5-2020

Using Imagery to Improve Program Quality in Computer Programming

Joseph S. Ditton
Utah State University

USING IMAGERY TO IMPROVE PROGRAM QUALITY IN COMPUTER PROGRAMMING

by

Joseph S. Ditton

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

John Edwards, Ph.D. Major Professor	Stephen Clyde, Ph.D. Committee Member
Dean Mathais, Ph.D. Committee Member	Richard S. Inouye, Ph.D. Vice Provost for Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2020

ABSTRACT

Using Imagery to Improve Program Quality in Computer Programming

by

Joseph S. Ditton, Master of Science

Utah State University, 2020

Major Professor: John Edwards, Ph.D.
Department: Computer Science

This research represents a new area of study in computer science that has never been investigated. We explore the idea that watching a screen recording of yourself writing computer code that solves a specific problem can increase the speed and quality of a subsequent attempt at solving a similar problem. A new software called Phanon has made this easier by providing an efficient way to store video-like recordings of software development. Due to the novelty of this research question, the study was designed as an exploratory study with the intent of discovering new research questions as well as to provide justification for future, large scale experimentation. We are not making any assertions about the statistical significance of the results but do find that the results provide insight into the meta-cognitive processes of the participants and suggest that using imagery can have a positive affect on student engagement and can potentially have a small affect on program quality.

(71 pages)

PUBLIC ABSTRACT

Using Imagery to Improve Program Quality in Computer Programming

Joseph S. Ditton

A common practice in sports is to review film footage of a game. This is done in the hope that individuals on the team or the a team as a whole might learning something about technique, form, process, or strategy that they either did well on or need to improve on. We believe that the benefits of doing this could extend to computer science, and more specifically, the process of writing a computer program. This has never been done in the past, as such this research is exploratory in nature. A new tool called Phanon can record someone writing a computer program at a small fraction of the cost of recording video, which will make it easier to reproduce this research later at a larger scale. In this paper we explore the effects that watching video recordings of software development can have on the quality of new software being developed. While the results not conclusive we do suggest that there are beneficial effects and that this area of study deserves further exploration.

To my wife, for pushing me along these last few steps...

ACKNOWLEDGMENTS

First, I would like to express my thanks to my advisor, Dr. John Edwards, for the hours and hours he has spent helping me get this point. He has been patient and encouraging throughout every step. Its hard to imagine a USU without him. I would also like to thank my amazing boss, Joel Duffin. His passion for education inspired me to return to school for my Master's Degree. This thesis never would have been possible without the flexibility given to me by Joel at work. Finally, I would like to express my gratitude and love to my family for all the support.

Joseph S. Ditton

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Background

Imagery is a technique used commonly in sports where the athlete rehearses their sport in real-time either in their mind, or through some other visualization activity like a video recording. We claim that some of the benefits of doing imagery could be applicable to computer science as a discipline. For example, a basketball player might watch a recording where they are subject that shows them shooting free-throws over and over. This would help cement the motion of shooting a free-throw in their minds. Computer scientists might be able to do the same thing, but instead of shooting free-throws they could be writing for-loops. More than just cementing programming language syntax we believe that imagery could also be used to provide insight into the software development and problem solving processes of each computer scientist. This could be useful not only in the field, but during the education process also.

Currently, the closest thing to imagery that we, as computer scientists, use is code review. However, self code review, without the real-time element, has limited uses. It can be valuable as a tool to help identify mistakes in an end product, but does little to help identify mistakes or areas of improvement in the software development process. In this research we compare the benefits of self code review with the benefits of real-time imagery. If using imagery in computer science proves to be successful then it could change how we teach and how we learn computer science. For example, as part of each homework assignment, students could be required to watch a recording of themselves completing the assignment and asked identify areas of improvement.

Using imagery in computer science is a completely novel idea that we thought of and it has not been explored before. This presents some obvious challenges as we will have little

data to compare our results to and few resources to help us avoid potential pitfalls. We carefully designed our research with this in mind.

1.2 Phanon

In the past, recording yourself doing software development was impractical and expensive due to the large amount of storage space required for video data. However, a new tool called Phanon has eliminated this problem by capturing keystroke data. With this new technology we can record hundreds of hours of software development at a small fraction of the cost of recording video. Phanon, which was developed by Dr. Edwards and myself, was designed to be used by introductory computer science students and gathers data with the purpose of learning about the efficacy of repetitive syntax exercises. Through analysis of collected keystroke data we learned that the data can be manipulated to display a video-like recording of the student's code editor.

The keystroke data could be more accurately described as editor change event data. For example, if a user enters "ctrl+v" to paste something into their program the event that will be recorded will be a "paste" event rather than a separate event for both the "ctrl" key press and the "v" key press. Other event types include, but are not limited to, "+input", "delete", and "cut". Each event has associated "added" and "removed" fields which contain information about how and where the state of the editor changed. Finally, a UNIX timestamp is recorded with each event. In addition to editor change events Phanon records whenever a student runs their program. This record includes the program input, program output, and whether the program encountered any errors.

This format provides some obvious advantages over a screen recording. The primary advantage is that only the "interesting" events are recorded. This is how Phanon reduces recording size. A video recording will capture all of the "uninteresting" events such as the time it takes fingers to move between keys. A majority of a video recording will be comprised of these uninteresting events. Another advantage is that the data can be interpreted by a computer in meaningful ways, not just to reproduce a recording. All of the recordings are also fully interactive, meaning that the viewer can pause the recording and run the

program at what ever state it was in when they paused. While paused they can also modify the program if they want to try something. Then when resuming playback the program will revert to the state it was in before it was paused.

1.3 Potential Applications

We believe that these recordings can be useful in many different contexts but are focusing on using them for educational benefits. Consider this example; A teacher assigns a homework assignment to student A and student B. The teacher expects this assignment to take around 5 hours to complete. Student A starts early and completes the assignment in 5 hours without any serious issues. Student B also starts early but runs into many issues and it takes them 20 hours to turn in a completed assignment. Both students turn in an excellent solution and both get a good grade. From the teacher's perspective, both students are doing well in the class when in reality student B is struggling and needs intervention. The tool can use the metadata stored with the recordings to identify that student B took longer than normal and bring this to the teacher's attention. The teacher could then watch the recording of student B at a fast speed and would quickly determine that student B needs help. Since the recorded data has information about what errors the student encountered the teacher could easily see what issues student B was blocked on and could provide personalized instruction to help that student improve. The tool could also use the number of errors a student encountered as a possible metric to identify struggling students.

Another useful application of this recording data is to help identify cases of cheating. Anomalies in the editor change data such as large pastes or long stretches of consistent typing could indicate that a student took something from the internet or that they are copying something from another students screen.

There has been a lot of research into the use of worked examples across many disciplines including computer science [1]. Worked examples come in many forms, for example: a worked example in math might go through the solution of a problem up to a certain point and then ask the student to perform the next step. Or, it will show the solution up to a

point and then ask the student to identify where the mistake is in the example. Currently, one of Phanon's features delivers worked example-like exercises to students by giving them part of a programming construct and asking them to fill in the blanks. These exercises, however, are intentionally designed to only teach programming syntax. We believe this will be effective as it removes problem solving and allows students to focus on the syntax. To supplement this, the Phanon recordings could be used as interactive worked examples that can walk students through complex problem solving. Referring back to the math example mentioned earlier, we could do something similar by having students watch a recording of a program that solves a problem up to a certain point, then ask them to finish the program.

The above examples are use cases that could be implemented at an institutional level. I, however, was interesting in also using these recordings at a personal level. As a individual, I would most likely only have access to recordings that I produced myself meaning that I was the subject of the recording. Is it possible to learn something by watching a recording of yourself?

1.4 Related Work

The area of research that is the most similar and serves as the inspiration to this thesis is the use of internal and external imagery in sports and physical performance [2]. A ubiquitous practice in sports is to review game film (external imagery) with the goal of self-evaluation. Hale suggests that this practice, while potentially beneficial, is not as beneficial as imagining oneself in the act of performing the physical task (internal imagery) be it lifting weights or throwing a ball. This idea is backed by research that shows that the primary motor cortex in the brain is stimulated when motion is imagined [3]. It is this functional equivalence that might explain why imagery has beneficial affects on the performance of athletes. Nevertheless, this thesis focuses on external imagery because it is easier to compare external imagery to existing practices in computer science like code review.

Using imagery as a meta-cognitive tool seems to have more of an affect on elite athletes rather than novice athletes [4]. Additionally, Arvenin-Barrow suggests that imagery is more

effective in closed-skill sports rather than open-skill sports. Closed-skill sports are those like golf or ice-skating. These sports involve performing a routine or repetitive action and the more consistent that action is the better the performance of the sport will be. Other sports like wrestling or basketball are open-skill sports. While these sports have some closed-skills, like shooting free-throws for example, the majority of the sport involves a broad range of often personalized skills that change their implementation based on game state. Environment also plays a greater factor in open-skill sports while closed-skills are largely environment invariant.

Thinking about these two types of sports, it makes sense that imagery would be more effective with closed-skills rather than open-skills. This is because it is much easier to imagine a correct performance of a closed-skill than an open-skill. Consider the sports of golf and basketball as an example. In basketball there are far more variables than in golf. Once the ball is in-bounded the correct action to perform is largely out of the ball handler's control. The actions and positions of opposing players and team mates, as well as the current score and remaining time on the clock are all arguments for the complicated function that will determine a players action. Golf players on the other hand, while they still have to account for variables such as distance and wind speed, the success of a shot is largely determined by how well the player can perform that singular swinging action. Thus, it seems logical that imagining a correct outcome is more plausible for closed-skills. This ability to imagine a correct performance is the same reason that imagery works better for experts than for novices, because experts have more of an idea of what a correct performance looks like than novices do.

In comparing the process of writing a computer program to a sport, computer programming seems to fall under the category of open-skilled rather than closed-skilled. Though, for a given programming language you could label many of the different program constructs as closed-skills. The processes of writing a for-loop or an if-statement are developed in much the same way that a closed-skill like a golf-swing would be developed. Consistent and repetitive practice is what gives players and programmers mastery over those skills.

As such, it stands to reason that both internal and external imagery could be effective in mastering skills in computer programming.

Some research shows that, while imagery is effective on its own, its effects can be enhanced by practicing imagery in the same conditions that you would be physically performing the action [5]. This enhanced imagery is referred to as PETTLEP-based imagery. The PETTLEP acronym stands for Physical, Environment, Task, Timing, Learning, Emotion, and Perspective. Consider this example; an ice-skater that is practicing using PETTLEP-based imagery will be at an ice-rink, standing on the ice, wearing their ice-skates, and wearing the same clothes they would normally wear during a performance at the same time of day they would be performing. If they are practicing a routine they might move to spot that they anticipate performing a certain skill before imagining themselves performing the action. This is an idea that can translate directly to using imagery in computer programming. Our experimentation was performed in a similar environment that programmers might be programming in normally. However, the purpose of our experiment was not to identify whether or not environment affects program quality. This could represent an area of future research.

Another interesting study shows that athletes were more likely to use imagery in the context of competition than in practice [6]. In that same context, they often imagined themselves winning or receiving a reward and almost never imagined themselves in a losing situation. This presents another possible research question; How can using imagery improve performance in a competitive programming or high pressure professional context?

This discussion of imagery leads into the broader category of meta-cognition. Meta-cognition is defined as "knowledge concerning one's own cognitive processes and products or anything related to them" [7]. This definition is widely referenced across a broad range of meta-cognition studies. Researchers at the University of Limerick in Ireland concluded that meta-cognition, imagery, and expertise are tightly coupled [8]. They also emphasize the importance of meta-imagery, which is thinking about your own imagery process and how to improve it.

Computer science is a discipline that suffers quite heavily from what some researchers call "the shallow learning problem" [9]. A comedic anecdote tells of programmers that state, "My code is broken and I have no idea why." Then after doing some more work they exclaim, "My code is working and I have no idea why!". This situation, which is likely familiar to any computer programmer, is a mostly accurate portrayal of the shallow learning problem. Students, especially in introductory courses will commonly write, or borrow, code they don't understand, despite knowing that it is essential. One study [9] reports that using self-explanation as a meta-cognitive strategy can help students combat, or at least bring awareness to the shallow learning problem. Using self-explanation could also be valuable in conjunction with watching Phanon recordings, and could represent yet another area of future research.

