©Copyright 2024

Matthew Woerner

Identifying and Addressing the Gap Between How Students and Professionals Read Code

Matthew Woerner

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science

University of Washington

2024

Committee:

David Socha

Mark Kochanski

Robert Dimpsey

Program Authorized to Offer Degree:

Computing and Software Systems

University of Washington

Abstract

Identifying and Addressing the Gap Between How Students and Professionals Read Code

Matthew Woerner

Chair of the Supervisory Committee:
David Socha
School of Science, Technology, Engineering & Mathematics

This project investigated and addressed the questions of: a) how do students and professional software developers read novel codebases, and b) how can we help students learn to better read code.

Our Spring 2023 study, seen in Appendix A, used semi-structured interviews and code reading exercises to identify and quantify several differences in the ways students and professional software developers read novel codebases. Students tended to face more difficulty with these reading tasks than the professionals due to an apparent lack of structured code reading process and an over reliance on making unverified assumptions about the code. We focused on three particular anti-patterns. Our interview data also indicated that the lack of a structured code reading process

complicates transitioning into a professional atmosphere post degree, requiring new professional software developers to learn these skills on the job.

Based upon the results, we developed a module to teach students a structured way to read code in novel codebases, and to assess their improvement. The module was integrated into the Fall 2023 quarter of CSS 390 (Software Engineering Studio). Students worked their way through a variety of formative exercises leading up to a final summative assessment where they were evaluated on their performance improvement throughout the module as well as how they compared to a prior group of students given a similar assessment in the Spring quarter. Comparing the number of code reading anti-patterns exhibited by both groups, we found that the students who completed the module were much less likely to trace into files outside of the code path, were more likely to follow all stack traces in a code reading challenge, and were less likely to make uncorrected misinterpretations about a codebase.

TABLE OF CONTENTS

Chapter 1. Introduction 1

Chapter 2. Previous Work 3

Chapter 3. Designing the Tool 8

3.1 Finding A Way To Track Student Improvement 9

3.2 Development of Concepts to Teach & Prototype Lessons 13

3.3 Prototyping a Summative Assessment 13

3.4 Prototyping Formative Assessments 16

3.5 Creating Lesson Plans and Exercises 20

3.6 Usability Testing 29

3.7 Creating the Final Module 35

3.8 Changes Made During Administration 42

Chapter 4. Results & Analysis 43

4.1 Student Exercise Scores 44

4.2 Anti-Pattern Analysis 48

4.3 Student Feedback and Sentiment 50

Chapter 5. Future Work & Conclusion 51

5.1 Future Work of the Module 51

5.2 Conclusion 53

Bibliography 55

Appendix A: CCSC-NW 2023 Paper 57

Chapter 1. INTRODUCTION

This paper reports on the results of a two-part study exploring the differences between how students and professionals read code, and then using those findings to explore how structured lessons could help students grow closer to a professional level of code reading ability. The first part of the study [1] was designed to answer the research question: How do undergraduate students and professional software developers differ in how they come to understand a codebase that is novel to them? That research topic came about as a suggestion from two University of Washington Bothell faculty who were teaching a course, Software Engineering Studio, whose purpose is to provide an "on-campus internship-like experience" [1] for students before moving into the workplace proper. That course included a sequence of badges that students completed, with each badge including a "challenge session" during which faculty met via Zoom with each student to assess the student's badge work, uncover gaps in the student's knowledge, and help fill in those gaps. As the course was administered, the faculty noticed that in these semi-structured badge challenge interviews with students [5] only a handful of the over 100 students were able to successfully follow a call stack to comprehend what source code in a novel codebase did. It was also noted that students tended to require direct instruction and guidance to help them fully understand the function of the codebase. These same dynamics were also noticed during my semi-structured interviews of 10 students outside of the Software Engineering Studio class, from different universities, and with varying levels of experience [1]. These 10 student interviews were compared with 11 semi-structured interviews with professional software developers from a variety of backgrounds to establish hypotheses for why students were having difficulties reading novel codebases.

The previous study identified that there existed several key differences in the ways students and professional software developers went about understanding a novel codebase. Among these differences were several anti-patterns that students exhibited and professionals avoided. These anti-patterns included the following:

1. Making uncorrected misinterpretations of the code
2. Not following the call stack through files
3. Examining irrelevant files/searching through files randomly

This led to the development of the hypothesis that if students were given clear instruction that helped to teach them structured methods to read code like professionals, then they would improve in their ability to read novel codebases and exhibit fewer anti-patterns in their readings. With this established hypothesis, we set out to answer a new research question: Would a structured series of lessons that teach students a structured code reading strategy help students bridge the gap between them and professionals in the field of code reading as measured by the number of anti-patterns exhibited in code reading done by students?

The remainder of this paper is organized into various sections documenting the process of developing the lessons given to students and the results found when the lessons were administered. Chapter 2 briefly discusses the previous work and research that led up to this project, more of which can be seen in Appendix A which contains the first paper on this topic co-authored by myself and two members of my capstone committee. Chapter 3 discusses the journey from the preliminary research described in Chapter 2 and Appendix A which concluded in June of 2023 to the administration of the code reading module in late October 2023. Chapter 4 showcases the results of the module and my takeaways from those results. Chapter 5 concludes

by discussing the future of this project and my overall conclusions after administering the module.

Chapter 2. PREVIOUS WORK

In the spring of 2023, I interviewed a series of 10 students and 11 professionals. These interviews showed noticeable differences in how students and professionals read novel codebases. In these interviews and subsequent code reading exercises, students tended to fall into similar anti-patterns, such as not following the call stack or making uncorrected misinterpretations, which led to incorrect readings of the code [1]. One of the hypotheses formed because of these interviews and exercises is that professionals were able to perform better due to their frequent checking of their own assumptions and their understanding of how to trace through call stacks.

In addition to the conducted interviews, a review of existing literature related to comparing how novice and expert programmers read code suggested that experts are far more efficient in how they read code [6, 7]. This could in part be due to more familiarity with how codebases are typically structured, which could lead to less cognitive overload and a faster and more comprehensive reading of the code. Other eye tracking studies suggest that professionals engage in more complex processing of code than novices [8]. These results hold a strong correlation to my interview findings in which students are spending a longer amount of time reading code and are not getting as good of an understanding as the professionals given the same tasks.

