanel.add(new JLabel("Blue"));
118   bSlider = new JSlider(0, 255, 0); // values 0 thr
119   bSlider.setOpaque(false);
120   rSlider.addChangeListener(colorChangeListener);
121   bPanel.add(bSlider);
122
123   bPanel.setMinimumSize(new Dimension(100, 50));
124   bPanel.setPreferredSize(new Dimension(300, 50));
```

Figure 4. Task 1 – rSlider should be bSlider (on line 120).

Task 2's problem stated that "moving the thickness slider does not change the pixel thickness". We showed the subjects this behavior by running the program. We showed them that the color sliders all work in this task and told them to find the bug in the code and fix it so that the thickness slider changes the pixel thickness correctly when moved.

The correct solution to Task 2 was to change the identifier instance of colorChangeListener in the thickness slider clone to thicknessChangeListener (on line 142, shown in Figure 5). Eclipse showed warning icons that said that "the field PaintWindow.thicknessChangeListener is never read locally" at the listener's declaration, but it was not obvious that any subject had noticed this warning, since they instinctively went straight to the section of the file that contains the slider clones. The bug in this task could have happened when the tSlider clone was created by copying and pasting one of the color slider clones. In this scenario, the programmer may have focused on renaming the identifiers to tSlider and tPanel, overlooking the single change of renaming colorChangeListener to thicknessChangeListener.

```
137   tPanel = new JPanel(new FlowLayout());
138   tPanel.setOpaque(false);
139   tPanel.add(new JLabel("Thickness"));
140   tSlider = new JSlider(1, 50, 5); // values 1 thr
141   tSlider.setOpaque(false);
142   tSlider.addChangeListener(colorChangeListener);
143   tPanel.add(tSlider);
144
145   tPanel.setMinimumSize(new Dimension(100, 50));
146   tPanel.setPreferredSize(new Dimension(300, 50));
```

Figure 5. Task 2 – colorChangeListener should be thicknessChangeListener (on line 142).

2) Modification Tasks: The next pair of tasks (Task 3 and Task 4) that the subjects were asked to complete were modification tasks, which we created to test Hypothesis 2 (in the list at the beginning of Section III). For these tasks, the subjects were given a bug-free Paint program, where they were asked to add a specific feature to it. The subjects were told that the modifications would be made to cloned (copied and pasted) code. Odd numbered subjects completed Task 3 with CnP clone visualization present and Task 4 without. Even numbered subjects had CnP for Task 4 and not Task 3. Similar to the debugging tasks, we set up the CnP clone visualization in advance and allowed the subjects to use CCFinderX (but did not require them to use it) for the task that they did not have CnP present.

For Task 3, the subjects were asked to add a titled border with the label "Pixel Color" to the color panel (colorPanel) and to also add a titled border with the label "Pixel Thickness" to the thickness panel (thicknessPanel). We gave all subjects written hints about how to create a titled border and how to set a border to a panel, in case they were unfamiliar with the TitledBorder package.

Task 4 asked the subjects to add color to the label of each color slider such that the color of the red slider's label's text would be red, the color of the green slider's label's text would be green, and the color of the blue slider's label's text would be blue. We gave all subjects written hints about how to create the colors red, green, and blue and how to set the foreground color of a label, in case they were unfamiliar with the exact syntax.

3) Renaming Tasks (with CReN): Next, the subjects were given the first pair of renaming tasks (Task 5 and Task 6), which were to test Hypotheses 3 and 4 (in Section III of this paper) that relate to our CReN feature. For these tasks, we gave the subjects a version of the Paint program that had all of the correct source code except for requiring renaming within one or more pasted code fragments (which we copied and pasted beforehand). The subjects were told that they would need to perform renaming in copied and pasted code. CReN was used by odd numbered subjects in Task 5 and by even numbered subjects in Task 6. For the tasks that subjects could not use CReN (Task 6 for odd subjects, Task 5 for even subjects), they were allowed to use any built-in Eclipse tools, such as Rename Refactoring or Find & Replace, or they could always do the renaming manually.

We interrupted the subject at a random time during the renaming tasks to increase the odds of inconsistent renaming. We simulated a regular kind of interruption that someone might experience at work. For example, we asked the programmers to do some simple paperwork like to sign another informed consent form that they could keep for their records or to fill out a form with their personal information so that they could receive payment for participation in the user study. The interruption was done on all renaming tasks so that it would not be a variable in the experiment. We expect that many of the inconsistent renaming that is done in practice is due to interruptions, so having the interruption in

the tasks shows CnP's usefulness in preventing these errors. We did not include the interruption time as part of the task completion time, since it was not actual time spent working.