1.5 Research Question

My research question is: How would watching a screen recording of yourself writing computer code that solves a specific problem increase the speed and quality of a subsequent attempt at solving a similar problem? This research question prompted the design and execution of the experiment described in the next chapter. While our primary goal was to answer this question we identified several other questions and hypotheses that our experiment could potentially provide some insight into.

CHAPTER 2
EXPERIMENT DESIGN

2.1 Hypotheses

We started this study with three hypotheses:

1. The test group would improve on the second challenge more than the control group. We will be measuring improvement by comparing each group's completion times on the second challenge with their times on the first challenge as well as comparing the number of errors each participant made. The null hypothesis is that there will be no statistical difference in improvement between the test group, who were given an imagery based activity to complete between challenge attempts, and the control group, who participated in a static, self code review.

2. The test group would find their review activity more engaging than the control group. This will be measured by asking the participants a likert-scale survey question that identifies how engaging each participant thought their review activity was. We will also measure this by coding the participants' answers to a free response question and analyzing how many participants mentioned that they thought their review activity was engaging or not engaging. The null hypothesis is that the participants in the test group will not find their imagery based activity any more or less engaging than than the control group finds their code-review activity.

3. The test group would approach the second challenge differently while the control group would approach both challenges more or less the same. We will measure this by sequencing the recordings produced by each participant and analyzing the frequency that certain patterns occur in each sequence. The null hypothesis is that there is no measurable difference in how each participant in the test group changed their approach

relative to the control group. This could mean that every participant changed their approach irrespective of group or none of the participants did so.

The experiment we performed required that a number of students, about 20, volunteer to participate and we had 19 volunteers. Because of the small sample size it is difficult to make any statistically significant assertions about the results. We anticipated this before hand and decided to cast the research as exploratory in nature. Given the novelty of the research question this is an appropriate approach. Performing several power analyses with different parameters lead to me estimate that I would need at least 120 participants in order to show a statistical difference between the two groups observed in the after-mentioned experiment. This experiment is also targeting those with an intermediate or higher skill level in computer programming. Given the observation that imagery works better with experts as opposed to novices, this seemed like the right target population. However, we still think it would be insightful to attempt the same experiment with introductory computer science students.

2.2 Experiment Design

The experiment performed was as follows: volunteers were split into two groups, a control group and a test group. Group assignment was performed automatically by the Phanon software. The test group was given a carefully designed Python programming challenge and had 15 minutes to complete it. The programming challenges are described in more detail in the following section and the complete challenges are in Appendix A. Upon completion, each individual in the test group then watched a recording of themselves doing the challenge. The recording played at 4x speed. This allowed the recording to be completed within five minutes. They were instructed to identify areas where they might improve their design and code quality. Once the test group was done watching the playback, they attempted a similar programming challenge with the same constraints.

The control group was given the same programming challenges with the same constraints as the test group. However, upon completion of the first challenge the control

group did not watch a recording of themselves. They had five minutes to review their completed challenge. During this review time they were also instructed to identify areas where they might improve their design and code quality.

Following the completion of the second challenge both groups were given a survey. The goal of the survey was twofold: First, it would help me assess the quality of the programming challenges, and second it would provide some insight on what the participants were thinking about while watching the playback or during the code review. The survey consisted of the following questions:

1. A likert question asking whether they thought the first or second challenge was harder.
2. A likert question asking whether they enjoyed the first or second challenge more.
3. A likert question asking how useful the review time was (either playback or code review).
4. A likert question asking how engaging the review time was (either playback or code review).
5. A free response question asking them to describe the thoughts they had about the review time (either playback or code review).

2.3 Programming Challenge Design

The problem with the programming challenges that are common at hack-a-thons and programming competitions is that they are designed to test problem solving and not necessarily programming. This results in the majority of time being spent thinking about or modeling the problem and not actually writing code. While these skills are valuable, Phanon cannot provide near as much insight into errors made in the problem-solving processes that don't involve writing code. To that end, each of the programming challenges were designed to maximize the amount of code that would be written during the 15-minute time limit. This resulted in programming challenges that were, theoretically, easier but took longer to implement.

In each of the programming challenges, participants implemented a relatively simple, but complete software system that was modelled after a real-world object. First, they implemented a software representation of a microwave oven, second was an elevator control system. Both challenges were largely inspired by my time as a graduate teaching assistant in Dr. Stephen Clyde's object-oriented programming class here at Utah State. Neither of these examples are technically difficult. However, the time constraint forced participants to think critically at the same time as writing the code. We both anticipated and observed that participants took a few moments at the start of each challenge to read and understand the problem space.

The primary motivation behind doing two different challenges instead of repeating the same challenge is that a second attempt on any problem will be significantly easier than the first. Some participants, especially those in test group, might have even memorized their solution after doing it once and simply regurgitated that solution for the second attempt. However, doing two separate challenges was not without risk either. They needed to be designed carefully enough that the difficulty was nearly identical between the two. If one was much harder than the other it would have been impossible to measure any kind of improvement between the two. After reviewing the survey results, we believe that the challenges were roughly equal in terms of difficulty. We will explore this data later in this thesis.

The actual process of designing these challenges was an iterative one. The design process took place in Phanon's authoring tools. At first, I had written a rigid set of instructions for each programming challenge. Using the microwave challenge as an example, the instructions told them what functions a microwave had, and how to use each function. They also described what each function should return and what the conditions were in which the function would execute either successfully or unsuccessfully. For example one line of the instructions read:

> "A MicrowaveOven has a method called insert_food which changes a flag in the MicrowaveOven that represents whether or not the MicrowaveOven has food

> inserted. You can only insert food if the door is open and the MicrowaveOven doesn't have food already inside of it."

Having instructions like that meant that participants would be spending much of the time reading and rereading the instructions which was contrary to our goal of maximizing the amount of code being written. These instructions also took away a lot of the problem solving required to complete the challenges. In the end we decided to give as little instruction as possible. Their final instructions told them names of each method they needed to implement with their arguments and return types, and that was all. We measured the correctness of their implementation through a series of unit tests and we decided to let the test output guide their problem solving. For example, if the participant's solution allowed food to be inserted into the microwave while the door was closed, then a unit test would fail and notify them why the failure took place. This was done in hopes that after one or two test failures they would have a good idea of what kinds of things they needed to think about while completing the challenges without having to spend a lot of time reading instructions.

2.4 Future Improvements of Experiment Design

During the experiment and while we were analyzing the data we identified a few things that were unanticipated when initially designing the experiment. While none of these issues affected the integrity of the data that we gathered, we could potentially have gathered more data or some different kinds of data if we had anticipated these things before hand. In this section we will address some of these issues and suggest how we would improve the experiment if we were to have to opportunity to do it again.

Initially we had decided to allow 20 minutes for each programming challenge but decided to instead allow only 15 minutes for each challenge. Our experiment took place in Dr. Edwards's CS 6830 class. This approach was beneficial for Dr. Edwards's class as he used it as an opportunity to teach the students how to gather and analyze data from an empirical study. The IRB board agreed that conducting the study in the classroom was ethical. This was also convenient because it provided us with our participants as well as our

venue. Unfortunately, it also limited the amount of time we had to perform the experiment because we could not go past the time when the class ended. We completed the experiment on time, but we had to rush the survey portion. Part of this was because we failed to ensure beforehand that all of the students would have a laptop. At the last minute we had to arrange for a laptop for one student, setting us back at least 5 minutes. Had we not had a hard schedule to keep, and had reminded the students to bring a laptop, we potentially could have got some more thorough responses during the free response portion of the survey. Should we have the opportunity to do this experiment again we would certainly want to do it when we have more time to allow for more thorough responses. Also, it would have been beneficial to allow for more time on each programming challenge. We anticipated some of the participants would not finish the first challenge but shortening the time from 20 to 15 minutes meant that only 3 participants completed the first challenge. This made it difficult to tell how much each participant improved between the two challenges.

In an effort keep participants from the control group from seeing what those in the test group were doing and visa versa we decided to physically separate the two groups during the experiment. The way we decided to do this was to display to each participant in the Phanon software UI which group they were in and then separate them accordingly. Because Phanon is being used by other people we did not want to modify the primary workflow and risk disrupting those clients. This combined with the fact that we were relying on Canvas to auto-provision a user for each participant meant that the only place in the UI that we could display the study group was on the screen where they did their first challenge. Before participants opened the challenge in Canvas they were instructed to not read any instructions on the page. They were only to look at which group they were in and move to the side of the room designated for their group. Once everyone was situated we started the timer and told them to begin by reading the instructions. Phanon recorded the time when they opened the programming challenge as their start time, however the first minute or so after they "started" was spent moving the their designated area. Normally, we would have used that timestamp to know when they started reading the instructions and we would

have been able to identify how long they spent reading the instructions before making their first change. Unfortunately, we have no way of knowing how much of that time was spent moving to their respective groups. When the results are reported we will calculate time spent from the timestamp of their first editor change. For future experiments, hopefully we will have the time to build some software that is dedicated to the experiment so we can change the workflow without disrupting existing clients.

Going along with the theme of not disrupting our existing clients, another sacrifice we had to make was that we couldn't allow any back door into the software that would allow participants to bypass any of the challenges. Our clients use Phanon to deliver syntax exercises to their students and we didn't want what I like to call the enterprising student (a student who is much more capable than their peers) to find this back door and exploit it. In order to bypass the tests for those who didn't complete the challenges, we authored in a back door into the unit tests for our specific programming challenges. If the participant typed a specific python command into their code it would bypass the tests allowing them to continue on with the next activity. In an effort to save time, we wrote this python command on the board roughly two minutes before time expired and instructed the participants to not add it to their code until time had expired. Unfortunately, one participant must not have heard this. They appear to have given up and added the code early. This doesn't really affect anything other than we now have to take this participant into account when we calculate how much time the participants spent. Having dedicated research software would have solved this problem as well.

The last issue we encountered didn't directly affect the data in any substantial way but did waste some of the valuable time we had to perform the experiment. The Phanon software is hosted on a small server and having 10 people load recordings of their first programming challenge all at once maxed out our memory allotment. Everyone in the test group was presented with an infinite loading screen. Fortunately, the resolution was simple. Instead of having every participant start the review at the same time we staggered them in groups of 2 or 3 to reduce the memory load on the server. This worked well and we just

had to make sure to offset the time allotment accordingly. The long term solution to this would be to ramp up the server to handle that load.

CHAPTER 3

ANALYZING EDITOR CHANGE EVENTS

3.1 Overview

In this chapter we will analyze the editor change events recorded for each participant. These events can help us determine how well each participant performed on each challenge. Generally speaking, we found no significant difference between the control group and the test group while exploring these results. We anticipated that this would be the case given our small sample size. However, there are some stories hidden in this data that are worth exploring more.

3.2 Time Spent

For the time spent metric we are measuring from the moment they make their first change to the moment they pass all of the tests. In the event they don't complete the challenge on time their completion time is determined to be 15 minutes. Fig. 3.1 and Fig. 3.2 display the distribution of time spent for each group. Note that each graph is comparing each group with itself for each challenge.

Like mentioned before, only three participants actually completed the first attempt, one from the test group and two from the control group. If you look closely at those graphs you might think that I have that backwards. In the control group there is only one obvious completion and this is the participant that finished the first challenge in 9 minutes. The second participant completed their first challenge 14.9 minutes so they got lumped in the bin for 15 minutes. Looking at the test group you might think there were two that completed the challenge also but in this case we had one participant that gave up before the time expired and used the bypass code when they weren't supposed to. We discussed this problem at the end of Chapter 2.