Based on the prior research conducted and the observations made of students and professionals during the exercises and interviews, I came to three interpretations that could explain why students exhibited the anti-patterns observed in the interviews. These interpretations

included INT1: a lack of experience reading large codebases, INT2: a fixation on figuring out what the program does rather than how the program accomplishes its task (lack of critical mindset), and INT3: an overall lack of knowledge of concepts for how to read the code examined such as language syntax or which direction to read each line (right to left vs left to right). As seen in Figure 2.1, each interpretation is backed up with several identified issues that themselves were designed to group together observations that were made in the semi-structured interviews. Following is a discussion of how observations led to issues which were grouped into interpretations, starting at the top of the observations listed in Figure 2.1 and moving downward.

Some of the more common observations were observed in over half of the student semi-structured interviews and related to the issue of students having difficulty navigating to and tracking code into different files (ISS1) and included:

- O1: Students not using namespaces to find source code.
- O2: Students randomly clicking on files to find source code.
- O3: Students not examining subdirectories for source code.

More common observations that did not fit under the umbrella of ISS1 but were still related to codebase navigation included:

- O4: Students not stepping down more than one level deep in a call stack.
- O5: Students not understanding how unit tests are structured in the project directory and meant to be navigated.
- O6: Students not using IDE tools to help themselves navigate.

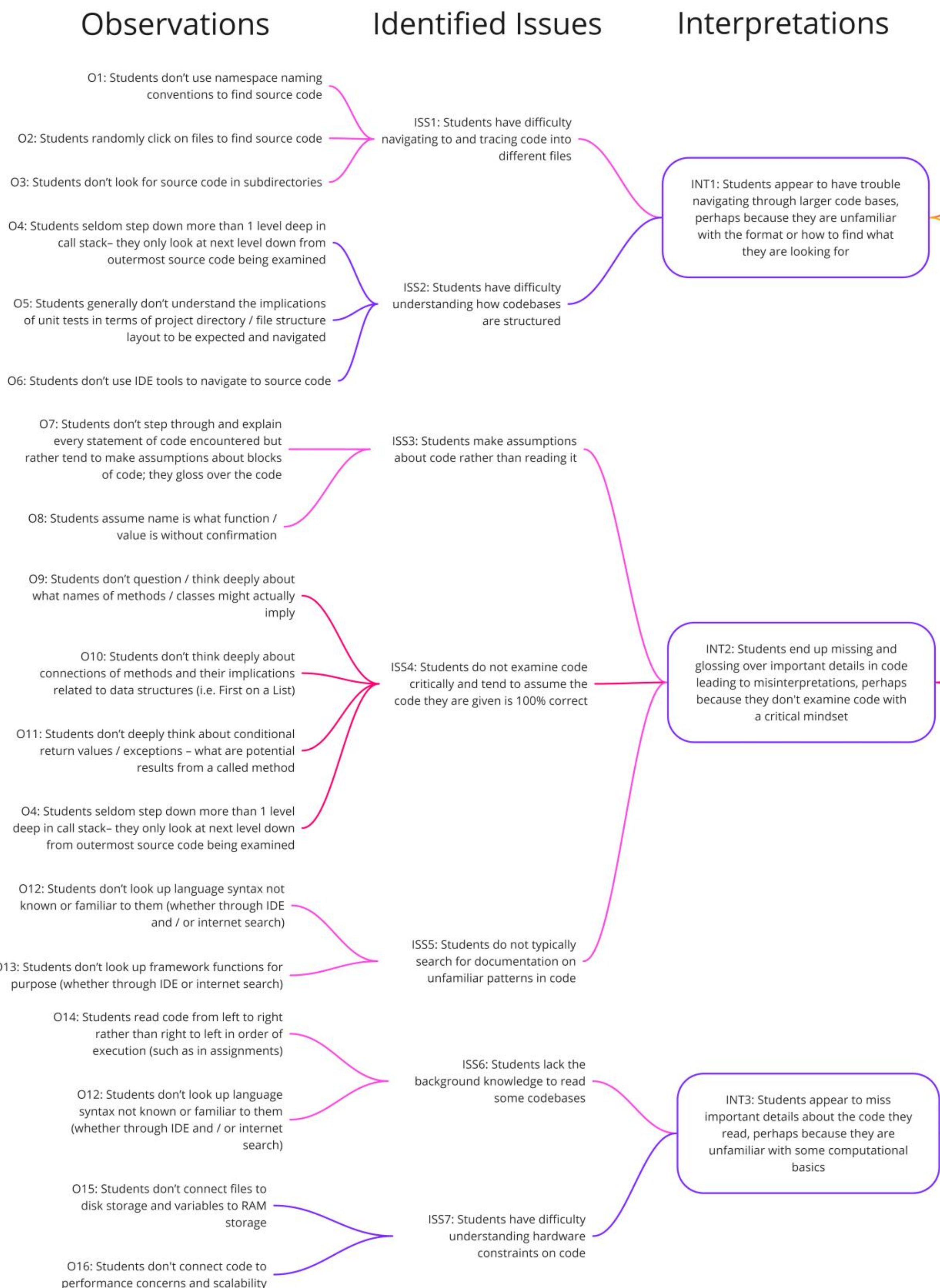


Figure 2.1. Code reading interview observations grouped and associated with underlying hypotheses/interpretations.

These three observations were also made in more than half of the student interviews and were consolidated under the more generic issue that students have difficulty understanding how codebases are structured (ISS2). ISS1 and ISS2 together form the evidence that led to the interpretation that students have trouble navigating through codebases due to a lack of experience (INT1).

Observations relating to student misinterpretations of code are also common occurrences in the student interviews, but are more diverse as students make misinterpretations in different ways. These observations were broken down into three core areas, each their own identified issue (ISS3-5). The issue of students making assumptions about code rather than reading it (ISS3), was used to group together the following observations:

- O7: Students do not step through and explain every statement of code encountered, but instead try to make assumptions and gloss over the code.
- O8: Students assume the name of a method completely describes its functionality without looking deeper.

Another issue, that students do not examine code critically and tend to assume the code is 100% correct (ISS4), groups together both rare and common observations, some already grouped in other issues, that showcase students not looking deeper into the code within a method. These observations include:

- O9: Students do not think about what the name of a method might imply about its function.
- O10: Students do not think deeply about the connections of methods and their implications related to data structures (i.e. first element in a list).

- O11: Students do not think deeply about conditional return values and what potential results from a called method could be (i.e. null).
- O4: Students not stepping down more than one level deep in a call stack.

The rest of the observations related to the general theme of code misinterpretations fall into another issue, that students do not typically search for documentation on an unfamiliar piece of code (ISS5). This issue was made clear from the following observations:

- O12: Students do not look up language syntax that is unfamiliar to them.
- O13: Students do not look up unfamiliar outside library functions for their purpose.