In Task 5, the subjects were told that the Paint program needs a thickness panel that contains the thickness slider and its panel, which is similar to the color panel that contains the red, green, and blue sliders and their panels. In this scenario, the color panel (colorPanel) has already been completed and the thickness panel (thicknessPanel) needs to be finished. As a last step to complete the thickness panel, the color panel was copied and pasted. For this task, the subjects had to rename all five instances of colorPanel to thicknessPanel and the one instance of rPanel to tPanel within the pasted code fragment only as shown in Figure 6 being done with CReN.

```
147
148     thicknessPanel = new JPanel();
149     thicknessPanel.setOpaque(false);
150     thicknessPanel.setLayout(new BoxLayout(thicknessPanel, Box
151     thicknessPanel.add(rPanel);
152
```

Figure 6. Task 5 – rename colorPanel to thicknessPanel.

Similarly, Task 6 explained that the Paint program had everything working, including the tool panel (toolPanel), but just needed the clear/undo panel (clearUndoPanel) to be completed (which is similar to the existing source code of the tool panel). The specific task required the subjects to rename all instances of toolPanel to clearUndoPanel, pencilButton to clearButton, and eraserButton to undoButton in the pasted code. Figure 7 shows how CReN would rename all six instances of toolPanel in this clone to clearUndoPanel when the programmer edits one of the instances.

```
158
159     clearUndoPanel = new JPanel();
160     clearUndoPanel.setOpaque(false);
161     clearUndoPanel.setLayout(new BoxLayout(clearUndoPanel, B
162     clearUndoPanel.add(pencilButton);
163     clearUndoPanel.add(eraserButton);
164
```

Figure 7. Task 6 – rename toolPanel to clearUndoPanel.

4) Renaming Tasks (with LexId): The last pair of renaming tasks that the subjects completed (Task 7 and Task 8) tested our Hypotheses 5 and 6 (see Section III), which are related to LexId. Like in the CReN tasks, the subjects received the Paint program with the copy and pastes already made in it and they were just required to perform the actual renaming of the identifiers or substrings only. Odd numbered subjects performed Task 7 with LexId (Task 8 without it) and even numbered subjects performed Task 8 with LexId (Task 7 without it). Subjects were given the same options for renaming when without LexId support (Eclipse renaming tools or manual edits) and the subjects were also interrupted during renaming. (In retrospect, it may have been more useful to not interrupt here, since we could then conclude that LexId is effective even when people are uninterrupted).

Task 7 involved the scenario where the Paint program had all source code including the code for the red slider (rSlider) and its panel (rPanel), but still needed the green and blue sliders (gSlider and bSlider) and their panels (gPanel and bPanel). As the final step in programming, the code for the red slider and its panel was copied and pasted twice and the labels and comments modified in advance (so that the subjects only had to focus on the actual renaming of identifiers and not other details). The subjects were told to rename rPanel to gPanel and rSlider to gSlider in the green slider clone (shown in Figure 8), and rPanel to bPanel and rSlider to bSlider in the blue slider clone. LexId would rename these twenty identifier substrings very quickly with one edit to each clone (the r to g in the green slider clone, and the r to b in the blue slider clone). This task is particularly good for LexId rather than CReN as CReN would need to do two edits for each clone, once for each of the different slider and panel identifiers.

```
103
104     gPanel = new JPanel(new FlowLayout());
105     gPanel.setOpaque(false);
106     gPanel.add(new JLabel("Green"));
107     gSlider = new JSlider(0, 255, 0); // values 0 thr
108     gSlider.setOpaque(false);
109     gSlider.addChangeListener(colorChangeListener);
110     gPanel.add(gSlider);
111
112     gPanel.setMinimumSize(new Dimension(100, 50));
113     gPanel.setPreferredSize(new Dimension(300, 50));
114
```

Figure 8. Task 7 (Part 1) – rename rPanel to gPanel and rSlider to gSlider in the green slider clone.