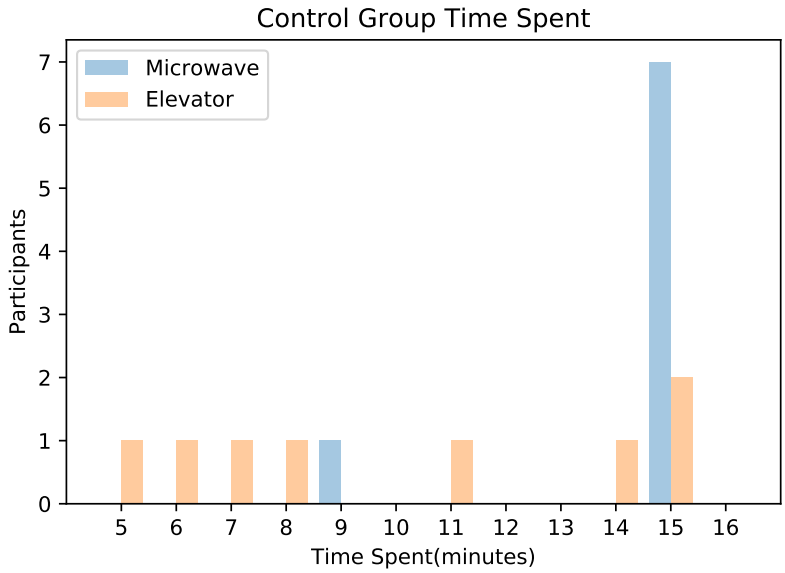

Fig. 3.1: Control Group Time Spent

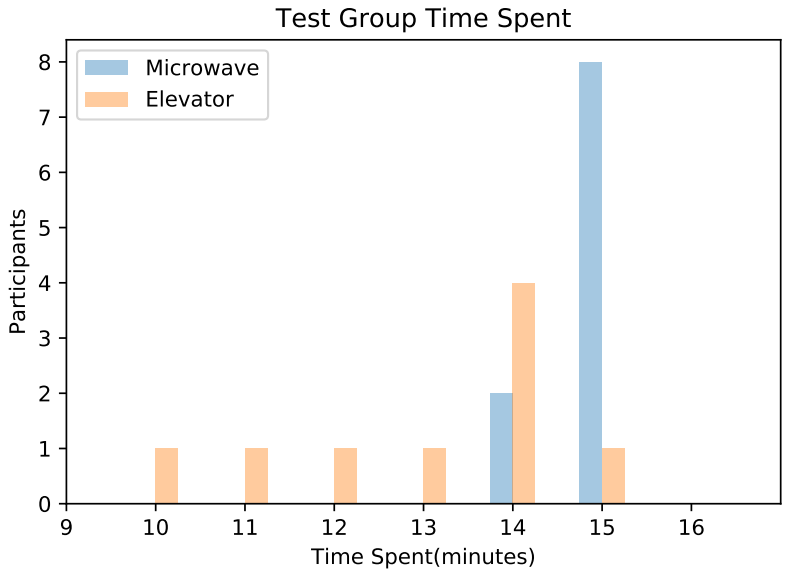

Fig. 3.2: Test Group Time Spent

As expected, both groups did better on the second challenge. Nine of the ten in the test group completed the second challenge with a median completion time of 13.3 minutes and six out of eight in the control group finished with a median of 10.1 minutes. If you remember, we had nine participants in the control group but we are only reporting the results for eight of them for this metric. While reviewing the recordings we discovered that one participant in the control group "cheated" and invalidated their results. While they didn't technically do anything that we strictly forbade them from doing, upon realizing that the microwave and elevator challenges were extremely similar they returned to the old challenge, copied their solution, then pasted it in to the second challenge. Maybe it would be more accurate to say that this participant was more clever than the others. Nevertheless, we opted to not report their results in this section to keep the data as pure as possible. This participant will still be included when exploring the results of the survey and of each of the recordings.

By enforcing a time-limit the results here are not very valuable because we cannot accurately measure how much each participant improved. Still, these results prompted exploration into some of the individual participants which could help inform future experiment design. Contrary to our hypothesis that the test group would improve more than the control group we observed that the control group improved more than the test group did. With this small of a sample size you could easily explain this by saying that the control group might have ended up with more capable people in it. To verify this, while exploring the recordings we focused in on the three participants in the control group that completed their second challenge in under eight minutes. One of these participants completed the first challenge in under 10 minutes so it's not surprising to see that participant finish in roughly 6 minutes on the second challenge. However, it's harder to explain what happened with the other two. Neither of them finished the first challenge but they were all close to completion when time expired. On the second challenge it must have just clicked for them and they breezed through it. All three had a few things in common: 1. They made very few if any syntax mistakes, 2. They made very few typing errors, and 3. They all developed the

program from top to bottom without pasting the example code. Implementing the program from top to bottom means that they fully implemented each method in the order that they were listed in the instructions. Compare this approach to someone who defined the shell of each method first then went back and fully implemented each method. It does appear that these three participants are among the more capable in our population, so that could help explain why the test group improved more.

3.3 Program Executions

Phanon recorded every time each participant executed their program along with the input provided to the program, the output the program gave, and whether or not there were any errors in the program. In this context errors specifically refers to syntax errors or run-time errors not whether or not the unit tests passed. Table 3.1 shows some statistics about program execution. The "Mic Total" and "Ele Total" columns refer to the median number of times each participant executed their program. The "Mic Error Ratio" and "Ele Error Ratio" columns show the median ratio of executions that resulted in an error.

	Mic Total	Mic Error Ratio	Ele Total	Ele Error Ratio
Control Group	13	.16	6.5	.2
Test Group	16.5	.14	13	.14

Table 3.1: Median Program Execution Statistics

Here we can see that the control group had a small increase in the ratio, but this is because they were not executing their programs near as often in the second challenge as the first challenge. The test group seemed to stay relatively consistent, they ran their programs a little less often but the ratio stayed about the same.

Fig. 3.3 and Fig. 3.4 display the distribution of the the ratio of executions that resulted in an error. When looking at the data from this perspective we can make some more observations.

We can see that two people in the control group and one in the test group encountered

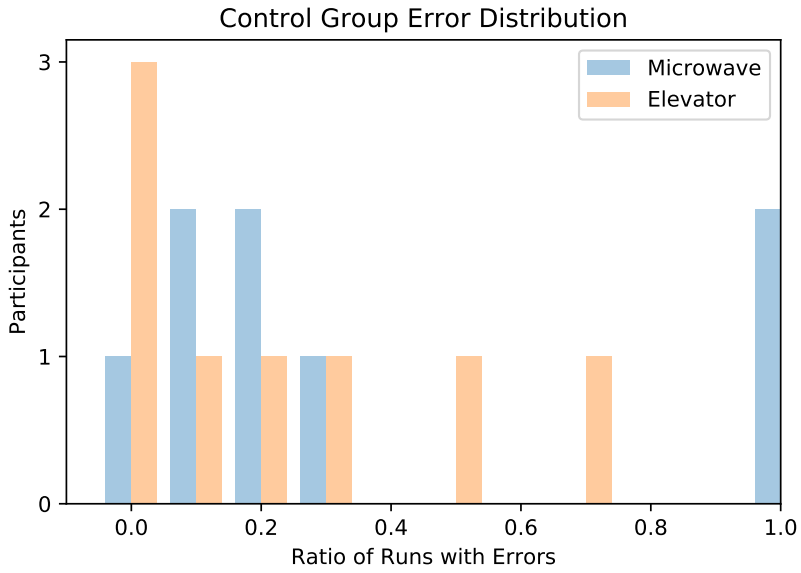

Fig. 3.3: Control Group Error Ratio

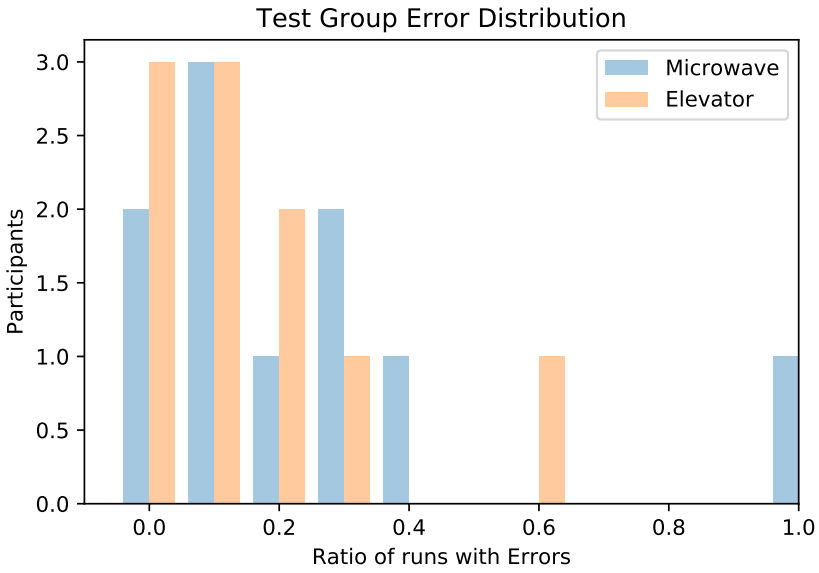

Fig. 3.4: Test Group Error Ratio

errors on every execution during the first challenge. Watching the recordings of these participants we saw that they each got hung up on syntax and they were never able to resolve it. One participant made a mistake on their first keystroke; they typed Class with capital "C" when declaring their microwave class.

3.4 Summary of Findings and Future Work

Unsurprisingly, these findings are not conclusive but do shed some light of the types of things that might be measurable if we were able to perform this experiment again with a larger population. The possibilities with the keystroke data are nearly limitless. One thing we would like to try would be to programmatically classify each programmer into one of a few categories based on their keystroke data. For example, a "tinkerer" is someone that doesn't encounter many errors but executes their program frequently and makes small changes in between program executions, meanwhile, a "hacker" could be someone that writes a lot of code between code executions. This work falls beyond the scope of this thesis, nevertheless, it could prove to beneficial to future research in this topic.

These results suffer due to some of the problems mentioned in the previous chapter. Specifically, shortening the challenges to 15 minutes made it impossible to compare the control and test group effectively because practically nobody finished the first challenge. At this point, assuming that future attempts at this experiment won't be under such a strict time constraint, I would suggest a small change to the experiment design. Rather than put a time limit on each challenge I would propose that we let each participant finish no matter how long it takes them, within reason obviously. This would give us a strong baseline to compare against. We would tell the participants that the goal is to finish as quickly as possible to encourage them to think critically at the same time as writing the code. This will make it easier for us to measure any effects that the playback might have on their programs.

CHAPTER 4

SURVEY RESPONSES

4.1 Overview

The survey was composed of four likert-scale questions and one free response question. The gathering of this data was fairly low-tech. The questions were asked in code comments presented to them in their text editor. The participants then just typed an "x" above the response they chose. Afterward, we went through each of the surveys and recorded their responses in a spreadsheet that allowed us to analyze the results. For each likert-scale question the participants' responses were recorded on an integer scale from 0 - 4. Each integer corresponds to one of the options for the question from left to right.

4.2 Challenge Difficulty

As mentioned earlier, it was crucial that the two programming challenges be similar in terms of difficulty or it would be impossible to measure any difference between the two. The tool we used to assess challenge difficulty was the survey that participants completed at the end of the experiment. The question that was presented to the student is visible in Fig. 4.1.

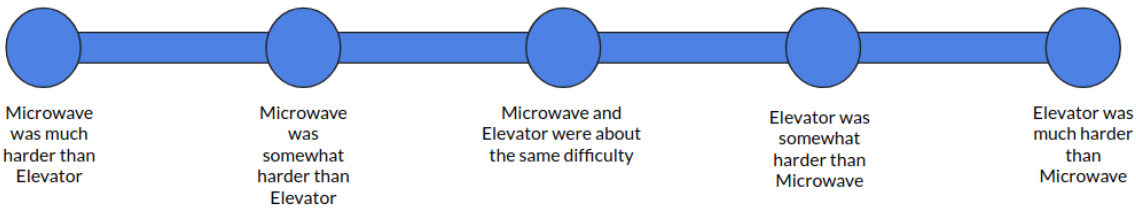

Fig. 4.1: Difficulty Survey Question

We found that, for the most part, participants agreed that the two challenges were similar in difficulty. Fig. 4.2 shows the distribution of the responses.