The student actions observed in ISS3-5 led to misinterpretations of code during the student interviews. Grouping these issues together helped develop the interpretation that students' lack of critical mindset and focus on how a program works leads to the missing of important information which results in the observed misinterpretations (INT2).

The remainder of the observations made during the student interviews are related to other times aside from misinterpretations and navigational issues that I needed to step in to help guide a student's code reading during the interview. The issue that students lacked the background knowledge to read some codebases (ISS6) was apparent in times I needed to instruct students to re-examine code a specific way or to do background research as seen in the observations:

- O14: Students read code from left to right (order of assignment), not right to left (order of execution).
- O12: Students do not look up language syntax that is unfamiliar to them.

A final issue, that students have difficulty understanding hardware constraints on code (ISS7) groups together observations from students I made after asking questions related to how efficiently the program that they were reading would run. These observations included:

- O15: Students do not connect long files to disk storage and variables to memory.
- O16: Students do not connect code to performance and scalability.

ISS6 and ISS7 both lean heavily towards the general idea that students had a lack of understanding of some computational basics such as how to read a line of code or search out information on code they were unfamiliar with, which led to interpretation that this lack of understanding resulted in students not fully understanding the code (INT3).

Chapter 3. DESIGNING THE TOOL

With the three possible interpretations created from my semi-structured interviews, work began on identifying the desired outcome for this project. We quickly determined that addressing the third root cause, that students had a knowledge gap preventing them from understanding the code, would be out of scope for the project as it touches more on students' prior computer science education rather than teaching them new skills to aid in the process of code reading.

Given the large number of observations of anti-patterns (uncorrected misinterpretations of code, lack of call stack tracing, etc.) made during the interviews, a reduction in the frequency of anti-pattern appearance during code reading by students was the clear-desired result of the teaching tool. We had several goals that we wanted students to achieve after using the teaching tool, specifically we wanted to see students:

1. G1: Navigate to/trace into different files directly from their predecessors on the code path without searching randomly
2. G2: Identify and research unfamiliar library functions
3. G3: Trace through the entire code path without skimming over methods
4. G4: Understand how codebases are typically structured

3.1 Finding A Way To Track Student Improvement

When trying to figure out how to quantify student improvement over these 4 objectives, several approaches were considered. The goal we had in mind at the start was to develop a final knowledge check at the end of the code reading module that we could have students perform and observe their behaviors much like was done in the semi-structured interviews. The problem with making the goal for the knowledge check "to observe a lack of antipatterns from students" is that without a reliable way to track those antipatterns I would be left needing to manually watch and record every student one on one as they complete the knowledge check. Knowing that would be an impractical solution, I started brainstorming ways to track a student's progress through a codebase to observe the desired outcomes laid out in G1-4.

The order in which students read the code and what files they examine needed to be tracked in order to observe the desired outcomes of G1 and G4, students' ability to understand the codebase structure and to navigate into files in the order they appear in the code path. One approach would be for students to somehow draw a line next to the code illustrating the path they traveled as they read the code. A similar proposal would have students document the lines they read in a table whilst specifying any jumps to different methods or files they visited throughout the process. Depending on how the teaching tool is delivered to students we could have also tracked clicks and cursor movements on specific files to get the order in which students navigated the codebase, but this would not have accounted for reading not mirrored by mouse activity. One approach that seemed to be possible regardless of delivery method would be to have students leave a trail in the form of comments down the path they traveled, with some way for them to mark the order in which the comments appeared. The "trail of comments" approach's agnostic relationship to teaching tool delivery method, the ease in which it could be

implemented, and the lack of any perceived burden on students when compared to the alternative delivery methods made it the most appealing option.

I also analyzed several approaches to quantify the desired outcome of G3: *students not skimming or skipping over areas of code* and the ability of students to correctly read the code in an application in such a way that they would be able to explain it to others. Having a series of multiple-choice questions tailored to the code being read and distributed throughout a code reading problem or at the end of a method would help verify correct readings. Such an approach could also target skimming if hidden side-effects in methods are present that would cause someone who just skimmed a method name to get a multiple-choice question wrong.

Another approach would be for a table that students would fill in at every method call describing what file they were on, where in the file they were, where they were about to jump to, and the current state of the code's execution. This table would be able to verify the order in which code was being read, check students' assumptions about the code, discourage skimming of the code due to the need to fill in the table, and verify how correct they were upon the examination of an instructor. However, usability tests I conducted of problems designed around the table showed that the table was slow to fill in, difficult for students to understand, and failed to track students reading files outside the code path which was an antipattern we identified previously.

Given these facts I was also concerned that a possible point of failure could be the students misunderstanding how to fill in the table rather than misunderstanding the code. Other approaches to quantify code skimming and reading correctness relied heavily on comments. One proposal would have students performing a matching between a bank of comments and a series

of blank spaces throughout the code path. Another would have students performing a multiple-choice selection between a series of comments for a given segment in a problem.

The final proposal was for a wide number of blank spaces for comments to be left in the codebase for students to fill in while they were reading code. This approach can help ensure skimming is tracked by ensuring all relevant areas of the code path have been commented and therefore not skimmed; the correctness of the comments would determine if a student had correctly read each snippet of code. This commenting approach would also help to track G2 as comment blocks could be placed before library functions, requiring students to explain the purpose of each in their response.

Hybrid approaches were also considered and could make it a simpler process to analyze the results of the teaching tool post-test by enabling automatic grading of multiple choice or matching questions as opposed to student written comments which would require manual grading for all students.

After much consideration, for the first draft of the final code reading knowledge check at the end of the module we would have a series of numbered empty comment blocks that students would fill out as they read the code. Blank comment blocks with ID numbers attached to them would be spread throughout the codebases I used in the final knowledge check in every critical section and in every file of source code. Students would be instructed to read though the codebase and for every blank comment block they come across, describe what the section after the comment block is doing and add it to a list of comments that they had already filled out to show the order in which they had done so, as seen in Figure 3.1.