For the last task of the user study (Task 8), the Paint program contained all of the color sliders' source code, but needed the thickness slider and its panel to be finished. We copied and pasted the source code of the blue slider and its panel to use as a base for the thickness slider clone. We then asked the subject to rename bPanel to tPanel and bSlider to tSlider in the pasted clone only. Figure 9 shows the ten common substring instances being renamed together with LexId. All labels, number values, comments, and the listener were updated in advance so that the subject did not have to worry about those kinds of changes, but only had to do the identifier substring renaming.

```
136     // THICKNESS slider
137     tPanel = new JPanel(new FlowLayout());
138     tPanel.setOpaque(false);
139     tPanel.add(new JLabel("Thickness"));
140     tSlider = new JSlider(1, 50, 5); // values 1 thro
141     tSlider.setOpaque(false);
142     tSlider.addChangeListener(thicknessChangeListener
143     tPanel.add(tSlider);
144
145     tPanel.setMinimumSize(new Dimension(100, 50));
146     tPanel.setPreferredSize(new Dimension(300, 50));
147
```

Figure 9. Task 8 – rename bPanel to tPanel and bSlider to tSlider in the thickness slider clone.

IV. USER STUDY RESULTS

From the user study, we gathered data for the amount of time that it took for the fourteen subjects to complete each of the eight tasks (Section IV.A – Time per Task), which we had recorded with the Morae software. Besides speed, we also looked at whether the subjects had a correct solution (Section IV.B – Solution Correctness). And, when reviewing the video recordings, we noticed some interesting differences in how each subject solved the tasks (Section IV.C – Method of Completion), which helped us learn about common programming practices and how they may have contributed to subjects' mistakes. The insignificant results that we found for CnP clone visualization (Tasks 1-4), in terms of task completion speed (Section IV.A), motivated us to examine more carefully how subjects performed the debugging and modification tasks (Sections IV.B and IV.C).

A. Time per Task

In order to test our hypotheses (listed in Section III), we needed to record the amount of time it took for subjects to complete the tasks (which includes code compilation, testing, and debugging time) until the point when the subject said aloud that he was finished with the task. We specifically compare the experiment group (with our tool) versus the control group (without our tool), shown in Table II. We paired each individual subject's time data together over two tasks (in a single task category). For example, we paired together Subject 1's data – Subject 1 would have one task with CnP and the other without CnP – for the debugging tasks (Tasks 1 & 2), and so on. As mentioned earlier, we made each pair of tasks within a task category similar to each other such that this would also not be a variable in the experiment. Pairing a particular subject's "with CnP" data with the same subject's "without CnP" data eliminates the variable between different people. This leaves the presence or non-presence of the software tool as the only variable that we should observe in the experiment.

TABLE II. COMPLETION TIME (MINUTES) FOR THE TASK PAIRS

		Time per Task				
		Modified Mean (minutes)	# of data points < the mean	# of data points >= to the mean	# of outlier pairs	# of erroneous pairs
Tasks 1 & 2	With	1.30	6	6	2	0
	Without	1.20	9	3		
Tasks 3 & 4	With	3.19	7	4	2	1
	Without	3.24	6	5		
Tasks 5 & 6	With	0.64	5	6	0	3
	Without	1.04	5	6		
Tasks 7 & 8	With	0.44	9	4	1	0
	Without	1.02	8	5		

Table II shows the average (mean) time in minutes that subjects took to complete the pair of tasks, with or without CnP, CReN, or LexId. This is a "modified mean", since it does not contain the pairs of outliers or erroneous data pairs in its calculation, as we eliminated these subjects' specific data pairs from all analysis. We determined an outlier to be any significantly different time compared to the rest of the subjects' completion times for the same task, as determined by a boxplot in MiniTab (Student Release 12). Erroneous data was determined when there was a tool error (bug) or a user error (from a misunderstanding of the task or the tool, or an observed lack of seriousness while completing a task).

In order to statistically analyze our time data, we first performed a Chi-Square Goodness of Fit Test in MiniTab (version 15) to determine the nature of the data. This statistical test required the data to be in frequencies, so we divided the data as data points less than the mean and data points greater than or equal to the mean (two columns in Table II). With such small sample sizes, we were unable to conclude for sure that the data fit a normal distribution.

