Solo Programming vs Pair Programming: Strategies for Debugging

by

Zhe Zhao

(Under the direction of Eileen T. Kraemer)

Abstract

Program debugging is a process to locate and fix the bugs or defects responsible for a symptom violation in a computer program, thus making it behave as expected. Pair programming is a methodology in which two programmers share the same employed device and environment, collaboratively working on the same design, algorithm, code, and test. Most academic effort in pair programming has been spent on how pair programmers design a program system and how they implement it, rather than on how they debug it. In our study, we recruited two kinds of groups, solo and pair, to perform a program-debugging task in a time-restricted lab session. Based on our collected study data, we carefully examined the performance and strategy differences between the solo and pair programmer. We found evidence that working collaboratively may more efficient, not only when programming, but also when debugging. Further more, we present several methodologies, that can be adopted during academic practice to help students improve their debugging skills.

Index words: pair programming, pair debugging, debugging strategy, testing

Solo Programming vs Pair Programming: Strategies for Debugging

by

Zhe Zhao

B.S., Beijing Institute of Tech., 2003

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Athens, Georgia

2014

Solo Programming vs Pair Programming: Strategies for Debugging

by

Zhe Zhao

Approved:

Major Professors:	Eileen T. Kraemer
Committee:	Tianming Liu
	John A. Miller

Electronic Version Approved:

Julie Coffield
Interim Dean of the Graduate School
The University of Georgia
August 2014

Solo Programming vs Pair Programming: Strategies for Debugging

Zhe Zhao

July 24, 2014

DEDICATION

To my parents

Acknowledgments

First of all, I would like to acknowledge all the friends who were always there for me, and helped me go through all the tough time in my graduate study and life.

Foremost, I would like to express my sincere gratitude to my advisor Dr. Eileen Kraemer for her continuous support of my study and research, for her patience, and immense knowledge.

Also, I am so deeply grateful for the support and help from Mr. Micah Cooper. It was an enjoyable three years working together with him in the Office of International Education.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support throughout my years of life. This accomplishment would not have been possible without them. Thank you.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer programming is a sophisticated process to systematically solve a real-world problem by computer. It usually involves activities such as understanding the original problem, designing a model of the problem, implementing an executable program, and testing the program based on the original design [1].

Good program design and implementation methodologies are consistently emphasized in the entire programming procedure, since good design and deliberate implementation can reduce the cost of producing and maintaining a software system. In another words, a good design and implementation leads to a more reliable and maintainable program.

Proposed by the Extreme Programming methodology research, different research groups across the entire world have studied pair programming over the past several decades [2, 3]. Numerous studies have testified that most programmers prefer to work with another programmer together in the programming tasks, e.g. design, implementation or debugging. They enjoy the task more and are more confident about their solution when programming in pairs than programming alone [4, 5]. Based on benefits discovered from these previous studies, more and more study results have been published about the following issues:

- How Pair programming helps the programmer produce higher quality programs, and makes their work more enjoyable [6].

- Pair Programming helps the student design more robust and stable programs in academic practice [7].

- Adoption of Pair programming into classroom practices improves student programming abilities [8, 9].

Our study mainly focuses on program debugging. The participants in this study work on a program-debugging task during a course lab session. Then the study data is carefully examined to define the performance and strategy when the programmers are debugging a buggy program.

The rest of the thesis is organized as follows. In Chapter 2, we performed a study survey on the related topics of pair programming, pair debugging and debugging strategies. Chapter 3 describes how we planned and designed of our study. Chapter 4 analyzes the results of the study. Chapter 5 summarizes our conclusion drawn from the study and presents the future work.

Chapter 2

Related Work

2.1 Pair Programming

The idea of pair programming is not a brand new idea, but can be traced back to the early 1970s[10]. However, it did not generate widespread interest until eXtreme Programming introduced it as a practical programming approach [11, 12]. Recently, pair programming has become a focus of discussion and research in computer science education, as described in the following paragraphs.

N. Nagappan and L. Williams studied student programmers in a "CS1" course working in a "closed-lab" setting (projects were completed during the 3-hour lab meeting)[13]. Some lab sections used pair programming; other sections used solo programming. They found that students who worked in pairs had a more positive attitude toward working in a collaborative environment, that pair programming reduced the burden on the laboratory instructors because the members of the pair helped each other, and that pair programming was no deterrent to overall student performance. Further, they found that pair programming helped in the retention of students in the computer science major.

In 2006, Robins et al.[14] reported on a study of solo programmers in which they recorded every student request for help during “CS1” closed labs over a 2 year period. B. Hanks replicated this study with pair programmers and found that pair programmers experienced 41% as many problems per lab as did solo programmers and asked for help only 25% as often[15].

L. Williams conducted another study, in which both paired and solo students were given standard assignments. Paired students reported that they would be more willing to work in pair groups than solo in the future. She also showed that paired students are able to complete programming assignment faster and with higher quality. Paired students were reported to be happier and less frustrated than solo students during the study, which the authors suggested could reduce the educator’s workload and also led to less student cheating [16].

According to Cockburn’s study [17], pair programming improves the quality of software design, reduces the deficiencies of the code, enhances technical skills, improves team communication and is considered to be more enjoyable for the participants.

In addition to the studies summarized above, many other studies[18, 19, 20] revealed that pair programming leads to the “conditioned” solo programmer (one who has pair programming experience) performing better in multiple scenarios, leading to better design or implementation of the given tasks.

2.2 Debugging Strategies

Another important topic in computer programming is program debugging. A number of studies have been conducted with the goal of gaining a deeper understanding of what strategies programmers use when debugging.

Gould and Drongowski [21, 22] observed that experts usually start to debug a program by employing some tactic selected based on available data (code listings, output, etc.), prior experience with this code or other code, and personal preference, which leads them to the bug, to a clue, or to neither. In the case of leading them to a clue, they apply an appropriate method to follow the clue. If no bug or clue has been found, then they choose a new tactic. Katz and Anderson [23], considered a program debugging task as a troubleshooting process consisting of four stages:

- Understand the system
- Test the system
- Locate the error
- Repair the error

They found that students' difficulties were largely in the stages up to and including locating the erroneous line of code. They found that students used a variety of bug-location strategies, and that the choice of strategy differed depending on whether they were debugging their own programs or those of other students. They also found that although the bug-location strategies affected which lines of the program were searched, that once students decided on a line, their ability to judge whether or not the line was correct and their ability to correct an error are not substantially affected by the strategy used to locate the line.

Vessey studied and compared expert and novice debugging processes using verbal protocol analysis[24]. She proposed that subjects usually followed a complete "strategy path" while debugging. A complete strategy path consisted of a choice between two options from each of four distinct intermediate strategy paths, resulting in the 16 possible complete strategy paths seen in Figure 2.1 [24].

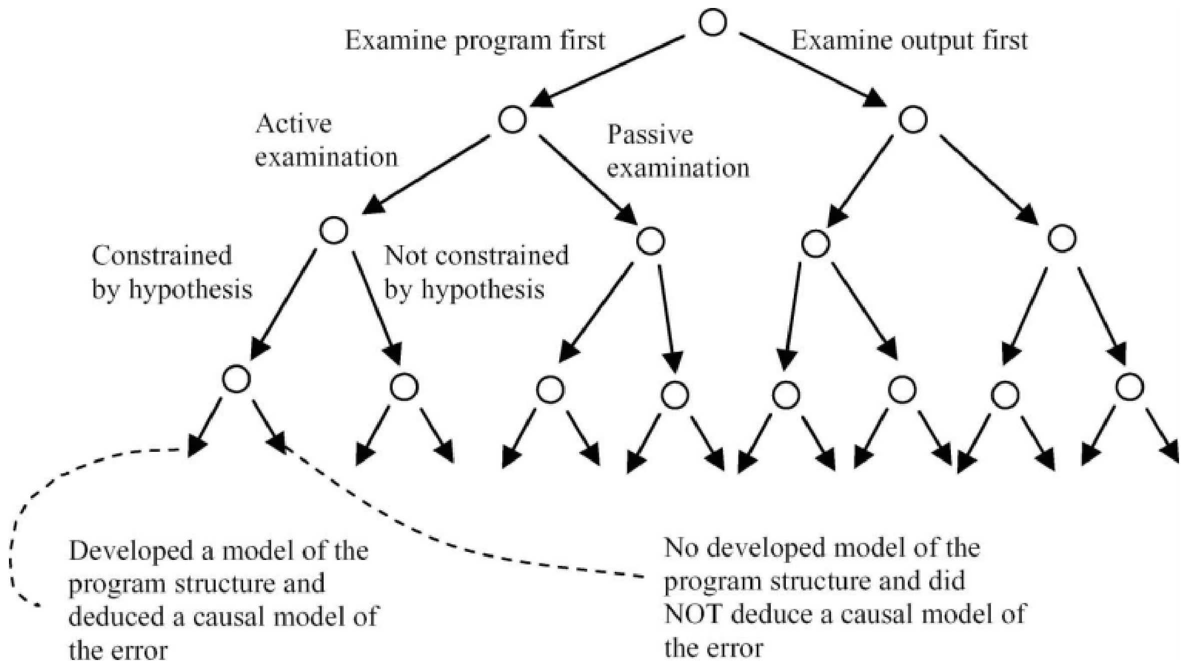
Examine program first
Examine output first
Active examination
Passive examination
Constrained by hypothesis
Not constrained by hypothesis
Developed a model of the program structure and deduced a causal model of the error
No developed model of the program structure and did NOT deduce a causal model of the error

Figure 2.1: Vessey's Possible Debugging Strategy Paths

Programmers were classified into experts and novices, based on their observed ability to effectively "chunk" the program they were required to debug. Then, significant differences in subjects' approaches to debugging were used to characterize programmers' debugging strategies. Comparisons of these strategies with the expert-novice classification showed programmer expertise based on chunking ability to be strongly related to debugging strategy. She then proposed that experts use breadth-first approaches to debugging, adopt a system view of the problem area, and are proficient at chunking programs. She proposed that novices use breadth-first approaches to debugging but are deficient in their ability to think in system terms, use depth-first approaches to debugging, and are less proficient at chunking programs.