With this data I could more easily follow a student's movement through the codebase in a format that required little explanation. This format also could be easily translated into mediums

such as Canvas through a simple quiz asking for the comments in the order they were written (as seen in Figure 3.1), allowing for an easy submission and evaluation of the assignment.

```
1 reference
public AccountCreationStatus CreateAccount(string username, string password, string confirmedPassword)
{
    //46:
    //
    if (password != confirmedPassword)
    {
        return AccountCreationStatus.PasswordMismatch;
    }

    //47:
    //
    if (this.accountLibrary.TryGetAccount(username, out Account _))
    {
        return AccountCreationStatus.AlreadyExists;
    }

    //48:
    //
    this.accountLibrary.AddNewAccount(username, password);
    return AccountCreationStatus.OK;
}
```

```
//46:
//check if password and confirm password match
//47:
//check if account already exist
//31:
//get all acounts and store them into accounts
//43:
//function is get all accounts in .json file and each element into a list
//48:
// create new account and return Status ok
```

Figure 3.1. Example of empty comment blocks in a codebase and a student filling in each comment block in order of execution within the box (blocks 31 and 43 are elsewhere in the code).

This format would ultimately be the one used in the final knowledge check and would be tied to 3 questions asking students to read through the code and fill in comments starting from 3 different endpoints if given a specific input. With a student's answers I could pick out antipattern usage by identifying comments filled in that resided out of the code path, incorrect descriptions in comments, and a wrong final output.

Despite this there remained a flaw intrinsic to all methods that do not require me to directly observe the students; that flaw is that students can lie on their answers to cover up antipatterns if they realized their mistakes later in the code reading process. To mitigate this flaw students were instructed to not erase any blank comment that was filled in erroneously, instead students should remark within the comment that the prior comment was made in error and give context as to why it was filled out in the first place and how the student realized their mistake.

3.2 Development of Concepts to Teach & Prototype Lessons

As shown in Figure 3.2, development of the practice problems started by examining the hypotheses formed from the issues identified in the interviews and prior research. Each hypothesis was broken down into different high-level code reading concepts we devised to make lesson development easier. Ideas for lessons were then outlined in a high level and associated with each identified high-level concept. These concepts and lessons were designed in a way to be language agnostic to all object-oriented languages as I was not yet sure what language the practice problems would be in. An added benefit of this was that the lessons the code reading module would seek to teach could be applied to any object-oriented language.

After breaking down each hypothesis into concepts, it was determined that the concepts of "lack of knowledge" and "hardware constraints" strayed too far from the goal of teaching code reading skills and were dropped from lesson consideration. With the concepts broken down into lessons, I now had a general idea of what was going to be taught to the students and how it was going to be taught and therefore sought to develop the final test of the module first to evaluate all outlined concepts before developing the lessons preceding it.

3.3 Prototyping a Summative Assessment

In the exercise given to each student and professional in the interviews I conducted, participants had to perform code reading problems in a codebase called "studio-quotes". This codebase is used by the instructors of Software Engineering Studio as a part of an early challenge in their course. The quotes challenge used in both the SES and my interviews served as an initial inspiration for what type of application the final test should be, i.e. a web API. I figured that comparing the results from my semi-structured interviews to results from a test on the same type

of application would serve as a more apt comparison.

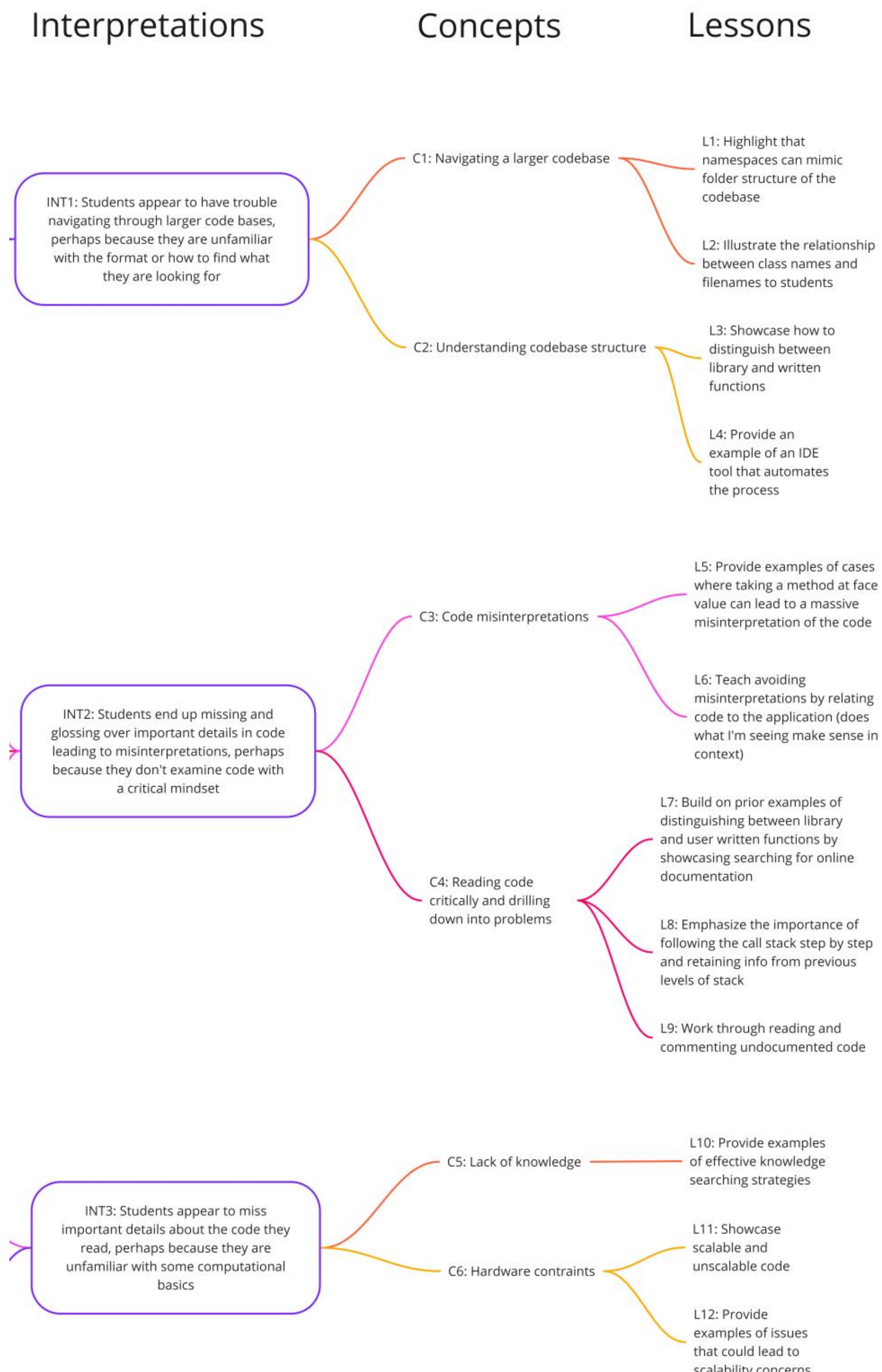


Figure 3.2. Code reading interview hypotheses/interpretations broken down into code reading concepts and then further into ideas for lessons to teach those concepts.

