UNIVERSITY OF CALIFORNIA,
IRVINE

User Assessment of Debugging Using a Software Visualization Tool Compared with Traditional Debugging Methods

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCES

in Software Engineering

by

Christina Lauren Rall

Thesis Committee:
Assistant Professor James A. Jones, Chair
Professor André van der Hoek
Professor Debra Richardson

2014

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my sincere appreciation for the time and guidance provided to me by my committee chair and academic advisor, Professor Jim Jones. Thank you for believing in me even when it seemed like I wasn't going to pull this off and thank you for all of your guidance and encouragement.

I would also like to thank my committee members Professor André van der Hoek and Professor Debra Richardson for their time and support. I would especially like to thank Debra for encouraging me to study Software Engineering at UCI and to take an interest in on-campus research.

Abstract of the Thesis

User Assessment of Debugging Using a Software Visualization Tool Compared with Traditional Debugging Methods

By

Christina Laruen Rall

Master of Science in Software Engineering

University of California, Irvine, 2014

Assistant Professor James A. Jones, Chair

Debugging is time and energy intensive. Many tools have been developed to help solve the problems associated with debugging, but programmers still rely on editing their code using traditional, manual techniques. One reason behind this is that many techniques succumb to the Isolation Flaw, where they isolate suspicious code to the point that it loses necessary context. Additionally, traditional debugging relies on the ways in which humans rely on the creation, testing and modification of hypotheses. An ideal tool will both avoid the Isolation Flaw while assisting the developers in their hypothesis cycle. This thesis consists of an empirical study that evaluates how debugging changes between traditional debugging and debugging with visualization assistance. The visualization chosen is based on the Tarantula fault localization tool. Each participant is given one program with the visualization and one without it to debug. Results imply that debug times using the tool to debug the programs in this study sometimes resulted in faster debugging, but usually was not significantly different from traditional debugging. The visualization decreased the average time between locating the file that the bug was in and fixing the bug for all bugs, implying that participants who

reached that file did so on a more accurate hypothesis about the cause of the bug. While the tool may not consistently improve the speed of debugging for programs of this size and bugs of this complexity, it offers promising results regarding the context-dependent learning that is missing from debugging tools that contain the Isolation Flaw.

Motivation and Related Works

Overview

The underlying motivation behind all software visualization work is summed up well by the overview of Stasko et al.'s book on Software Visualization[1]. High quality interfaces are becoming more advanced and standardized; "yet the overwhelming majority of programmers edit their code using a single font within a single window and view code execution via the hand insertion of print statements." Software visualization uses visual representations to enhance the understanding of various aspects of a software system[2]. This thesis investigates software visualization used particularly in the field of fault localization and debugging, an area of particular importance given estimations such as that of the National Insititute of Standards and Tehcnology that testing and debugging constitute 30-90% of software development costs and that an average error can take 17.4 hours to locate and fix[3].

This thesis was originally intended simply as a user study for Tarantula[4], a test information visualization tool created by Jones, Harrold, and Stasko. It sought to evaluate the tool in comparison with traditional debugging methods, since these traditional methods are the bread and butter of debugging in practice. However, when considering the existing literature, it became clear that the scope of the project extended beyond a simple user study; it could also be used to better understand how and whether having visualization assistance helps participants to locate and fix faults.

In this study, we used an experimental design to compare debugging results with and without a visualization. This included an analysis of whether or not the participant was able to find and fix the bug, as well as the time it took for participants to locate each of the following: the correct file, the correct method, and the correct fix. These measures were chosen in order to look at debugging by its process, not just by its results.

Analysis of Related Works

The paper that is most pertinent to this thesis is Parnin and Orso's investigation of whether automated debugging techniques actually help programmers[5]. They claim that the main flaw with existing automated tools is a natural consequence of attempting to reduce the number of statements developers need to examine. For the sake of this paper this issue will be referred to as the *Isolation Flaw*, which is defined as the mistaken belief that isolated statements provide enough contextual information for developers to understand and fix bugs.

Fault localization tools aim to walk a careful line between showing the developer too much and showing the developer too little. On one hand, showing the developer too much is an obvious problem. If a tool were to show every line of the source code as suspicious, then the developer is not any closer to finding the fault. Unfortunately, even dramatically reducing the amount shown is not always enough. A tool that cuts code down to 1% of its original size would still leave five thousand lines for a developer to inspect on a 500,000 line program. This is certainly a more manageable size, but may still not save the developer enough time or energy. On the other hand, code reduction comes with potential

consequences as well. Based on my interpretation of related work, the *Isolation Flaw* is one potential result of over-pruning, but it is not the only issue. There is also the very real risk that the tool could eliminate the actual bug in the process of pruning (causing the developer to expend even more time and effort than they would without the tool). More simply put, too much output makes the bug difficult to find and too little makes it difficult to fix.

The goal of a fault localization tool is thus to reduce the information the developer must inspect while creating as few consequences as possible. Visualizations allow these tools to do more than just reduce the problem space, by providing a means of adding meaningful information, while still walking the balance between providing too little and overwhelming the user.

Three popular ways to account for the scope of the information the developer must inspect are (1) Slicing, (2) Delta Debugging, and (3) Differentiating Techniques. While not always present, visualizations and user interfaces have been created based on all three of these techniques and will be referenced within the category they fall under.

(1) Slicing

The *Isolation Flaw* is common in program slicing tools, which have been around since Weiser developed the technique over 30 years ago[6] [7]. Slicing creates useful, but ultimately overly large[8], sets. In order to reduce sets to a manageable size, research has been conducted to investigate various slicing techniques that produce smaller data sets [9] [10] [11]. These smaller data sets remove large amounts of information, opening the door

for the *Isolation Flaw* to manifest. Additionally, even despite these efforts, they are often still too large to be used in practice.

(2) Delta Debugging

Delta debugging, similar to slicing, works to reduce the problem space[12]. It works by doing a binary search to minimize input that causes failures and to minimize the body of code affected by the failures. There are multiple delta debugging techniques, but all seek to eliminate something: line of code, suspicious inputs, etc.

(3) Differentiating Techniques

Another group of automation techniques relies on differentiating between the characteristics of passing and failing test cases such as path profiles[13], model checking[14] [15], statement coverage[4][16][17], predicate values[18][19], and various clustering techniques based on these characteristics[20][21][22][23]. Tarantula falls into this category; it is a visualization tool that bases suspiciousness on statement coverage. Differentiating techniques also run into the *Isolation Flaw* because they reduce the number of statements developers need to look at.

Given that the point of automated debugging is to reduce the amount of code a developer needs to inspect, the question starts to emerge of whether it is even possible to have a fault localization tool that does not run into this *Isolation Flaw*. In order to answer this, two other questions must first be answered: (1) What about traditional debugging better allows developers to access the understanding of the problem that they need in order to find and fix

the bugs? and (2) How can the benefits of traditional debugging be leveraged so that the developer has a reduced number of statements to evaluate, without losing the necessary contextual information?

The first question asks about human understanding of the debugging problem they are facing. Traditional debugging obviously does not fall victim to the *Isolation Flaw* because there is no isolation — all of the code is present. However, this can be a double-edged sword; since *all* of the code is present, developers are almost inevitably dealing with more information than can be handled at a single time, unless they are debugging an extremely small program. This suggests that while debugging traditionally, developers must be using some heuristic for reducing the problem space.

The research behind the Whyline debugging tool, a visualization tool based on slicing, discusses the logical process humans use to debug traditionally. Failures produce some "observable symptom of failure" and developers debug by guessing about the cause of that symptom and then testing their hypothesis until they find a solution[24] [25]. This specifies that the heuristic that developers use is a hypothesis heuristic to reduce the problem space.

The intuition behind delta debugging is also based on this hypothesis model: programmers debug by creating a hypothesis: "this or that?", testing their hypothesis, and then narrowing down their hypothesis until they find the minimal element causing the error.

Winslow's research on the differences between novice and expert programmers, while not specific to debugging, can also provide insight. He found that novices approach programming "line by line" rather than at the level of meaningful program chunks or structures[26]. It seems reasonable to assume that this difference in approach would hold true between novice and expert debuggers as well. This ties in well with the *Isolation Flaw* — if programs are debugged line-by-line, then the context of the meaningful chunks is lost. The more global approach used by expert debuggers prevents context loss.

Together, this means that programmers using traditional methods are able to heuristically reduce the problem space using hypotheses and are able to mentally retain the context to fix the bug by processing programs as meaningful chunks or structures. Based on this conclusion, a useful automated debugging tool should:

- Reduce the problem space without actually removing any code. (This can be achieved by directing the user's focus, reducing the amount seen, or adding information. Meaningful supplementary information can be helpful as long as it is added in a way that is not overwhelming.)
- Maintain the structural integrity of the program (rather than resorting to something like sorting lines by suspiciousness, which dramatically reduces context and makes it more difficult to understand how pieces fit together)
- The tool should either explicitly or implicitly help developers to create hypotheses
- The tool should help developers to confirm or reject hypotheses they form

This Thesis

As stated in the overview, we compare debugging results with and without visualization assistance in order to better understand how the presence of a visualization tool changes the ways in which people debug. Tarantula presents the entire code of the program with every line color coded according to the suspiciousness of the code. It was a good tool for this study based on the conclusions of the analysis of related work because it does not remove code or change the structure of the program and the suspiciousness values were intended to help participants create and evaluate their hypotheses.

Empirical Study

This thesis attempts to evaluate whether and in what way software visualizations can assist in fault localization and debugging. This section will discuss the participants, design, and implementation of the empirical study.

Participants

We recruited undergraduate students enrolled in a Software Testing class at UCIrvine as well as master's students enrolled in Software Engineering because these students should have enough understanding of programming and software testing to successfully debug code. It also provided a larger sample size, providing more generalizable results. There were 20 participants in the preliminary run and 94 participants in the actual study. It is unclear why, but there were 97 responses to the survey during the actual study despite there only being 94 participants. It is possible that some participants did not turn in their data at the end of the session. Only data that was collected was analyzed.