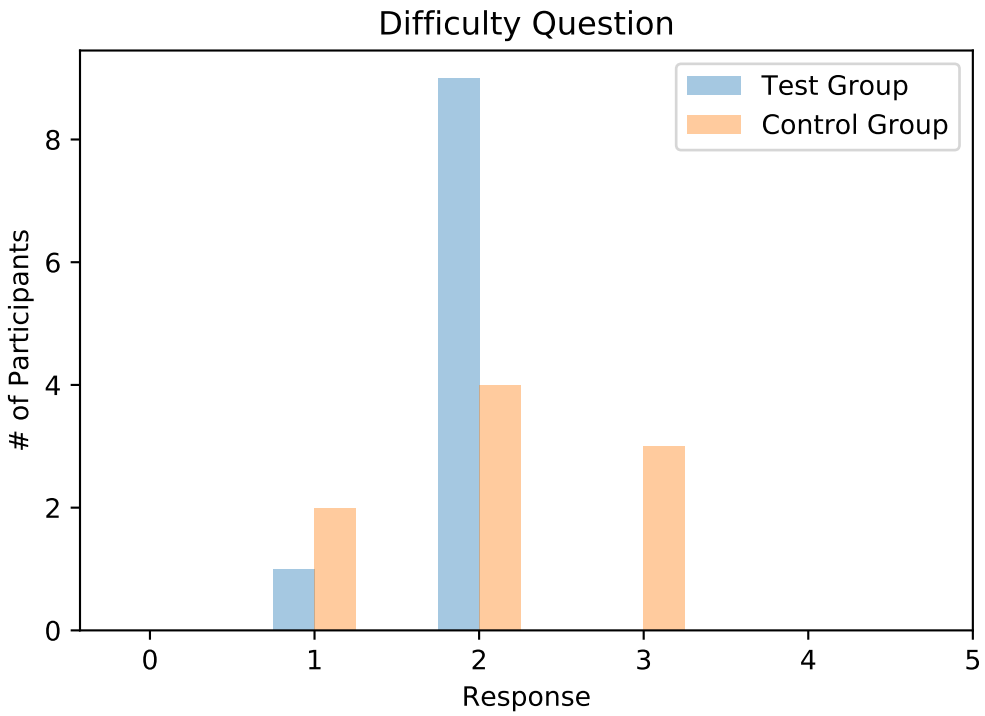

Fig. 4.2: Difficulty Survey Response Distribution

Now, it is worth noting that the survey was completed after the participants had completed both challenges as well as participated in their respective review activities. This could potentially mean that the review had some affect on the perceived difficulty of the second challenge and could explain why nobody in the test group identified the second challenge as harder at all while the 1/3 of the control group thought that the second challenge was somewhat harder. These are speculations not hard conclusions but it would be worth paying attention to this in future experiments with larger sample sizes. That being said, I feel confident based on the survey responses that the difficulty was similar between the two challenges.

4.3 Challenge Enjoyment

Fig. 4.3 shows the question that the participants were asked.

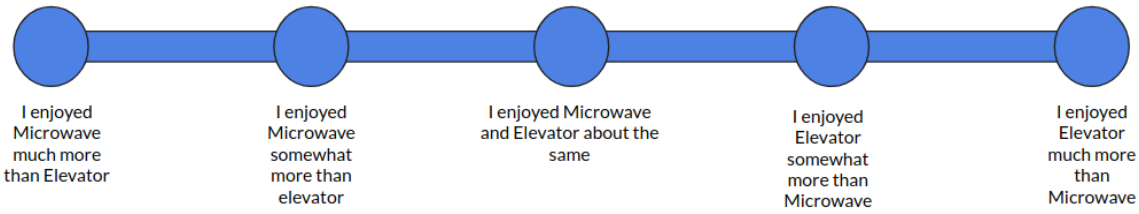

Fig. 4.3: Enjoyment Survey Question

Fig. 4.4 shows the distribution, its worth pointing out that one participant in the test group did not answer this question, most likely on accident.

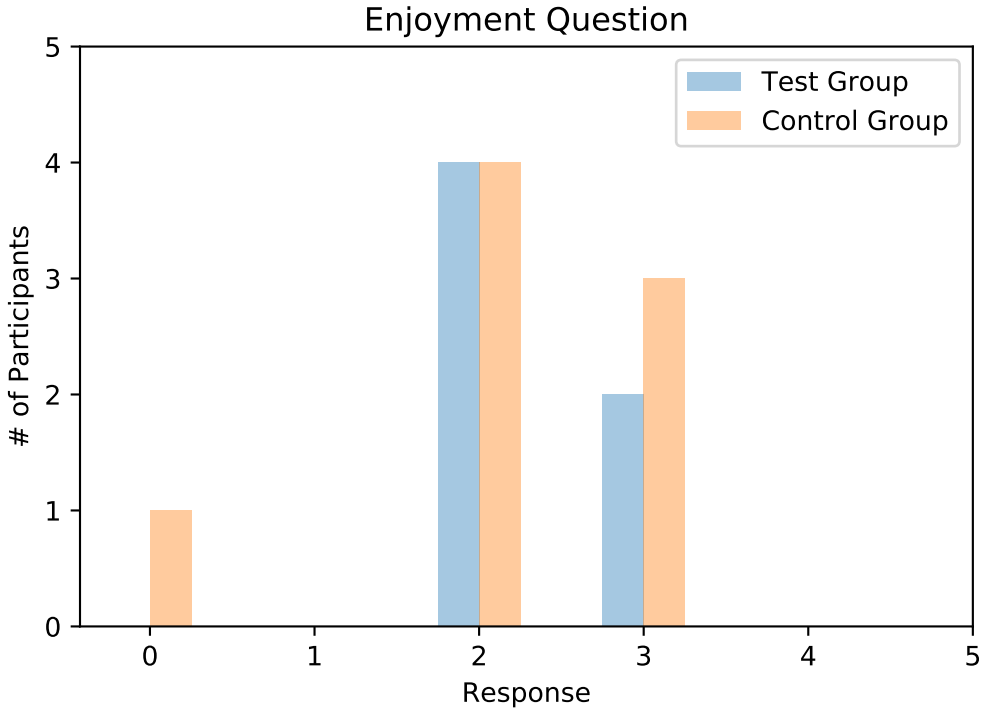

Fig. 4.4: Difficulty Survey Response Distribution

We see a similar distribution on this question as we did with the difficulty question where most felt they enjoyed the two challenges about the same. We spent some time digging in to the outlier that answered that they thought that the microwave challenge was much more enjoyable than the elevator challenge. This participant is also one of the

three that thought that the second challenge was harder than the first. While watching the recordings for this participant we noticed that they copied and pasted the starter code we gave them for the first challenge, but then wrote it from scratch on the second challenge and made several mistakes doing so. This participant is also not a very strong typist, in that they type relatively slow and make a lot of mistakes so they frequently backspace. So the decision to type out the starter code instead of pasting it given all of these things is interesting. Where this student was in the control group they didn't get the chance to watch the video of themself doing the first challenge. This makes me wonder if their approach on the second challenge would have been different had they been in the test group.

4.4 Usefulness Question

The third survey question asked them to identify how useful they thought the review time was. The exact question is displayed in Fig. 4.5.

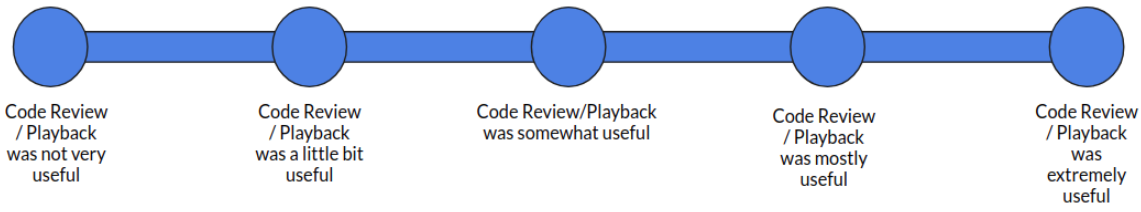

Fig. 4.5: Usefulness Survey Question

Fig. 4.6 shows the distribution of the responses, which, coincidentally, are identical between the two groups aside from the fact that there is one more participant in the test group. These results become more interesting when we look at the free response answers. The language of the free response answers seem to contradict some of the data here. We will go into this more when we discuss the free response answers in detail.

With this data we were also interested in some of the specific participants. Two participants in each group answered that they thought the review was useful, while half of each group thought the review was entirely useless. We wanted to see if we could identify if their were any noticeable differences between those who thought it was useful and those who

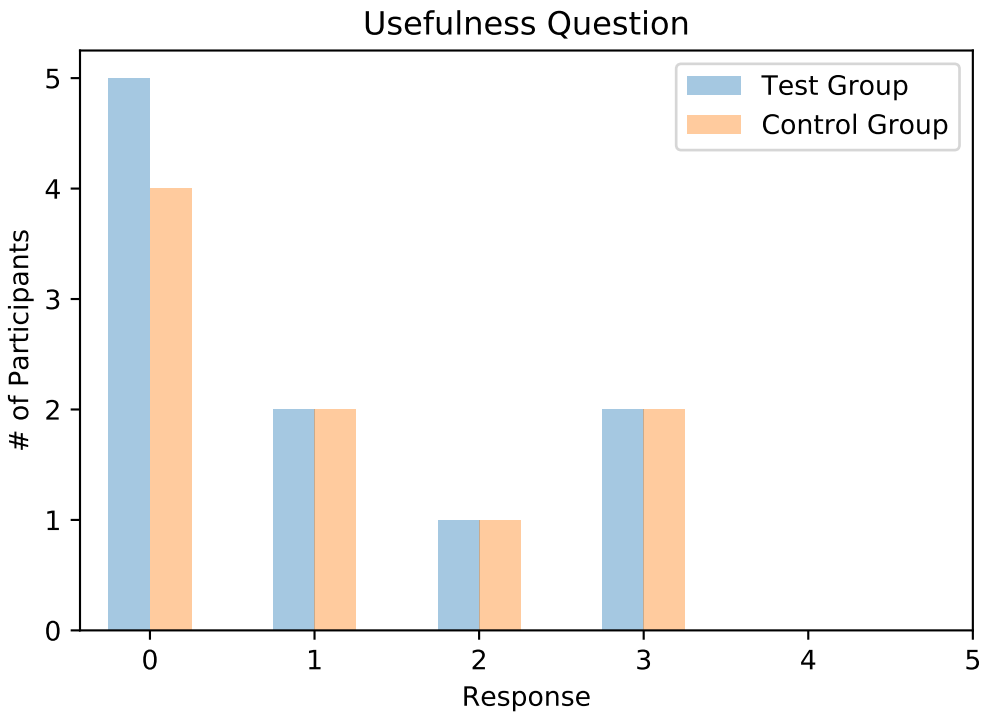

Fig. 4.6: Usefulness Survey Response Distribution

didn't. However, this proved to be difficult. Of the users that finished the first challenge, the one from the test group thought that the review was useful while the two from the control group did not. There also was not any significant difference how quickly the groups completed the second challenge. Potentially with more participants we would find some patterns but with our small sample size it is hard to find any difference between them.

4.5 Engagement Question

The final likert-scale question they were asked dealt with how engaging they thought the review activity was. Fig. 4.7 shows the full question.

This question yielded the most significant results and seems to confirm our hypothesis that the test group would find their review time more engaging than the control group. We feel like, should we get to do this experiment again with a larger sample size, this could be statistically significant. Fig. 4.8 shows the responses to this question.