According to [28], a user study with a design of one factor (in our case, each task category), two treatments (with and without our tool), and a paired comparison can be analyzed with the Wilcoxon non-parametric test. We performed the Wilcoxon Signed Rank Sum Test as our statistical method of choice and since our number of observations/pairs was large enough, we also used a normal approximation, in Table III. The Wilcoxon test focuses on the median difference, with the null hypothesis (H0) stating that the median difference is equal to zero and the alternative hypothesis (Ha) stating that the median difference is less than zero. In other words, for the alternative hypothesis, we are looking at whether subjects "with CnP" take a significantly lesser amount of time to complete the tasks than "without CnP". The results showed that in all of the renaming tasks (both CReN and LexId), subjects completed the tasks quicker (in less time) with the tool than without the tool (p value of 0.0017 for CReN tasks and p value of 0.0139 for LexId tasks), just as we expected. We could not come to that same conclusion for the debugging and modification tasks (p value of 0.3632 and p value of 0.4801).

TABLE III. HYPOTHESIS TESTING ON THE PAIRED TIME DATA

	Wilcoxon Signed Rank Sum Test Normal Approximation alpha of .05, one-tailed test				
	Mean	Std Dev	z	P value	Reject H0?
Debugging Tasks (Tasks 1 & 2)	39	12.74	-0.35	0.3632	No
Modification Tasks (Tasks 3 & 4)	27.5	9.81	-0.05	0.4801	No
CReN Tasks (Tasks 5 & 6)	33	11.25	-2.93	0.0017	Yes
LexId Tasks (Tasks 7 & 8)	45.5	14.31	-2.20	0.0139	Yes

B. Solution Correctness

We determined whether subjects had correct solutions to the tasks because this is part of our hypotheses (Section III). We wanted to test whether the features of CnP help prevent inconsistencies/errors that can happen without them present.

In Table IV, we show the number of subjects who had no errors at the time the program was run and when they announced that they were finished with the task. For example, for Task 1, it shows that twelve subjects (out of the twelve who ran the program) had no errors when they ran the

program for the first time, that no subjects ran the program more than once for Task 1, and two people did not run the program at all (but both had a correct solution when finished). Subjects who did not run the program may have been very confident in their solution's correctness.

We saw cases where the programmers made copy-paste-modify errors that they caught before they were finished and other times when they did not notice the mistakes at all (Table IV). All subjects who finished the debugging tasks (Tasks 1 & 2) had correctly spotted the bugs in the code. The first modification task (Task 3) had the most people without correct solutions. Four subjects with CnP visualization support and one subject without it made the same mistake of adding the titled border around tPanel instead of thicknessPanel (thicknessPanel contains the single panel tPanel, Figure 3). For Task 4, many subjects ran the program multiple times to check the correctness of partial solutions.

Over all of the twenty-eight renaming cases (Table IV, Tasks 5-8), there were fourteen incorrect states of the program (either when running the program or when finished). Thirteen of these were due to renaming mistakes made by subjects when CReN and LexId were not used (sometimes because the interruption would make the subjects forget what they were working on). These errors could have been prevented with CReN or LexId. The other incorrect state was due to an engineering oversight. Specifically, CReN undid a group of consistently renamed identifiers one-by-one after a subject did a series of "undo" operations in Eclipse during Task 6. In the future, we can modify CReN to undo all previous renaming together as a single transaction rather than one at a time. Since CReN is still a proof-of-concept, we believe that this oversight is acceptable.

TABLE IV. CORRECT STATES ON PROGRAM RUN OR WHEN FINISHED

	# correct on 1st run of program	# correct on 2nd run of program	# correct on 3rd run of program	# correct on 4th run of program	# correct on 5th run of program	# correct when finished with task
Task 1	12/12	N/A	N/A	N/A	N/A	14/14
Task 2	12/12	N/A	N/A	N/A	N/A	13/14
Task 3	8/14	3/3	N/A	N/A	N/A	8/14
Task 4	6/14	6/8	3/4	2/2	1/1	12/14
Task 5	9/10	1/1	N/A	N/A	N/A	12/14
Task 6	9/11	2/2	N/A	N/A	N/A	13/14
Task 7	8/11	1/1	N/A	N/A	N/A	12/14
Task 8	7/8	N/A	N/A	N/A	N/A	12/14

C. *Method of Completion*

Although we did not hypothesize about how the subjects would complete each task, we found interesting differences between subjects' methods of completion that may have had some effect on their time per task or solution correctness.