Once it was decided to develop an API, several ideas were brainstormed before finally setting on an API for a rudimentary social media application. I felt as if the concept of a social media application would lend itself well to an exercise like this as the students would likely be able to understand what the application should be able to do at a high-level without much explanation and because such an application could have a variety of features which gave me a multitude of ways that I could design problems to challenge students.

At a high-level, the API was initially designed to perform a few simple operations: account creation, post creation, and post lookup. In the account creation step, a request is made with a username, password, and password confirmation. The API would check to see if the password and password confirmation matched and if so, checked a local JSON account storage to determine if an account with the same username existed. If no such account existed, a new account would be added to the local storage with the provided username and password, while an OK response would be returned to the user. To create a post, a user would provide their account credentials (username and password) as well as text for the post. Like the accounts, the posts would be written to a file as JSON. Unlike accounts, posts are stored alongside an ID number and their creation time within the local storage. Getting posts from a user required a username as input, which would be used to filter the posts from storage. All post information that was stored when the post was initially created including the ID and creation time as well as the post text are returned with this endpoint. These are easy operations to understand at a surface level, but digging into the code multiple layers would require students to demonstrate the extent of their code reading proficiency. The first draft of this codebase contained over 20 files and roughly 500 lines of code. Care was taken in the design to utilize various design patterns students may be unfamiliar with as an additional measure to make the codebase more novel to the students. One

example came in the form of relying on a Singleton design pattern for the classes that dealt with accounts and posts, requiring students to navigate through private and public constructors to understand how certain classes are initialized and later called.

With a suitable codebase created, time was spent brainstorming how best to assess that students have learned from the various lessons in the course. In our research we had identified several anti-patterns which we wanted to address in this module, as such it seemed the most logical course of showcasing the benefits of this module would be to place the students in a similar environment to the interviews where those anti-patterns were discovered and have them not exhibit those same anti-patterns to the same degree. The codebase for the summative assessment, or final knowledge check as it grew to be called, was designed as a web API for this reason as well. In addition, the codebase was designed to present more opportunities for students to exhibit anti-patterns than the codebase from my prior interviews, so it seemed to already be a more than fair comparison. The comment block tracing method from Chapter 3.1 was also included in the codebase to track anti-patterns in student responses.

3.4 Prototyping Formative Assessments

With a completed final knowledge check and several anti-patterns to teach students to avoid, I started drafting smaller codebases in which I could add exercises to reinforce the knowledge throughout the lessons and to better prepare students for the final knowledge check. Like the social media app used for the knowledge check, I wanted to use an API designed around a simple concept. After some thought and searching through phone apps for inspiration, I decided on creating a note taking application, an application to help plan meals, and a calculator.

All applications would be used as examples for showcasing how to navigate through larger codebases, avoid common reading mistakes, and to serve as practice for the knowledge check. In

addition to their shared goals, the notes app would be designed to help introduce students to the singleton design pattern and how to read code that performs file I/O operations. The Meal Planner application would serve to introduce students to the process of looking up unfamiliar functions and navigating through a codebase with dozens of files. The calculator was designed to help teach students about endpoints of an application and where to start reading code, but given the breadth of content in the course and the reality that any of the additional 2 codebases could also be used to teach the same topic led to the calculator app being scrapped from the final lesson plan despite the codebase for it being completed.

The initial concept for the Meal Planner app was to create a program that, when given a set of ingredient names supported by the application and an integer representing the calorie target for the meal, picked a meal containing only provided ingredients names that was as close as possible to the target number of calories. In development, this output changed to become all meals that could be made with the set of ingredients provided in the input along with their caloric values. I went with this concept because I felt that it could be implemented through a factory design pattern with dozens of ingredient and meal classes, which would be a useful tool to help teach students to trace into files whose classes may not be immediately clear, such as variables with an interface as a type that are assigned a class by a factory. I also thought that if I could teach students how to navigate through a codebase dense with files, then they would be less likely to jump around to random files searching for methods or variables in future codebases as outlined in the Figure 3.2 lessons L1. With the idea to use factories to fetch specific meals or ingredient objects planned, I also sought to include difficult to understand outside libraries to heavily nudge students towards looking up official documentation for support (lessons L3, L7).

In deciding how to go about this I drew on my own past experiences and chose to include some methods from the System.Reflection C# library [9] to create instances of classes. Specifically, the constructor of one of the business logic heavy classes defines a dictionary that maps every meal's name to a list of ingredients contained in that meal. Meal objects all inherit the same interface and have their own implementations of a GetIngredientNames method and GetName method, both of which are needed from each meal to populate the dictionary. As such reflection is used to load every type in the assembly, these type names are then filtered down to only the classes in the Meals namespace.

Each of these classes is instantiated through their type names and their various getters are called to add a key value pair to the dictionary. At this point in their education, students likely have not been taught anything about reflection and will be unsure what the methods called to get these types from the assembly do. As such with some prior instruction, my hope is that this will prompt them to look up official documentation on these methods to figure out what they do (lessons L3, L7). As for the rest of the Meal Planner app flow, when given a comma delimited string of ingredients, the API does the following:

1. Splits the string into a list by the commas
2. Loops through all meals we placed into the dictionary earlier
3. Checks each meal to see if every ingredient required by the meal is in the list of ingredients we have
4. If we have all the ingredients to make the meal, loop through the meal's ingredient names and get the ingredient object associated with each ingredient name through the FoodItemFactory
5. Grab the calorie count for each ingredient and add them together

6. Add a new object containing the meal data and the meal calorie count to a list
7. Returns the list of all found meals as a response

This code is not designed to be the most elegant or efficient way to perform the functions of this API, and due to the limitations of relying on separate objects for every meal and ingredient, we are limited on the number of meals we can return and ingredients that we can accept as input without expanding the size of the codebase to a hard to maintain state. All of this was done intentionally to provide an environment where students can practice proper code reading and apply the skills they will be taught in lessons.

The notes app was designed to get students used to exploring files that do not contain source code as a part of their tracing. In this codebase, students would be required to read into a JSON file containing stored notes. This codebase was also meant to introduce students to the singleton design pattern before the final knowledge check to remove the possibility of any unfamiliar design patterns being the reason students may fail to properly read the code rather than anti-patterns. This is the reason why one of the main classes where the business logic for this API occurs is instantiated by the controller via a singleton design pattern. The notes app's CreateNote endpoint accepts an object containing strings that represent the title and contents of a note. It checks the title against all titles in the note storage and if the note title already exists, then the note's content is changed to match the new content passed to the endpoint and is written back to the notes JSON file. If the title does not exist, then the new note with its title and contents is appended to the end of the note's JSON file. The GetNote endpoint works in a similar manner to the create endpoint in the sense that it also checks the local storage for a note title. If the note is found then the note's contents are returned to the caller.