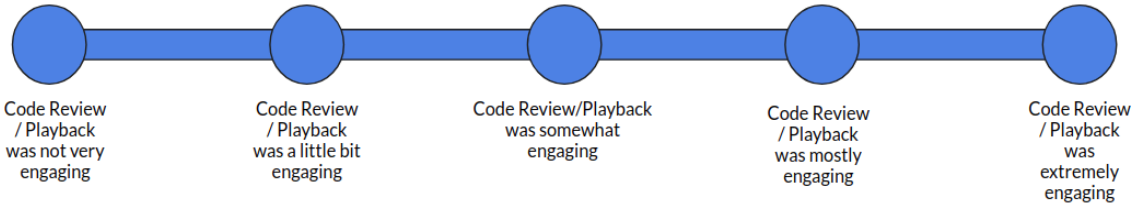

Fig. 4.7: Engagement Survey Question

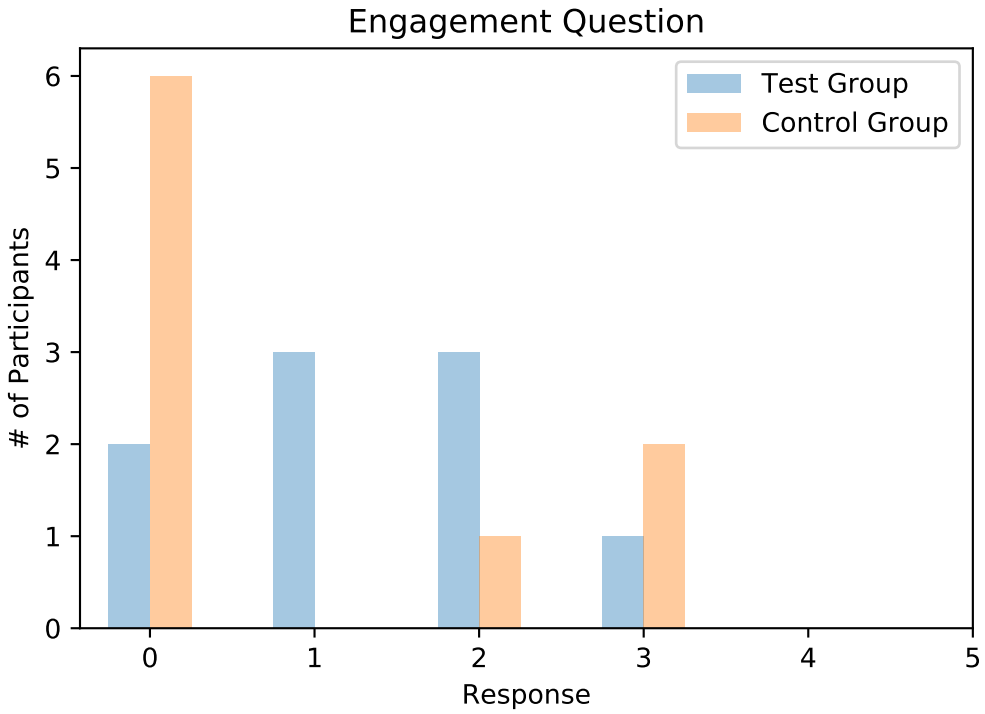

Fig. 4.8: Engagement Survey Response Distribution

The majority of the control group thought the the code review was not very engaging, while the majority of the test group thought it was at least a little bit engaging or somewhat engaging. Naturally, we looked into the two participants from the test group who didn't find the playback engaging at all. When coding the free response answers, which we will explore more in the next section, we found that one of these student's answers identified that they did so poorly on the first challenge that there was ultimately no value to be found in the review. Unfortunately, the other participant didn't write anything for their free response question so we cannot know how they felt about it. So we turned to the video recording of this participant's first challenge and discovered that they too did so poorly on the first challenge that the playback, in my opinion, was not very engaging. By the end of the first challenge this participant only had passed less than 10% of the unit tests. This implies that some level of expertise, relative to the task, is required for the playback to be useful. This seems to corroborate existing research that shows that imagery is a more effective tool when used by experts [4]. That being said, it would still be interesting to try this experiment with introductory programmers with challenges that have been scaled down to an appropriate difficulty.

4.6 Coding Free Response Answers

When authoring the free response question we ultimately settled on the simple prompt: "Tell us your thoughts about the playback/code review". One of the risks these kinds of survey questions pose is that the question prompt can potentially introduce bias into the responses. We felt that this prompt allowed participants to freely express their thoughts and experiences without being guided into a specific genre of response.

Before doing any analysis on the responses they needed to be coded. Coding is a process that is used to turn qualitative data into quantitative data so that you can analyze it. To begin the coding process we developed a code book. The code book is a predefined list of categories that the survey responses might fall into. The code book we developed is outlined in Table 4.1.

Once we had developed the code book we got a few people together. Individually, we

categorized each response into one or more of the categories mentioned in the code book. Afterwards we compared our responses, discussed any discrepancies, and finalized the coded responses. Throughout this process we worked very closely with Dr. Hillary Swanson from the College of Education at USU to ensure that we were doing this correctly. She also helped us develop a similar technique that we used to code the participants' Phanon recordings into categories. Those results will be explored in the next chapter.

While coding the responses its important to note that each response was explicitly labelled with their respective codes. The inclusion or exclusion of a code doesn't imply the opposite. For example: if a response was not coded as "generally engaging" you cannot assume that the participant thought the review time was not engaging, only that there was not specific language in their response that indicated that they thought it was engaging.

Table 4.1: Code Book for Coding Survey Responses

	Code	Code Name	Description
	I	Not Enough Progress	Response mentions that participant did poorly so the review was not helpful
See/ Visualize	II.A	Errors	Response mentions that they saw or were able to visualize errors that they made
	II.B	Process	Response mentions that they were able to see or visualize their process
	III	Desire to Improve	Response mentions that they tried to improve or that review helped them to improve in some way
	IV	Nothing New Learned	Response mentions that the review didn't show them anything they didn't already know
	V	Reinforced Knowledge	Response mentions that the review reminded them of or reinforced existing knowledge
	VI	Typing Errors	Response mentions that they made typing errors
	VII	Slow Down	Response mentions that they that they needed to slow down and think
General Attitude	VIII.A	Positive	Response has a generally positive attitude about the review
	VIII.B	Negative	Response has a generally negative attitude about the review
	VIII.C	Engaging	Response generally describes review as engaging
	VIII.D	Not Engaging	Response generally describes review as not engaging
	IX	Wanted Direction	Response indicated that they wanted more guidance / direction on what to do during review
	X	Wanted Interaction	Response mentions that they wanted to be able to interact with review more
	XI	Found Errors	Response mentions that they found errors

4.7 Analyzing Coded Free Response Answers

Fig. 4.9 shows a selection of coded responses by group.

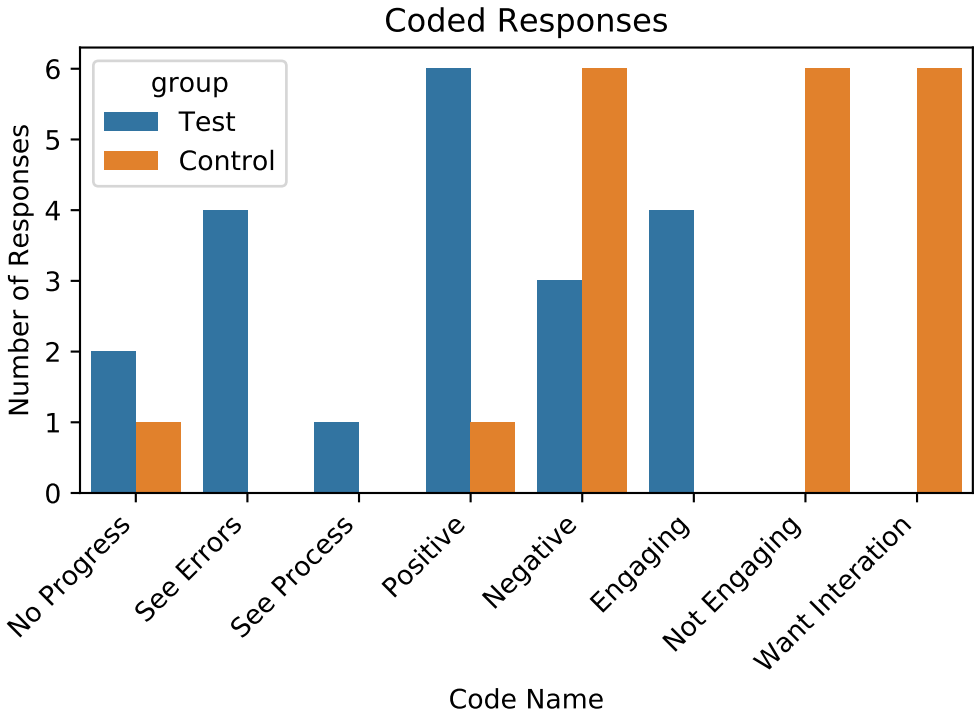

Fig. 4.9: Coded Responses

It's worth noting at this point that one participant in the test group and two participants in the control group did not write anything for the free response question. In Fig. 4.9 we can see that there is a clear division between the responses of the test and control groups.

Two responses from the test group and one from the control group were labelled with the "no progress" label, indicating that they did not make enough progress to get any use out the review. Each of these responses were also labelled, unsurprisingly, as being generally negative.

With only one exception, every response from the participants in the control group had a negative tone. We explored in detail the one participant in the control group that had a generally positive response about the review time. This is what they said:

> "At first I thought it was pointless because I was not able to type any of my thoughts. After I got over my initial frustration about not being able to type I then looked through my code and found a few mistakes that I made sure I did not make in the next program."

The coders assigned the "not engaging" code to this response but this participant was one of the few that thought the review time was useful and engaging based on the other survey questions. So there appears to be some inconsistent results here. Nothing remarkable appeared while reviewing this participant's recordings either.

Coming back to the results, again, with one exception, every participant in the control group expressed that the review generally wasn't engaging. The only participant's response that didn't get coded into that category got coded with the "no progress" code. Not a single participant from the test group indicated that they did not find the playback engaging. On the contrary, four participants from the test group had specific language that indicated that they did find the playback engaging.

Looking at the "want interaction" code, every participant in the control group, except the same one from the previous paragraph, explicitly mentioned they wanted to interact in some way with the review. You could summarize every response from the control group with the response of one participant:

> "Without commenting, changing things, and running, I feel like reviewing it was pointless."

Every single response from the control group, with the exception of the one who got the "no progress" code, specifically mentioned one or more of those three things: commenting, changing, running. This is a strong result: the affect that student engagement has on learning effectiveness has been widely explored, and it is generally accepted that there is positive correlation between engagement and student success [10]. This is certainly something that is worth exploring further.

The "wanted direction" category and "found errors" category are unique to the control group as well. Two of the control group participants specifically mentioned that they wanted

some sort of guidance during the review. Its not surprising that no one in the test group expressed the same desire. The playback acted as a step-by-step guide during the review time for the test group.

Turning our attention to the test group we found that the responses were more varied than the control group's responses, though there were some common themes. Four of the nine responses specifically mentioned that the playback helped them see or visualize errors or process. This highlights one of the distinct advantages that real-time imagery has over static imagery, as static imagery can provide little or no insight into how something happens.

The majority of the test group's responses were labelled as generally positive. The remaining three were labelled with generally negative. Two of these participants were labelled as "no progress" and the third felt the playback was useless. Watching the recordings of this participant didn't provide any insight as to why this participant felt that way. They performed about as well as the rest of the participants did on the first challenge.

Only one participant pointed out that they made a lot of typing errors – something that code review could never provide. It's surprising to me that more participants didn't mention this as this is first thing I notice when watching any of my recordings. While watching each of the recording we did, in fact, notice that many of the participants make frequent and obvious typing mistakes.

4.8 Inconsistencies in the Responses

We mentioned in a previous section that there was some inconsistency with the language of the free response and the results of the likert-scale question about how useful the review time was. One participant from the test group responded with the following:

> "I liked being able to see my thought process as I worked through the problem. When I approached the Elevator problem, I tried to improve things I saw myself do that I could have done better."

This participant only responded with a one when asked how useful the playback was, which seems to contradict their comments. This could represent a potential problem in

the language of the likert-scale question. Perhaps it would have been better to have two negative responses, one neutral response, then two positive responses instead of one negative then four positive.

4.9 Summary of Survey Results

Here I will summarize some of the more notable results. First, the results are consistent with our hypothesis that the test group would find the review time more engaging. We saw that the control group almost universally, in both the likert-scale question and in the free response section identified that they wanted more interaction during review time. Meanwhile, the test group thought the playback was generally engaging. We also discussed the relative difficulty of two challenges and concluded that the challenges were roughly equal in difficulty. For future experiments we would like to explore the following things:

1. Explore what makes the playback more engaging than the code review to help us potentially find ways to make the playback even more engaging.

2. Explore why participants generally thought that playback wasn't useful.

3. Try to improve survey questions to get more accurate or better detailed results.

CHAPTER 5

ANALYZING PLAYBACK

5.1 Overview

Phanon captured recordings for both of each participant's challenge attempts, regardless of which group they were in. This was done so that we could go back and watch each recording with the hope of learning something about each participant's software development process. There are many different ways to define "process" and that made it somewhat difficult to approach these recordings at first. We reached out to Dr. Hillary Swanson from the College of Education at USU. She works a lot with middle-school students and had done some research on modeling thought process [11] using qualitative methods. We met with her and developed a technique for sequencing each of the recordings. After watching and sequencing all 38 recordings some patterns emerged. Some of these patterns seem to be affected by whether or not a participant was in the test or control group. These patterns can be potential measurements of how well a participant performed, especially when looked at on an individual level.

5.2 Sequencing the Recordings

You could use the word "summarizing" as a synonym for "sequencing" in this context, as the result of sequencing is a high-level summary of each recording. Each sequence is a flat list of "moves". This terminology comes from Dr. Hillary Swanson's paper on characterizing student theory building referenced earlier. A move is a singular, sometimes continuous, action performed by the participant. Table 5.1 shows each move with its description.