For the debugging (Tasks 1 & 2) and modification (Tasks 3 & 4) tasks, we found that the subjects used different methods to locate the source code region that the task applied to and then different ways to inspect the area for bugs or inconsistencies. Table V shows that most subjects manually scanned the source code file, scrolling to find their region of interest (eleven or twelve out of fourteen, shown in column 2), but a few (two or three, shown in column 1) used the Find part of the Find & Replace tool or the Search References tool in Eclipse to jump to a specific location in the file. For the debugging cases, the subjects often went directly to the areas of the source code that had to do with either the change listeners' declaration or its use, indicating that these subjects probably had prior experience with such code. In the first task, half of the people compared the broken blue slider code with the red and green slider clones that they knew worked. For the first modification task (Task 3), five subjects who had an incorrect final solution all appeared to "inspect the problem region only" as they did not notice that colorPanel and thicknessPanel were clones (not colorPanel and tPanel).

TABLE V. NUMBER OF EACH LOCATION AND INSPECTION METHOD USED FOR DEBUGGING AND MODIFICATION TASKS

	Location		Inspection	
	Use Find / Search Tools	Manual Scan	Compare Clones	Inspect Problem Region Only
Task 1	3	11	7	7
Task 2	3	11	3	11
Task 3	3	11	9	5
Task 4	2	12	13	1

For the renaming tasks (Tasks 5-8), it was interesting to see which methods the subjects used when not using CReN or LexId for renaming. Table VI shows that subjects used Find & Replace, copying and pasting, or manual typing when without CnP. Some subjects used the default settings of Find & Replace, which finds and replaces one occurrence at a time. Subjects who may have wanted to be more efficient chose to configure Find & Replace to only rename within selected lines (the code fragment of interest). Some subjects copied and pasted substrings, but the more efficient programmers often chose to copy and paste whole identifier names (even when a substring was the only change to make) because it is quicker to select whole identifiers in Eclipse rather than to select part of an identifier with the mouse. Sometimes subjects started using one method, but switched to another during the same task (shown in the table with .5 increments). One subject even tried Rename Refactoring (not shown in the table) to do the renaming for Task 5, but it was unsuccessful, since refactoring renamed the declaration and instances within the copied code fragment, too. Still some programmers always chose to just type really quickly (manually) rather than copy and paste or use any other tool.

TABLE VI. NUMBER OF TIMES EACH RENAMING METHOD WAS USED

	Renaming without CReN or LexId			
	Find & Replace (one at a time)	Find & Replace (selected lines)	Copy & Paste / Manual (identifier)	Copy & Paste / Manual (substring)
Task 5	2	2	2	1
Task 6	1	0	3	3
Task 7	2	3	1	1
Task 8	0	0	0.5	6.5

V. Discussion and Threats to Validity

Based on the outcomes of this user study experiment, we have noticed some areas where we could refine the experiment design and improve our tool design. In particular, we analyzed factors that may have masked the effect of clone visualization during debugging and modification tasks.

A. Confounding Factors for Clone Visualization

Based on our current data from the user study, we were unable to confirm our original hypotheses about CnP's clone visualization feature, but we did see the usefulness of clone visualization. For example, if subjects had made use of cloning information in Task 3, they would have produced correct solutions (modifying the thicknessPanel clone rather than the tPanel clone). We realize that even when clone visualization was provided for the debugging and modification tasks, it was not forced on the user, so the programmer may not have used it (unlike CReN and LexId, which had to be used when present). Although it was difficult to tell for sure whether the subjects actually used the visualization, we found that some subjects seemed to compare clones (Table V) and that cloning information can help avoid incorrect solutions when used.

Subjects' experience levels can have an effect on their debugging strategies. For the debugging tasks, some programmers appeared to use a typographic debugging strategy (where they examined code line by line), while others used a more symptom-driven approach (where they went directly to the problem region) [7, 21, 23, 24, 26]. The subject's choice of debugging method seemed to have an effect on whether he compared clones. If the subject knew the code pattern due to prior experience, he would know which code belongs to which behavior and he would quickly go to its location in the file without using the clone visualization. Visualization would be more helpful to programmers without as much prior knowledge.

It was also observed that the subjects who were more familiar with the Eclipse IDE itself had used it to their advantage to gain efficiency (using Eclipse's occurrence marking and code completion features). And, sometimes subjects did not choose to run the program if they were more confident in their solution. While this would save time, it could leave an undetected error, as it did for some subjects.