After both codebases were developed, I looked back to the Social Media app used for the final knowledge check and felt that I needed to make some changes to make the knowledge check a true culmination of all the prior learning students had done in the course. As such I chose to incorporate a factory design pattern as a part of a new series of “post action” endpoints. I defined a post action as an interface that defined a PerformPostAction method which took in a list of all posts and a specific post id. Depending on the type of post action a different operation, such as incrementing a field in the post object, would be performed on the post matching the id and app posts would be written back to the JSON storage they resided in. To facilitate new actions, I added “likes” and “views” fields to each post stored in JSON and decided to theme my actions around those 2 fields. I created 2 actions designed to simply increment the count associated with each of the fields by 1. Both actions were then set to be resolved through a factory, that way the section handling the business logic for posts only needed to call the factory to get the correct post action, and call the PerformPostAction method of whatever the factory returned. The like endpoint passed in a string that would cause the like action class to be returned, whereas the view endpoint would do the opposite and pass in a string that would cause the factory to return the view action class.

3.5 Creating Lesson Plans and Exercises

With exercises set up to reinforce the lessons I wanted to teach, I sought to write the lessons themselves. My goal with the organization of these lessons was to start with what was in my view the simplest topics and slowly build towards the tougher material. As a result, I started the lessons off with a section on codebase structure, class names, and namespaces. These lessons were meant to use C# throughout but to also be applicable to any object-oriented language,

which helps a great amount with the accuracy of many of my statements on code style as C# has extensive official style guidelines I linked to directly in the lessons for reference.

Due to the issues students had with finding methods and classes in my interviews, I described how class names typically have a 1:1 relationship with their filenames (lesson L2). I also described the relationship between a class's source code's file path and its namespace (lesson L1). According to style conventions, namespaces have different segments separated by periods which correspond to the project/folder that contains a file of source code. With the knowledge of a class's namespace and the name of a class, students should be able to determine what file the source code of a class resides in.

Finally, I explained that students could identify what namespaces in a file of source code belonged to the current project/solution. Namespaces that share a common prefix share a part of their file path according to the best practices, therefore any namespace used that shares elements with the namespace of the class they are examining is a namespace located elsewhere in the source code. On the contrary I also explained that namespaces that lacked this prefix were likely to be outside packages or libraries that were used and could be looked up if one was unsure of their purpose (lesson L3).

Throughout these early sections I endeavored to provide links to official sources to back up my style guideline claims and included several pictures to illustrate my points as seen in Figure 3.3 (lesson L4). To reinforce this early lesson, I asked students to identify the different namespaces in my NotesApp API only by looking at the file structure of the repository as a part

of a quiz which can be seen in Figure 3.4. Once the student answers they would be provided with the correct answer and an explanation.

Codebase Structure

Introduction

Most codebases you come across are likely to follow the coding style guidelines of the language they are written in. In the case of C# you can find a number of these guidelines in the language's official documentation (https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions).

For this section of the module, we will mainly be focusing on the naming guidelines and conventions you are likely to see in C# code or any similar object oriented language (https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines).

Codebase Structure - Namespaces

Namespaces

Namespaces are present in several mainstream programming languages, including C#, Javascript, Java, C++, Python, etc. Namespaces are used to organize classes into groups and are also used by classes to reference other classes that reside in different namespaces.

In C# you will see two types of namespaces:

1. Namespaces that refer to code inside of the project/solution being read
2. Namespaces that refer to outside libraries and packages

An easy way to differentiate between the two types is to compare them to the namespace of the class currently open. It is best practice for namespaces that are part of the same project/solution to share a common prefix.

In the below image of the NotesLibrary class from the previous example, **NotesApp** is defined as the namespace of the file on the first line. We can also see a few lines later that we state the class **uses** code from classes in the **NotesApp.Models** namespace. We can tell that this namespace resides in our codebase due to their sharing of the prefix **NotesApp**. We can also tell that the two other namespaces **(Microsoft.Extensions.Caching.Memory** and **System.Text.Json)** are outside libraries because they do not start with the prefix **NotesApp**.

```
namespace NotesApp
{
    using Microsoft.Extensions.Caching.Memory;
    using NotesApp.Models;
    using System.Text.Json;
```

IDEs (integrated development environments) like Visual Studio have tooltips that display the namespace an object belongs to when a cursor is hovered over them. This is an easy way to discover the origin of an object used in a program.

```
private readonly IMemoryCache recentNotesCache;
private List<Note?>? [
private static NotesL
```

interface Microsoft.Extensions.Caching.Memory.IMemoryCache
Represents a local in-memory cache whose values are not serialized.

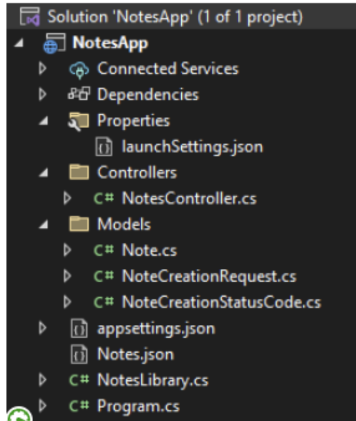
Figure 3.3. Examples of Codebase Structure lessons linking to official language documentation and illustrations being used to teach codebase structure concepts.

Once I explained namespaces and how to locate classes and get information on outside libraries, I set out to explain the basic structure of APIs. This was done to ensure that students feel more comfortable identifying code entry points; in the case of APIs, this would be the controller. The controller was highlighted as the area where web requests are received and that headers above a method help to indicate the URL that each method is designed to serve and that

the parameters accepted by the method are a part of the request body. I then asked some questions using the Meal Planner codebase to try and reinforce how the API works to the

Quiz Instructions

The following questions relate to the NotesApp codebase (GitHub Link) which we will also be using in later exercises. For convenience, a screenshot of the codebase's file structure is below:



Answer all questions using this image.

(Hint: Program.cs does not define a class so it has no namespace)

Question 1 1 pts

Given the file structure in the codebase above, how many different namespaces are used to categorize the source code?

Question 2 1 pts

Given the file structure in the codebase above, what are the names of the namespaces used in this codebase?

