Effect of clone information on the performance of developers fixing cloned bugs

Saman Bazrafshan Rainer Koschke
University of Bremen, Germany
{saman.bazrafshan, koschke}@informatik.uni-bremen.de

Abstract—Duplicated source code—clones—is known to occur frequently in software systems and bears the risk of inconsistent updates of the code. The impact of clones has been investigated mostly by retrospective analysis of software systems. Only little effort has been spent to investigate human interaction when dealing with clones. A previous study by Chatterji and colleagues found that cloned defects are removed significantly more accurately when clone information is provided to the programmers. We conducted a controlled experiment to extend the previous study on the use of clone information by investigating the effect of clone information on the performance of developers in common bug-fixing tasks. The experiment shows that developers are quite capable to compensate missing clone information through testing to provide correct solutions. Clone information does help to detect cloned defects faster, although developers may exploit semantic code relations such as inheritance to uncover cloned defects only slightly slower if they do not have clone information. If cloned defects lurk in semantically unrelated places however, clone information helps to find them faster at statistical significance. Developers without clone information needed 17 minutes longer on average or 140% more time in relative terms to complete the task successfully.

I. INTRODUCTION

Copy&paste programming is a frequent practice in software development. While this strategy of developing cases and accelerates the development of software systems in the short term, it creates code redundancy and may lead to problems in the mid and long term. Even though there are some studies showing that clones may have a negative effect on maintainability in certain cases [1], [2], [3], [4], overall there is still not sufficient evidence that cloned code causes indeed more maintenance effort than non-cloned code in general. The majority of empirical studies on the effect of duplicated code is commonly based on retrospective analysis using source code repositories or defect tracker databases. Although such data may provide valuable insights towards the relation of clones and maintainability, other important factors are left out since retrospective analysis considers only the result of a possibly more complex sequence of actions. For instance, a bug fix with preceding tedious localization effort may appear as a simple and small change in the source code. To observe and investigate the benefits of clone-management tools and human interaction in dealing with clones within maintenance activities such as defect reproduction, program comprehension, code changes, and testing, other approaches are necessary [5].

Controlled experiments are appropriate to bridge the gap and help to investigate such aspects in a controllable manner.

Human-based studies on the effect of clones in defect localizing and repairing tasks are very rare in clone research [6], [7], [8]. None of these studies has sufficiently evaluated the support clone-management tools can offer in such tasks yet (a more detailed review of these works follows in Section II). Either the developers were not supported by any kind of clone management tool in these studies or the level of tool support was minimal, that is, only a simple clone report as a textual list of clones without any further integration in an integrated development environment (IDE) was provided. So these studies did not investigate the full potential clone management tools can offer.

We conducted a controlled experiment that evaluates how a programmer’s performance—in terms of time and correctness—is affected when clone information is provided while dealing with a common bug-fixing task involving a cloned defect. To overcome the shortcomings of pure textual information, we offer a more realistic tool support in our experiment that provides contextual clone information integrated in the IDE (i.e., only the clones of the currently inspected file are shown) and enables the user to easily browse through existing clones and to identify the location of each clone fragment. This kind of tool support is more realistic for assessing the effect of clone management tools that try to compensate for risks imposed by clones and leverages the user’s experience with his or her familiar IDE.

Plenty of clone management tools have been introduced to support developers to detect clones [9], [10], support refactorings [11], [12] and change propagation [13], [14] as well as monitoring to prevent unwanted inconsistencies [15]. Still, clone management has not yet become an integral part of the daily work of programmers. In our study, we analyze whether clone information provided in an easy-to-use interface affects the performance of programmers regarding common bug-fixing tasks in a positive way. For that matter, we prepared the tasks such that the defective code segment is cloned and, hence, occurs twice in the source code of the software system.

Contribution. In this paper, we present our controlled experiment that aims to investigate the effect of clone-management toolsing on software maintainability—more precisely on bug-fixing. Compared to the wide field of empirical research based on historical data, controlled experiments are rarely applied in the domain of software clones. Specifically, we investigate how the removal of a cloned defect is affected by the fact that clone information is provided in the IDE and...
how missing clone information is compensated by developers. We are answering the following two research questions:

**Question 1** — Does the time needed to remove a cloned defect decrease when clone information is available?

It may seem obvious that it may take less time to fix a defect that is cloned when clone information is accessible. Yet, this clone information must be interpreted, filtered, and inspected by a developer, which causes some overhead. By answering this question, we aim to provide evidence as to whether it takes more time to fix a cloned defect without any clone information. Moreover, if a difference exist, we investigate the extent of the difference. The second question we answer is as follows:

**Question 2** — Does the probability of incorrect or incomplete removals of cloned defects decrease when clone information is available?

Question 2 considers the correctness of a defect removal. A commonly stated threat is that clones may lead to inconsistent or incomplete changes. By answering this question, we aim to provide empirical evidence as to whether the risk of inconsistent changes is reduced by providing clone information.

In addition to our research questions, we report and discuss insights gathered from our experiment, which can be useful to improve further research on code clones based on experiments.

**Outline.** The remainder of this paper is organized based on the guidelines from Jedlitschka and Pfahl [16]: Section II presents related work. The experiment design is described in Section III. Section IV documents the execution before Section V summarizes the results statistically and tests our hypotheses. The results are discussed in Section VI and the threats to validity in Section VII. Finally, section VIII concludes.