Different from Vessey's strategy paths, Ducasse and Emde proposed that four possible strategies are employed during program debugging:[25]

- Fltering (tracing algorithms, tracing scenarios, path rules, slicing/dicing);

- Checking computational equivalence of the intended program and the one actually written (algorithm recognition, program transformation, assertions);
- Checking the well-formedness of the actual program (language consistency checks, plan recognition);
- Recognizing stereotyped errors, those that can be easily recognized from experience (bug cliche' recognition).

Overall, researchers have generally focused on comparisons of experts and novices, though both experts and novices have been found to engage in different behaviors with their own programs versus those of others and to perform better on code that they have previously seen, even when seeded with new errors. Also, experts are more likely than novices to focus first on gaining an understanding of the program's purpose and behavior before attempting to localize and diagnose the error.

2.3 Pair Debuggging

Pair programming and program debugging have been well studied from different research perspectives, but relatively little prior work exists on the debugging behavior of pairs. In one study on this topic, Murphy et al. performed discourse analysis on the verbal interactions of five pairs of introductory programming students as they debugged Java programs. They identified types of discourse including extensions, feedback requests, critiques and completions as the most common types of transactions between the pairs. Results suggested that pair programmers who talked more and used complete transactions more often attempted more problems, but those who critiqued more frequently successfully debugged more problems[26].

Thippaya Chintakovid conducted another interesting study, on the debugging of a spreadsheet program. This study suggested that collaboration makes students more engaged in the debugging task. Students in pair groups tended to communicate with each other with a strategy of questions and hypotheses to elicit discussion and push their debugging forward [27].

In Knuth's report on the debugging and evolution of TeX over a ten year period[28] and Perkins and Martin's [29] notion of fragile knowledge, both of these suggest that for most students with some expertise the difficulty of debugging is not in repairing the error but rather in the earlier stages of the troubleshooting process (understanding the system, testing the system, or locating the error).

Among all these studies, to our best knowledge, there is no prior work comparing solo and pair programmers working on the same debugging tasks. We conducted a study to explore the differences in solo versus paired student programmers engaged in debugging tasks.

Chapter 3

Experiment Design

Our study is designed to observe and analyze student programmers engaged in debugging programs, in order to explore differences in the strategies employed by solo programmers and pair programmers. We recruited students from a sophomore-level computer science course, who have completed one semester of a "CS1" course using Java and have either completed or are concurrently enrolled in the "CS2" course, also using Java. The subjects worked on small-scale program-debugging tasks during a 50-minute "closed" lab session. Pretests, post-tests, and demographic surveys were conducted. We captured screen video and user audio during the debugging lab session and also collected their final debugging result. We evaluated performance based on the total time spent on debugging and the total number of bugs fixed. We also evaluate subjects' satisfaction, confidence and enjoyment by carefully examining the video and audio records captured during the lab session.

3.1 Experiment Setting

Course Structure

Our study was conducted in the Computer Science Department of the University of Georgia. We recruited participants from the sophomore-level course ***UNIX Systems Programming*** taught in the spring semester 2014. This four-hour course covers C and C++ and the basics of UNIX systems programming, including file and directory structures, basic and advanced file i/o, process creation, and inter-process communication. A basic C/C++ debugging tool gdb (GNU Project Debugger) had previously been taught and practiced in one prior lab session.

Subjects

Forty-two students signed up for the study, all at the undergraduate level. Most of the students were sophomores or juniors. They were not expected to have a strong background in programming or related computer science topics. We can define them as novice programmers based on their previous study experience, although some of them may have taken some other programming course in advance, which could be found from the Demographic Surveys completed before lab session. In general, they are beginners of computer programming.

The students were informed in class of the study and had the option to participate or not to participate. No penalty was associated with non-participation. No reward was associated with participation.

Procedure

After the students signed up for the study, the subjects were asked to complete a Demographic Survey online before the lab session.

The course consists of three lab sections. We randomly selected one of the sections to work solo and the other two sections worked in pairs. On the day of the study, students in the paired groups were assigned partners based on similar midterm scores.

Prior to starting the debugging exercise, students were asked to complete a pretest designed to evaluate their familiarity with concepts that appeared in the debugging session, as well as their background knowledge on programming.

At the beginning of the lab session, the instructor gave them a handout about how to download the buggy code from the server, how to turn on Camtasia to capture their screen activities, and how to save the Camtasia file and submit the final work. After all the setup had been performed, they could start to work on the debugging task. The TAs walked around to answer basic questions, but not those directly related to how the bugs in the program.

In the lab session, we provided the students a gdb debugging environment. As mentioned above, the instructor had previously covered the concepts about debugging and a gdb how-to in the class lecture. The students had also finished one lab session about how to use gdb to debug some simple buggy program, although we found it did not work well in the later practice. We will revisit this part in Chapter 4.

Camtasia screen capture software was used to capture the screen activities while the participants were working on the tasks provided. The conversation between the two students in the pair group was recorded, and the solo students were asked to engage in "think aloud" during the debugging session.

The recorded audio and video were later coded and then analyzed to detect patterns in debugging behaviors and to compare the pair programmers with the solo programmers.

For the following 30 minutes, all the groups worked on the debugging session. They were not allowed to use the internet, but were provided with a "cheat sheet" about how to use gdb.

After they had finished the debugging exercise or when time was called, they were guided to save all their work and the recorded files on the server. They also finished a post-test through eLC, online courseware used by the university.

Materials

There were four prepared materials that both the paired and solo groups worked on:

1. Demographic survey

2. Pre-test

3. Buggy program

4. Post-test

The demographic survey, adapted from our previous studies, is intended to collect basic information such as their gender, age, programming experience, and courses they have taken previously. The survey contains 16 questions. The demographic survey provides general background information about the subjects in the study. Our main focus was on their knowledge and programming background.

The pre-test was designed to test the subjects' background knowledge on concepts that are necessary for the debugging task at hand. In the pre-test we asked the subjects answer the following question:

... ...

What is the result of the statement:

scanf("%d", &x);

(a) the value of x is stored at the location entered into %d

(b) a double value is placed at the address stored in variable x

(c) a double value is placed into the memory associated with variable x

(d) an integer value is placed into the memory associated with variable x

... ...

We slightly revised the code referred to in the question, and generated one bug in our buggy program as follows:

```
... ...
   void Read2DArray(int a[][3], int ROWS) {
         printf("Please input the elements for an integer array with %d elements \n",
                3 * ROWS);
         printf("After each input please hit ENTER: \n");
         int r,c;
         for (r = 0; r < ROWS; r++){
               for (c = 0; c < 3; c++){
                     scanf("%f", &a[r][c]);
               }
         }
   }
    ... ...
```

This main material for our study is the buggy code. It has several auxiliary functions and a main function. The main function is designed to receive keyboard input and to load the particular case the subject wants to debug.

The other functions are designed to manipulate arrays, which are pre-input or input by subjects while the program is running, and perform operations such as printing the array to the screen, or displaying some other meaningful output about the array such as the max value, the sum of one row, etc. Figure 3.1 shows the structure and data flow of the program.

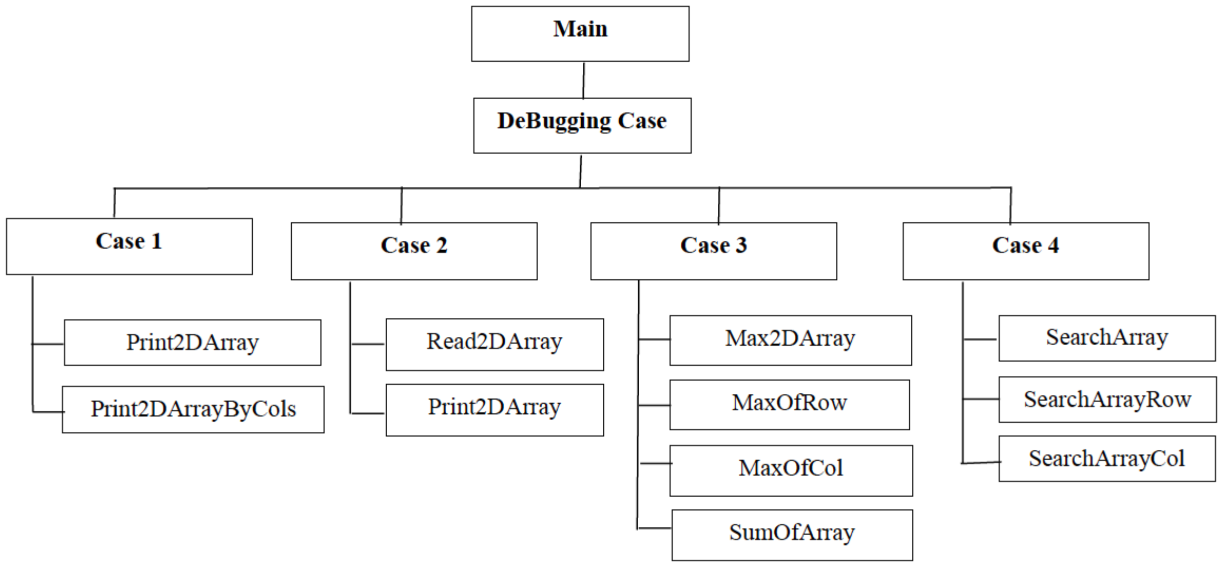

Figure 3.1: Program Structure

There are four sub-cases for debugging purposes. Below is the table for the bug types and difficulties.