Table 1

Programming experience in years for overall, professional and java

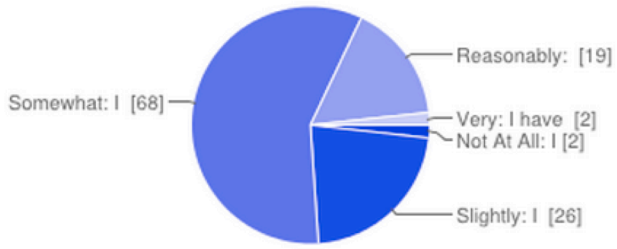
	Years programming	Professional programming	Java experience
mean	4.3077	1.0970	2.2959
max	25	15	7
min	0.5	0	0

Information regarding programming experience, obtained from a pre-questionnaire, is shown in Table 1. Years of programming ranged from 6 months to 25 years with an average of 4.3 years experience. Professional programming experience ranged from not at all to 15 years, with an average of just over one year's experience. Since the debugging tasks given in this study were coded in Java, participants

were also specifically asked about their level of experience programming in Java. This ranged from no experience to 7 years of experience with an average of 2.3 years.

Since the visualization provided relies on unit tests, participants were also asked about their experience with unit testing. The results of this are shown in Figure 1. Most participants had at least some hands on experience using unit tests (76%). A remaining 22% had 'slight' experience with unit testing, indicating that they had heard of it but not used it themselves and the remaining 2% had no experience with unit testing what so ever.

Figure 1



Not At All: I do not know what unit testing is	**2**
Slightly: I have heard some about it but not used it myself	**26**
Somewhat: I have used it a few times	**68**
Reasonably: I have used it on multiple occasions and am comfortable using it	**19**
Very: I have used it frequently and/or in a professional setting and am confident with it	**2**

Task Overview

Each participant had to complete four tasks: two debugging tasks bookended by two quick questionnaires. Participants were allowed to move on to the next debugging task if they spent more than 30 minutes on a task. Most participants did this, although some chose to spend a few minutes extra if they felt they were close to a solution.

The tasks were entirely self-guided. Participants were given a printed sheet (shown in Figure 2) that contained a link to a Google Drive folder and the following instructions:

> Download *theCorrectFileForYourSession.zip*
>
> Unzip the file and read DebuggingPracticumInstructions.pdf completely before starting. Raise your hand if you have any questions.

There were eight possible zip files, which are discussed further in the next section. The instructions enumerated the four tasks: Pre-Questionnaire, Debugging with Visualization Assistance, Traditional Debugging and the Post-Questionnaire, and provided instructions for each which are discussed in more detail in the following subsection

Figure 2

This handout was given to all participants. There were 8 versions of this handout each with a different zip file listed at the top for participants to download.

Download **PracticumA1.zip** from https://drive.google.com/a/uci.edu/folderview?id=0B5aKFN583n0keXBDejJ2TTJ2OFk&usp=sharing

Unzip the file and read DebuggingPracticumInstructions.pdf completely before starting. Raise your hand if you have any questions.
The first line below is an example of how to fill in this sheet.

Program Name	Time Found	File Name	Method Name	Bug Fix
Pacman	4:18 pm	AbstractGhostMover	FakeMethodName	Line 5: i++;

Debugging Instructions

The following instructions were provided for both debugging tasks to guide participants through the debugging

1. Import *projectToDebug* into Eclipse
2. Record the time you started on the hand out given to you. You have UP TO 30 MINUTES to try to locate the bug using whatever method(s) you want.
    a. You can run *pacman* by right clicking src/main/java > Run As > Java application
    b. You can run the full suite of unit tests by right clicking src/test/java > Run As > JUnit Test
3. Every time you have an idea what method the bug is in, fill it in on your handout and write the current time.
4. Once you think you've found a bug, try to fix it.
5. If you find a solution, record it. If you decide you were wrong about the bug's location, keep looking and go back to step 4.
6. Repeat until you find the bug or 30 minutes has elapsed since you started then go on to the next task.
7. You are given the following bug report: *"Bug Report"*

The *projectToDebug* and *Bug Reports* referred to above are described in the next section. For the task with Visualization Assistance, the participants were given this step prior to the import instructions:

1. Read the one-page explanation of the visualization provided in *VisualizationAssistance.pdf* then scroll down to view the visualization.

Details on the different visualizations is discussed in the *Visualizations* subsection of the next section. The instructions on the first page of the Visualization Assistant document is as follows:

> Figure 3 shows code on the left with corresponding unit tests on the right. The values entered for the unit tests are at the top and the P or F at the bottom indicates which unit tests passed and which failed. While these 13 lines of code are fairly simple to debug, the mass of black makes it more difficult to distinguish the effects of a unit test. Colors are used to make it easier to view the unit test results at a glance. If a statement is only executed during failed executions, it is colored red; if a statement is only executed during passed executions it is colored green; statements that are not executed are not given a color. If a statement passes some of the time and fails at other times it will appear somewhere on the spectrum between green and red: yellow or orange (Figure 4). The color value is calculated by looking at:
>
> $$\text{color(s)} = \text{low color (red)} + \frac{\%\text{passed(s)}}{\%\text{passed(s)} + \%\text{failed(s)}} * \text{color range}$$
>
> %passed(s) — the ratio of passed test cases that executed line s to the total number of passed test cases
>
> %failed(s) — the ratio of failed test cases that executed line s to the total number of failed test cases

The bug is not necessarily in the reddest statement. Sometimes a bug earlier in the code can cascade, causing later lines of code to result in test failures. Any line that is not completely green is at least somewhat suspicious. Zoom the PDF out to view more of the code at once.

Figure 3

Unit tests

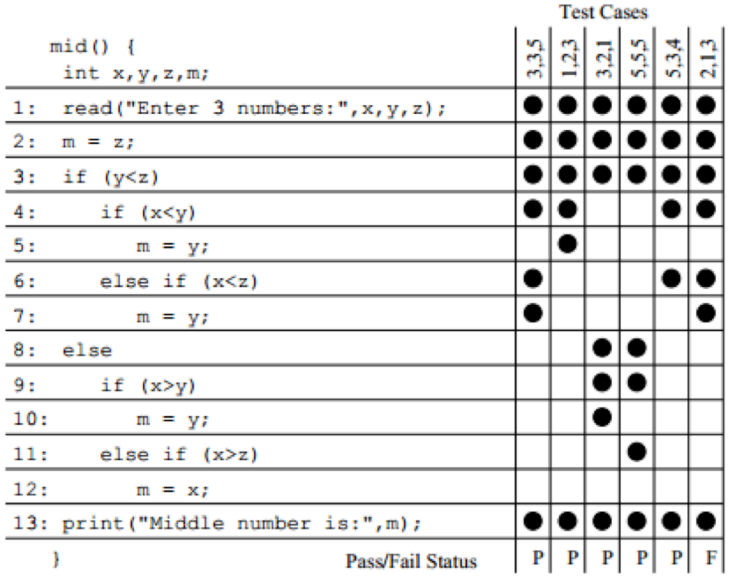
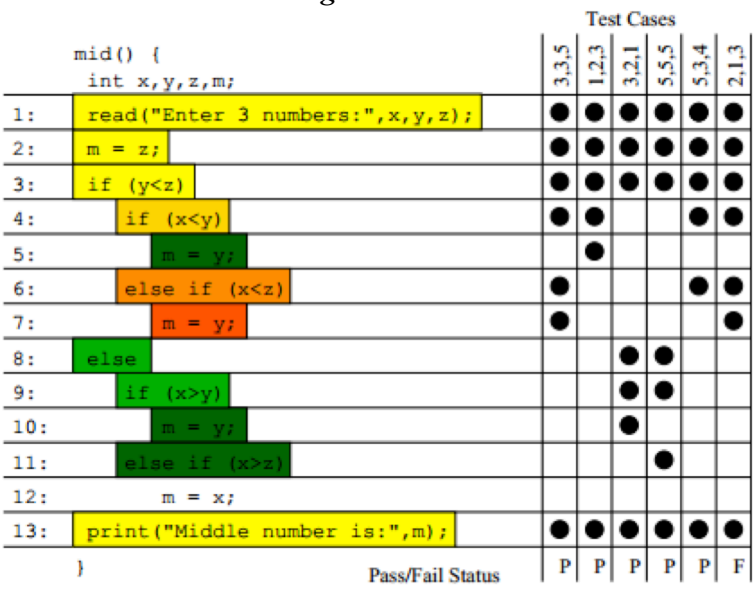


Figure 4

Unit tests with line coloring



Experimental Conditions

Programs to Debug

Two programs were chosen for debugging: *pacman* and *tetris*. Each participant performed one debugging task in each program. While using a single program for both tasks would allow for a direct comparison between debugging times, this option would have produced too many sources of error. Most prominently, participants might have been able to just compare files to see where they differed. Even assuming this would not occur, participants would likely debug the second task faster due purely to increased familiarity with the code. With two different programs, these problems are avoided. Additionally, it provides more diverse data due to the differences between the programs themselves.

The version of *pacman* used was the jpacman-framework created by Arie van Deursen, which was created with the intention of being used for teaching software testing[27]. The code is broken into 31 class files and includes interfaces and abstract classes. Not including the interfaces and abstract classes there are roughly 3,000 lines of code. Additionally, participants were given 69 unit tests written in JUnit.

The version of *tetris* used is from Per Cederberg and is available under the GNU General Public license[28]. The code has 6 class files and roughly 2,400 lines of code. The programming style of the game did not lend itself well to unit tests, so instead a list of 20 manual tests was given. These test cases included items like:

> test 5: start the program, start the game, push <L,R,U,D>
>
> test 9: start the program, start the game, get a *tetris* (use left, right, space, down, up)".

The participants were told that "getting a *tetris*" refers to filling in an entire horizontal row with blocks, which, in code without bugs, triggers the entire line to clear. These test cases were given in a PDF with all test cases listed. Next to each test case is either a green P or a red F which indicates whether the test passed or failed. This was to simulate having a JUnit test suite with the JUnit output describing which test cases passed or failed. One sample of this is shown in Figure 5. This allows participants to see the results without having to iterate through every manual test and still provides the list of manual tests so that participants can perform them if desired.