Structure your answer as namespaces separated by commas with no spaces ex: namespace1,namespace2

Figure 3.4. Example of brief quiz used to reinforce code reading concepts taught in lessons.

students. The students are asked to identify the file where the web requests are serviced (the controller), the method that receives the request, and the URL that method services. Students are also asked to identify the namespaces in the Meal Planner repository to further reinforce the earlier lesson on namespaces and code structure.

With the basics covered, the topic of code tracing was tackled next. I wanted to provide students with a formal set of instructions to read code in the hope that this would lower the cognitive load on them (lessons L8, L9). I devised a table (see Figure 3.5) that I planned to have students fill out as they progressed through the codebase.

Call Stack Depth	Line #	Filename	Method	Called Filename (or Called Library)	Called Method	Parameters	Return Value

Figure 3.5. Trace table students were required to fill in during their first code reading exercise.

The table itself is designed to get students to focus on the flow of the code's execution from start to finish while also keeping track of important values such as parameters and return values. I made sure to define each of these fields as thoroughly as possible for students as some are not readily apparent. For example, Call Stack Depth refers to the number of method-calls the current location of the trace is from the start of the trace. If the controller, which has a depth of 0 as it is the starting point, calls a method the depth would be 1 once we trace into that new method, if that method calls another the depth would increase to 2 and so on. The purpose of this was to help students keep track of their position in the code, as when they finish in a method they could simply go back up in the table to the nearest row with a lower depth and continue reading from that spot. My hope was that after going through the process of filling out a table like this, as arduous as it may be in the moment for a student, it will commit the importance of a methodical approach to code reading and that students will be actively filling out the table in their own heads

moving forward. This pillar of structure in a chaotic torrent of code should help to keep students grounded and focused on the task at hand rather than getting overwhelmed in the face of unfamiliarity.

With the introduction of the table format completed, I took time in the lessons to highlight some important hurdles that students needed to keep an eye out for when reading code. From my interviews, I noticed that when students were overloaded or tried to skim through an area of code, they tended to make several missteps in their readings (Figure 2.1, O7). As such, I took time to emphasize the importance of carefully reading constructors so that we are sure of a program's starting state. Overloads were another area of concern, in my interviews some students ended up straying from the code path by entering an overload of a method different than the one called. At this point I also made sure to add various method overloads to the Social Media and Meal Planner codebases to provide a test for students as seen in Figure 3.6 (Lesson L6). Finally, the importance of pacing oneself when reading variable names was emphasized; there were a few students in my interviews that were not careful reading names and mistook a type name for a variable because they shared the same name but with different capitalization. I made sure to also include several variables like this throughout my codebases as a result (Lesson L5). For example: `Note? Note`.

```
public void CreatePost(string username, string password, string postText)
{…}
public void CreatePost(string username, string postText)
{…}
public PostItem? GetPostFromUser(string username)
{…}
public List<PostItem?>? GetPostsFromUser(string username)
{…}
```

Figure 3.6. Examples of overloads and deceptive naming practices used in the final knowledge check to try and elicit an incorrect trace from students.

With some feedback from my capstone committee chair I decided to provide a step by step written example showcasing the first few rows of the table being filled in using the create note endpoint from the Notes App. In this example I made sure to start with the constructors which run before API receives the HTTP request to show by example how important it is to understand the state of a program on startup. From there I briefly went into some of the steps taken after the request is received by the endpoint and hand the table off to the students to complete. Once they finish filling out the table, I presented them with the solution before having them fill out 2 additional tables. One of these tables asks students to trace through the get note endpoint for a note that exists in our JSON storage and the other asks students to trace through the GetNote endpoint in the case that we ask for a note that does not exist. After these exercises students are once again shown the correct solution in the hope that they will learn from any mistake they may have made should they arrive at an incorrect answer.

After the students have a few exercises of practice tracing through code with the table, we transition into some less structured code reading exercises. By this point I expected that students would understand how to trace between files and navigate through a codebase based on the prior lessons. For the final exercise before the knowledge check I present the students with the Meal Planner codebase, which has been modified to have several blank comment blocks throughout the codebase as seen in Figure 3.7. Each of these comment blocks is labeled with a unique ID number, and they are numerous enough to cover every critical operation in the codebase; the blocks even exist in areas of so-called “dead code” such as the overloads I introduced as a test for students. As students traverse the codebase, they would be expected to fill in each comment block they come across with two bits of information: the order in which they came across this comment block as they read the code (i.e. first, second, etc.) and an explanation of the code that

```
[HttpGet("GetPossibleMeals")]
0 references
public ActionResult<MealsResponseItem> Get(string mealIngredients)
{
    //2:
    //
    List<string> ingredientList = mealIngredients.Split(',').ToList();
    MealsResponseItem mealsResponseItem = new MealsResponseItem
    {
        MealInformation = this.mealSearcher.GetPossibleMeals(ingredientList)
    };

    //3:
    //
    if (mealsResponseItem.MealInformation != null && mealsResponseItem.MealInformation.Count > 0)
    {
        return Ok(mealsResponseItem);
    }
    else
    {
        return NotFound();
    }
}
```

Figure 3.7. Endpoint of the Meal Planner Controller with blank comment blocks added.

comes after the comment block, as described in Chapter 3.1. I chose to have students fill out these comments rather than setting them loose on the codebase and with an input and asking for the output because I wanted to better keep track of how they are reading the code. If students fill in these comments, they are essentially leaving a trail that shows the order in which they read the codebase. From that I can see if they read through the code in the order of execution or not, and if they exhibited any anti-patterns such as searching through unused files or methods outside the code path such as my added overloads. The comments also let me see the students' interpretation of the code, which can later be assessed for accuracy and provide a better idea of why a student arrived at the output whether it is correct or incorrect. Once a student fills in their comments and submits what they believe the Meal Planner's output will be given a specific input string, they are shown an example showcasing comments being filled out in the order of execution and the correct output of Meal Planner for the input given.