	Difficulty(1=easy,3=hard)	**Error Type**
Case 1	1	Boundary error
Case 2	2	Format specifier error
Case 3	2	Logic error
Case 4	3	Flow control condition error/Operand error

Table 3.1: Program Bugs

The subjects can run the program and then enter the case number. They typically debug the cases sequentially. They may find and fix the errors by reviewing the code or running and debugging the program in gdb.

The difficulties for each case is based on the nature of each cases:

Case 1 is a boundary error, If the subjects are careful enough it will be obvious either when reviewing the code or running gdb. But since it will be the first bug after the system setup, we were not expecting them to fix it immediately.

Case 2 is slightly more difficult than case 1. Since there are not many situations that require the students to focus on the output format, and some variations on the output format exist, we defined it is as harder than case1.

Case 3 is a logic error, which is actually not too difficult. But with an increase in scale of the code in case 3, the subjects need to read more code. Meanwhile there is no obvious sign about what is the wrong with the output of the of the program, which increases the difficulty of this case.

Case 4 is a well-known C/C++ bug in both industry and academia. It is a common tough bug to find in a program. We placed this code at the end of the assignment to avoid having subjects spend most of their time on this bug and not work on the rest. There will not be many students who have time to work on this one, since this bug is placed at the end of the program.

The dependency of each case is as followed (Figure 3.2).

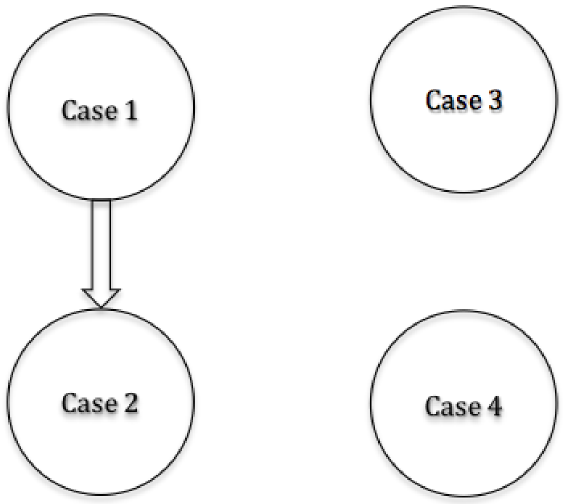

Figure 3.2: Case Dependency

Only case 1 and case 2 are dependent, the students were free to skip around but typically tackled the bugs in order. We did not have any restriction on it.

The post-test consisted of 8 questions, designed to make sure the student fixed the bug based on a rational reasoning, instead of luck. The questions inquire about the nature of each error.

3.2 Hypothesis

We evaluate our results from two perspectives, performance/enjoyment and debugging strategy. These hypotheses are derived mostly from the literature and study survey we covered previously, but also on the studies conducted by other students in our research group. Some of them have been shown true in a small scale sample space in some studies; some of them are just observation result; some of them are still discussion result with student programmers without any further verification. We divide these hypotheses into two categories:

Performance/Enjoyment and Strategy.

Performance/Enjoyment

We assume the pair groups will fix more bugs in general. In other words, that the 14 pair groups will solve more cases than the 14 solo groups within a particular time period. Previous pair programming studies showed during the programming stage that pairs performed better than solos. We investigate whether pair debugging shows the same beneficial trend.

We are also interested in the efficiency of the overall performance, which we define as the total time spent on the fixed cases divided by the teams who have solved the problem. Intuitively, it would seem that the pair group should perform better than the solo group. However, the pair debuggers have communication costs, requiring time to exchange their opinions, which could slow the bug fixing process.

Debugging, or in general, trouble shooting can be a challenging, but frustrating process. Some programmers enjoy it but others do not. L. Williams stated that most developers enjoy programming in a pair. We will keep a open mind about this issue and draw a conclusion after we go through the study record.

Strategy

Debugging could be considered as a trouble shooting process. Katz [23] proposed a general trouble shooting model (Figure 3.3).

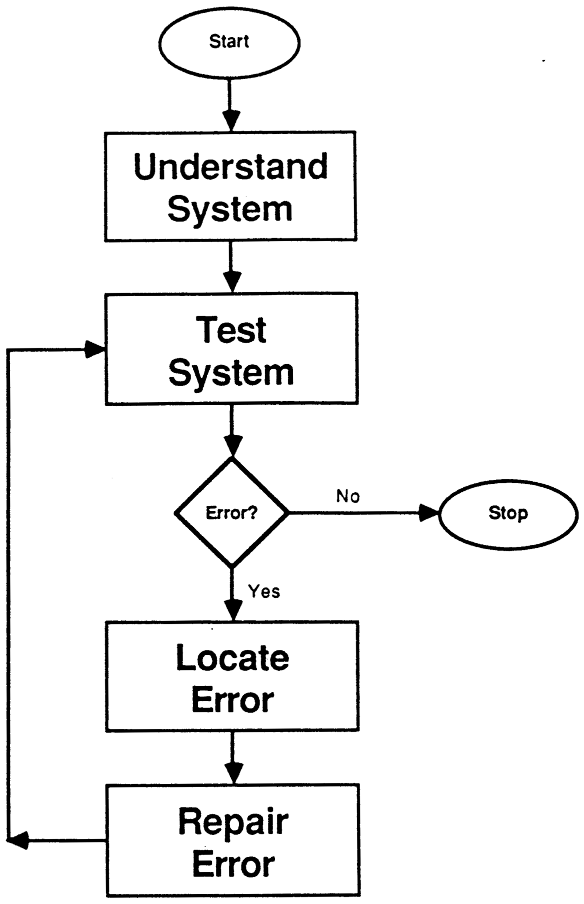

Figure 3.3: Simplified Model of General Troubleshooting

Furthermore, he concluded there are four steps during debugging:

(a) Comprehend the program.

(b) Test the program and detect that it isn't behaving incorrectly.

(c) Locate the erroneous line or lines of code.

(d) Rewrite the buggy code.

Main theories about program comprehension include top-down, bottom-up and the combination of them [30]. For the error locating stage, Katz proposed strategies known as forward-reasoning and backward-reasoning.

We believe both the solo and pair students will most likely employ Katz's model during their debugging. For the large scale program, e.g. in an industrial scenario, not only locating the error, trying to understand the program and find a optimal way to fix it are critical and difficult. For small-scale academic/educational practice, the most challenging phase is how to locate the error, and the student will likely fix the bug easily once they get to know where the bug occurs.

Summarizing Jeffries [31], Kessler & Anderson [32], Gugerty & Olson [33], and Klahr & Carver [34], Katz proposed two bug-locating strategies in his work: forward-reasoning and backward-reasoning. We apply this framework to our study and look for differences between the pair and solo programmer, from the forward and backward reasoning perspective.

Chapter 4

Evaluation

Forty-two students signed up for the study and participated in the lab session. All of the students successfully submitted the study required materials and saved the video and audio records. Some of the video and audio records were shorter than others, likely because these subjects were preparing for the debugging session, taking the demographic survey or the pre-test in the lab session, instead of finishing it before. We take this into consideration during the analysis.

Before the subjects actually proceeded to the debugging session, the top priority issue was how to pair the volunteers. The 42 students had slightly different programming backgrounds and capacities. A simple method would be to pair the subject randomly, in spite of their differing programming capacities. However, based on prior study recommendations, we attempted to conduct our study under conditions in which the programmers in each pair were of similar ability. In this case, they will feel motivated to contribute during the debugging session, and also avoid frustration that may result if one partner is much more capable. Accordingly, we paired students based on their midterm grades, pairing students with similar midterm exam scores, to the extent possible.

Below is the study result in general, which gives us some brief idea about the comparison of the pair and solo group.

	Solo		Pair	
	Mean	SD	Mean	SD
Midterm	67.43	20.93	66.27	14.34
Pre-test	2.64	1.01	3.25	0.94
Post-test	2.29	2.87	3.57	2.23
Prep time	4.25	3.46	5.77	5.10
Case 1	7.3	8.15	5.66	4.73
Case 2	2.61	4.92	3.2	3.20
Case 3	1.04	2.78	2.38	4.93
Case 4	1.36	3.5	1.30	2.87