**II. RELATED WORK**

Software clones have been subject to recent studies and are still an active field of research. A vast number of various approaches to detect and manage clones have been introduced—an overview is given in the surveys by Koschke [9] and Roy and colleagues [10], [17] and most recently by Roy, Zibran, and Koschke [18]. This section summarizes previous work that is most closely related to ours. We focus on human-based studies. Due to lack of space, we refer the reader regarding other studies about the relation clones and maintainability to the above surveys which summarize these other studies.

Carver and colleagues [5] consider human-based studies an important but yet largely neglected approach to complement findings that are based on analytical studies only. They emphasize that speculating about human behavior using analytical data is not adequate enough. Instead, claims need to be validated more thoroughly focusing on human-based studies.

In an ethnographic study Kim and colleagues [19] analyzed *Copy&Paste* programming practices of developers. Observing programmers using an instrumented Eclipse IDE, a taxonomy of *Copy&Paste* usage patterns has been derived. They found that developers often recall those clones they have created themselves, but that this knowledge is difficult to transfer to other developers. Based on their findings, they propose a set of tools to reduce maintenance problems incurred by *Copy&Paste* programming and to better support the intents of commonly used *Copy&Paste* scenarios.

In another observational study, Chatterji and colleagues [6] addressed the questions on whether and how developers use clone information in bug localization tasks and whether there is any difference between novice and professional developers in this regard. They observed developers in two bug localization tasks in systems they were unfamiliar with. One bug was contained in cloned code, the other one was not cloned (intended as a placebo). The participants were not required to actually fix the defect. The participants received a clone report created by CCFinder [20]. The clone report was available only in textual form. No further clone-management support was given and no integration of the reported data in any type of IDE was available. Most of their participants did not use the clone report correctly—possibly because of insufficient training or missing tool support. Their study indicates that there is a relationship between correct use of the clone report and effectiveness (i.e., those who used the clone report correctly were more effective in detecting defects), as well as that clone-detection ability is heavily reliant on experience, as professionals tended to use the clone report correctly more often than novices, although the design of the study does not truly allow to establish causal relationships. They conjecture that the clone report might not necessarily help developers locate the initial defect in a large software system, but it will help them locate clones of that defect. Our experiment is designed to validate this hypothesis.

The first controlled experiment to evaluate how a programmer’s performance is affected by clones in specific maintenance tasks has been conducted by Harder and Tiarks [7]. In a controlled experiment with a total number of 33 participants—including students as well as experts in the field of software clones—they analyzed how the removal of a defect is influenced by the fact that it is cloned or not. No clone information, clone tools or other hints related to clones have been given to the subjects. Based on the results they used statistical significance tests to investigate the performance in terms of time and correctness. The results of the statistical tests did not indicate a significant difference in the time needed to correct cloned defects or the correctness of the solutions. Nonetheless, they observed many cases in which cloned defects were repaired incompletely. That is, often only one occurrence of the cloned defect was located and fixed. In contrast to our study, no clone-specific information has been provided to the subjects in the experiment by Harder and Tiarks. That is, the developers were not supported by any kind of clone management tool.

Chatterji and colleagues [8] replicated the controlled experiment by Harder and Tiarks with a different sample of participants. This replication neither yielded any significant difference in the time needed. In contrast to the original study, however, the replication showed that it was significantly more difficult to correctly fix a cloned bug than a non-cloned one. Moreover, they extended the previous study by an additional
task for which the programmers were provided with a clone report in textual form and a training on how to use it. The results of this extension indicate that developers perform significantly better when a clone report of the software system is available than without any clone information.

We think that some design decisions of Chatterji and colleagues regarding the extension of their study should be reconsidered to allow for a more accurate inspection of the effect of clone information on software maintenance. First, they used a textual clone report as in their previous study. To investigate the effect of clone information on the performance of programmers, a more realistic scenario in terms of daily work and development environment is necessary. Therefore, we provide clone information and clone browsing using a simple tool integrated in an existing and well-known IDE. Besides the missing tool support, the participants were most probably biased by a learning effect because the same subjects and software systems were used for the replication of the study by Harder and Tiarks and the extension with all participants required to solve the tasks related to the replication first. Afterwards, they were already familiar with the source code as well as the defective features for both systems because the defects changed only in the fact as to whether they occurred in form of a clone or as a non-cloned abstraction. The learning effect of the subjects probably lead to a better performance in the tasks related to the study extension. Moreover, performance was analyzed for the extension in terms of correctness only. No data were collected that allowed for conclusions on the efficiency in terms of time. We recorded the time needed to solve the tasks in our experiment and analyzed whether the needed amount of time decreased using the clone-management tool. Finally, Chatterji and colleagues assigned the subjects to two groups such that each participant was required to solve a cloned defect in one of the systems and a non-cloned defect in the other one. To evaluate the effect of clone information on the performance of developers when fixing cloned bugs, it is more appropriate that each participant is required to fix a cloned bug in each system and to switch the order for which of the systems clone information is provided to the participants. We chose exactly this experiment design—each defect occurs twice in both subject systems. This enables us to compare the performance of the programmers repairing a cloned defect once with and once without the help of clone information.

III. EXPERIMENTAL DESIGN

This section describes the experimental design of our controlled experiment. Based on the hypotheses and variables as well as on the selection of appropriate subjects and objects we investigate our two research questions. Finally, we also describe the instrumentation of the experiment in this section.