We watched each recording at normal speed at first, then once we were more comfortable with the process we watched them up to 32x speed. Sequencing each recording turned out to be more difficult than we anticipated. If we made a mistake we had to start the

Code	Name	Description
P	Paste	User pastes code that adds substantially to their program
W	Write	User writes code that adds new code to their program
T	Test	User executes their program
M	Modify	User modifies existing code
R	Remove	User completely removes some code
O	Other	User was doing something other than typing for a significant amount of time

Table 5.1: Code Book for Sequencing Phanon Recordings

recording from the beginning and start the sequencing again because it was too difficult to rewind and maintain context. Additionally, each category is entirely subjective. What actually comprises the difference between writing new code and modifying existing code? We ended up watching each of the recordings twice because it took the first time around to get a feel for it. Fortunately, once we had established a good system we could generally get through each recording quickly and accurately.

Despite the challenges, it didn't take very long for some interesting patterns to emerge. Some of these patterns stood out from the others, these patterns are as follows:

- Test-Modify-Test (T-M-T)
- Test-Other-Modify-Test (T-O-M-T)
- Write-Test-Write (W-T-W)

Table 5.2 show a selection of the fully sequenced playbacks and we can see some of these patterns repeat over and over again in the sequences.

W-P-W-M-W-T-O-M-T-M-T-M-T-M-T-M-T-O-M-T
W-T-O-M-T-M-T-M-T-M-T-M-T-M-T-M-T-M-T-M-T-M-T-M-T-M-T-M-T
W-T-W-T-W-T-W-T-W-T-W-T-W-M-T-M-T

Table 5.2: Sequence Examples

In the following sections we will look at these patterns in detail. Because not very many participants completed the first challenge we will be focusing on the recordings for elevator challenge. This will let us compare side by side with the participants' completion times.

5.3 Test-Modify-Test

This pattern is by far the most common pattern, occurring 120 times during the microwave challenge and 82 times in the elevator challenge. This pattern shows a participant testing their code then immediately making a quick modification. When done multiple times in quick succession, we generally observed that a participant was trying what I like to call the spaghetti method of code writing. An old trick to test whether or not noodles are done cooking is to take a random noodle out of the boiling water and flick it against the wall. If it sticks, then the noodles are done, if it bounces off then they need to cook longer. This development strategy involves trying something random just to see if it "sticks" and when it doesn't some other seemingly random change is made and tested. Clearly this is not an advantageous development strategy and we generally observed that those who had a high occurrence of this pattern took more time to complete the second challenge. Fig. 5.2 shows that there could be a positive correlation with duration.

You might think that because each occurrence of the pattern takes time it is obvious that if they have more occurrences then it will take them longer to complete the challenge. This might be true for this pattern but not necessarily for all of the patterns. An occurrence of a pattern will have a negative impact on time spent if the result of that pattern didn't move the participant any closer to the end goal. An example of this is the programmer that employs the spaghetti method mentioned earlier. However, if one of these patterns moves you closer to the end goal then repeating that pattern might be able to get you to the goal faster. You could say that each occurrence of a pattern has an associated "cost" that is positive or negative.

Looking at that graph it appears that the cost for each occurrence may be higher for the control group than the test group. This idea of cost will be explored more in a later

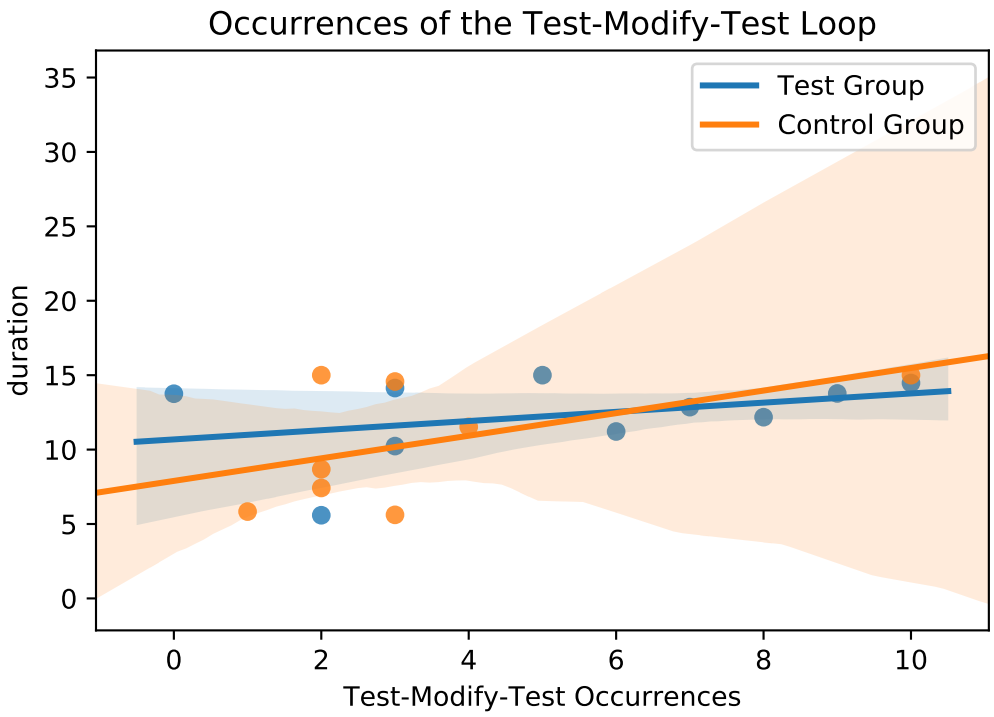

Fig. 5.1: Test Modify Test

section.

5.4 Test-Other-Modify-Test

This pattern is less common than the test-modify-test pattern, appearing 32 times in the microwave challenge and 21 times in the elevator challenge code. This pattern is similar to the last pattern but different in that after running their code they spent a substantial amount of time doing something other than typing. We assume that they are reading and trying to understand either error or test output. This pattern could indicate a couple of things depending on surrounding context. On common place to see this pattern is after a string of test-modify-test patterns, for example: T-M-T-M-T-O-M-T. This likely shows someone doing the spaghetti method then realizing it would probably be a good idea to pay closer attention to the test output. When this pattern is repeated it is generally at the tail end of a challenge right before the participant runs out of time.

This pattern also seems to be positively correlated with the amount of time spent on the second challenge. Fig. 5.2 shows that correlation.

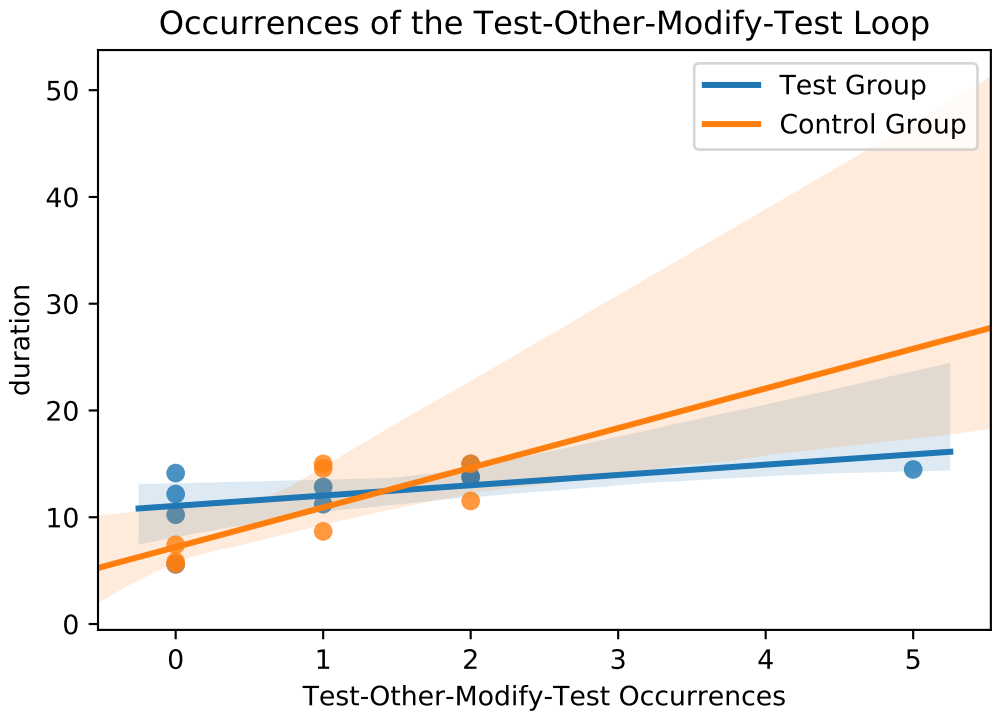

Fig. 5.2: Test Other Modify Test

Just like in the previous section, the cost of each occurrence appears to be higher for the control group than it is for the test group.

5.5 Write-Test-Write

In terms of cost, this is the only pattern of our three that appears to have a negative cost. This pattern is not very common; it only appears 9 times in the microwave challenge, and 8 of those were in one recording. It appears 17 times in the elevator challenge. This pattern indicates that a participant spent some time writing code, confirmed that it worked, then continued writing. This seems to be a generally good approach. The recordings where this pattern is repeated are genuinely satisfying to watch, and unsurprisingly, some of the

fastest challenge attempts had this pattern in them repeatedly. Fig. 5.3 shows there could be a small negative correlation with duration.

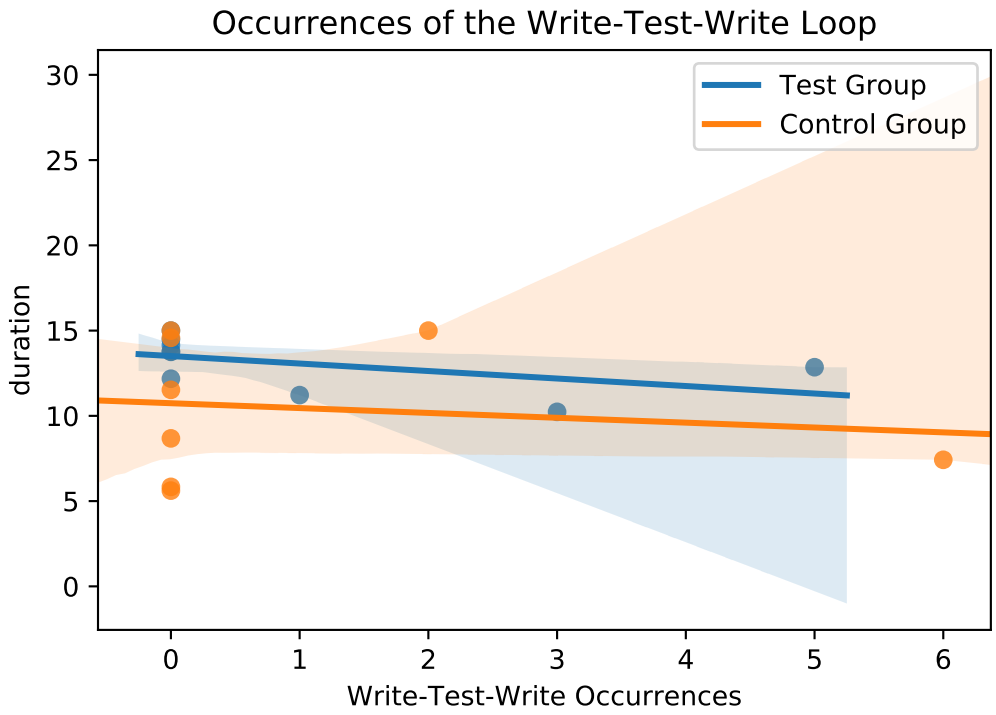

Fig. 5.3: Write-Test-Write Occurrences

This pattern does only seem to have a true negative correlation to duration if the pattern occurs in succession. Compare the 2 sequences in Table 5.3.

Sequence	Duration
W-T-W-T-M-T-W-T-W-T-W-O-W-T-W-T-M-T-M-W-T-W-T-M-T-O-M-T-M-T-M-T-M-T-M-T-M-T	14
W-T-W-T-W-T-W-T-W-T-W-T-W-T-W-M-T	8

Table 5.3: Write-Test-Write Comparison

5.6 Other Insightful Patterns

A couple of other interesting patterns showed up, though they were much less common.