Table 4.1: Student Grades

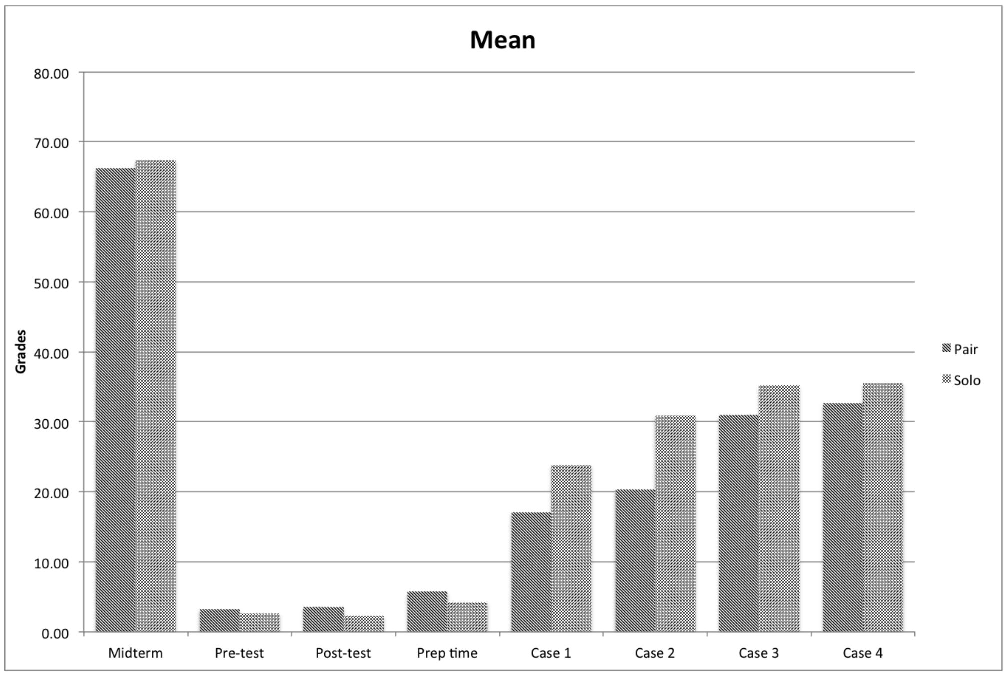

Figure 4.1: Students Grades Mean

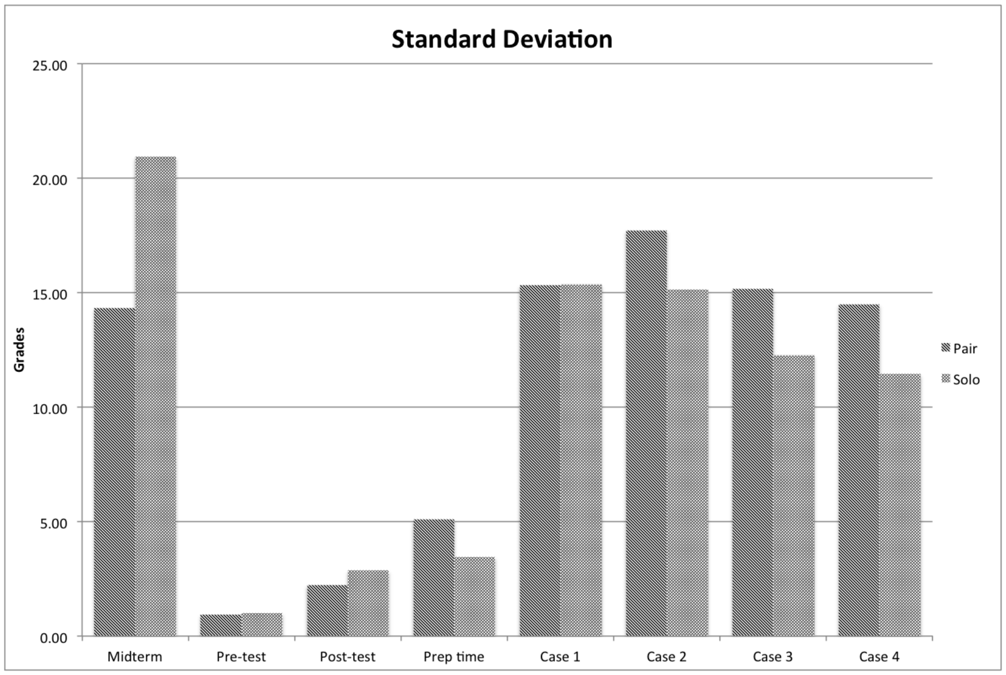

Figure 4.2: Students Grades Standard Deviation

We summarize performance based on the debugging result in section 4.1. Then, based on the think aloud protocol/pair conversation and screen activities, we seek to discover their debugging strategy in section 4.2.

4.1 Debugging Performance

Subjects Profile

Eighty-two students were enrolled in the class at the time of the midterm, and the uncurved, average score was 75.25, which is slightly higher than that of the volunteers in our study. We recruited forty-two students out of eighty-two. Twenty-eight students worked as 14 pairs, and the remaining 14 students worked solo. The average score of the pair groups in our study was 66.3, and the average score of the solo groups was 67.4 (Figure 4.1). No significant difference existed between the two groups. the

As we mentioned above, we paired students with similar midterm score, and no significant difference existed between the pair group and the solo group.

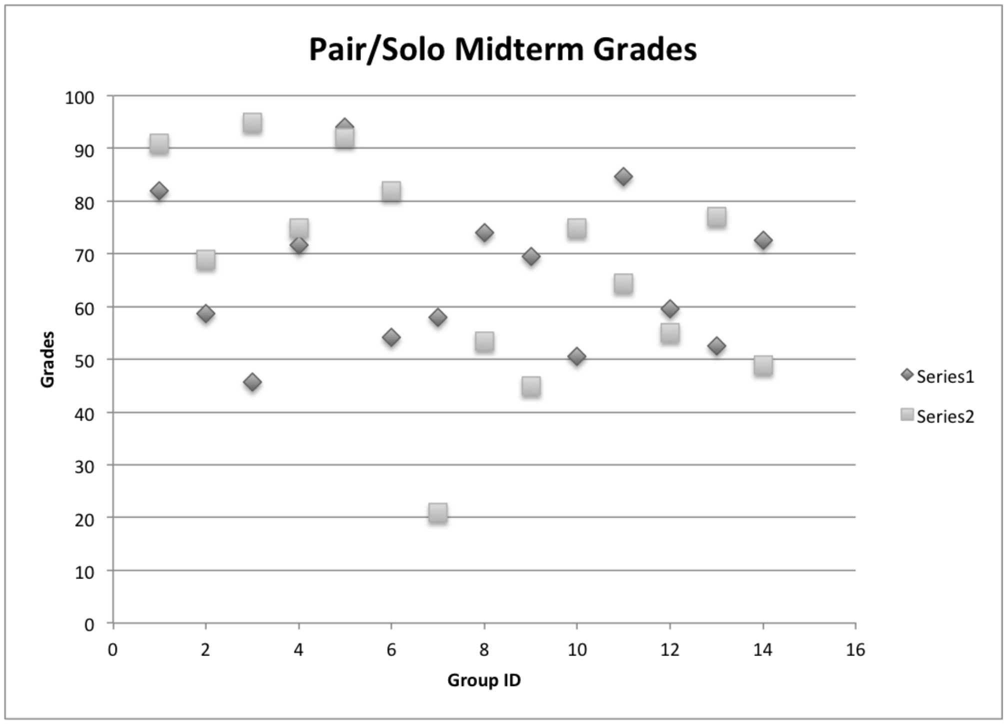

Figure 4.3: Pair/Solo Group Midterm Grades

	Pair	Solo
Mean	66.27	67.43
Standard Deviation	14.34	20.93
p-value	0.866	

Table 4.2: Students Midterm Grades

As is demonstrated in these figures and table, the solo and pair groups are evenly distributed in the coordinate system. Not only we can conclude the subjects have been divided equally into two kinds of groups by the midterm grade, but we can also verify this strategy worked well after they took the pre-test. The solo group scored 2.6 out of 10 in the pre-test, and the pair group score 3.2 of 10, which was not a significant difference (Table 4.2).

	Pair	Solo
Mean	3.25	2.642857143
SD	0.935414347	1.008208072
p-value	0.110606457	

Table 4.3: Pre-test Scores Mean/SD

Performance/Enjoyment

Table 4.3 lists the total bugs that the 14 pairs fixed and those that the 14 solo subjects fixed. In total, the pair group fixed 24, and the solo group fixed 13.5. This result shows the pair groups have fixed bugs twice as many as the solo groups. To reveal more details of the productivity, we also compared the pair and solo group in the case that each group is further sub-divided based on their midterm score, into those who scored 70 or above and those who scored below 70. The fixed bug result supports the difficulty ratings previously established. Although the time is sufficient, the resolved bugs remaining decrease, in both pair and solo groups.