A. Hypotheses and Variables

Our hypotheses are directly derived from our two research questions. The alternative hypotheses are both 1-tailed. Table I shows the relation of the hypotheses to our research questions. Since we want to investigate the impact of clone information on the performance of programmers solving bug-fixing tasks, the sole independent variable of our experiment is whether clone information is provided or not. The dependent variables are based on our research questions and used in similar experiments [7], [8]: (1) the time needed to complete a given task and (2) the correctness of the solution. The correctness of a solution is further distinguished in the following two factors:

a) Addressed. The performed actions corrected at least one occurrence of the defect.

b) Complete. The performed actions correct all existing defects that are part of the given task.

This distinction implies that every complete solution is also addressed, but only complete solutions are regarded as correct results. That is, fixing just one of two defects in cloned code—addressed solutions—does not suffice in terms of correctness. Nonetheless, we will provide and evaluate data of both addressed and complete solutions in the results section.

B. Design

For the execution of our experiment we use two small open-source Java games, namely, FrozenBubble\textsuperscript{1} and Pacman\textsuperscript{2}. Both have been used in previous similar experiments [7], [8]. Harder and Tiarks [7] developed a laboratory package including the games, defective versions of the source code with and without cloned bugs and corresponding bug reports. We used that package which is freely available\textsuperscript{3} and derived our maintenance tasks from it. For each game, we defined one maintenance task that requires the developers to fix a cloned defect. The tasks are denoted as \textit{tFB} for FrozenBubble and \textit{tPM} for Pacman in the following. For each of these tasks, we either provided information regarding existing clones in the systems or not. The different variants of provision of information constitute two levels of the sole independent variable in our experimental design, which is whether clone information is available for the programmers or not. Task \textit{tFB}_\textit{c} is the variant of FrozenBubble for which clone information is provided, while the variant without any clone information is denoted as \textit{tFB}_\textit{nc}. Likewise, we will refer to the task variants for Pacman as \textit{tPM}_\textit{c} and \textit{tPM}_\textit{nc}.

The subjects were separated into two groups A and B. The participants of each group have been either assigned to the task variant \textit{tFB}$_\textit{c}$ and \textit{tPM}$_\textit{c}$ or to the contrariwise variant \textit{tFB}$_\textit{nc}$ and \textit{tPM}$_\textit{nc}$. This division of subjects results in a $2 \times 2$ factorial design [21] and allows for comparison of the results. Table II shows the assignment of groups to the corresponding tasks.

C. Subjects

The participants of our experiment were all computer science students from the University of Bremen. We selected the participants based on two main requirements: First, basic Java knowledge was needed because both software systems under

\textsuperscript{1}http://www.frozen-bubble.org/

\textsuperscript{2}http://code.google.com/p/pacman-rkant/

\textsuperscript{3}http://www.softwareclones.org/experiment
study are written in Java. As a basic principle of the University of Bremen, all computer science students learn Java as primary programming language in their first two terms. Beyond that we did not ask students to participate who were in their early terms, that is, students who did not take part in any software engineering, software architecture or software quality courses yet. Finally, by asking the subjects to self-assess their skills on a scale from 1 to 100 with 100 being perfect, we did not find major differences regarding their skills. Second, participants needed basic knowledge in using the Eclipse IDE for Java development because we wanted everybody to use the same set-up during the execution. This makes it easier to prepare a standardized programming environment in which unexpected actions by the participants are minimized and relevant data can be reliably collected. In total, 29 students participated in the experiment of which all were randomly assigned to the groups A and B with no further mechanism of blocking.

### TABLE II

**Assignment of Groups to Tasks**

<table>
<thead>
<tr>
<th>Task 1</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 2</td>
<td>IFB</td>
<td>IFB</td>
</tr>
</tbody>
</table>

#### D. Objects

We already described that we prepared different task variations upon two small open-source Java games. By using existing software systems that have not been artificially created by us we wanted to provide a maintenance scenario that is as close and thoroughly as possible to real world problems. The given tasks demanded the participants not only to fix the defects but also perform various steps that usually belong to such a maintenance task, such as reading a defect report and getting familiar with related features and the corresponding source code, locating and correcting the defects by applying appropriate changes to the code and testing whether the applied changes cause the desired correcting effect. If not, correcting and testing are repeated until success is achieved.

By covering all necessary activities of a common bug-fixing approach it is possible to measure the time needed to complete a whole maintenance task, and moreover, to accurately verify the correctness of the solution. If we required the subjects to perform only certain activities, for instance, to locate the defects without fixing and testing, they would not be able to check whether they overlooked a defect or its clone. By asking the subjects to also fix and test their solutions we avoided this problem. Note, that in contrast to the studies of Harder and Tiarks and Chatterji and colleagues we excluded the time participants spent on reading the bug reports from the data evaluation. The reading speed of subjects is very different as far as we experienced and is not an activity that is directly related to locating or changing clones.

To emulate a whole maintenance task of a real world scenario as good as possible the selected objects needed to meet two essential requirements. As a result of the limited time of an experiment like this and the variables that are to be analyzed the selected software systems must not be too small and not too big at the same time and simple enough to be understood by the subjects—also allowing to reproduce the defects—but still be realistic.