According to the questionnaire, no subjects normally use clone detection tools to debug, and the one subject that used CCFinderX during the user study reached the time limit for the debugging task (his completion time was removed from analysis as shown in Table II, since it was too long compared to the others). The subjects said that the clone detection tool had too steep of a learning curve to use, and that they normally debug Java source code by mentally tracing through it, using print statements, or using the built-in Eclipse debugger if necessary. The one subject who used the Eclipse debugger during the user study also took a much longer time to finish the task than others.

B. Threats to Validity

As said in Section III.A, all subjects were required to have some familiarity with Java and Swing in order to participate in the study. However, it was clear that some subjects who performed very well may have had more experience or prior knowledge than others. This substantial distribution of programming skills is a universal research issue not unique to us. While we had both graduate and senior undergraduate students as subjects, four of them had non-trivial industrial experience. Thus, while our sample of subjects may not represent the whole programmer population, they should at least be a good representation of the entry-to-medium level subset. Furthermore, subjects with industrial experience also made mistakes when performing tasks without the tool. Although we cannot generalize our claims to all programmers, our study demonstrates the tool's usefulness in clone-related maintenance tasks.

We do not believe that the order of tool support made a difference is subjects' task completion (for example, having the CnP tool present for the first task in a pair did not necessarily benefit the subject compared to a subject who completed the tasks without the tool first). We introduced the CnP tools to all subjects at the beginning of the user study, so everyone had the same level of prior knowledge about the tools, including their potential benefits and usage scenarios.

While the user study's programming tasks may not represent all possible scenarios of cloning, they were fairly close to real-world tasks in GUI software development and maintenance. (We deliberately chose relatively simple debugging and modification tasks in order to measure completion time). CnP's features can be applied throughout source code evolution, making it a long-term benefit.

C. Tool Design

The feedback from the subjects about the tool features was overwhelmingly positive. All subjects said that they would use the tool to help them prevent copy-and-paste-related errors. Their main suggestions for changes to the tool are: (1) to have the visualization optional (able to be turned off), while continuing to track clones in the background, and (2) to know exactly what was renamed by CReN and LexId.

We chose colored bars for visualization because it is a simple way to highlight a clone region in the files that the programmers are already working with. We are looking into improving our current visualization so that many (sometimes overlapping) clones' bars do not start to clutter the source code files. One suggestion that a subject gave is to only show the clones that are at the mouse cursor's current position. We are also considering users' suggestions to be able to disable the clone visualization until it is needed. Some users said that they prefer not to have so many different colors in their editor, while one subject said that he could not differentiate between the colored bars, since he is color blind.

It was evident from the user study that subjects often overestimated what was renamed. At least two subjects did not know whether CReN renamed the identifier's declaration also (which was outside of the clone) like refactoring would. We also observed that when CReN automatically renamed a lot of instances of the currently edited identifier, subjects sometimes prematurely thought that they were done renaming everything, but there were still other identifiers left

to be renamed. We could improve the tools to display an optional pop-up window that tells exactly what was renamed.

VI. Related Work

In addition to the related work cited elsewhere in this paper, we briefly mention related work in the areas of clone visualization and identifier renaming, and leave additional detail, especially in renaming, in our prior work [8, 9, 10].

Tools such as [4, 6, 27] use colored bars, markers, or rulers to show clones like CnP does. Many other methods of clone visualization, as described in [3], use separate views that programmers need to invoke or relatively complex graphs that they need to learn and understand.

Eclipse contains a feature called Rename Type Refactoring that allows similarly named variables and methods within a class to be updated when the class name is renamed. For example, when a class "Bar" is renamed to class "Foo", the "fBar" variable and "createBar" method (both of type "Bar") within the class are renamed to "fFoo" and "createFoo" if the programmer checks this option when invoking refactoring in Eclipse. LexId can do this same thing, but does not limit itself to the class scope.

When applied to clones, CReN [9] and LexId [10] track the relationships of the identifier instances between the related clones and the relationships between the identifier instances of the same program element or name within each clone. A related tool, VACI, detects contexts and identifiers, and then forms translation classes that can be used to detect an inconsistency in naming [13]. While VACI can detect possible renaming inconsistencies, CReN and LexId instead proactively prevent the inconsistencies from occurring at all.