	Case 1	**Case 2**	**Case 3**	**Case 4**
Pair	10	8	4	3
Solo	8	4	2	2

Table 4.4: Bugs Fixed per Cases

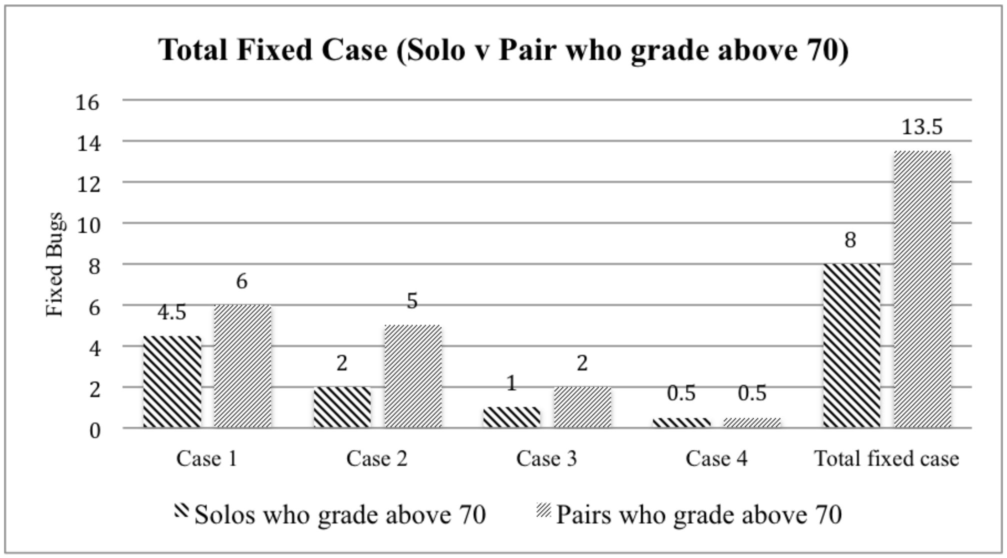

Figure 4.4: Bug Fixed by Groups Midterm Grades Above 70

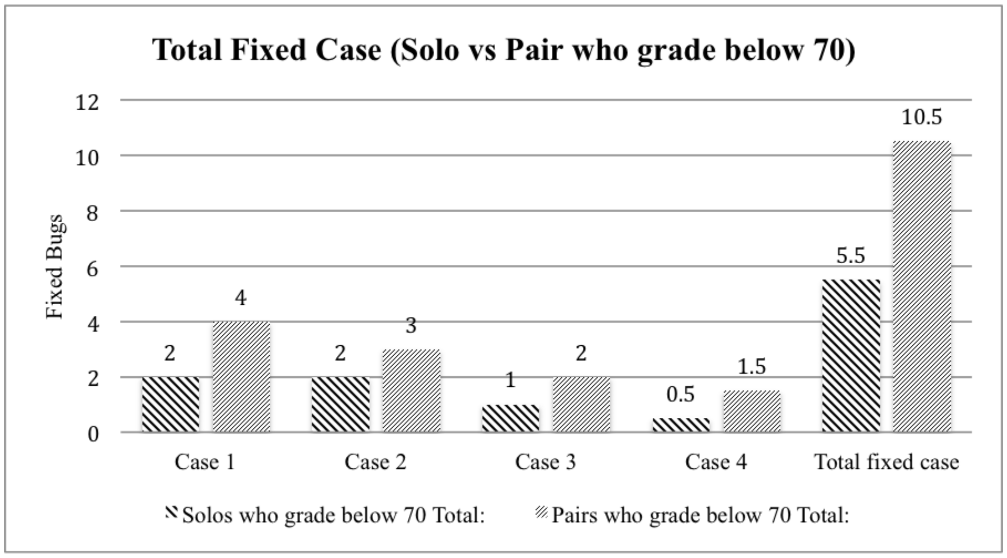

Figure 4.5: Bug Fixed by Groups Midterm Grades Below 70

A threat exists for the productivity result above, since the solo group and pair group could be working on these bugs with different time period, because the lab instructors and TAs didn't force the students to start the debugging at the same time, although they stopped on call. So we decided to measure the performance by calculating their efficiency, which means the result output per unit time. Figure 4.5 and Table 4.4 show the comparison of the pair and solo. The pair programmers show higher efficiency (less time per bug fix) on three of the four cases.

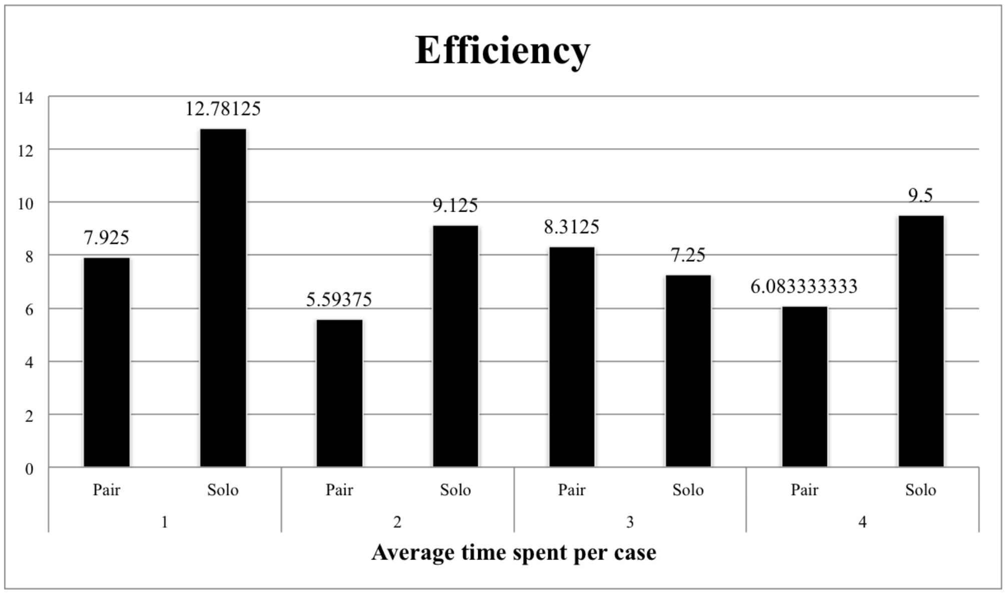

Figure 4.6: Debugging Efficiency

Case number	**1**		**2**	
Group Types	Pair	Solo	Pair	Solo
Amount of Successful Group	10	8	8	4
Total Time	79.25	102.25	44.75	36.5
Average time spent per case	7.93	12.78	5.59	9.13

Table 4.5: Efficiency per Case(a)

Case number	**3**		**4**	
Group Type	Pair	Solo	Pair	Solo
Amount of Successful Group	4	2	3	2
Total Time	33.25	14.5	18.25	19
Average time spent per case	8.31	7.25	6.08	9.5

Table 4.6: Efficiency per Case(b)

	Pair	Solo
Case 1	7.93	12.78
Case 2	5.59	9.13
Case 3	8.31	7.25
Case 4	6.08	9.5
Mean	6.98	9.67
SD	1.34	2.30
p-value	0.045	

Table 4.7: Efficiency in Total

	Pair	Solo	Pair	Solo	Pair	Solo	Pair	Solo
Mean	7.93	12.78	5.59	9.13	8.31	7.25	6.08	9.50
Standard Deviation	3.52	6.59	1.94	5.07	6.27	3.18	3.13	2.12
p-value	0.045		0.130		0.399		0.122	

Table 4.8: Result per Case

Based on the video and audio record, we noted two roles naturally emerged in the pair groups. Some students were more willing to control the editing job, which we call the driver role. Some students were more interested in thinking and guiding the other student, which we call the navigator role. In 2 out of 14 cases, the subjects agreed to change roles, after one of them proposed it, since he/she didn't feel comfortable about his/her role. Also we found the pair programmers were more encouraged once they fixed one bug and received agreement from his/her partner.

Although we didn't design a survey of satisfaction, we decided to examine the students' enjoyment during the debugging, by listening to their conversation from pair programmers and the record from solo programmers. Most complaint came from the solo students. Some of them complained the code is too many lines, some students complained the bugs are too hard. By comparing the verbal information, we conclude that in most cases pairs enjoyed the debugging more than solos.

4.2 Debugging Strategies

Generally, the pair programmers spent more time on the preparation period than the solos. They discussed some basic issues and came to an agreement before they proceeded further, issues such as how to start to debug the program (code review or gdb), and what are their first impressions and understanding of the program. This is understandable, since the pair group needs to set some ground rules or debugging strategy frameworks for the later collaboration.

We can draw a pattern of the debugging session from both pair and solo groups. They can try to understand the program in two ways: review the code or run the program. Then, they go to the next stage to locate the errors. After they find where the bug is located, they make the modification and test it.

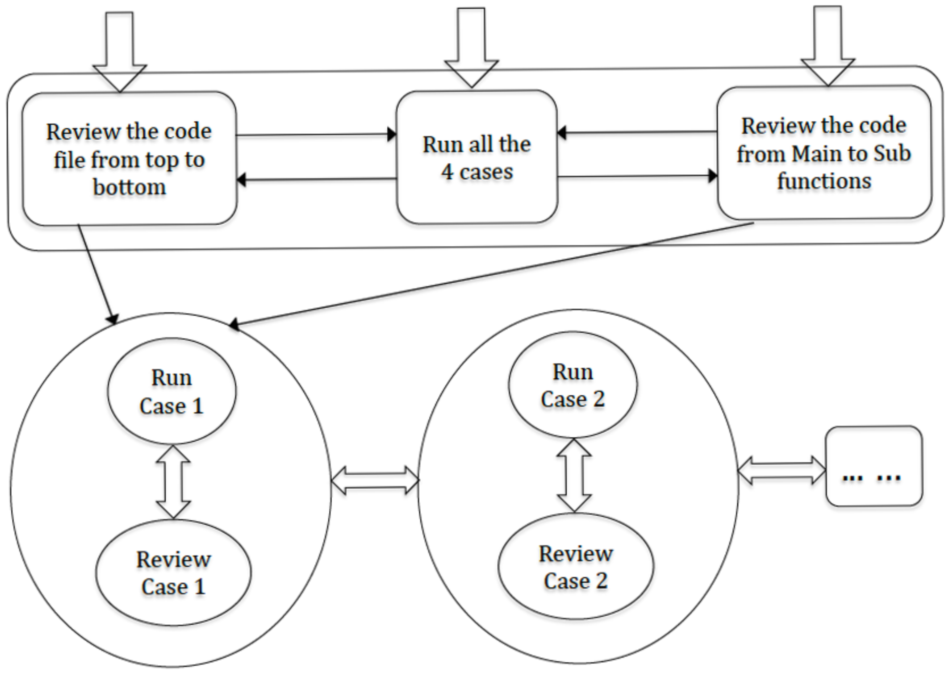
Review the code file from top to bottom
Run all the 4 cases
Review the code from Main to Sub functions
Run Case 1
Review Case 1
Run Case 2
Review Case 2
... ...

Figure 4.7: Debugging Strategy

Before we start to analyze the pair and solo debugging strategy, and the bug-locating strategy in particular, we want to step back and illustrate the concepts of forward-reasoning and backward-reasoning. In general, the programmers' debugging behaviors may be divided into two strategies. The first strategy is when the programmer tries to comprehend the program at the beginning of the debugging session. They tend to build a representation of the program (paraphrasing the code during the debugging). Then, based on their understanding, they try to execute the code mentally as if they are the computer. This is called forward-reasoning. First they understand the whole system, then they execute it, and find what is the wrong with it.

The second strategy is backward-reasoning. In this method, the programmers start off to execute the program, and try to define the incorrect behaviors. During the execution, they could follow the hints to some particular line of code, and then based on their program knowledge, find the bugs.

Strategy	**Type**
Forward-reasoning	Comprehension
	Hand-Simulation
Backward-reasoning	Simple Mapping
	Causal Reasoning

Table 4.9: Katz's Debugging Strategy

Katz explained that forward reasoning and backward reasoning have the sub-types in the above table, and he gave the follow definition about the each sub-types [23].