Figure 5

Example test suite result sheet for tetris

Tetris test cases:
P = Passed F = Failed
these make sure the program is working, it can open and close without error
P **test 0:** start the program, wait 5 seconds, and close it
P **test 1:** start the program, start the game, wait 5 seconds, and close it

these just have you push buttons without regard to what objects are doing, just push the buttons with a 1 sec gap and then close
P **test 2:** start the program, start the game, push <L,L,L,L>
P **test 3:** start the program, start the game, push <R,R,R,R>
P **test 4:** start the program, start the game, push <L,R,R,L>
F **test 5:** start the program, start the game, push <L,R,U,D>
P **test 6:** start the program, start the game, push <space, space, space, space>

these have you take into consideration where objects are (if something is broken, ie the game freezes, blocks wont move, etc, close)
P **test 7:** start the program, start the game, get a tetris (only use left and right)
P **test 8:** start the program, start the game, get a tetris (only use left, right, space,up)
F **test 9:** start the program, start the game, get a tetris (use left, right, space, down,up)

these have you use the game over stuff, and then the restart
P **test 10:** start the program, start the game, let blocks stack to game over
P **test 11:** start the program, start the game, let blocks stack to game over, then restart
F **test 12:** start the program, start the game, let blocks stack to game over, restart, get a tetris (use L, R, up down)

these have you use the games pause
P **test 13:** start the game, pause the game, un pause
P **test 14:** start the game, pause the game, the push <L>
P **test 15:** start the game, pause the game, the push <R>
P **test 16:** start the game, pause the game, the push <U>
F **test 17:** start the game, pause the game, the push <D>
P **test 18:** start the game, pause the game, un pause, push <L, R>
F **test 19:** start the game, pause the game, un pause <D, space, R, L, U>

Bugs

There were a total of four different possible bugs in the study, two possibilities for each program (though rach participant was only given a single bug per program.). This was done for two reasons: 1) to reduce the odds of cheating from adjacent participants and 2) to

provide more diverse bugs to increase generalizability of the results. They are named as follows: *Pacman1*, *Pacman2*, *Tetris1*, and *Tetris2* (also referred to as *P1*, *P2*, *T1*, *T2*).

Table 2

The bug report, class, method, bug, fix, and detailed result explanation for each bug

	Pacman1	**Pacman2**	**Tetris1**	**Tetris2**
Bug Report	Arrow keys are not behaving as expected	Pacman can move in ways that are not expected	Full rows are not behaving as expected	Down arrow is not behaving as expected
Class	Board	Game	SquareBoard	Figure
Method	tileAtDirection	movePlayer	removeLine	moveAllWayDown
Bug	return tileAtOffset(t, dir.getDy(), dir.getDx());	if (tileCanBeOccupied(target) || thePlayer.isAlive()) {	for (int x = 1; x < width; x++)	yPos++
Fix	return tileAtOffset(t, dir.get**Dx**(), dir.get**Dy**());	if (tileCanBeOccupied(target) **&&** thePlayer.isAlive()) {	for (int **x = 0** x < width; x++)	**y**++
Detailed Result	All direction keys trigger swapped coordinates (x,y). For example, the right key (0,1) now does (1,0) so hitting the right key makes pacman move down	The player can now move into a tile if it can be occupied OR if the player is alive. This means that the player can always move through tiles. Pacman can walk through walls	Line removal starts counting at the second square in the row, so when a row clears, all elements clear except the leftmost square.	On tetris, hitting the down arrow is supposed to move the piece to the bottom of the screen. This bug makes the change to a temporary value so the piece does not move.

Each of the four bugs altered a small portion of a single line of code. The four bugs can be seen in Table 2. Each of the bugs causes a visible change to the UI. In order to help participants to locate the bugs in the brief time allotted without giving the error away, participants are given vague error messages. This was added after some participants in the preliminary trial run attempted to debug things like "the level does not increment" which was not a feature of the code to begin with. It is a realistic assumption that anyone in a real world setting would have some vague inclination of the nature of the bug when debugging. While every bug has an intended fix, for *Pacman1* participants rarely found it. Instead of editing the method in tileAtDirection, participants tended to go to the Direction class and change the values stored in the enums, swapping all the x and y values. In a work setting,

this would not be an optimal fix, as it is essentially debugging code by adding a second bug that cancels out the first. That being said, this alternative bug fix did cause all unit tests to pass. Participants were told they could move on to the next task once they fixed the bug; thus, a participant who saw all passing unit tests would have no reason to keep debugging and to realize that her solution was not optimal. Additionally, only 15% of those who were given *P1* found the true solution, where 43% found the enum solution. Given these factors, I made the decision to accept this fix as correct.

The bugs were always given in matched pairs. Participants were given either *P1* and *T1* or *P2* and *T2*. The combinations chosen were arbitrary. The choice to pair bugs was made to reduce the number of different zip files and thus to reduce both the overhead of saving and checking double the number of zips and to reduce the potential for accidentally saving files incorrectly.

Versions

There were a total of eight versions. Four of these were a combination of the two bug sets and the two independent variables (with and without the visualization assistance). The remaining four were the counterbalanced alternatives to the original four. This was to reducing priming and fatigue effects. Priming effects could result in participants doing better on the second debugging task because they were already 'warmed up.' Fatigue effects would have the opposite result, where participants do worse on the second debugging task because they are tired or frustrated after the first. Alternating which task came first means that the data should not reflect these effects even if they are present because each task was the second task half of the time.

The conditions can be seen in Table 3. Every version has a unique combination of task order, the program that has the associated visualization, and the bug number. The bug number refers to the numbers used in the previous section. This means that following this table A1 had *pacman* first followed by *tetris*, *pacman* had a visualization with it and Tetris did not, and the participant was given the bugs *Pacman1* and *Tetris1*.

Table 3

The eight condition versions

Version Name	First Task	Visualization Task	Bug Number
A1	Pacman	Pacman	1
A2	Pacman	Pacman	2
B1	Tetris	Pacman	1
B2	Tetris	Pacman	2
C1	Tetris	Tetris	1
C2	Tetris	Tetris	2
D1	Pacman	Tetris	1
D2	Pacman	Tetris	2

Visualizations

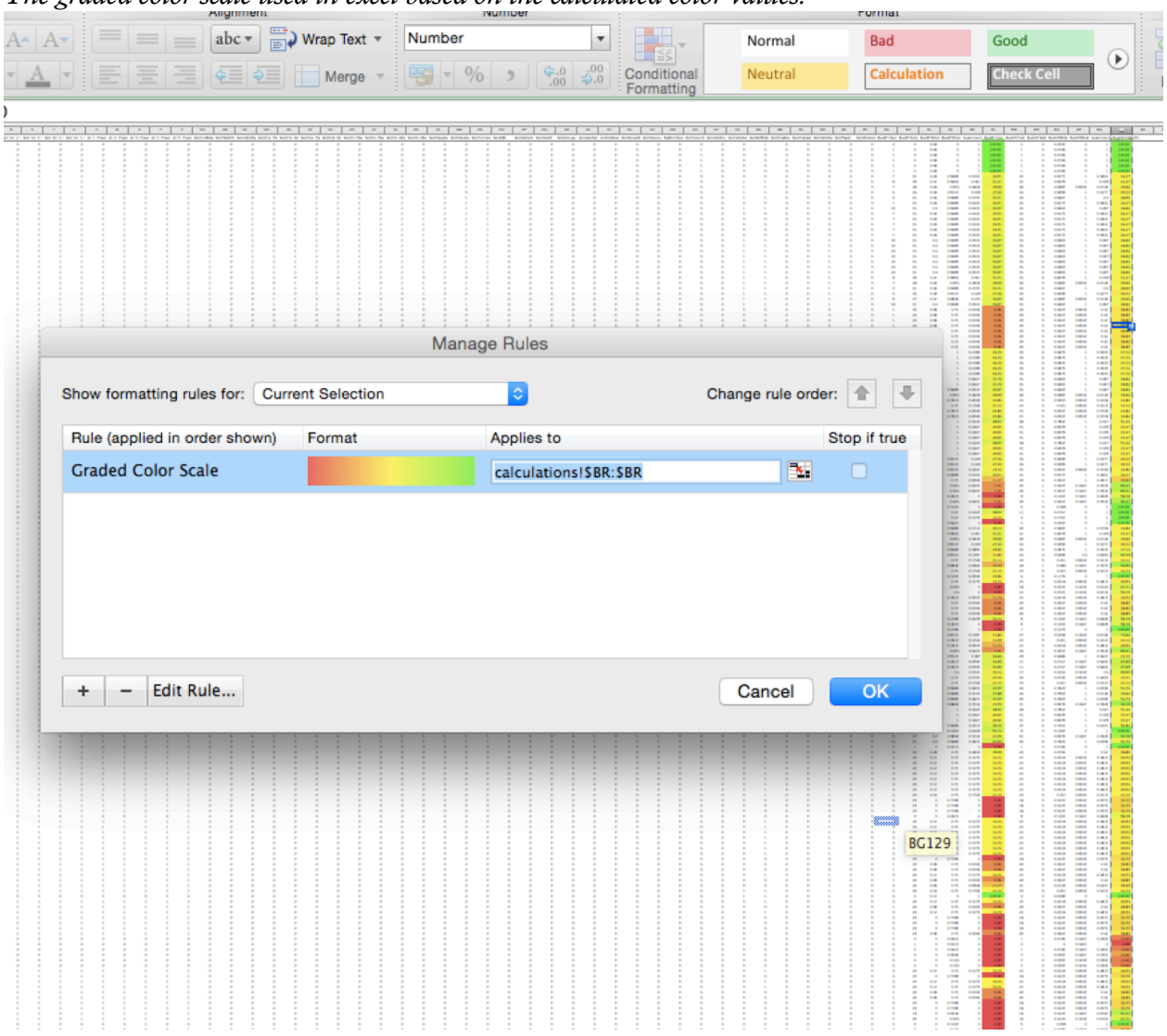
The visualizations were different for each of the four bugs. They were computed in the same way described in the Task Overview section of this paper. Actual computations were done using Excel formulas. Both programs were filled with print statements that contained the class name and the line number for each line of executable code. These print traces were saved to text files for each unit test (*pacman*) or manual test (*tetris*). The pass/fail ratios for each bug were used to compute the correct color values. The number calculated using this formula:

$$\text{color(s)} = \text{low color (red)} + \frac{\%\text{passed(s)}}{\%\text{passed(s)} + \%\text{failed(s)}} * \text{color range}$$

was then given an actual color by using Excel's built in conditional formatting tool, the Graded Color Scale. This method is shown from a zoomed-out view in Figure 6.