One of these patterns was Test-Other-Test. This pattern showed up several times, almost exclusively in those participants who, for the free response question, mentioned they did so poorly that the review time was useless. This pattern might indicate that a participant is stuck to the point that they don't know what to try next. So they run their program, read the test or error output and just run their program again without changing anything as if it will suddenly start working. I suspect that if we were to sequence some recordings of novice programmers then we might see this pattern more often.

The beginning of each sequence has to start in one of two ways, with a "write" or with a "paste". Beginning with a "write" was by far the most common way to start the program, with 33 of the 39 sequences starting that way. We hypothesised that after the first challenge that we see a lot of participants change their approach, specifically in the test group, but this just wasn't the case. Only two participants changed their general approach. The first changed from writing all of the code out to pasting the example code we gave them, and the other change from pasting the example to writing it all out. We talk a little more about this participant in the last chapter when discussing the enjoyment likert-scale question. Other than those two, everybody seemingly took the same approach for both of their challenges.

5.7 Cost of Pattern Occurrences

In the previous sections we observed that the cost of each pattern of occurrence seems to be different between the control group and the test group. I recognize that without being able to compare this data with data from the microwave challenge that we cannot make any assumptions about whether or not the playback had any affect on this. However, it does provide us with a unique metric that we could potentially use to measure the effectiveness of the playback during future experimentation. We also might be able to have Phanon automatically compute this cost for us which will be important as we scale up.

5.8 Summary and Future Work

Coding each of the recordings revealed some interesting and insightful patterns. We found that each of these patterns has an associated cost either positive or negative. These

results could could help measure the impact that imagery has during future experiments. One thing that these sequences can't tell us is the distribution of how much time was spent on each move. The "test" move is discrete but the "write" move and "modify" move are generally continuous. Assigning a cost to each move might help us understand in more detail what each participant spent the most time on without having to watch all of the recordings. In the future we would like do the following things:

1. Improve the software to do the sequencing of each of the recording automatically, this will be necessary to accommodate a larger sample size.

2. These sequences could be useful as an early warning system to help identify when a student is struggling. We would like to explore this as an educational tool.

CHAPTER 6

CONCLUSION

We began this research with three hypotheses:

1. The test group would improve on the second challenge more than the control group.
2. The test group would find their review activity more engaging than the control group.
3. The test group would approach the second challenge differently while the control group would approach both challenges more or less the same.

While analyzing the keystroke events, the results seemed to contradict the first the hypothesis because we saw that the control group finished faster than the test group did. These results are not conclusive but did shed some light on some of the shortcomings of our experiment design that we would like to correct during future attempts. Specifically, we would like to make sure that everybody finishes each challenge so we have a stronger baseline to compare to.

The survey results provided us strong evidence that supports our second hypothesis. Nearly everybody in the control group specifically mentioned wanting to be able to interact with the review more. Meanwhile, nearly everybody in the test group praised the playback for how engaging it was. In the future we would like to explore this more. We also would want to gather some more data to help us understand why, despite being engaged with the playback, most found it to be not useful in the likert-scale question.

While analyzing the recordings we were able to identify several insightful patterns that were repeated throughout each recording. While, we didn't find any evidence that supports our third hypothesis, it might be possible that playback had an affect on the cost of each of the patterns. For future experiments we would like to explore this more and potentially use that as a metric to measure how much each participant improved between challenges.

Ultimately, we cannot confirm or reject any of these hypotheses with the data we have. However, they did serve as a useful guide for this exploratory study and we did discover that there could be potential benefits of using imagery that might be uncovered with more data.

If imagery can be shown to improve outcomes then it could be integrated into computer science curriculum. New types of engaging activities, such as playback based quiz questions or peer review, could be used to assess student success. These tools could be downloaded and used offline or on mobile devices so that students can practice imagery even when they are not sitting at computer with a strong internet connection. Imagery can be used by teachers to assess and improve their own program quality and they can practice meta-imagery in order to better be able to teach students how to improve their own imagery processes. Playback could also be useful for teachers to assess and potentially even grade the process of each student. At a professional level it can be used to improve program quality in the same way we suggest it can for students. It could also be useful for companies to assess the competency of potential hires or be used part of periodic performance reviews.

6.1 Future Work

One of the primary goals of this thesis was to discover areas of future work. We have already discussed some of these discoveries in the previous chapters but we will summarize those findings here.

The keystroke data that we gathered could help us automatically label different students with one of a few categories. For example, a “tinkerer” might be someone that doesn’t encounter many errors but executes their program frequently and makes small changes in between program executions, meanwhile, a ”hacker” could be someone that writes a lot of code between code executions. These categories might be useful in predicting student success or could help teachers identify struggling students.

While analyzing the survey results we saw some interesting things that deserve more attention. We would like to explore the following things:

1. Explore what makes the playback more engaging than the code review to help us potentially find ways to make the playback even more engaging.

2. Explore why participants generally thought that playback wasn't useful.

3. Try to improve survey questions to get more accurate or better detailed results.

During this research we developed a technique for sequencing programming recordings that has untapped potential. As we continue to develop this process we would like to do the following:

1. Improve the software to do the sequencing of each of the recording automatically, this will be necessary to accommodate a larger sample size.

2. These sequences could be useful as an early warning system to help identify when a student is struggling. We would like to explore this as an educational tool.

This thesis represents the beginning of research into this topic, not the end. Throughout our research we have identified many new research questions, developed new techniques for gathering data, and written state-of-the-art software that has untapped potential. All of these things give us many possible avenues to approach future experiments that could show the effect that using imagery can have in our discipline.

REFERENCES

[1] B. Skudder and A. Luxton-Reilly, "Worked examples in computer science," in *Proc. ACE '14 Proceedings of the Sixteenth Australasian Computing Education Conferences*, Jan. 2014, pp. 59–64.

[2] B. D. Hale, *Imagery in sports and physical performance.* Amityville, NY: Baywood, 1994, ch. 5, pp. 75–96.

[3] M. Roth, J. Decety, M. Raybaudi, R. Massarelli, C. Delon-Martin, C. Segebarth, S. Morand, A. Gemignani, M. Decorps, and M. Jeannerod, "Possible involvement of primary motor cortex in mentally simulated movement: a functional magnetic resonance imaging study," *Neuroreport*, vol. 7, pp. 1280–1284, 1996.

[4] M. Arvinen-Barrow, D. A. Weigand, S. Thomas, B. Hemmings, and M. Walley, "Elites and novices athletes' imagery use in open and closed sports," *Journal of Applied Sport Psychology*, vol. 19, pp. 93–104, 2007.

[5] D. Smith, C. Wright, A. Allsopp, and H. Westhead, "It's all in the mind: PETTLEP-based imagery and sports performance," *Journal of Applied Sport Psychology*, vol. 19, pp. 80–92, 2007.

[6] C. R. Hall, W. M. Rogers, and K. A. Barr, "The use of imagery by athletes in selected sports," *The Sports Psychologist*, vol. 4, pp. 1–10, 1990.

[7] J. H. Flavell, "Metacognition and cognitive monitoring," *American Psychologist*, vol. 34, pp. 906–911, 1979.

[8] T. E. MacIntyre, E. R. Igou, M. J. Campbell, A. P. Moran, and J. Mathews, "Metacognition and action: a new pathway to understanding social and cognitive aspects of expertise in sport," *Frontiers of Psychology*, vol. 5, 2014.

[9] V. A. Aleven and K. R.Koedinger, "An effective metacognitive strategy: learning by doing and explaining with a computer-based cognitive tutor," *Cognitive Science*, vol. 26, pp. 147–179, 2002.

[10] R. M. Carini, G. D. Kuh, and K. Stephen P, "Student engagement and student learning: Testing the linkages," *Research in Higher Education*, vol. 47, pp. 1–32, 2006.

[11] H. Swanson, K. Martin, B. Sherin, and U. Wilensky, "Characterizing student theory building in the context of block-based agent-based modeling microworlds," 2020.

APPENDICES

APPENDIX A

PROGRAMMING CHALLENGES

A.1 Microwave Challenge Instructions

```
### Microwave Oven
In this exercise you will write a class called `MicrowaveOven`. When you click the run
↪  button a hidden program will attempt to use the `MicrowaveOven` class you have created
↪  and will verify that you have implemented everything correctly. You can run your
↪  program as often as you like and can use the test results to identify issues in your
↪  program. It is very important that everything is named correctly or the tests wont
↪  pass.

A `MicrowaveOven` has the following methods:
* `open_door()`
* `close_door()`
* `is_door_open(): boolean`
* `insert_food(): boolean`
* `remove_food(): boolean`
* `has_food(): boolean`
* `cook_food(time: integer): boolean`
* `how_long_has_current_food_cooked(): integer` return `-1` if there is no food in the oven

Implement each of these methods and use the test output as necessary to help guide you on
↪  what each method should do.

Use the following for help related to class syntax in python:

```
class ClassName:
    # class variables
    variable_name = value
```
```

```
    # constructor, dont forget the self!
    def __init__(self):
        pass

    # class methods, dont forget the self!
    def method_name(self, arg1, arg2):
        pass
```