Both systems FrozenBubble as well as Pacman meet the requirements regarding the size of the source code with 3,000 and 2,400 SLOC, respectively. In addition, the defect reports contain some hints that allows to limit the search space. The second constraint is approached by the fact that we used simple and intuitive games—that are also widely known—which allows to verify their correctness by playing them for just a few seconds. For that reason, the tasks were designed such that the defects cause wrong program behavior, but not an entire crash. We deliberately chose rather small and localized defects, because Harder and Tiarks [7] found that participant require too much time to solve more complex defect scenarios.

Another important requirement for the design of our experiment was that each task includes two defects that need to be fixed and that one defect is a clone of the other one. This scenario corresponds to a reactive clone management, that is, clones are handled only when they need to be modified. Proactive clone management, on the other hand, would try to eliminate clones when they are detected even if it is not clear whether they need to be modified in the future. Reactive clone management is a more realistic setting especially for large systems with many clones.

To provide the subjects visible symptoms for both defects of each task we designed the tasks such that each defect of one task has its own visual symptom it is causing. Only if both occurrences were fixed, all symptoms would disappear. This way, we enabled the participants to check whether they fixed all existing defects. If the subjects did not remove both symptoms, the solution is considered incomplete, hence, only if the defect and its clone for each task have been located and fixed, the outcome is considered a correct solution.

Both systems contained different clones already that could be used instead of introducing artificial clones to the existing code. That is, for both variants of the given tasks, code fragments that existed multiple times were used. We inserted the defects at appropriate clone fragments. In the following we describe the two tasks and the defects in more detail.

**Task IFB.** The point of the game FrozenBubble is to elim-
We inserted the same defect in these clone fragments by changing the \( i + 1 \) to \( i \) in lines 6–8. As a consequence, only the bubble is loaded whose index corresponds to the index 1—a gray colored bubble. Therefore, only gray bubbles appeared, thus, all bubbles at the top, the one on the launcher and the one from the preview. Since there is no distinction between bubbles and their visual representation, the defects cause that all bubbles are eliminated by the first shot, because a gray bubble is shot on a group of also gray bubbles at the top.

To emulate a real-world maintenance task we provided the following defect report to the subjects: There are eight differently colored bubbles in the game. When a level starts, only gray bubbles appear on top. The launcher at the bottom will also fire only gray bubbles. Obviously not all available bubbles are loaded. Since, Harder and Tiarks [7] found in their experiment (as well as in a pilot test to it) that subjects may spend a long time seeking the defect in wrong classes, the reports provided some additional hints where not to look. For instance, the following hint was used similarly for both tasks: The bug is located in the default package. You do not have to modify the packages lib, manager, or screens. We intended to exclude libraries and packages drawing the game scenes from possible defect locations. This information has been given because of the time constraints in such an experiment.

**Task tPM.** Pacman is a very well known arcade game in which the player navigates the main character Pacman through a right-angled maze. The goal is to collect different items while fleeing from four ghosts that kill Pacman on collision. Based on this scenario two characters of different kind are involved: the Pacman and the ghosts. Both characters are implemented as classes, each extending a generic class called **Actor.** Although, the character classes inherit from the same generic class, the movement routines are implemented independently in the classes **Player** and **Ghost** and form clones of each other. In contrast to the previous task, the two clones are now linked in the code through the inheritance hierarchy. Based on this link we can investigate whether programmers may be guided by semantic code relations in their search for duplicated defects if clone information is not provided by a tool.

For the movement of both characters two coordinates are used. The first one is the position of a character on a coarse grained invisible grid. This coordinate is used to check for actor collisions and possible movement directions (gridX and gridY) from the current position. The second coordinate is a movement delta, relative to the center of the current grid cell (deltaX and deltaY). Based on two loops all characters are successively moved on the screen pixel by pixel in their current direction. As soon as a character reaches the center of the next grid cell, it is assigned to this cell. The original movement routine of the Pacman character without the inserted defect is shown in Figure 2 as pseudo code. The same code can be found in the **Ghost** class except for the check whether a movement is possible—lines 3, 13, 23, and 33—because the ghosts move on pre-computed paths and cannot get stuck. Therefore, the defective code segments are type-3 clones of each other. For tPM we inserted the defect in both characters by changing the subtractions in lines 8 and 38 to additions.

```java
1 switch currentDirection
2 case up
3 if canMoveTo(gridX, gridY - 1) then
4 deltaX = 0
5 deltaY = deltaY - speed
6 if |deltaY| >= CELL_SIZE then
7 deltaY = 0
8 moveTo(gridX, gridY - 1)
9 end
10 end
11 break;
12 case right:
13 if canMoveTo(gridX + 1, gridY) then
14 deltaX = deltaX + speed
15 deltaY = 0
16 if |deltaX| >= CELL_SIZE then
17 deltaX = 0
18 moveTo(gridX + 1, gridY)
19 end
20 end
21 break
22 case down
23 if canMoveTo(gridX, gridY + 1) then
24 deltaY = 0
25 deltaX = deltaY + speed
26 if |deltaY| >= CELL_SIZE then
27 deltaY = 0
28 moveTo(gridX, gridY + 1)
29 end
30 end
31 break;
32 case left:
33 if canMoveTo(gridX - 1, gridY) then
34 deltaY = 0
35 deltaX = deltaX - speed
36 if |deltaX| >= CELL_SIZE then
37 deltaX = 0
38 moveTo(gridX - 1, gridY)
39 end
40 end
41 break
42 end
```

Fig. 2. Pseudo code for Task tPM
The following defect report was presented to the subjects for task tPM: For all game characters, the movement up and left does not work correctly. Instead of moving up or left the characters move in the opposite direction in a flickering motion. Moving down and right works fine for all characters. As with FrozenBubble, the bug report provided hints where not to look for the defects. In addition, a rough overview on how the movement is implemented by describing the two coordinate systems and how they relate was given in the report.