- Comprehension: A subject finds bugs while building a representation of the program
- Hand-simulation: Subjects evaluate the code as if they were the computer
- Simple Mapping: The program's output points directly to a specific line (or type of line) being incorrect
- Causal reasoning: The subject searches backward from the program's output, using their knowledge of the program and programming to find the error.

Furthermore, he stated that if a subject refers to successive lines of the program, without considering the output of the program, the protocol may be categorized as reflecting forward reasoning, otherwise if the subject makes predictions about the location of the bugs in a program based on the program's output or uses the program's output to limit his search of the program, that subject's protocols maybe categorized as reflecting backward reasoning.

He also mentioned that subjects prefer to use backward-reasoning when debugging their own code, but forward-reasoning when debugging others. We examine and evaluate this hypothesis in this context.

In our study the students spent about 5 minutes to get familiar with the program itself. After this period of time, they have the essential knowledge about the functionality of the program.

Our classification of the reasoning type of each group will depend on the dominant strategy of the programmers employed, some groups might be using a combination of the forward and backward reasoning, but mostly there is a mainly used reasoning method, which can define their debugging strategy preferences.

Those who tried to use case review at the beginning of the debugging will be more likely to be categorized into forward reasoning. e.g. pair 2, the subjects started off by running the code once, then reviewed the entire code file, and tried to figure out the structure of the whole program. They tried to answer some questions. e.g.

> *... ...*
>
> *"what are the data flow and control flow?"*
>
> *"How the program behaves? '*
>
> *... ...*

These questions above could conclude this groups belongs to the forward reasoning group.

Those who started off by running gdb will be more likely to be categorized into backward reasoning. They want to know the output of the program in the first place.

Those who were trying to understand the code, and then run the program to verify their understanding will be categorized into forward reasoning. The subjects will talk to their partners something like the followings. e.g.

> *... ...*
>
> *"Do you want to read through it?"*

"Let's see what does the program do?"

... ...

Those who ran the code first, then went back to buggy code trying to define why the program generated a output like that, will be categorized into backward reasoning. In this categorization, subject will ask the following questions after them run the program. e.g. during debugging case 1, the subject asked:

... ...

"what does the fourth column suppose to be?"

... ...

Basically we will go through the cases one by one and define the groups by their conversation and think-aloud data.

Group ID	**Forward**		**Backward**	
	Comprehension	**Hand-Simulation**	**Simple-Mapping**	**Causal-Reasoning**
Pair 1				X
Pair 2	X			
Pair 3			X	
Pair 4				X
Pair 5		X		
Pair 6				X
Pair 7				X
Pair 8	X			
Pair 9				X
Pair 10			X	
Pair 11				X
Pair 12		X		
Pair 13	X			
Pair 14				X

Table 4.10: Pair Debugging Reasoning

Group ID	Forward		Backward	
	Comprehension	Hand-Simulation	Simple-Mapping	Causal-Reasoning
Solo 1	X			
Solo 2				X
Solo 3	X			
Solo 4		X		
Solo 5				X
Solo 6	X			
Solo 7			X	
Solo 8	X			
Solo 9		X		
Solo 10				X
Solo 11				X
Solo 12	X			
Solo 13				X
Solo 14			X	

Table 4.11: Solo Debugging Reasoning

In Katz's study, he argued that the debugging strategy the experts prefer are more related to the authorship of the program they are working on. If they are more familiar with the algorithms or how the program is implemented, they will prefer backward reasoning, which means they will choose the weak side of their knowledge back to the strong side. On the contrary, if they do not know how the program executes, they will use forward reasoning instead, which also is a strong to weak pattern.

From the pair table, we can see 5 pairs used forward reasoning, and 9 used backward reasoning during debugging, which means the pair programmer will be more willing to use backward reasoning than forward reasoning.

This is understandable, since the pairs are more familiar with the program after the preparation phase, because of the exchange of opinions and discussion between the pairs. But the debugging result shows that although the pairs have a strategy preference, the 5 pairs who used forward solved 13 bugs in total, while the 9 pairs used backward reasoning solved 12 bugs in total. So, in this case, the forward strategy appears to have been more productive.

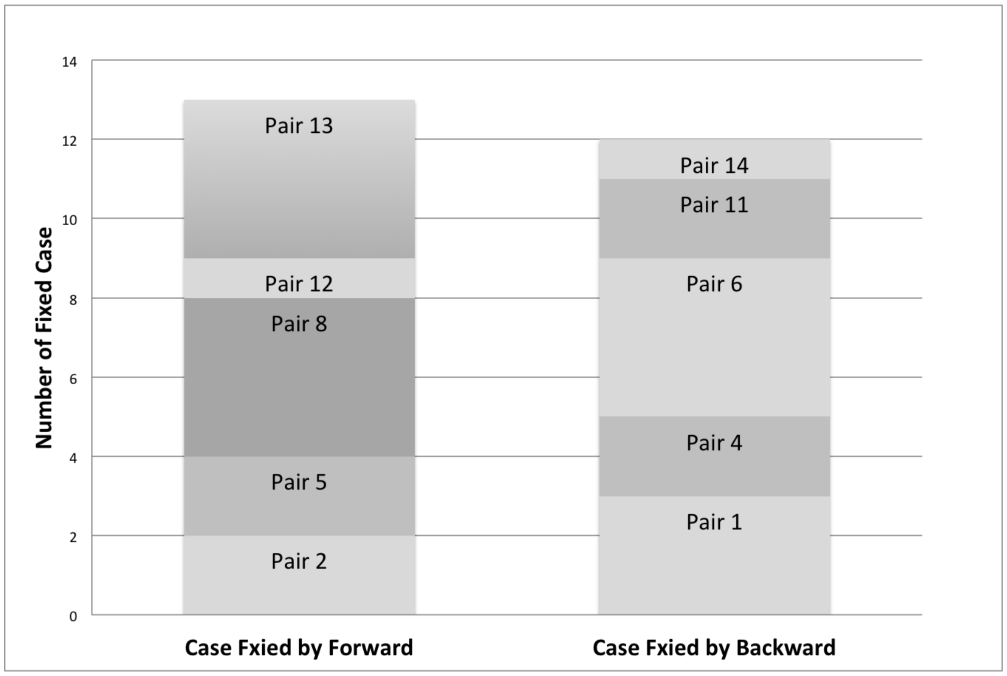

Figure 4.8: Pair Reasoning Result

For the solo groups, the numbers who used forward reasoning and backward reasoning were equal. This tells us that when a programmer is debugging solo, he does not have a obvious preference. But 5 solos who used forward reasoning, fixed 12 cases, and 3 solos who used backward reasoning, fixed 4 cases, which is significant. This debugging result suggests that when a novice debugging a small scale program, the backward reasoning bug locating strategy might be a overall better strategy.

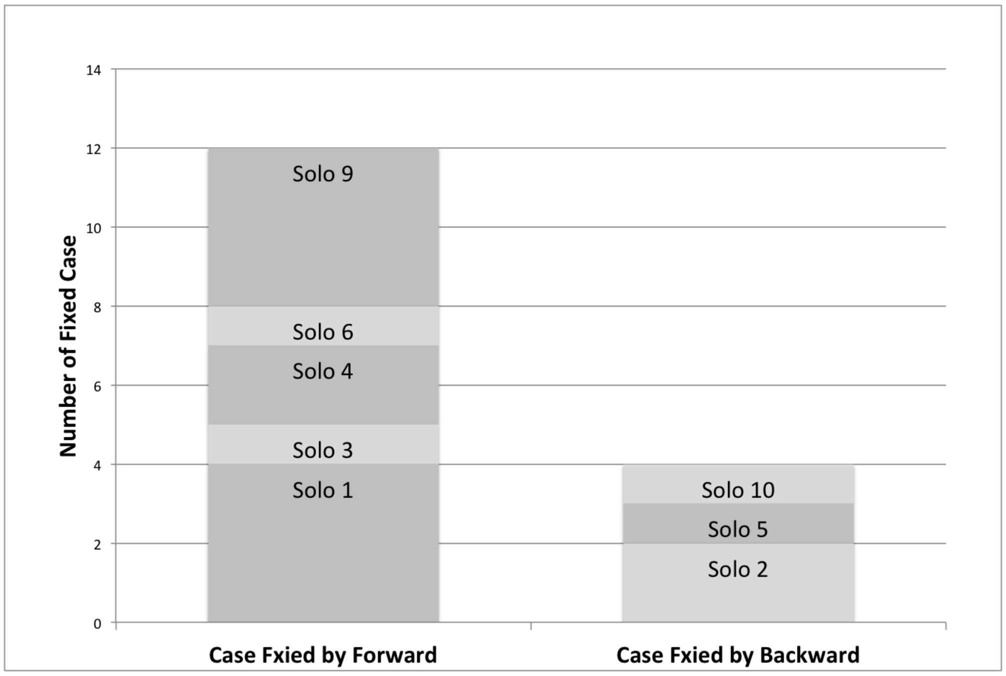

Figure 4.9: Solo Reasoning Result

For the pair groups, there were 4 groups did not fix any bugs, and all of them were among the 9 backward reasoning groups.

For the solo groups, there were 2 forward reasoning groups and 4 backward reasoning groups did not fix any bugs.

The preference of the pair group also shows that working together helps students understand the program itself more deeply and comprehensively.

Based on Katz's statement, those programmers who understand the program itself prefer to use backward reasoning, those who understand how the program execute prefer forward reasoning instead. Although different solo groups will differ in the comprehension about the program, this difference of the understanding of the program did not cause the strategy difference in their practice. The working in pair seems improve the students understanding of the program significantly, which directly lead the pairs more prefer to use backward reasoning as their prime debugging strategy.