Figure 6

The graded color scale used in excel based on the calculated color values.



The visualizations themselves were different for every bug. Figure 7-Figure 10 show zoomed out views of the visualization assistant pdfs for each of the four bugs. The blue rectangle surrounds the line of code where the fault is located. As is shown in the figures, the

Figure 7

Trace of the Pacman1 bug. The actual bug is surrounded by a blue box.

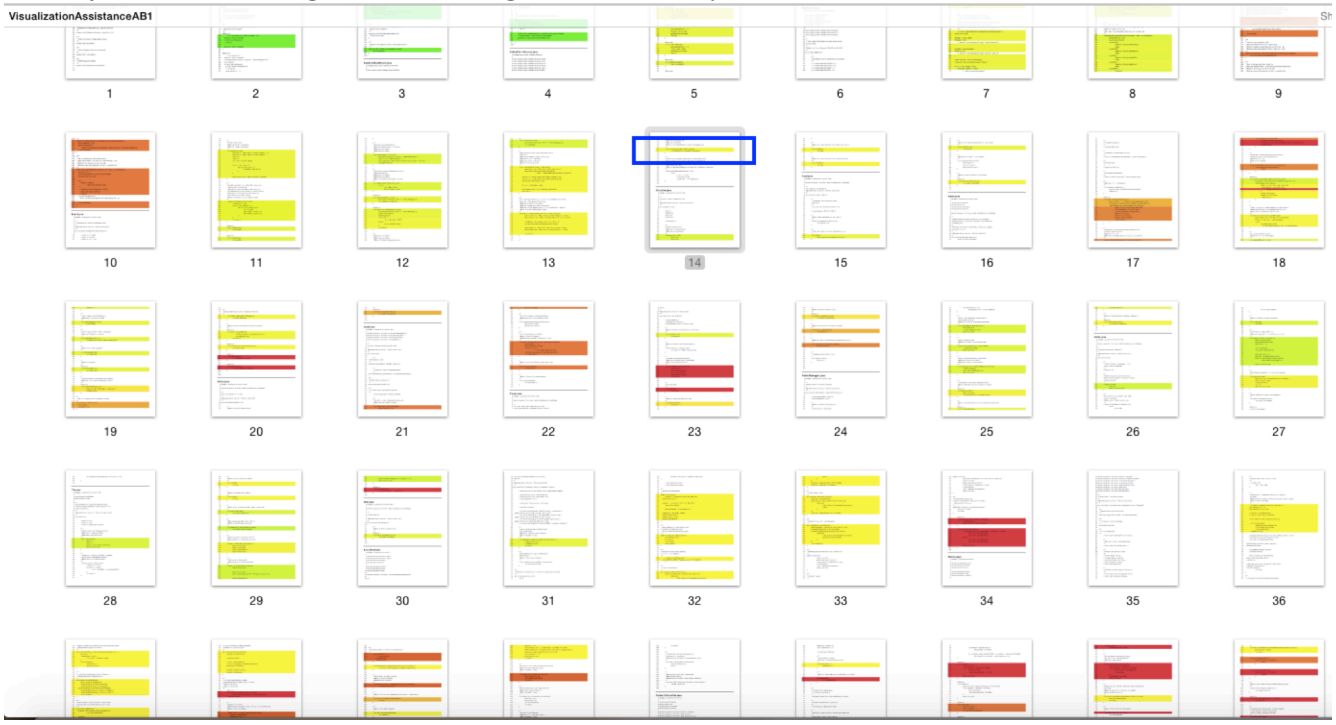


Figure 8

Trace of the Pacman1 bug. The actual bug is surrounded by a blue box.

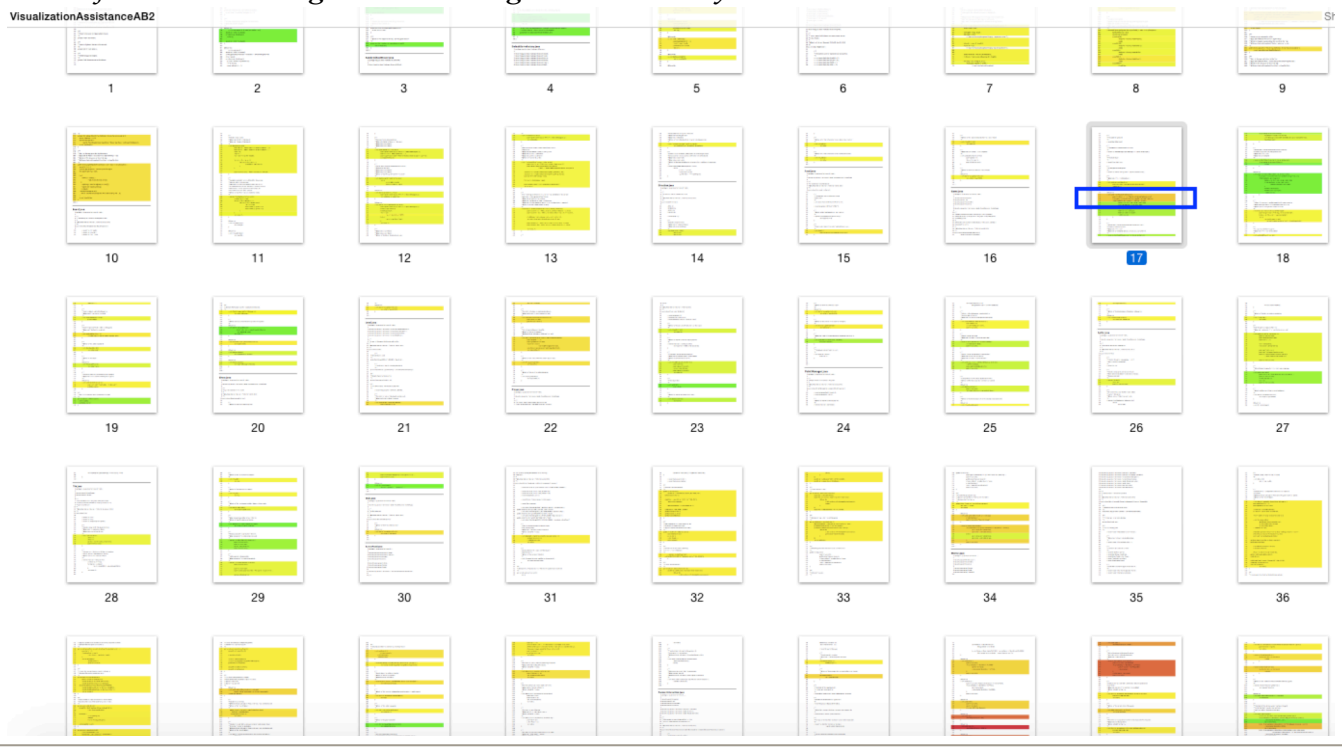


Figure 9

Trace of the Tetris1 bug. The actual bug is surrounded by a blue box.