VII. Conclusion

We presented a user study that tested our CnP tool's visualization and renaming features. The user study confirmed that our renaming tools (CReN and LexId) both perform statistically quicker than without them on similar tasks and help prevent inconsistent renaming that otherwise happens. Although the study results did not allow us to directly conclude statistically that CnP's clone visualization improves a programmer's performance on debugging and modification tasks, by analyzing the experiment data and considering the debugging and modification micro-processes, we revealed relevant factors that have masked the effect of clone visualization on such maintenance tasks. This analysis helps us better determine whether and when a programmer may exploit clone information. We are not aware of any other similar analysis of the role of clone information in maintenance tasks, and, thus believe that the analysis in and of itself can be a contribution. Our analysis can be used in the design of future experiments.

References

[1] B.S. Baker, "Parameterized duplication in strings: algorithms and an application to software maintenance", *SIAM Journal on Computing*, 1997, pp. 1343-1362.

[2] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone smells in software evolution", *ICSM*, 2007, pp. 24-33.

[3] A. Chiu and D. Hirtle, "Beyond clone detection", *University of Waterloo*, Course Project, 2007, pp. 1-21.

[4] M. de Wit, A. Zaidman, and A. van Deursen, "Managing code clones using dynamic change tracking and resolution", *ICSM*, 2009, pp. 169-178.

[5] F. Detienne, "Reasoning from a schema and from an analog in software code reuse", *ESP*, 1991, pp. 5-22.

[6] E. Duala-Ekoko and M.P. Robillard, "Tracking code clones in evolving software", *ICSE*, 2007, pp. 158-167.

[7] M. Ducasse and A. Emde, "A review of automated debugging systems: knowledge, strategies and techniques", *ICSE*, 1988, pp. 162-171.

[8] D. Hou, F. Jacob, and P. Jablonski, "Exploring the design space of proactive tool support for copy-and-paste programming", *CASCON*, 2009, pp. 188-202.

[9] P. Jablonski and D. Hou, "CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE", *ETX*, 2007, pp. 16-20.

[10] P. Jablonski and D. Hou, "Renaming parts of identifiers consistently within code clones", *ICPC*, 2010, in press.

[11] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs", *ESEC and FSE*, 2007, pp. 55-64.

[12] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?", *ICSE*, 2009, pp. 485-495.

[13] T. Kamiya, "Variation analysis of context-sharing identifiers with code clones", *ICSM*, 2008, pp. 464-465.

[14] R. Kerr and W. Stuerzlinger, "Context-sensitive cut, copy, and paste", *C3S2E*, 2008, pp. 159-166.

[15] M. Kim, L. Bergman, T.A. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL", *ISESE*, 2004, pp. 83-92.

[16] A.J. Ko, H.H. Aung, and B.A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks", *ICSE*, 2005, pp. 126-135.

[17] K. Kontogiannis, "Managing known clones: issues and open questions", *Dagstuhl Seminar*, 2006, pp. 1-5.

[18] R. Koschke, "Frontiers of software clone management", *FoSM*, 2008, pp. 119-128.

[19] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: a tool for finding copy-paste and related bugs in operating system code", *OSDI*, 2004, pp. 289-302.

[20] B. Liblit, A. Aiken, A.X. Zheng, and M.I. Jordan, "Bug isolation via remote program sampling", *PLDI*, 2003, pp. 141-154.

[21] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas and C. Zander, "Debugging: a review of the literature from an educational perspective", *Computer Science Education*, Taylor & Francis, 2008, pp. 67-92.

[22] R.C. Miller and B.A. Myers, "Interactive simultaneous editing of multiple text regions", *USENIX Annual Technical Conference*, 2001, pp. 161-174.

[23] J. Rasmussen, "Models of mental strategies in process plant diagnosis", Human Detection and Diagnosis of System Failures, *Plenum Press*, 1981, pp. 241-258.

[24] J. Reason, Human Error, *Cambridge University Press*, 1990.

[25] M. Toomim, A. Begel, and S.L. Graham, "Managing duplicated code with linked editing", *VLHCC*, 2004, pp. 173-180.

[26] I. Vessey, "Expertise in debugging computer programs: a process analysis", *IJMMS*, Academic Press, 1985, pp. 459-494.

[27] V. Weckerle, "CPC: an eclipse framework for automated clone life cycle tracking and update anomaly detection", *Free University of Berlin*, Master's Thesis, 2008.

[28] C. Wohlin, P. Runeson, M. Hoest, M.C. Ohlsson, B. Regnell, and A. Wesslen, Experimentation in Software Engineering: An Introduction, *Kluwer Academic Publishers*, 2000.