Chapter 5

Conclusion

In this work, we focused on investigating the advantages or disadvantages of pair debugging in an educational practice. We conducted the study to make the comparison between the pair and solo groups about how they debugged a small-scale buggy code, and what did they perform during the debugging. We discussed not only what are the differences between them, but how and why these differences came about.

Our study data are collected by both the writing, video and audio methods. The writing data are collected online, the video and audio data are captured by software installed in computer. Then we use the statistic method to analyze the performance of the debugging result of these two groups, and employed an existing framework to evaluate the debugging strategies.

We believed that the pair programming is a better way to make the debugging more successful and enjoyable. Not only the programmers will enjoy debugging more than solos, consequentially, they are more efficient when they work on the debugging task collaboratively, rather than alone.

For the debugging strategy, our conclusion is the pair programmer will potentially prefer backward reasoning, instead of forward reasoning, because of the natural advantage of it.

This natural advantage means, the programmers who are debugging in pair will naturally start to discuss the program and help each other to understand it. The programmer usually gains better understanding of the system, when they are debugging in pairs instead of solos, which unconsciously leads them to satisfy the conditions of forward-reasoning.

This is our first attempt trying to analyze the behavior of the novice programmer when they are handling some buggy programs, generated by other programmer. Although the sample space is considered large enough, but the debugging tasks that the subjects worked on are still not much convincing. To be more accurate and solid, we could design a series of lab sessions, gradually increasing the difficulties of the buggy programs, train the subjects' GDB skills well, and define our own analysis framework in the future. Also, we could add a survey of satisfaction in the post-test to draw a conclusion about the enjoyment. This study could be an inspiration and basis for the future work.

Chapter 6

Appendix

6.1 Study Materials

6.1.1 Demographic Survey

Demographic Survey

1. Please choose your gender.

- ☐ Male
- ☐ Female

2. What is your age?

- ☐ 18
- ☐ 19
- ☐ 20
- ☐ 21
- ☐ 22
- ☐ 23
- ☐ 24
- ☐ >24

3. Is English your primary language?

- ☐ Yes
- ☐ No

The following two questions follow the U.S. Census questionnaire format for collecting data on race and ethnicity.

4. Are you a person of Hispanic, Latino or Spanish origin? (optional)

- ☐ No, not of Hispanic/Latino/Spanish origin.
- ☐ Yes, Mexican, Mexican American, Chicano
- ☐ Yes, Puerto Rican
- ☐ Yes, Cuban
- ☐ Yes, another Hispanic, Latino, or Spanish origin — Argentinean, Colombian, Dominican, Nicaraguan, Salvadoran, Spaniard, and so on (Please specify).

  ____

5. What is your race? (optional)

- ☐ White
- ☐ Black, African American or Negro
- ☐ American Indian or Alaska Native (Please specify name of enrolled or principal tribe)

  ____
- ☐ Asian Indian
- ☐ Chinese
- ☐ Filipino
- ☐ Japanese
- ☐ Korean
- ☐ Vietnamese
- ☐ Other Asian (Please Specify Race)

  ____
- ☐ Native Hawaiian
- ☐ Guamanian or Chamorro
- ☐ Samoan
- ☐ Other Pacific Islander (Please Specify Race)

  ____

☐ Some other race (Please Specify)
____

6. When did you finish high school?

   Year: __________ Month: __________

7. Do you remember the zipcode of your high school?

   ☐ Yes, it is: ___________
   ☐ No. Then please write down the name, city and state of your high school.
   ____

8. What is the zipcode of your permanent home address?

   Zip Code: ___________

9. When did you enter the University of Georgia?

   Year: __________ Semester: __________

10. What degree/degrees are you pursuing at the University of Georgia? (Please specify all of your degrees if you have multiple degrees)

    a. _________________________
    b. _________________________
    c. _________________________
    d. _________________________
    e. _________________________

    Please select your minors if you have one.

    a. _________________________
    b. _________________________
    c. _________________________
    d. _________________________
    e. _________________________

11. How many credits have you completed at the University of Georgia?

    ____

12. Please check below all courses that you have already taken.

    ☐ CSCI 1100-1100L Introduction to Personal Computing
    ☐ CSCI 1210 Computer Modeling and Science
    ☐ CSCI 1301-1301L Introduction to Computing and Programming
    ☐ CSCI 1302 Software Development
    ☐ CSCI 1730 Systems Programming
    ☐ CSCI 1900 Computer Science Special Topic
    ☐ CSCI 2150-2150L Introduction to Computational Science
    ☐ CSCI(MATH) 2610 Discrete Mathematics for Computer Science
    ☐ CSCI 2670 Introduction to Theory of Computing
    ☐ CSCI 2720 Data Structures
    ☐ CSCI 3030 Computing, Ethics, and Society

- ☐ CSCI 4300 Web Programming
- ☐ Others, please specify
  ____
  ____

13. Please choose your level of expertise for the following programming languages(Expert/Intermediate/Novice).

- ☐ Java: ____________________
- ☐ Javascript: ________________
- ☐ C/C++: ___________________
- ☐ Objective C: _______________
- ☐ C#: ______________________
- ☐ Perl: _____________________
- ☐ Python: ___________________
- ☐ Visual Basic: _______________
- ☐ PHP: _____________________

14. Do you have your own website?

- ☐ Yes (Please write the URL)
  ____

- ☐ No

15. How many class projects have you completed that involved writing 100 lines of code or more?

A: _______________

16. Have you finished any programming projects outside of classes?

- ☐ Yes, how many of them involved writing 100 lines of code or more_____________
- ☐ No

6.1.2 Pre Test

Assume the following code exists in a function in a C program:

```
…
int r, c, x;

int testArray[3][4] = {{1},{2, 3, 4},{5, 6, 7}};

scanf("%d", &x);

for (r = 0; r<10; r++){
   for (c = 0; c < 10; c++){
         do_something();
   }
}
…
```

1. How many times would `do_something() be` invoked in the above code snippet?

2. Which of the curly braces are essential?
   a. those associated with the outer loop
   b. those associated with the inner loop
   c. both the outer and the inner loop braces
   d. neither the out nor the inner loop braces

3. What is the value of testArray[0][3]?
   a. 0
   b. 2
   c. 3
   d. undefined

4. What is the value of testArray[3][0]?
   a. 0
   b. 1
   c. 5
   d. undefined

5. What is the value of testArray[4][0]?
   a. 0
   b. 1
   c. 5
   d. undefined

6. What is the result of the statement:

    ```
    scanf("%d", &x);
    ```

    a. the value of x is stored at the location entered into %d
    b. a double value is placed at the address stored in variable x
    c. a double value is placed into the memory associated with variable x
    d. an integer value is placed into the memory associated with variable x

7. What is the compiler flag needed to compile a program named "testing.c" so that it runs in the debugger?

8. What is the command required to run gdb on program "testing"?

9. What is the gdb command to set a breakpoint at a particular line number in program testing?

10. What is the gdb command to display the value of a variable?

6.1.3 Post Test

Post Test

Note: You may review your revised file buggy.c while answering these questions.

1) What is the nature of the bug in *Print2DArray*?

   a) The ROWS parameter value is incorrect
   b) Loop control variable *r* has the wrong start value.
   c) Loop control variable *r* goes out of bounds.
   d) Loop control variable *c* goes out of bounds.
   e) Loop control variable *c* has the wrong start value.
   f) I was not able to determine the nature of the error
   g) *Print2DArray* does not contain an error

2) What is the nature of the error in function *Read2DArray*?

   a) loop control variable *r* goes out of bounds
   b) loop control variable *c* goes out of bounds
   c) scanf should be reading into a[r][c] rather than &a[r][c]
   d) scanf is reading into the wrong type
   e) the ROWS parameter value is incorrect
   f) I was not able to determine the nature of the error
   g) *Read2DArray* does not contain an error

3) What is the nature of the error in *Max2DArray*?

   a) loop control variable *r* accesses the wrong number of rows
   b) loop control variable *c* accesses the wrong number of columns
   c) the comparison of a[r][c] with max should have used <=
   d) the comparison of a[r][c] with max should have used >
   e) the comparison of a[r][c] should have used >=
   f) I was not able to determine the nature of the error
   g) *Max2DArray* does not contain an error

4) What is the nature of the error in *SumOfArray*?
   a) the **for** loop using variable *r* accesses the wrong number of rows
   b) the **for** loop using variable *c* accesses the wrong number of columns
   c) the outer for loop is missing curly braces
   d) the inner for loop is missing curly braces
   e) the comparison of a[r][c] should have used >=
   f) I was not able to determine the nature of the error
   g) *SumOfArray* does not contain an error

5) What is the nature of the error in *MaxOfRow*?
   a) the **for** loop accesses the wrong number of rows
   b) the **for** loop is missing curly braces
   c) the **if** statement is missing curly braces
   d) the loop condition should be **`c <= 3`**
   e) the **if** condition should (**`a[searched_row][c] > max`**)
   f) I was not able to determine the nature of the error
   g) *MaxOfRow* does not contain an error

6) What is the nature of the error in "*SearchArray*"?
   h) the **for** loop using variable *r* accesses the wrong number of rows
   i) the **for** loop using variable *c* accesses the wrong number of columns
   j) the outer for loop is missing curly braces
   k) the inner for loop is missing curly braces
   l) the printf statement should be outside any loop
   m) I was not able to determine the nature of the error
   n) *SearchArray* does not contain an error

7) What is the nature of the error in "*SearchArrayCol*"?

   a) the **for** loop accesses the wrong number of rows
   b) the input paramater ROWS in incorrect
   c) the input parameter searched_col is incorrect
   d) the comparison of a[r][searched_col] should have used ==
   e) the comparision of a[r][searched_col] should have used >=
   f) I was not able to determine the nature of the error
   g) *SearchedArrayCol* does not contain an error

8) In function *DebuggingCase* ,
   o) elements of b[][] for which values are not provided are initialized to 0.
   p) elements of b[][] for which values are not provided have random values.
   q) different rows contain different numbers of elements
   r) different columns contain different numbers of elements
   s) the failure to initialize array elements causes a compiler error
   t) the failure to initialize array elements causes a runtime error
   u) none of the above

Bibliography

[1] D. Knuth, "The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching," 1968.