E. Instrumentation

We provided all subjects a standardized development environment including all tools to execute the experiment and to analyze the outcome afterwards. The use of tools other than those provided by us was not allowed to ensure equal prerequisites for all participants and simultaneously avoid bias based on different levels of experience with other tools. For this reason, the Eclipse IDE was used as development environment and all subjects were required to have basic knowledge in programming with Eclipse. For our purpose we extended the Eclipse IDE by two plug-ins. The first plug-in was particularly developed to support the execution of such experiments and collect relevant data for later analysis. It was successfully used in previous experiments [22], [7], [8] already. In detail the plug-in provides functionality to: guide the participants through the whole experiment by step-by-step instructions, for instance, displaying the task descriptions and defect reports; automatically log user actions, e.g., record the usage of various Eclipse features like searches and debugging as well as to display surveys and collect answers; record the time needed to complete each task.

We used this plug-in for two main reasons: First, it allows the subjects to work autonomously on the tasks which avoids bias by potential feedback from us and minimizes the extent to which the performance of the participants is influenced caused by the awareness that they are monitored by us. Second, the automated collection of relevant data to evaluate the results is more accurate than a manual approach, for instance, by taking our own notes or ask the subjects to protocol their proceeding and the time required to complete the tasks.

The second plug-in we integrated in the Eclipse IDE allows to navigate through code clones of a software system. It is based on the results of our clone detector iClones\(^4\). We pre-configured iClones to detect clones of type 1, 2, 3 with a minimum total length of 50 tokens consistently to our former studies [23], [24], [25], [26], [27]. However, we provided the resulting clone information for both games such that the subjects needed to know neither anything about the configuration nor the detection of clones itself. Therefore, our experiment design does not depend on iClones, hence, any other tool providing clone information can basically be used. Certainly, it is required that the defective clone fragments are detected at least. The plug-in provided the subjects a simple user interface with a list of existing clones grouped in clone classes (a clone class contains fragments that are clones of each other) as a table in a split-screen window below the source code. The table presents each fragment together with basic information: a unique id, the path of its source file and its start and end line within the file and, the level of similarity (type 1, 2, or 3). We implemented two main mechanisms in the plug-in to provide the user with context-related clone information only. First, the list of clones covers only clone classes which contain at least one fragment that is located in the currently opened source file in the IDE. All irrelevant clones with respect to this file are invisible, which makes it much easier to overview the clone information. Moreover, we use markers next to the source code—at the left edge where usually line numbers are displayed in editors—to indicate that the code segment the user currently inspects is a clone, similarly to the familiar error indicators in Eclipse. The markers can be used by clicking to jump to the corresponding clone in the clone table to identify the other clone fragments of the same clone class. Finally, by clicking fragments in the table, it is possible to jump directly to their location. The corresponding source file is opened in the editor (if not yet opened), the focus is set to the position of the selected fragment, and the cloned code is highlighted. By providing such context-related clone information we increase the usability of clone data which usually suffers from a large amount of detected clones and potential false positives.

IV. Execution

We executed the experiment several times, each time different subjects participated because various time constraints did not allow us to bring all participants together at one time. Each time the experiment was executed with small groups of participants or in individual appointments together with at least one supervisor. To establish the same conditions for subjects of different runs, a repeatable workflow was used. Mainly, this workflow consists of the four different phases: Introduction, Installation, Execution, and Data Collection. In the following we describe each phase in detail.

Introduction. We started by introducing the subjects using a slide presentation together with a scripted talk. Information was given on the course of the experiment, time constraints, how to use the delivered materials and what they are supposed to do. Describing the tasks, no information on code clones was provided in any way, hence, the subjects were not aware that the experiment focused on clones. Finally, we gave an introduction to the integrated plug-in which allows to navigate through existing clones. Again, we did not use the term clone here. We presented the plug-in as an extension to the search feature of the Eclipse IDE that enables the user to locate similar or identical code fragments. We chose to avoid the term clone and presented the plug-in as an advanced version of an already existing feature to avoid bias—talking about clones or a clone-management tool may affect the expectations of the subjects. The plug-in was developed such that it can be used very similar compared to other features in Eclipse to foster an intuitive usage of the tool based on existing experience and further mitigate possible bias. The subjects were informed that

\(^4\)http://www.softwareclones.org/iclones.php
the tool will only be available for one of the two tasks and that there is no dictate to use it. Therefore, all the standard features of Eclipse could be applied without further constraints.

**Installation.** The participants received a pre-configured Eclipse IDE including the additional plug-ins for the experiment and a workspace including the two tasks. Based on our dependent variable, the participants got a workspace including clone information either for task \( tFB \) or task \( tPM \). Since Eclipse is Java based, no additional installation or configuration was needed—everything was prepared to work out of the box. Moreover, a handout with a short summary of the Introduction was provided.