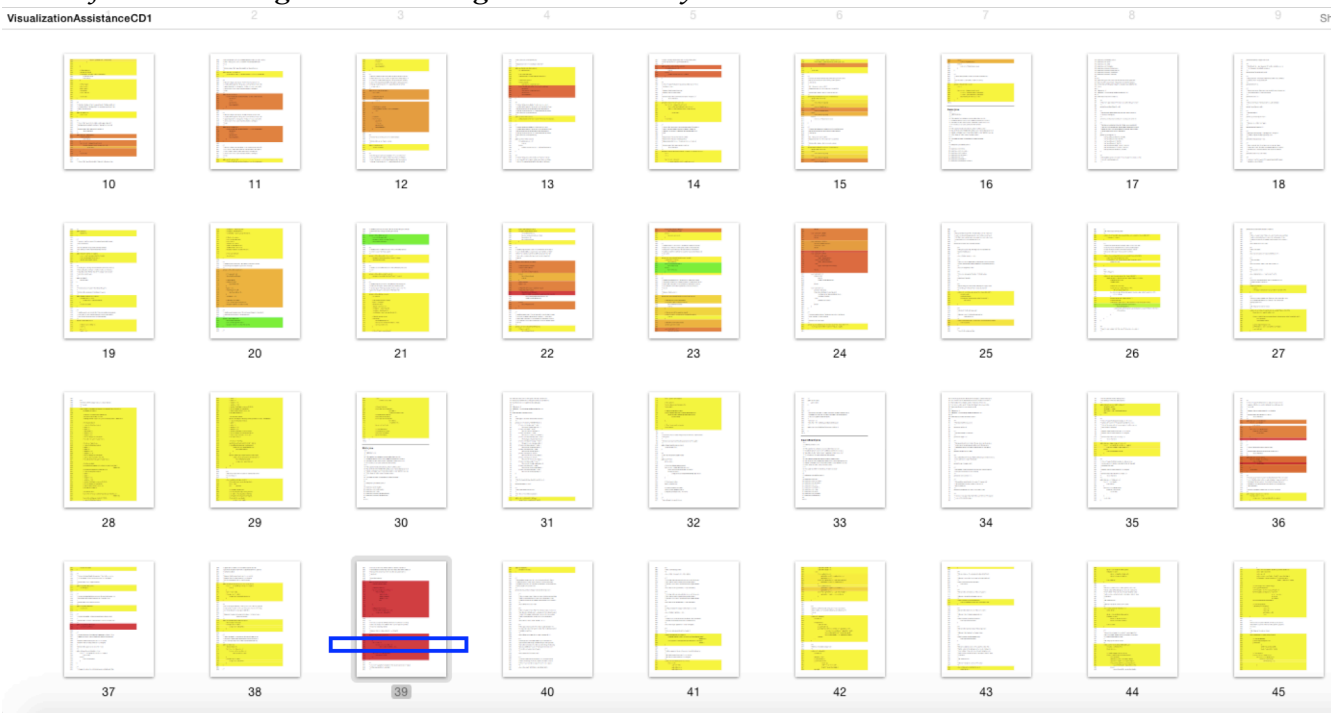
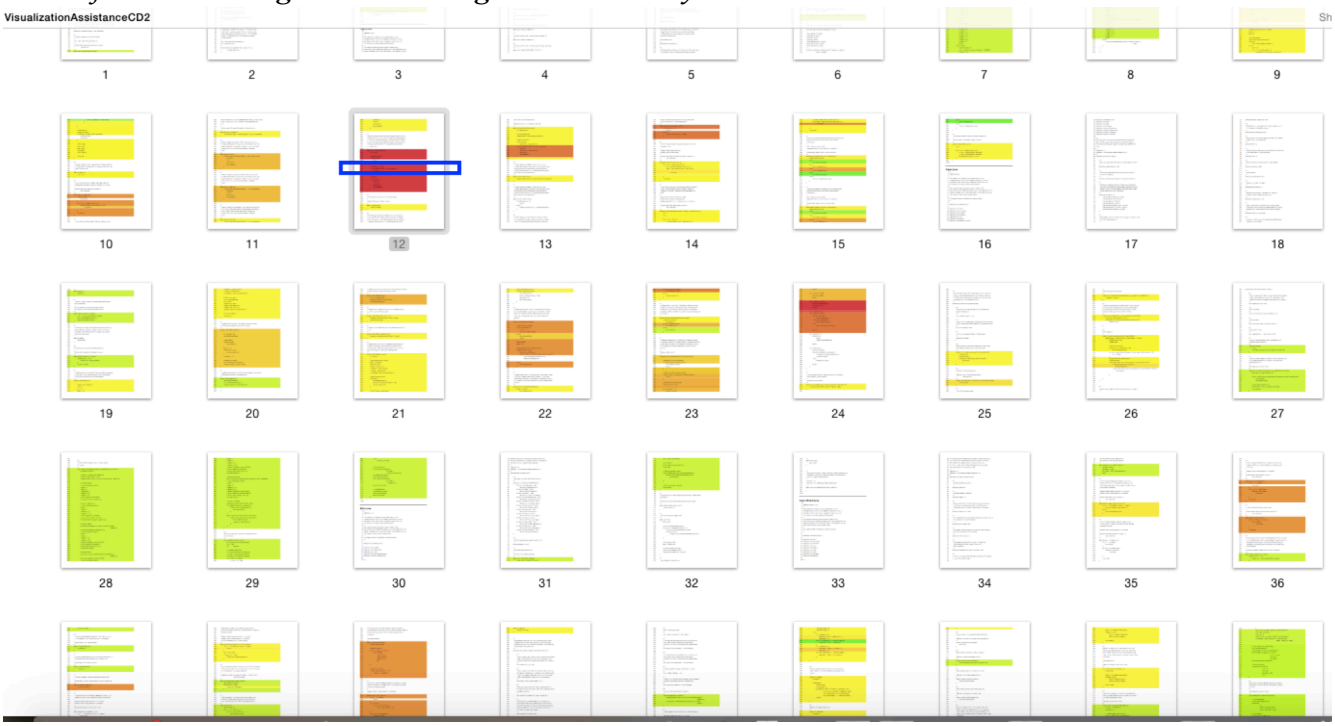


Figure 10

Trace of the Tetris2 bug. The actual bug is surrounded by a blue box.



suspiciousness level was different for different bugs. Specifically, the two *pacman* bugs ranged from yellow to orange, where the *tetris* bugs both appeared in the brightest shade of red. The bugs were chosen prior to seeing the color traces. The selection process based purely on the criteria that the bugs chosen must cause some, but not all, of the unit tests to fail.

The color distribution for every program trace ranged from pure green to pure red. The color differences in the actual bugs was incidentally fortunate, because it provided both the 'ideal case' in which the bug itself was labeled as completely suspicious (red) and the less ideal case where the bug is colored as partially suspicious (yellow/orange).

Implementation

There were two separate versions of the study run. The first was the preliminary trial run, which involved 20 participants and acted as a test of the methods. The second was the final implementation, which was essentially the same as the preliminary trial run but with a few changes made based on results and feedback from the trial.

Difference between the Final Implementation and the Preliminary Trial Run

The preliminary trial run revealed several flaws with the original design. This lead to the addition of the following in the final implementation: bug reports, expected results for key presses, emphasis on recording start times, and recording participant numbers.

Bug reports were added because less than a third of participants in the trial were able to fix bugs they identified, and because their comments and solutions indicated that they were attempting to fix bugs that did not exist. For example, multiple participants tried to fix the

fact that levels did not increment in *tetris* rather than the bug they were given. In order to help guide the participants without giving away details about the bug, they were given vague bug reports that can be seen back in Table 2.

Participants in the trial were also unclear on what the buttons were supposed to do. To solve this, the expected results of the most used key presses: (up, down, left, right, and spacebar) were written on a whiteboard at the front of the computer lab in the final implementation so that participants had enough familiarity with the code to recognize bugs.

We also observed that participants did not do a good job of documenting start times despite the fact that this was listed in their instructions. In the final implementation, this was emphasized verbally at the beginning of the study. Unfortunately, many participants still did not document start times. This was problematic because it reduced the number of data-points that could be used in a comparison of the amount of time spent to find the bugs.

The last addition was to record participant numbers in the questionnaires and on the handouts so that analysis could be conducted comparing factors such as debugging techniques or programming experience on the results. However, due to the issues with documented start time, there were not enough participants who generated correct fixes for further subdivision to reveal useful trends. Additionally, this issue was not found in the pilot; it was found at the end of the first day of sessions so it could only be applied to the second day of sessions.

Independent and Dependent Variables

Independent

The independent variable is the debugging style. The two conditions of this variable are visualization assisted and traditional debugging. This independent variable is considered separately for each of the four possible bugs.

Dependent

The four dependent variables are three recorded times and number fixed. The time dependent variables are calculated separately at each of these points: when the participant locates the correct file, when the participant locates the correct method, and when the participant correctly fixes the bug. The number fixed is a discrete variable with two options: fixed or not fixed. As discussed in the *Bugs* subsection earlier in this section, each bug had one correct fix with the exception of *P1,* which had two correct fixes.

Results and Discussion

Quantitative Results

Table 4

Numbers of participants who were able to fix the bugs compared with the total number of participants assigned to that bug based on condition.

		P1	P2	T1	T2
Fixed	visualization	15	7	10	15
	traditional	14	9	11	11
Total	visualization	27	23	22	22
	traditional	22	22	27	23
Percent	visualization	55.56%	30.43%	45.45%	68.18%
	traditional	63.64%	40.91%	40.74%	47.83%

Fix or Not

The first comparison looked at whether or not participants were able to fix the bug for each of the four different bugs. The numbers are shown in Table 4. For both *pacman* bugs, a larger percentage of participants were able to find the bugs, 64% compared to 56% for *P1* and 41% compared to 30% for *P2*. For *tetris*, the opposite was true with a larger percentage of people in the visualization condition finding the bug, 45% to 41% for *T1* and 68% to 48% for *T2*. These percentages are out of a relatively small number of people since there were 22-27 people in each category.

Time-to-Fix for Each Bug

Differences between conditions were computed separately for each of the four bugs using an unpaired t-test. We chose to use and unpaired test because each participant was given two different bugs. Paired data would imply that the same participant was given the same bug twice, once with visualization and once without. The t-test determines whether the two groups (visualization assisted and traditional) are significantly different from one another. A

difference is considered significant as long as $p<.05$; in other words, less than a 5% chance that the differences between the two groups were due purely to chance.

Looking only at participants who were at least able to find the correct file, Table 5 shows the number of participants per group, the average time to find the correct file, method, and fix, and the computed p-values. The only statistically significant difference between groups was on the *Pacman1* bug ($p=.04$), where the average total time to locate and fix the bug was 19.3 minutes without the visualization and 11.7 minutes with the visualization. The remaining three bugs did not show significant differences as far as fix time is concerned. There were no statistically significant differences between visualization and traditional debugging for the other three bugs ($p>0.4$ meaning that there is at least a 40% chance that differences between groups are not caused by the study variables). This indicates that for these three bugs there are no relevant differences in time-to-fix based on the presence or lack of visualization assistance.

Table 5

For participants who are able to locate the correct file, the number of participants (N), the mean, the standard deviation (STDEV), and the standard error (SE) are calculated for both traditional debugging and debugging with visualization. These values are calculated for the times in minutes from the start time until the location of the correct file, from the start time to the location of the correct method, and from the start time to the location of the correct fix.

Pacman1		Correct File	Correct Method	Correct Fix
With Visualization	N	10	9	9
	Mean	11.4	11.5556	11.6667
	STDEV	8.2892	8.7765	8.6603
	SE	2.6213	2.9255	2.8868
Traditional	N	11	11	11
	Mean	17.0909	19.0909	19.2727
	STDEV	6.8477	6.8184	6.7837
	SE	2.0647	2.0558	2.0454
p-values		0.1014	0.0442	0.0407

Tetris2		Correct File	Correct Method	Correct Fix
With Visualization	N	10	10	8
	Mean	18.1	18.1	18.75
	STDEV	7.8944	7.8944	9.0040
	SE	2.4964	2.4964	3.1834
Traditional	N	7	7	7
	Mean	13.4286	14	15.1429
	STDEV	7.1381	6.9761	7.4258
	SE	2.6979	2.6367	2.8067
p-values		0.2315	0.2873	0.4169

Pacman2		Correct File	Correct Method	Correct Fix
With Visualization	N	4	4	4
	Mean	16	16	16
	STDEV	8.9815	8.9815	8.9815
	SE	4.4907	4.4907	4.4907
Traditional	N	11	10	8
	Mean	14.0909	14	14.375
	STDEV	10.1139	9.1409	9.5160
	SE	3.0495	2.8906	3.3644
p-values		0.7456	0.7168	0.7825

Tetris2		Correct File	Correct Method	Correct Fix
With Visualization	N	13	13	12
	Mean	14.3077	15.3077	16.5833
	STDEV	6.3821	5.4065	6.4450
	SE	1.7701	1.4995	1.8605
Traditional	N	10	9	7
	Mean	12	11.7778	19.7143
	STDEV	7.0396	5.9325	9.6560
	SE	2.2261	1.9775	3.6496
p-values		0.4201	0.1632	0.4064

Finding the File but not a Fix

There were a total of 10 participants were able to find the first file within the half hour time limit but were not able to find the correct solution. Of these, 6 were using traditional debugging and 4 were using the visualization. There is insufficient data present to do a

statistical analysis of this, but at least in this instance, fewer participants who were using the visualization were unable to fix the bug after locating the correct file than with traditional debugging.