[2] L. Williams and R. Kessler, *Pair programming illuminated.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] G. D. Bergland, "A guided tour of program design methodologies," *Computer*, vol. 14, no. 10, pp. 13–37, 1981.

[4] L. A. Williams and R. R. Kessler, "All i really need to know about pair programming i learned in kindergarten," *Communications of the ACM*, vol. 43, no. 5, pp. 108–114, 2000.

[5] J. T. Nosek, "The case for collaborative programming," *Communications of the ACM*, vol. 41, no. 3, pp. 105–108, 1998.

[6] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the case for pair programming," *IEEE software*, vol. 17, no. 4, pp. 19–25, 2000.

[7] J. C. Carver, L. Henderson, L. He, J. Hodges, and D. Reese, "Increased retention of early computer science and software engineering students using pair programming," in *Software Engineering Education &amp; Training, 2007. CSEET'07. 20th Conference on.* IEEE, 2007, pp. 115–122.

[8] E. Mendes, L. Al-Fakhri, and A. Luxton-Reilly, "A replicated experiment of pair-programming in a 2nd-year software development and design computer science course," in *ACM SIGCSE Bulletin*, vol. 38, no. 3. ACM, 2006, pp. 108–112.

[9] L. L. Werner, B. Hanks, and C. McDowell, "Pair-programming helps female computer science students," *Journal on Educational Resources in Computing (JERIC)*, vol. 4, no. 1, p. 4, 2004.

[10] E. Arisholm, H. Gallis, T. Dyba, and D. I. Sjoberg, "Evaluating pair programming with respect to system complexity and programmer expertise," *Software Engineering, IEEE Transactions on*, vol. 33, no. 2, pp. 65–86, 2007.

[11] K. Beck, "Embracing change with extreme programming," *Computer*, vol. 32, no. 10, pp. 70–77, 1999.

[12] ——, *Extreme programming explained: embrace change.* Addison-Wesley Professional, 2000.

[13] N. Nagappan, L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller, and S. Balik, "Improving the cs1 experience with pair programming," in *ACM SIGCSE Bulletin*, vol. 35, no. 1. ACM, 2003, pp. 359–362.

[14] A. Robins, P. Haden, and S. Garner, "Problem distributions in a cs1 course," in *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52.* Australian Computer Society, Inc., 2006, pp. 165–173.

[15] B. Hanks, "Problems encountered by novice pair programmers," *Journal on Educational Resources in Computing (JERIC)*, vol. 7, no. 4, p. 2, 2008.

[16] L. Williams and R. L. Upchurch, "In support of student pair-programming," *ACM SIGCSE Bulletin*, vol. 33, no. 1, pp. 327–331, 2001.

[17] A. Cockburn and L. Williams, “The costs and benefits of pair programming,” *Extreme programming examined*, pp. 223–247, 2000.

[18] E. Mendes, L. B. Al-Fakhri, and A. Luxton-Reilly, “Investigating pair-programming in a 2 nd-year software development and design computer science course,” in *ACM SIGCSE Bulletin*, vol. 37, no. 3. ACM, 2005, pp. 296–300.

[19] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald, “Pair programming improves student retention, confidence, and program quality,” *Communications of the ACM*, vol. 49, no. 8, pp. 90–95, 2006.

[20] C. McDowell, L. Werner, H. Bullock, and J. Fernald, “The effects of pair-programming on performance in an introductory programming course,” in *ACM SIGCSE Bulletin*, vol. 34, no. 1. ACM, 2002, pp. 38–42.

[21] J. D. Gould and P. Drongowski, “An exploratory study of computer program debugging,” *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 16, no. 3, pp. 258–277, 1974.

[22] J. D. Gould, “Some psychological evidence on how people debug computer programs,” *International Journal of Man-Machine Studies*, vol. 7, no. 2, pp. 151–182, 1975.

[23] I. R. Katz and J. R. Anderson, “Debugging: An analysis of bug-location strategies,” *Human-Computer Interaction*, vol. 3, no. 4, pp. 351–399, 1987.

[24] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, “Debugging: a review of the literature from an educational perspective,” *Computer Science Education*, vol. 18, no. 2, pp. 67–92, 2008.

[25] M. Ducasse and A.-M. Emde, "A review of automated debugging systems: knowledge, strategies and techniques," in *Proceedings of the 10th international conference on Software engineering.* IEEE Computer Society Press, 1988, pp. 162–171.

[26] L. Murphy, S. Fitzgerald, B. Hanks, and R. McCauley, "Pair debugging: a transactive discourse analysis," in *Proceedings of the Sixth international workshop on Computing education research.* ACM, 2010, pp. 51–58.

[27] T. Chintakovid, S. Wiedenbeck, M. Burnett, and V. Grigoreanu, "Pair collaboration in end-user debugging," in *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on.* IEEE, 2006, pp. 3–10.

[28] D. E. Knuth, "The errors of tex," *Software: Practice and Experience*, vol. 19, no. 7, pp. 607–685, 1989.

[29] D. Perkins and F. Martin, "Fragile knowledge and neglected strategies in novice programmers," in *first workshop on empirical studies of programmers on Empirical studies of programmers*, 1986, pp. 213–229.

[30] A. von Mayrhauser and A. M. Vans, "Industrial experience with an integrated code comprehension model," *Software Engineering Journal*, vol. 10, no. 5, pp. 171–182, 1995.

[31] R. Jeffries, "A comparison of the debugging behavior of expert and novice programmers," in *Proceedings of AERA annual meeting*, 1982.

[32] C. M. Kessler and J. R. Anderson, "A model of novice debugging in lisp," in *Proceedings of the First Workshop on Empirical Studies of Programmers*, 1986, pp. 198–212.

[33] L. Gugerty and G. M. Olson, "Comprehension differences in debugging by skilled and novice programmers," in *Papers presented at the first workshop on empirical studies of*

programmers on Empirical studies of programmers. Ablex Publishing Corp., 1986, pp. 13–27.

[34] D. Klahr and S. M. Carver, "Cognitive objectives in a logo debugging curriculum: Instruction, learning, and transfer," *Cognitive Psychology*, vol. 20, no. 3, pp. 362–404, 1988.

[35] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[36] M. M. Müller, "Are reviews an alternative to pair programming?" *Empirical Software Engineering*, vol. 9, no. 4, pp. 335–351, 2004.

[37] M. W. Berkowitz and J. C. Gibbs, "Measuring the developmental features of moral discussion," *Merrill-Palmer Quarterly (1982-)*, pp. 399–410, 1983.

[38] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[39] J. E. Hale, S. Sharpe, and D. P. Hale, "An evaluation of the cognitive processes of programmers engaged in software debugging," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 2, pp. 73–91, 1999.

[40] C. Kehoe, J. Stasko, and A. Taylor, "Rethinking the evaluation of algorithm animations as learning aids: an observational study," *International Journal of Human-Computer Studies*, vol. 54, no. 2, pp. 265–284, 2001.

[41] C. Gama, "Helping students to help themselves: a pilot experiment on the ways of increasing metacognitive awareness in problem solving," in *Proceedings of the International Conference on New Technologies in Science Education. Aveiro, Portugal.* Citeseer, 2001.

[42] J. Hughes and S. Parkes, "Trends in the use of verbal protocol analysis in software engineering research," *Behaviour &amp; Information Technology*, vol. 22, no. 2, pp. 127–140, 2003.

[43] T. VanDeGrift, "Coupling pair programming and writing: learning about students' perceptions and processes," *ACM SIGCSE Bulletin*, vol. 36, no. 1, pp. 2–6, 2004.

[44] J. Vanhanen and C. Lassenius, "Effects of pair programming at the development team level: an experiment," in *Empirical Software Engineering, 2005. 2005 International Symposium on.* IEEE, 2005, pp. 10–pp.

[45] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 721–734, 2002.

[46] Y. B.-D. Kolikant and M. Mussai, ""so my program doesn't run!" definition, origins, and practical expressions of students'(mis) conceptions of correctness," *Computer Science Education*, vol. 18, no. 2, pp. 135–151, 2008.

[47] E. A. Chaparro, A. Yuksel, P. Romero, and S. Bryant, "Factors affecting the perceived effectiveness of pair programming in higher education," in *Proc. PPIG.* Citeseer, 2005, pp. 5–18.

[48] B. Simon, D. Bouvier, T.-Y. Chen, G. Lewandowski, R. McCartney, and K. Sanders, "Common sense computing (episode 4): debugging," *Computer Science Education*, vol. 18, no. 2, pp. 117–133, 2008.

[49] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander, "Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies," in *ACM SIGCSE Bulletin*, vol. 40, no. 1. ACM, 2008, pp. 163–167.

[50] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers," *Computer Science Education*, vol. 18, no. 2, pp. 93–116, 2008.

[51] R. E. Mayer, "The psychology of how novices learn computer programming," *ACM Computing Surveys (CSUR)*, vol. 13, no. 1, pp. 121–141, 1981.

[52] D. E. Knuth, "Computer programming as an art," in *ACM Turing award lectures.* ACM, 2007, p. 1974.

[53] M. E. Fagan, "Design and code inspections to reduce errors in program development," in *Pioneers and Their Contributions to Software Engineering.* Springer, 2001, pp. 301–334.

[54] W. W. Royce, "Managing the development of large software systems," in *proceedings of IEEE WESCON*, vol. 26, no. 8. Los Angeles, 1970.