**Execution.** In the Introduction and the handout, all relevant information was presented to the participants to start the experiment and then to continue by following the step-by-step guide of our Eclipse plug-in. All activities of the participants to complete the experiment were set, except for the time spent on the two tasks. That is, the duration the participants worked on each task was not limited to avoid incomplete or incorrect results due to lack of time. Such results could not be compared to the timing and correctness of those that have been produced within a given time limit. Harder and Tiarks [7] observed in pilot studies to their experiment that a time limit was a major mortality threat to obviate if the number of available subjects is rather small. However, subjects were allowed to skip a task if they were not able to locate or fix the bugs.

**Data Collection.** As soon as a participant finished the tasks, his or her workspace including all relevant data for our analysis was automatically archived. By automatically packing the workspace the possibility of flaws during a manual copying process have been bypassed.

V. **ANALYSIS**

From the 29 subjects that participated in our experiment we analyzed only the data of solutions complying with our requirements. Therefore, we excluded the result from one student who stated missing Java or Eclipse skills in the self-assessment. Furthermore, three participants who gave up on both tasks were excluded from our analysis. Solutions of participants who specified that they failed on the tasks cannot be compared to solutions specified to be finished regarded timing and correctness in a meaningful way. In total, we analyzed 25 results of subjects who fulfilled the requirements and stated that they finished both tasks.

Several participants did not use the provided clone information, which resulted in an uneven distribution of participants to the given tasks. For instance, if a subject with the task combination \( tFB \) and \( tPM \) chose not to use the clone information to locate the defects, but instead searched them manually or used other Eclipse features such as the debugger, we counted their results to the tasks \( tFB_{nc} \) and \( tPM_{nc} \). As mentioned before, the participants were free to choose how to approach the given tasks to avoid bias, and therefore, it was not compulsory to use the clone plug-in even if it was available. Comparing the performance of those participants to the other ones and also considering their self-assessment towards programming experience, we did not find any indication that they chose to do the tasks without the clone information based on superior skills or experience as their results have neither been remarkably better nor worse. We will discuss the reasons for doing the tasks without the tool support in Section VI.

A. **Descriptive Statistics**

We use statistical tests to investigate the performance of the subjects and pursue our first research question. Table III shows the average times needed by the participants to solve the tasks. Figure 3 visualizes the distribution of the timing data.

**TABLE III**

<table>
<thead>
<tr>
<th>Correctness</th>
<th>( tFB_{c} )</th>
<th>( tFB_{nc} )</th>
<th>( tPM_{c} )</th>
<th>( tPM_{nc} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>addressed</td>
<td>743 &lt; 1.636</td>
<td>599 &lt; 630</td>
<td></td>
<td></td>
</tr>
<tr>
<td>complete</td>
<td>743 &lt; 1.781</td>
<td>539 &lt; 611</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 3](image)

(a) \( tFB \)

(b) \( tPM \)

Fig. 3. Performance of the participants in seconds

For both tasks the subjects were faster on average when clone information was provided. The difference is most distinctive for \( tFB \), where the group without access to clone data needed more than twice as much time on average to finish the task. The average time for \( tFB_{c} \) and \( tFB_{nc} \) differs by 1,038 seconds—more than 17 minutes—regarding complete solutions. In relative proportion, the average time increased by 139.7 %. The respective box-and-whisker plots in Figure 3a indicate that the difference in the mean (denoted by the * symbol) is not caused by outliers. Instead, a shift of the distributions can clearly be observed. The variance of the time measures for addressed and complete results is rather small. This was expected, since the majority of subjects succeeded on both tasks and addressed solutions include complete ones which implies similar timings among both groups.

The time difference measured for the \( tPM \) task is much less distinctive than for \( tFB \), even though the results again show
that the subjects required less time on average to solve the task using clone information. The average time for \( tPMc \) and \( tPM\text{nc} \) differs only by 72 seconds for complete solutions. In relative proportion, the average time increased only by 13.4\%. Figure 3b shows that the mean values are affected by an outlier in \( tPM\text{nc} \) (denoted by the \( o \) symbol). Comparing the timings of only addressed solutions to complete ones, we again observe rather small differences which again is mainly due to very high numbers of complete results.

The investigation of our second research question is based on the numbers for correct and incorrect solutions provided. To assess correctness, we manually inspected the changes to the source code of each subject using the Unix diff tool. Furthermore, we executed the programs to assure they are running properly. For each task we analyzed whether any of the defect occurrences were corrected. If not, the subject was eliminated from the data evaluation because these were submissions on which the subjects gave up and did not perform any changes at all. There is no meaningful way to compare such results to addressed or complete ones. Accordingly, we checked for each task whether both defect occurrences were corrected to distinguish between addressed and complete results. Table IV shows the results of our manual inspection.

<table>
<thead>
<tr>
<th>Correctness</th>
<th>( tFB )</th>
<th>( tFB_{\text{nc}} )</th>
<th>( tPM )</th>
<th>( tPM_{\text{nc}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>addressed</td>
<td>9 (100%)</td>
<td>4 (25%)</td>
<td>1 (16.7%)</td>
<td>5 (83.3%)</td>
</tr>
<tr>
<td>complete</td>
<td>12 (75%)</td>
<td>5 (83.3%)</td>
<td>18 (94.4%)</td>
<td>15 (97.5%)</td>
</tr>
</tbody>
</table>

Generally, most participants were able to locate and fix both defects inserted in the software systems. In \( tFB \), all participants who used the available clone information corrected both defects. If no clone information was given or was not used, 25\% of the subjects fixed only one of the two defects. Similarly, the solutions for the \( tPM \) task were predominantly correct. The relative success rate for the task \( tPM_{\text{nc}} \) is with 94.4\% very high—surprisingly even higher than the success rate for \( tPMc \).