Find-to-Fix Times

For all four bugs, the average time between locating the correct file and correctly fixing the bug is shorter for participants who used the visualization than for those who did not as shown in Table 6. This finding was only marginally significant with p=.10; p-values between .05 and .10 can be considered marginally significant for small data sets. There were only 34 participants total in each condition (traditional versus visualization) across all bugs. With such a sample size, that it is likely that it would likely be significant given a larger sample.

Table 6

The differences between the average time to locate the correct file and the average time to find the correct fix for all 4 bugs with and without visualizations

		Time to Find Correct File	Time to Find Correct Fix	Difference
P1	Visualization	11.4	11.6667	0.2667
	Traditional	17.0909	19.2727	2.1818
P2	Visualization	16	16	0.0000
	Traditional	14.0909	14.375	0.2841
T1	Visualization	18.1	18.75	0.6500
	Traditional	13.4286	15.1429	1.7143
T2	Visualization	14.3077	16.5833	2.2756
	Traditional	12	19.7143	7.7143

Of the 34 participants who fixed bugs using traditional debugging, 6 had a gap that was larger than 5 minutes between the time they located the correct file and the time they fixed the bug. Of the 34 participants who fixed bugs using the visualization assistance, 2 had more than a 5-minute gap between the time they located the correct file and the time they fixed

the bug. Across all bugs, traditional debugging took an average of 2.9 minutes from file-to-fix and the visualization condition took an average of 1.2 minutes from file to fix.

If the three outliers (participants who took 20+ minutes to fix the bug after locating the correct file) are removed, the values are statistically significant (p=.03). In this case, the average time from file-to-fix for traditional is 1.6 minutes and for visualization it is 0.4 minutes. Outliers affect datasets less with a larger participant pool, so the significant result without outliers supports the notion that the difference between conditions would be significant.

Qualitative Results

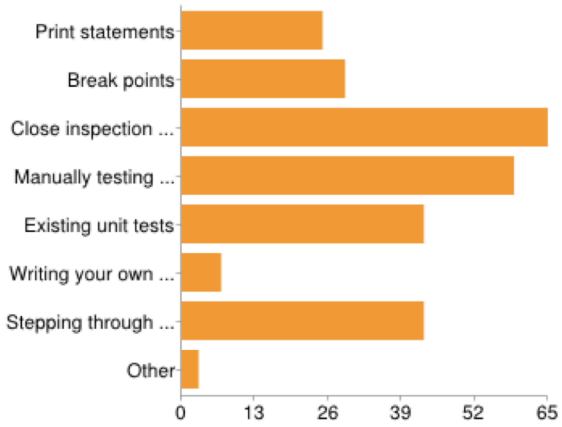
Qualitative results are computed from the results of the post questionnaire. The questionnaire asked about debugging techniques that were used for each of the trials and then asked participants to supply a value from 1 to 5 for questions like "Finding the fault was easier with" and answers ranging from "1 Visualization Assisted" to "5 Traditional". This set up was done to help prevent bias due to question phrasing.

Debugging techniques for traditional and visualization assisted from the 112 survey respondents are shown side by side in Figure 11. Print statements, break points, and stepping through the code were used more during traditional debugging than with the visualization assistance: 25 to 14, 29 to 14, and 43 to 37 respectively. Close inspection of the

Figure 11

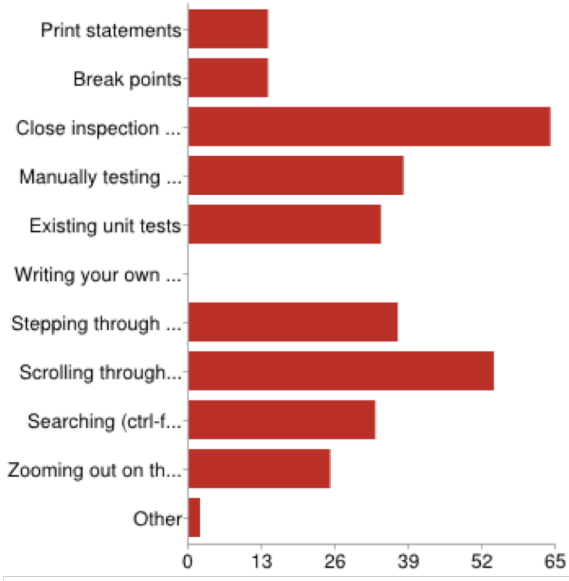
Participant responses when asked what techniques they used while debugging.

Traditionally



Print statements	**25**	22%
Break points	**29**	26%
Close inspection of code (reading for bugs)	**65**	58%
Manually testing input	**59**	53%
Existing unit tests	**43**	38%
Writing your own unit tests	**7**	6%
Stepping through the code	**43**	38%
Other	**3**	3%

With Visualization Assistance



Print statements	**14**	13%
Break points	**14**	13%
Close inspection of code (reading for bugs)	**64**	57%
Manually testing input	**38**	34%
Existing unit tests	**34**	30%
Writing your own unit tests	**0**	0%
Stepping through the code	**37**	33%
Scrolling through the Visualization pdf	**54**	48%
Searching (ctrl-f) the Visualization pdf	**33**	29%
Zooming out on the Visualization pdf	**25**	22%
Other		

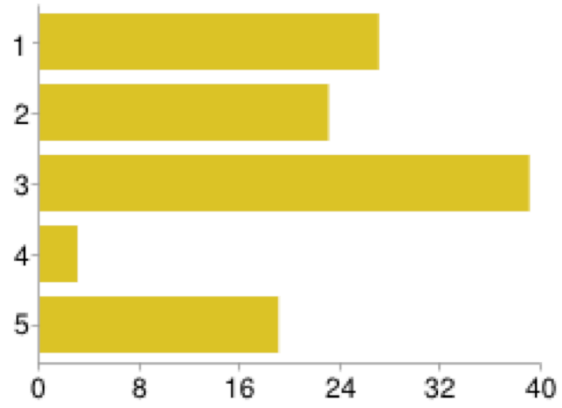
code was consistent for both groups with 64-65 people stating that they used this technique. Unit test were also relied on more in traditional debugging, 43 to 34 for existing test and 7 to 0 for writing their own tests. For those who answered "Other", the items listed were "common sense," "exploratory testing," and "using the *tetris* test spec document."

Still in Figure 11, the results specific to the visualization assistance showed that 54 participants scrolled through the visualization, 33 conducted a direct search, and 25 zoomed out the view.

The remaining four Likert-style questions had the following results. As shown in Figure 13,

Figure 13

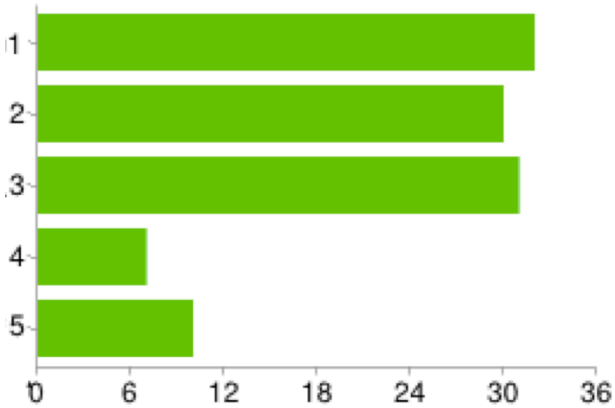
Survey results (ranked 1 to 5) for "Finding the fault was easier with:"



1 Visualization Assisted	**27**	24%
2	**23**	21%
3 They were the same	**39**	35%
4	**3**	3%
5 Traditional	**19**	17%

Figure 12

Survey results (ranked 1 to 5) for "The visualizations were:"



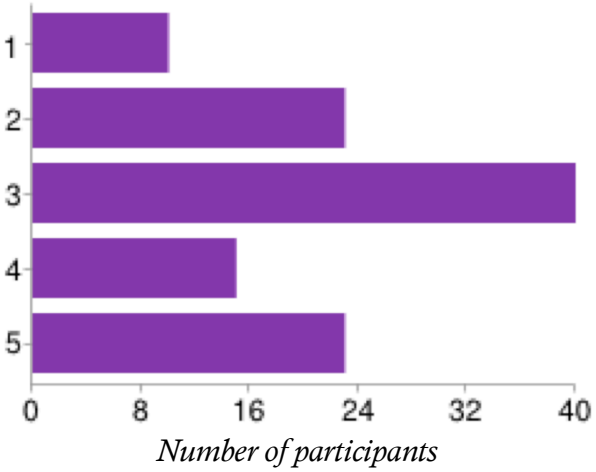
1 Helpful	**32**	29%
2	**30**	27%
3 Neither helpful nor unhelpful	**31**	28%
4	**7**	6%
5 Unhelpful	**10**	9%

45% of participants felt that finding the fault was easier with visualization assistance, 20% thought it was easier traditionally and 35% though that they were the same. As shown in Figure 12, 56% said that the visualizations were helpful, and 28% said they were neither helpful nor unhelpful. As shown in Figure 14, 30% of participants thought that the visualizations were difficult to understand, 34% said that they were easy to understand, and 36% were in the middle. As shown in Figure 15, 44% of participants said they would choose to use the visualization assistance again, 16% said they would not choose to use the visualization assistance again, and 38% stated that they would maybe choose to use it again.