A.2 Microwave Challenge Unit Tests

```
continue_tests = True
def pass_all_tests(password):
    if password == "asdfasdf":
        global continue_tests
        continue_tests = False
        phanon_test_pass("Tests bypassed")
### BEGIN_STUDENT
### END_STUDENT

student_output = phanon_get_stdout()
student_program_code = phanon_get_program()

"""
Tests
"""
if continue_tests:
    try:
        oven = MicrowaveOven()
        phanon_test_pass("MicrowaveOven was successfully constructed")
    except NameError:
        print("Couldn't find a class called MicrowaveOven")
        phanon_test_fail("Could not successfully construct a MicrowaveOven")
        continue_tests = False
```

```
    except Exception as inst:
        print(inst)
        phanon_test_fail("Could not successfully construct a MicrowaveOven")
        continue_tests = False


if continue_tests:
    method_names = ['open_door', 'is_door_open', 'close_door', 'insert_food', 'has_food',
    ↪  'remove_food', 'how_long_has_current_food_cooked']
    for method_name in method_names:
        try:
            getattr(oven, method_name)()
            phanon_test_pass("{} exists and executes without error".format(method_name))
        except AttributeError as inst:
            print("Could not find a method on MicrowaveOven called {}, or you forgot to
            ↪  initialize a class member variable".format(method_name))
            print(inst)
            phanon_test_fail("MicrowaveOven should have a method called
            ↪  {}".format(method_name))
            continue_tests = False
        except Exception as inst:
            print(inst)

            phanon_test_fail("An error occured when calling {}".format(method_name))
            continue_tests = False


    try:
        oven.cook_food(10)
        phanon_test_pass("cook_food exists and executes without error")
    except TypeError:
        print("Cook food should accept one argument")
        phanon_test_fail("The cook_food method should accept an argument")
        continue_tests = False
    except AttributeError as inst:
```

```
        print("Could not find a method on MicrowaveOven called cook_food, or you forgot to
        ↪  initialize a class member variable")
        print(inst)
        phanon_test_fail("MicrowaveOven should have a method called cook_food")
        continue_tests = False
    except Exception as inst:
        print(inst)
        phanon_test_fail("An error occured when calling cook_food")
        continue_tests = False



if continue_tests:
    new_oven = MicrowaveOven()
    try:
        continue_tests = new_oven.is_door_open() == False
        phanon_test(continue_tests, 'Door was closed when expected', 'Door was open when it
        ↪  should have been closed')
        if continue_tests:
            new_oven.open_door()
            continue_tests = new_oven.is_door_open() == True
            phanon_test(continue_tests, 'Door was open when expected', 'Door was closed
            ↪  when it should have been open')
        if continue_tests:
            new_oven.close_door()
            continue_tests = new_oven.is_door_open() == False
            phanon_test(continue_tests, 'Door was closed when expected', 'Door was open
            ↪  when it should have been closed')
    except Exception as inst:
        print(inst)
        continue_tests = False
        phanon_test_fail("An error occured when testing door functionality")

if continue_tests:
    new_oven = MicrowaveOven()
    try:
```

```
continue_tests = new_oven.has_food() == False
phanon_test(continue_tests, 'has_food returns the correct value when no food should
↪  be in oven', 'has_food returns incorrect value when no food should be in oven')
if continue_tests:
    continue_tests = new_oven.insert_food() == False
    phanon_test(continue_tests, 'insert_food correctly returns False when door is
    ↪  closed', 'insert_food incorrectly returns true when door is closed')

if continue_tests:
    new_oven.open_door()
    continue_tests = new_oven.insert_food() == True
    phanon_test(continue_tests, 'insert_food correctly returns true when door is
    ↪  open', 'insert_food incorrectly returns false when door is open')

if continue_tests:
    continue_tests = new_oven.has_food() == True
    phanon_test(continue_tests, 'has_food returns the correct value when food
    ↪  should be in oven', 'has_food returns incorrect value when food should be
    ↪  in oven')

if continue_tests:
    continue_tests = new_oven.insert_food() == False
    phanon_test(continue_tests, 'insert_food correctly returns false when food is
    ↪  already in oven', 'insert_food incorrectly returns true when food is
    ↪  already in the oven')

if continue_tests:
    new_oven.close_door()
    continue_tests = new_oven.remove_food() == False
    phanon_test(continue_tests, 'remove_food correctly returns false when door is
    ↪  closed', 'remove_food incorrectly returns true when door is closed')

if continue_tests:
    new_oven.open_door()
    continue_tests = new_oven.remove_food() == True
```

```
    phanon_test(continue_tests, 'remove_food correctly returns True when door is
    ↪  open', 'remove_food incorrectly returns False when door is open')

if continue_tests:
    continue_tests = new_oven.remove_food() == False
    phanon_test(continue_tests, 'remove_food correct returns false when no food is
    ↪  in the oven', 'remove_food incorrectly returns true when no food is in the
    ↪  oven')

if continue_tests:
    continue_tests = new_oven.has_food() == False
    phanon_test(continue_tests, 'has_food returns the correct value when no food
    ↪  should be in the oven', 'has_food returns the incorrect value when no food
    ↪  should be in the oven')

if continue_tests:
    continue_tests = new_oven.how_long_has_current_food_cooked() == -1
    phanon_test(continue_tests, 'how_long_has_current_food_cooked correctly returns
    ↪  -1 when there is no food in the oven', 'how_long_has_current_food_cooked
    ↪  returned something other than -1 when there is no food in the oven')

if continue_tests:
    new_oven.close_door()
    new_oven.cook_food(10)
    new_oven.open_door()
    new_oven.insert_food()
    continue_tests = new_oven.how_long_has_current_food_cooked() == 0
    phanon_test(continue_tests, 'cook_food did not increase cook time when there
    ↪  was no food in the oven', 'cook_food incorrectly increased cook time when
    ↪  there was not food in the oven')

if continue_tests:
    new_oven.insert_food()
    new_oven.close_door()
    new_oven.cook_food(60)
```

```
            continue_tests = new_oven.how_long_has_current_food_cooked() == 60
            phanon_test(continue_tests, 'how_long_has_current_food_cooked returned the
            ↪  correct value after cooking', 'how_long_has_current_food_cooked returns the
            ↪  incorrect value after cooking')

        if continue_tests:
            new_oven.cook_food(900)
            continue_tests = new_oven.how_long_has_current_food_cooked() == 960
            phanon_test(continue_tests, 'how_long_has_current_food_cooked returned the
            ↪  correct value after cooking', 'how_long_has_current_food_cooked returns the
            ↪  incorrect value after cooking')

        if continue_tests:
            new_oven.open_door()
            new_oven.cook_food(10)
            continue_tests = new_oven.how_long_has_current_food_cooked() == 960
            phanon_test(continue_tests, 'cook_food correctly does not cook food when door
            ↪  is open', 'cook_food cooked the food when the door was open!')

        if continue_tests:
            new_oven.remove_food()
            new_oven.insert_food()
            continue_tests = new_oven.how_long_has_current_food_cooked() == 0
            phanon_test(continue_tests, 'removing food correctly resets the amount of time
            ↪  the current food item has cooked', 'removing food item does not correctly
            ↪  reset the amount of time a current food item has cooked')
    except Exception as inst:
        print(inst)
        phanon_test_fail("An error occured when testing inserting, removing, and cooking
        ↪  functionality")
```

A.3 Elevator Challenge Instructions

```
### Elevator
```

```
In this exercise you will write a class called `Elevator`. When you click the run button a
↪  hidden program will attempt to use the `Elevator` class you have created and will
↪  verify that you have implemented everything correctly. You can run your program as
↪  often as you like and can use the test results to identify issues in your program. It
↪  is very important that everything is named correctly or the tests wont pass.

An `Elevator` has the following methods:
* `__init__(max_occupancy: integer)`
* `open_door()`
* `close_door()`
* `is_door_open(): boolean`
* `insert_person(): boolean`
* `remove_person(): boolean`
* `get_number_of_occupants(): integer`
* `go_to_floor(floor_number: integer)`
* `get_current_floor(): integer`

Implement each of these methods and use the test output as necessary to help guide you on
↪  what each method should do.

Use the following for help related to class syntax in python:

```
class ClassName:
    # class variables
    variable_name = value
    # constructor, dont forget the self!
    def __init__(self):
        pass

    # class methods, dont forget the self!
    def method_name(self, arg1, arg2):
        pass
 ```
```

A.4 Elevator Challenge Unit Tests

```
continue_tests = True

def pass_all_tests(password):
    if password == "qwerqwer":
        global continue_tests
        continue_tests = False
        phanon_test_pass("Tests bypassed")
### BEGIN_STUDENT
### END_STUDENT

student_output = phanon_get_stdout()
student_program_code = phanon_get_program()

"""
Tests
"""
if continue_tests:
    try:
        elevator = Elevator(3)
        phanon_test_pass("Elevator was successfully constructed")
    except NameError:
        print("Couldn't find a class called Elevator")
        phanon_test_fail("Could not successfully construct a Elevator")
        continue_tests = False
    except Exception as inst:
        print(inst)
        phanon_test_fail("Could not successfully construct a Elevator")
        continue_tests = False

if continue_tests:
    method_names = ['open_door', 'is_door_open', 'close_door', 'insert_person',
    ↪  'get_number_of_occupants', 'remove_person', 'get_current_floor']
    for method_name in method_names:
        try:
```

```
            getattr(elevator, method_name)()
            phanon_test_pass("{} exists and executes without error".format(method_name))
        except AttributeError as inst:
            print("Could not find a method on Elevator called {}, or you forgot to
            ↪  initialize a class member variable".format(method_name))
            print(inst)
            phanon_test_fail("Elevator should have a method called {}".format(method_name))
            continue_tests = False
        except Exception as inst:
            print(inst)
            phanon_test_fail("An error occured when calling {}".format(method_name))
            continue_tests = False

    try:
        elevator.go_to_floor(10)
        phanon_test_pass("go_to_floor exists and executes without error")
    except TypeError:
        print("go_to_floor should accept one argument")
        phanon_test_fail("The go_to_floor method should accept an argument")
        continue_tests = False
    except AttributeError as inst:
        print("Could not find a method on Elevator called go_to_floor, or you forgot to
        ↪  initialize a class member variable")
        print(inst)
        phanon_test_fail("Elevator should have a method called go_to_floor")
        continue_tests = False
    except Exception as inst:
        print(inst)
        phanon_test_fail("An error occured when calling go_to_floor")
        continue_tests = False


if continue_tests:
    new_elevator = Elevator(3)
    try:
```

```
            continue_tests = new_elevator.is_door_open() == False
            phanon_test(continue_tests, 'Door was closed when expected', 'Door was open when it
            ↪  should have been closed')
            if continue_tests:
                new_elevator.open_door()
                continue_tests = new_elevator.is_door_open() == True
                phanon_test(continue_tests, 'Door was open when expected', 'Door was closed
                ↪  when it should have been open')
            if continue_tests:
                new_elevator.close_door()
                continue_tests = new_elevator.is_door_open() == False
                phanon_test(continue_tests, 'Door was closed when expected', 'Door was open
                ↪  when it should have been closed')
        except Exception as inst:
            print(inst)
            continue_tests = False
            phanon_test_fail("An error occured when testing door functionality")

    if continue_tests:
        new_elevator = Elevator(3)
        try:
            continue_tests = new_elevator.get_number_of_occupants() == 0
            phanon_test(continue_tests, 'get_number_of_occupants returns the correct value when
            ↪  no occupants should be in elevator', 'get_number_of_occupants returns incorrect
            ↪  value when no occupants should be in elevator')
            if continue_tests:
                continue_tests = new_elevator.insert_person() == False
                phanon_test(continue_tests, 'insert_person correctly returns False when door is
                ↪  closed', 'insert_person incorrectly returns true when door is closed')

            if continue_tests:
                new_elevator.open_door()
                continue_tests = new_elevator.insert_person() == True
                phanon_test(continue_tests, 'insert_person correctly returns true when door is
                ↪  open', 'insert_person incorrectly returns false when door is open')
```

```
if continue_tests:
    continue_tests = new_elevator.get_number_of_occupants() == 1
    phanon_test(continue_tests, 'get_number_of_occupants returns the correct value
    ↪  when person should be in elevator', 'get_number_of_occupants returns
    ↪  incorrect value when person should be in elevator')

if continue_tests:
    new_elevator.insert_person()
    new_elevator.insert_person()
    continue_tests = new_elevator.get_number_of_occupants() == 3
    phanon_test(continue_tests, 'get_number_of_occupants returns the correct value
    ↪  when 3 people should be in elevator', 'get_number_of_occupants returns
    ↪  incorrect value when 3 people should be in elevator')

if continue_tests:
    continue_tests = new_elevator.insert_person() == False
    phanon_test(continue_tests, 'insert_person correctly returns false when
    ↪  elevator is at maximum capacity', 'insert_person incorrectly returns true
    ↪  when elevator is at maximum capacity')

if continue_tests:
    new_elevator.close_door()
    continue_tests = new_elevator.remove_person() == False
    phanon_test(continue_tests, 'remove_person correctly returns false when door is
    ↪  closed', 'remove_person incorrectly returns true when door is closed')

if continue_tests:
    new_elevator.open_door()
    continue_tests = new_elevator.remove_person() == True
    phanon_test(continue_tests, 'remove_person correctly returns True when door is
    ↪  open', 'remove_person incorrectly returns False when door is open')

if continue_tests:
    new_elevator.remove_person()
```

```
    new_elevator.remove_person()
    continue_tests = new_elevator.remove_person() == False
    phanon_test(continue_tests, 'remove_person correct returns false when no food
    ↪  is in the elevator', 'remove_person incorrectly returns true when no one is
    ↪  in the elevator')

if continue_tests:
    continue_tests = new_elevator.get_number_of_occupants() == 0
    phanon_test(continue_tests, 'get_number_of_occupants returns the correct value
    ↪  when no occupants should be in the elevator', 'get_number_of_occupants
    ↪  returns the incorrect value when no occupants should be in the elevator')

if continue_tests:
    continue_tests = new_elevator.get_current_floor() == 1
    phanon_test(continue_tests, 'get_current_floor correctly returns 1 by default',
    ↪  'get_current_floor returned something other than 1 by default')

if continue_tests:
    new_elevator.close_door()
    new_elevator.go_to_floor(10)
    new_elevator.open_door()
    new_elevator.insert_person()
    continue_tests = new_elevator.get_current_floor() == 1
    phanon_test(continue_tests, 'go_to_floor did not change floors when there was
    ↪  no one in the elevator', 'go_to_floor incorrectly changed floor when there
    ↪  was no one in the elevator')

if continue_tests:
    new_elevator.insert_person()
    new_elevator.close_door()
    new_elevator.go_to_floor(60)
    continue_tests = new_elevator.get_current_floor() == 60
    phanon_test(continue_tests, 'get_current_floor returned the correct value after
    ↪  changing floor', 'get_current_floor returns the incorrect value after
    ↪  changing floor')
```

```
        if continue_tests:
            new_elevator.go_to_floor(25)
            continue_tests = new_elevator.get_current_floor() == 25
            phanon_test(continue_tests, 'get_current_floor returned the correct value after
            ↪  changing floor', 'get_current_floor returns the incorrect value after
            ↪  changing floor')

        if continue_tests:
            new_elevator.open_door()
            new_elevator.go_to_floor(10)
            continue_tests = new_elevator.get_current_floor() == 25
            phanon_test(continue_tests, 'go_to_floor correctly does not change floor when
            ↪  door is open', 'go_to_floor change floors when the door was open!')

    except Exception as inst:
        print(inst)
        phanon_test_fail("An error occured when testing inserting, removing, and changing
        ↪  floor functionality")
```