### B. Hypothesis Testing

To test our hypothesis we use statistical tests to evaluate the recorded data. To support our first hypothesis \( H_1^{\text{time}} \) regarding the time needed to fix both defects, the null hypothesis \( H_0^{\text{time}} \) has to be rejected based on the results of the statistical test. We use the parametric Student’s T-test as well as the non-parametric Mann-Whitney U-test for independent samples on the time data. Since, we cannot assume a normal distribution the U-test is the more adequate significance test in our case, although it does not take into account the degree of the time differences but only their ranking. For both statistical tests we use the 1-tailed variant, because the alternative hypothesis \( H_1^{\text{time}} \) postulates that the time needed to correct a cloned defect is larger if no clone information is provided. The results are shown in Table V. Values below the commonly used threshold for statistical significance \( p < 0.05 \) rejecting the null hypothesis \( H_0^{\text{time}} \) are printed bold.

<table>
<thead>
<tr>
<th>Correctness</th>
<th>Student’s T-test</th>
<th>Mann-Whitney U-test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( tFB )</td>
<td>( tPM )</td>
</tr>
<tr>
<td>addressed</td>
<td>0.0000</td>
<td>0.3871</td>
</tr>
<tr>
<td>complete</td>
<td>0.0000</td>
<td>0.3667</td>
</tr>
</tbody>
</table>

Statistical significance is reached for \( tFB \) based on the T-test as well as the U-test. In contrast to \( tFB \), no statistical significance has been reached for the task \( tPM \) using both tests. The p-values of \( tPM \) are rather high. Based on the different results of the two tasks regarding statistical significance \( H_0^{\text{time}} \) cannot be rejected and, hence, the \( H_1^{\text{time}} \) cannot be clearly supported.

To test the statistical significance regarding our second hypothesis towards correctness, the null hypothesis \( H_0^{\text{corr}} \) has to be rejected to support \( H_1^{\text{corr}} \). We use Fisher’s and Barnard’s exact tests to test statistical significance of the correctness. Both tests can be used to test significance of categorical data in the form of contingency tables. Moreover, both tests support small sample sizes like ours. Similar to the timing, \( H_1^{\text{corr}} \) is a 1-tailed hypothesis, because it postulates that the probability of a correct removal of a cloned defect is lower when clone information is used. The 1-tailed variants of the two statistical tests are used accordingly. Table VI shows the results.

<table>
<thead>
<tr>
<th>Test</th>
<th>( tFB )</th>
<th>( tPM )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fisher</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Barnard</td>
<td>0.0854</td>
<td>0.3200</td>
</tr>
</tbody>
</table>

VI. DISCUSSION

In this section we discuss the results and interpret peculiarities. Apart from the recorded data of our Eclipse plug-in, we asked every participant to write down his solving strategy and also talked to them in a feedback session subsequent to the experiment. Furthermore, we remark on observations that may help to gain important insights towards further research.

The results of our experiment did not show a statistically significant benefit of the clone information on correctness, but that was expected. The participants were allowed to execute and test their modified version, so testing was able to compensate for missing clone information. That is, the participants were able to continue their search for the cloned defect in case they fixed only one of the two defect occurrences.

We expected the participants to find cloned defects faster if clone information was available. We found this to be true for both programs as the means of the time needed to fix the defect is smaller for the experimental group with clone information. Yet, the difference is statistically significant only for time \( tFB \) of FrozenBubble. The difference in \( tPM \) of Pacman is marginal. We assume the inheritance hierarchy reduced the time needed to locate the cloned defect when there was no clone information—which we expected to observe when preparing the defect scenarios. Both tasks have in common that the cloned instances of the bugs are located in different classes, however, the inheritance hierarchy of Pacman seems
to helped the participants to identify the corresponding class just by looking at the file tree or type inheritance tree. The cloned defects in Pacman were in two sibling classes (ghost and player character) as opposed to FrozenBubble where the defects were in a cloned loading sequence of images in the otherwise semantically unrelated classes FrozenGame and LaunchBubbleSprite. So the inheritance hierarchy gave hints on where to look for the defects only in Pacman. This explanation was also stated in several strategy reports of the participants. This is probably the reason why nearly 50% of the participants who were offered the clone tool decided to do without it for the task tPM compared to only 25% for tFB.

Apart from the complexity of the two tasks, participants who decided to do without using the tool quoted that they preferred to use other features of Eclipse. The feature that was referred to the most is the keyword search functionality of Eclipse—which was also commonly used to locate the defects when no clone information was provided. Besides being familiar with it, the participants stated that after they located the first defect it has been obvious to them what to search for. Hence, there was no need to use another feature. This feedback shows two things: (1) the subjects preferred to search the source code of the games by specifying their own patterns based on the information they gained by browsing through the code before, (2) the advantage of clone information was recognized based on the degree of complexity regarding the given task.

Some of the participants who did not use the clone information also reported that a substantial training or a more eye-catching presentation of the clone tool may increase the probability of being used. We deliberately chose a rather short introduction to the plug-in and a presentation that is characteristic to Eclipse features to avoid bias in the expectations of the subjects and support an intuitive usage. Giving a more detailed training probably improves the performance of the subjects, but the challenge is to find the right balance to minimize bias in such human-based studies as much as possible.