Subjective feedback was collected explicitly via the "Do you have any other comments to

Figure 14

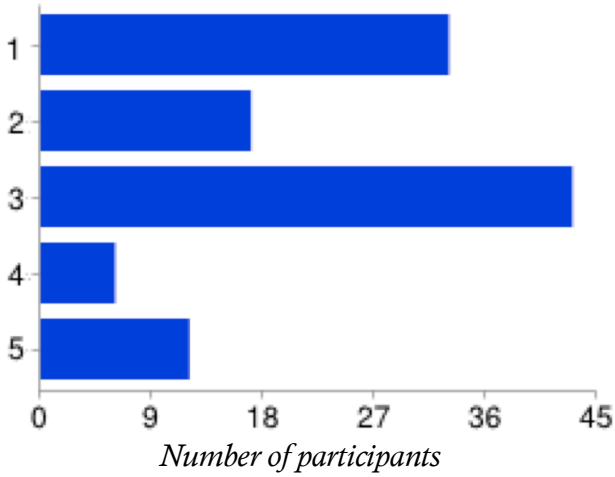
Survey results (ranked 1 to 5) for "The visualizations were:"



1 Difficult to understand	**10**	9%
2	**23**	21%
3 Neither difficult nor easy to understand	**40**	36%
4	**15**	13%
5 Easy to understand	**23**	21%

Figure 15

Survey results for "I would _____ to use the visualizations again:"



1 Choose	**33**	29%
2	**17**	15%
3 Maybe choose	**43**	38%
4	**6**	5%
5 Not choose	**12**	11%

add?" free answer question at the end of the survey and implicitly from comments participants made in person as they returned their handout.

It included a range of positive and negative comments about the visualization including the following excerpts:

- "I only used the visualization assistance a couple times because I was mostly lazy to search through each folder and find the specific method I wanted to inspect."
- "The visualization did help more even though I found the *pacman* bug quicker. Maybe the *pacman* bug was simpler or I got lucky finding it."
- "Visual assistance helps no doubt. But more than that a unit test suite helps"
- "Surprisingly useful. If this is a real program, I'll gladly use it in the future."
- "Two cases were not similar in the size and also style of coding"
- "I was able to find the error through executing the code. Utilizing the pdf file would have been effective had the bug been more profound in nature."
- "Visualizations helped to narrow down where the problem area was with a big code base. Traditional methods worked for Tetris since the issue was clear and only a few methods were involved. My only reservation with the visualization was that to some degree the colors drew attention to areas that the bug wasn't actually in. By this I mean the bug was in a "green" area, and my first instinct was to carefully examine all red code. But examining the red code led me to find the relevant green code block, which allowed me to find the bug."
- "The visualization didn't add anything. It felt like extraneous info."

- "[The visualization's] main use for me was to confirm problem areas when I found them via other methods."

and some more general comments such as:

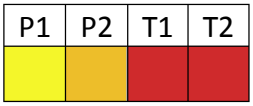
- Tetris was hard to debug because they were not good enough at playing to find the bug
- They have only ever debugged their own code and found debugging unfamiliar code to be strange or challenging
- Ran out of time
- Did not like how long it took to set up the workspace
- Would prefer that the visualization tool were a plugin
- Did not have enough experience coding to debug

Discussion

Starting with the quantitative results, a larger percentage of participants were successful at debugging *pacman* traditionally and at debugging *tetris* with the visualization assistance. This could be due to a number of factors. First, the suspiciousness colors for *tetris* bugs were completely red, whereas the suspiciousness colors for the *pacman* bugs were yellow and light orange as shown in Figure 16. It would make sense that more people would be able to find the bug with a visualization that pointed directly to the bug. Participants who were given visualizations for *pacman* might have used up more of their time investigating red lines of code that did not contain the

Figure 16

Suspiciousness color assigned for each bug at the line of the bug



bug. Since participants were given up to 30 minutes to find and fix the bugs, it is possible that they would have found the bug if given more time.

A larger percentage of participants who had the visualizations with *tetris* were able to fix the bug than those using the visualization with *pacman*. This might imply that redder bugs are easier to find with the visualization. However, when considering how long it took to fix the bug, the only bug with significant differences between fix times for visualization versus tradition was *P1*, which is the yellowest bug, as shown in Figure 16. For the yellowest bug, participants were able to fix the bug on average 7.6 minutes faster with the help of the visualization. This was also the bug with the alternative solution of swapping the x,y values in the Direction enum. The enum itself showed up in white on the visualization, since enums are not shown in the trace. It is possible that by looking over the red blocks of code, participants could quickly see that references to the direction enum were associated with failures, thus allowing participants to hypothesize that the enum was the cause.

For the remaining three bugs, no significant differences were found. This implies that while the visualization will not significantly increase the time spent debugging, it will only sometimes make debugging faster. It is hard to extrapolate from null results. It is possible that the visualization does not actually improve debugging time except in very specific circumstances. However, it is also possible that these numbers could completely change if participants were asked to work until they found a solution rather than stopping after 30 minutes. Since only 30-68% of participants were able to fix bugs within the half hour limit

(depending on condition and bug), the results could change fairly dramatically with that design.

The most consistent difference between the visualization and the traditional conditions was the find-to-fix trend — the difference in the time it took from finding the correct file to finding the correct fix. Though there were not significant differences in overall fix times for most of the bugs, for all four of the bugs, those in the visualization condition spent less time on average between finding the file and fixing the bug. It is reasonable to assume that this difference in time indicates that participants had a greater understanding of the fault and its location by the time they located the correct class file and were thus able to fix the fault more quickly once they reached this point. This might indicate that the real benefit of the visualization is not so much faster fault localization, but greater understanding of the fault. In more complex code (or code with more complex bugs than were presented in this study), it is possible that having a better understanding of the fault would have a greater impact on how quickly participants were able to fix the bug.

This would make sense, since using the visualization involves starting in the reddest areas, then moving back if it seems as though bug earlier in the code can cascade, causing later lines of code to result in test failures. By the time participants reach the correct method, they have seen a larger amount of code that is either directly or indirectly relevant to the bug. This experience would lead to an increased understanding of how the bug influences the code and would likely improve the transition from finding its general location to actually fixing it.

As far as the qualitative results are concerned, participants showed the expected affinity towards traditional debugging techniques such as close inspection of the code, manually testing input, and stepping through the code. Yet, in all cases (except close inspection of the code) fewer participants used these techniques when using the visualization assistance. Instead, they scrolled through the visualization.

All questions in the survey were listed as optional, so not all participants answered regarding the techniques they used, but of those that answered, only about 50% claimed to have used the visualization PDF. Some participants claimed to only use a single debugging technique (where most other participants used 3 or more techniques each) so it is possible that not all participants realized that they were able to select multiple items for the question and only cited the technique they relied most heavily upon (usually "close inspection of the code"). However, some participants only listed scrolling through the visualization PDF. With the data available, it is not possible to determine the reason that not all participants used the visualization tool.

Searching the PDF and zooming out to see more of the code at once in the visualization condition was used with similar frequency to print statements and break points in the traditional condition, implying reasonable adoption of more profound feature use than just scrolling.

As far as the Likert questions are concerned, there was always a group of 9-17% who opted for traditional debugging, found the visualizations to be unhelpful or difficult to understand, or who would not choose to use the visualization again by selecting the most extreme value (1 or 5 depending on the question). Otherwise, the overall skew seemed to indicate positive feelings towards fault localization, its helpfulness, and choice to use the visualization again.

As far as the subjective comments provided by participants, it seems as though there were some who found the visualization helpful and some who did not. It was interesting that some people chose to explicitly mention that the visualization made debugging easier even though it took them more time or that they would have used it more if they had a more profound bug to solve. Some of the general feedback was related to issues like the fact it was not a plugin or how long the workspace took to set up and were not necessarily related to the results of the study itself. It also showed that perhaps a few of the participants were too inexperienced with programming to be ideal participants for this exercise despite the fact that they were all either Software Engineering Masters students or undergraduates enrolled in a software testing course. While some participants clearly did not like the visualization, others seemed to find it useful.

With these results in mind, we can analyze the Tarantula-based debugging technique against the four key properties of a useful debugging tool that were discerned from related works in the first section of this paper:

1. Reduce the problem space without actually removing any code
2. Maintain the structural integrity of the program

3. The tool should either explicitly or implicitly help developers to create hypotheses
4. The tool should help developers to confirm or reject hypotheses they form

The first pair of properties is related to maintaining context in order to avoid the *Isolation Flaw*. The second pair deals with the hypothesis heuristic that people use to debug code.

As already discussed in the first section, Tarantula's visualization style takes into account (1) and (2) by leaving the program completely intact and simply overlaying colors on the existing code.

The greatest evidence for property 3 is the reduced time differences between locating the file and fixing the bug when using the visualization because it implies that by the time participants reached the correct file, they had a better understanding of the problem and solution and were able to more quickly solve the problem at this point. This is further supported by one participant who explicitly described using the visualization to assist in backwards navigation of the code: "examining the red code led me to find the relevant green code block, which allowed me to find the bug. It is likely that the mechanism by which examining the red code was able to "lead" someone to green code was because the contextual information it provided assisted in the creation hypotheses about what other portions of code were causing those blocks to fail.

Property 4 is difficult to evaluate from the quantitative data. It could be interpreted based on the number of hypotheses recorded by each participant, but almost all participants wrote only one or two hypotheses. It is unclear whether this was because participants were able to

locate the bug on a single hypothesis or if participants simply did not write down every idea that occurred to them. I suspect the latter. One reason I believe this is based on personal experience with programming interviews. For these interviews, one of the most commonly reinforced pieces of advice is for interviewees to talk through their logic as they code. Given how much this needs to be emphasized, I believe that accurately verbalizing or recording every hypothesis does not come naturally and is mostly extracted through deliberate effort. In that case, a better method for determining how useful the tool was for hypothesis checking might be simply asking participants for their opinions on the matter. This was not a question on the survey, so data is not available regarding how many participants felt that the tool was useful for hypothesis checking. One participant made the following comment regarding the visualization: "Its main use for me was to confirm problem areas when I found them via other methods." This feedback suggests that the Tarantula-style visualization used in this study exhibits property 4, but more data would be needed to generalize this.