Another factor which may have contributed not to use the clone tool and, moreover, which may explain the high number of correct solutions is that in our experiment the defective clones occurred only twice. We experienced that clones in industrial and open-source systems have often multiple occurrences. The chance of a complete removal of a cloned bug in a real scenario is most probably lower than in our study because more instances of the bug may exist at various locations. Incomplete fixes of such clones may leave the system in a defective state or even introduce new or other defects based on inconsistent changes. In such cases, it may be more beneficial to be provided with clone management tools to make sure that all occurrences of a cloned bug have been fixed or at least are known and kept track of. This will reduce the risk that a system remains defective after applying changes to clones.

Apart from the keyword search feature that was frequently used, the logs related to the activities of the participants do not show clear strategies or patterns. In contrary, different approaches were used from participant to participant. Some used features like searching or debugging whereas others just browsed the code and heavily switched between files without using tool support. Similarly, testing was done quite extensively or just very selectively.

VII. Threats to Validity

In this section we discuss the threats to validity related to our study and how we tried to mitigate these.

Construct Validity. Our Eclipse plug-in is designed such that only the time needed to complete a whole task is recorded. A more fine grained measurement capturing the time the subjects spent on each individual activity (e.g., program comprehension, bug reproduction and localization, changing and testing the code) may expose further insights. However, in this study we are interested in the overall time needed to finish a complete maintenance task including all involved activities and in the final results produced for each task.

Internal Validity. Selection. The subjects were randomly assigned to two groups A and B without any blocking. However, we ensured that all subjects participated for the first time in a controlled experiment related to code clones. Due to the small number of participants and the simple randomized sampling used, the groups may not be balanced regarding experience and skills of the subjects. The risk of unequal experience was minimized by the fact that all subjects were required to be familiar with Java and Eclipse. Moreover, the tasks were designed such that no knowledge beyond the standard Java class library was needed.

Maturation. To avoid the risk of learning bias, the subjects were provided with different systems for each task. Both groups were provided with clone information for the systems in switching order to rule out a bias based on a different complexity of the systems. The order in which clone information was available may have affected the expectations for the second task to work on.

Instrumentality. Due to organizational reasons all subjects were required to use the Eclipse IDE prepared for the experiment. Although experience in working with Eclipse was mandatory to participate different levels of knowledge with it may result in differences related to the performances. However, the recording of data (e.g., time) was done in the background and did not influence the subject’s behavior.

Experimenter bias. To avoid bias based on the interaction between the subjects and us, we gave a scripted talk at the beginning of each session that was identical for all participants. Afterwards the subjects were guided through the whole experiment by instructions on the Eclipse plug-in. The relevant data to analyze the results were automatically recorded by the plug-in as well and, therefore, not exposed to our subjectivity.

External Validity. In this study we used two rather small software systems from the same domain with rather low complexity. The given tasks were quite simple, too. However, the costs of preparing, executing and evaluating such experiments is still quite high. Each participant spent about three hours considering the introduction, execution as well as a feedback session afterwards. A more complex experiment design (more systems, more clones, etc.) would have gone
beyond the time constraints of such an experiment. Still, we prepared the defects such that they are quite realistic—wrong indices in loops, for instance, off-by-one errors, and incorrect uses of signs when computing movement patterns are fairly common defects. Nonetheless, most industrial as well as opensource systems are larger in size and normally of higher complexity. The clones in these systems occurred only twice in the source code, but from our experience many clones occur more than twice in most systems. Due to the high cost of involving participants for human-based studies the number of subjects was rather small and all of them were students. A different (especially larger) sample of professional subjects may produce different results. These threats suggest that our results may not necessarily be generalized to other systems and populations. Further studies are necessary to reduce these threats and allow for a more general understanding.

VIII. CONCLUSIONS

We conducted a controlled experiment to investigate the use of clone information on the performance of developers in common bug-fixing tasks in terms of time and correctness. For this purpose, we extended a previous study on the use of clone information by changing the experiment design to overcome various shortcomings. To the best of our knowledge, we are the first to evaluate the support of contextual clone information integrated in an IDE in a human-based controlled experiment. The experiment shows that developers are quite capable to compensate missing clone information in certain situations through testing to provide correct solutions (although our experimental design does not allow to generalize this finding to cases in which defects were cloned more than once). Clone information potentially helps to detect cloned defects faster, although developers may exploit semantic code relations such as inheritance to uncover cloned defects only slightly slower if they do not have clone information. If cloned defects lurk in semantically unrelated places however, clone information helps to find them faster at statistical significance. Developers without clone information needed 17 minutes longer on average or 140% more time in relative terms to complete the task. Although the findings might be regarded quite expected, they still need to be proven by empirical research to rule out possible misjudgments [28].

In future research further human-based studies should be directed to the use of clone information and clone management tools to widen and deepen the insights gained. For instance, maintenance tasks other than bug-fixing need to be looked at to capture a more complete picture of how programmers work with clones. Also, more effort is needed to research the impact of different code structures and design patterns, such as inheritance, on the performance of developers when dealing with cloned code. Our work is preliminary regarding this aspect.

ACKNOWLEDGMENTS

The authors would like to thank the students who participated in the experiment. This work was supported by research grants of the Deutsche Forschungsgemeinschaft (DFG).

REFERENCES