Threats to Validity

Construct Validity

The amount of time it took participants to fix the bugs as well as simply whether or not people were able to find the bugs was used to measure whether or not the visualization improved performance. This are fairly accurate measures of performance when the concern is the amount of time it takes to debug and the ultimate goal of debugging. The limiting factor here is the 30 minute time limit. These constructs might be more generalizable if participants were asked to continue until they completed they task.

The independent variable looked at difference between debugging traditionally and debugging with a visualization. Traditional debugging should be very generalizable since the code was written in Java, a commonly used language, and debugging was done in Eclipse, a common IDE. The visualization was specifically Tarantula-based and may not generalize to other automated debugging techniques or to other visualizations.

Internal Validity

Potential subject bias may have occurred since students were obtained from classes taught by my advisor. However, the free response questions seemed to show a range of positive and negative feedback that seemed to indicate honest feedback. The study used experimental design with counterbalancing to reduce error. Some potential sources of error may have come from participants in the visualization condition choosing to ignore the visualization or from the possibility that some participants were colorblind. Additionally, one of the tasks did not load well on some of the school computers, requiring participants to

import JAR files. This may have increased frustration of fatigue and may have affected the results. Finally, there was the issue that not all participants recorded start times, resulting in missing data about how long tasks took to complete. It is possible that results would be very different with this additional data.

External Validity

The study included only students, limiting the generalizability. All students were from graduate or upper-division undergraduate courses, which makes them only a year or two estranged from being members of the workforce. Especially given that participants had an average of 1 year of professional experience, it is likely that these results are generalizable to the equivalent level of experience of those in entry-level job positions. However, this cannot be determined with certainty unless a study on that population is conducted.

Furthermore, all tasks were under 5,000 lines of code, used java, and were games with visual output, limiting the generalizability to only programs that meet these same requirements. Since only two programs were evaluated, it is unknown whether the results are fully representative, even for programs with these restrictions.

Future Work

If this study were to be conducted again, it might be worthwhile to ask participants to keep going until they find the bug. However, not everyone will necessarily be able to find the bug and there are difficulties with incentivizing such a study. Additionally, improvements could be made to try to increase the use of the visualization. Multiple monitors might encourage use of the tool since code could be open on one side with the visualization open on the other. Also, the PDF opened at full size. If there were a way for the visualization to open fully zoomed out so that an overview of the code is visible, that would likely provide a better overview of the system and allow participants to focus in more quickly than by scrolling. Based on one comment that described the bug as "green", future studies should also be careful to either screen for color blindness or provide an option that will be visible for those who have it.

Technical improvements to this study could implement the visualization as an Eclipse plugin, as was suggested by a number of participants. It would be interesting to see how (and if) results change if the visualization was presented in a plugin rather than a pdf. It would likely reduce context switching and the time cost of alternating between views.

Future work should definitely consider the vast diversity of bug finding situations. This study used programs with two to three thousand lines of code that participants were not familiar with and which produced visual output (games). Future studies could compare different code lengths, code with text output, code without output, or code written by the users (for example a case study done in a work environment). Additionally, this study

focused on a student population. Results might be different for more expert programmers. Other areas of study might include the effect of multiple bugs.

More theoretical future work could investigate how different visualizations or automated testing tools are able to deal specifically with the *Isolation Flaw*, and how they can best accommodate the hypothesis model used in program testing. Another area of potential investigation is better ways of quantifying properties that make a debugging tool genuinely useful.

Conclusion

This thesis investigated the differences between traditional and visually assisted debugging sessions using an experimental design. The visualizations provided to participants were based on the methods used for the Tarantula program [4] and consisted of assigning color-coded suspiciousness values to every line of code executed by the unit tests.

The research sought to better understand how the presence of the visualization tool changes the way that people debug. It was concerned specifically with the *Isolation Flaw*, a common problem for debugging tools that isolate suspicious code to the point that they remove necessary contextual information. Based on this flaw and other surrounding literature it was determined that a useful debugging tool should have the following feature set:

1. Reduce the problem space without actually removing any code
2. Maintain the structural integrity of the program
3. The tool should either explicitly or implicitly help developers to create hypotheses
4. The tool should help developers to confirm or reject hypotheses they form

Both quantitative and qualitative data were collected. The quantitative data for each bug included the number of participants who were able to fix it, the time it took to: find the correct file, find the correct method, and correctly fix the bug, and the differences between these times. The qualitative data was collected from pre and post questionnaires and included descriptive statistics about the participants and participant opinions on traditional and visually assisted debugging.

The results showed that visually assisted debugging can sometimes increase the speed of debugging, but that with most of the bugs it did not. From the results analyzed, there was nowhere in the study that the visualization assistance decrease the speed of debugging significantly, implying that it sometimes helps but did not hurt, at least as far as the bugs in this study were concerned. Additionally, the time between finding the correct file and finding the correct fix were shorter with visually assisted debugging than with traditional debugging for all four bugs.

The results implied that, even though visualization did not always improve the speed of debugging, it did a fairly good job of meeting the four requirements for a useful debugging tool. For the first two points, the Tarantula style visualization did not remove any context or structure from the original code. For the second two points, between qualitative participant feedback and the quantitative results that revealed the shorter times between locating the correct file and fixing the bugs, it seems as though the visualization tool helped participants to gain a better understanding of the fault and to apply that knowledge to creating better hypotheses while debugging.

References

[1] Stasko, J.T., Brown, M.H., and Price, B.A. 1998. Software Visualization Programming as a Multimedia Experience. Cambridge, Mass.: MIT Press/

[2] Ghanam, Y. and Carpendale, S. 2008. A survey paper on software architecture visualization. Technical Report, University of Calgary, pages 1–10.

[3] Tassey G. 2002. *The economic impacts of inadequate infrastructure for software testing.* National Institute of Standards and Technology, RTI Project Number 7007.011.

[4] Jones, J., Harrold, M.J., and Stasko, J. 2002. Visualization of test information to assist fault localization. *Proceedings of the International Conference on Software Engineering (ICSE 02),* pages 467–477.

[5] Parnin, C. and Orso, A. 2011. Are automated debugging techniques actually helping programmers? *Proceedings of the 2011 International Symposium on Software Testing and Analysis. (ISSTA '11)* Pages 199-209

[6] Weiser, M. 1981. Program slicing. Proceedings of the International Conference on Software Engineering *(ICSE 81),* pages 439–449.

[7] Weiser, M. 1984. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357.

[8] Zhang X., Gupta, N. and Gupta, R. 2006. Pruning dynamic slices with confidence. *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 06),* pages 169–180.

[9] DeMillo, R.A., Pan, H. and Spafford, E.H. 1996. Critical slicing for software fault localization. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 96)*, pages 121–134.

[10] Gyimothy, T., Beszedes, A. and Forgacs, I. 1999. An efficient relevant slicing method for debugging. *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 99),* pages 303–321,

[11] Zhang, X., Gupta, R. and Zhang, Y. 2003. Precise dynamic slicing algorithms. *Proceedings of the International Conference on Software Engineering (ICSE 03)*, pages 319–329.

[12] Zeller, A. 2009. Why Programs Fail: A Guide to Systematic Debugging. 2nd ed. Burlington, Mass.: Morgan Kaufmann.

[13] Reps, T., Ball, T., Das, M. and Larus, J. 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 97)*, pages 432–449.

[14] Groce, A., Kroening, D., and Lerda, F. 2004. Understanding counterexamples with explain. *Proceedings of the International Conference on Computer-Aided Verification (CAV 04)*, pages 453–456.

[15] Ball, T. Naik, M. and Rajamani, S.K. 2003. From symptom to cause: localizing errors in counterexample traces. *Proceedings of the Symposium on Principles of Programming Languages (POPL 03),* pages 97–105.

[16] Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E., 2002. Pinpoint: problem determination in large, dynamic internet services. *Proceedings of DSN'02, Washington, DC, USA, June 2002, IEEE Computer Society*, pp. 595–604.

[17] Abreu, R. Zoeteweij, P., Golsteijn, R., and van Gemund, A.J.C.2009. A practical evaluation of spectrum-based fault localization. *The Journal of Systems and Software* 82:1780–1792

[18] Liu, C., Yan, X., Fei, L., Han, J., and S. P. 2005. Midkiff. SOBER: Statistical model-based bug localization. *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 05*), pages 286–295.

[19] Liblit, B., Naik, M., Zheng, A.X., Aiken, A., and Jordan, M.I. 2005. Scalable statistical bug isolation. *Proceedings of the Conference on Programming Language Design and Implementation* (PLDI 2005).

[20] Hao, D., Pan, Y., Zhang, L., Zhao, W., Mei, H., and Sun, J. 2005. A similarity-aware approach to testing based fault localization. *Proceedings of the International Conference on Automated Software Engineering (ASE 05)*, pages 291–294.

[21] Jones, J.A. Harrold, M.J. and Bowring, J.F. 2007. Debugging in parallel. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 07), pages 16–26.

[22] Liu, C. and Han, J. 2006. Failure proximity: A fault localization-based approach. *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE 06)*, pages 286–295.

[23] Renieris, M. and Reiss, S. 2003. Fault localization with nearest neighbor queries. *Proceedings of the International Conference on Automated Software Engineering (ASE 03),* pages 30–39.

[24] Ko A.J. & Myers B.A. 2004. Designing the Whyline: a debugging interface for asking questions about program failures. *ACM Conf. on Human Factors in Computing Systems (CHI),* 151-158.

[25] Ko, A., J., & Myers, B., A. (2009). Finding causes of program output with the Java Whyline. *In proc. of SIGCHI Conference on Human Factors in Computing Systems*, pp. 1569-1578.

[26] Winslow, L. 1996. *Programming pedagogy – a psychological overview*. SIGCSE Bulletin 28, 17-22.

[27] van Deursen, A. 2003. jpacman-framework. Retrieved from: https://github.com/avandeursen/jpacman-framework-v5

[28] Cederberg, P. 1994. Tetris. Retrieved from: http://www.percederberg.net/games/tetris/