For the final knowledge check I wanted to make use of the comment block strategy of evaluating students used in the last exercise. This was done not only to get the information provided by this method, but also to not introduce new procedures to the students that could contribute to a misunderstanding or a mistake being made not due to any misinterpretation of the code, but due to responding to an unfamiliar problem format. As such, comment blocks were added into the Social Media codebase in much the same way that comments were added into the Meal Planner codebase. Like the last problem, students are asked to navigate through the Social Media app codebase given a specific input to the controller at a given endpoint. Students should trace through, filling out comments as they go and provide the correct result once the code finishes and returns from the controller. In this final section, students are made to trace through three different sets of endpoints (Figure 3.8) and inputs that are designed to take students down various code paths which contain elements of all prior lessons taught up to this point (Figure 3.9). Given the size of the Social Media codebase, this exercise also puts students through a more difficult code reading experience than the students in the interviews were given, making it a more than fair comparison. After the students finish tracing and providing outputs, I ask a few brief short answer questions prompting students to provide their own inputs that would lead to an endpoint responding with a specific error state. These are done to have students not only

```
[HttpPost("CreateNewAccount")]
0 references
public ActionResult<string> CreateNewAccount(AccountCreationItem accountCreationItem)
[HttpPost("CreatePost")]
0 references
public ActionResult<string> CreatePost(PostCreationItem postCreationItem)
[HttpPost("LikePost")]
0 references
public ActionResult<string> LikePost(int postId)
```

Figure 3.8. Endpoints of the Social Media codebase used for tracing problems in the Final Knowledge Check.

demonstrate their understanding of the codebase, but also to hopefully along with the comments identify students who worked together on the exercise against the directions provided.

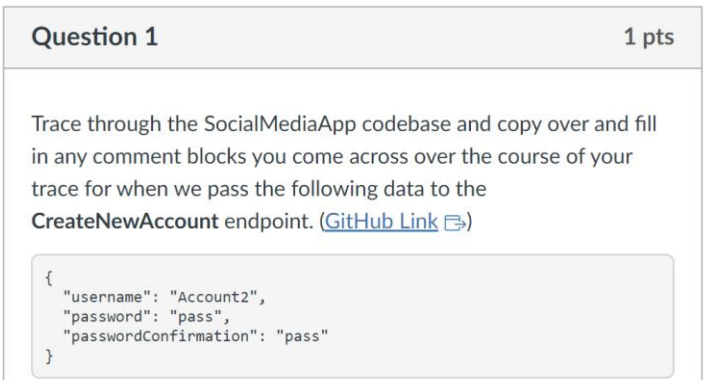


Figure 3.9. Example of input being provided for a problem asking a student to trace through the Social Media codebase's CreateNewAccount endpoint.

3.6 Usability Testing

As one can see from Figure 3.10, the lessons and exercises were compiled into a 30-page word document with a variety of text lessons and image examples. I reached out to two students from the earlier semi-structured interviews to provide feedback on the lesson plan and they agreed. Both students were in their junior year of their undergraduate degrees in Computer Engineering when they participated in their interviews and were now about to start their senior year of their degrees after a summer internship at a large software company. These students had no exposure to C# other than the previous interview in the Spring, so I felt they would be excellent candidates for evaluating the course's ability to teach inexperienced students code reading strategies even if they were not familiar with the language in the course.

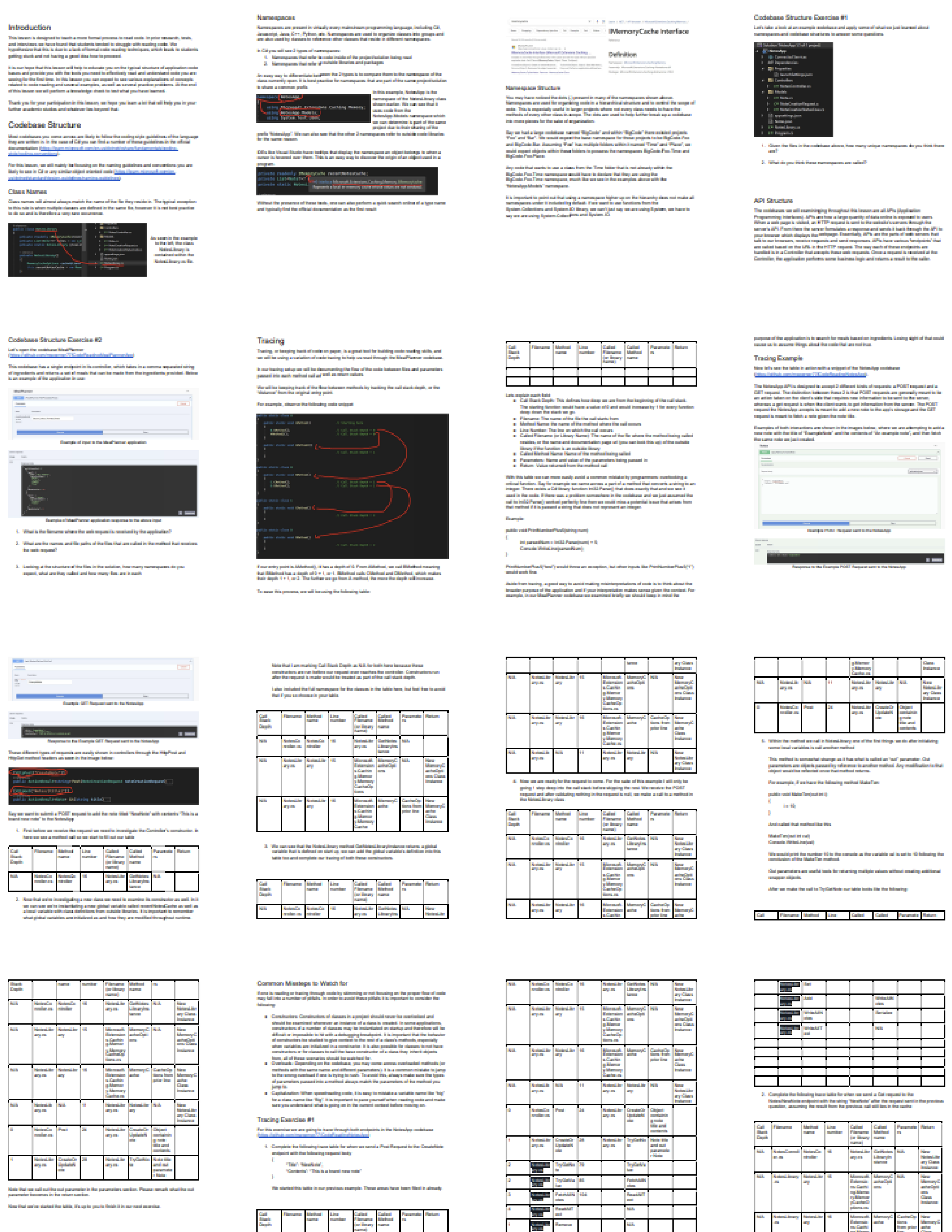


Figure 3.10.1. Condensed "shape" of the word document containing prototype lesson plan and student responses from usability testing.

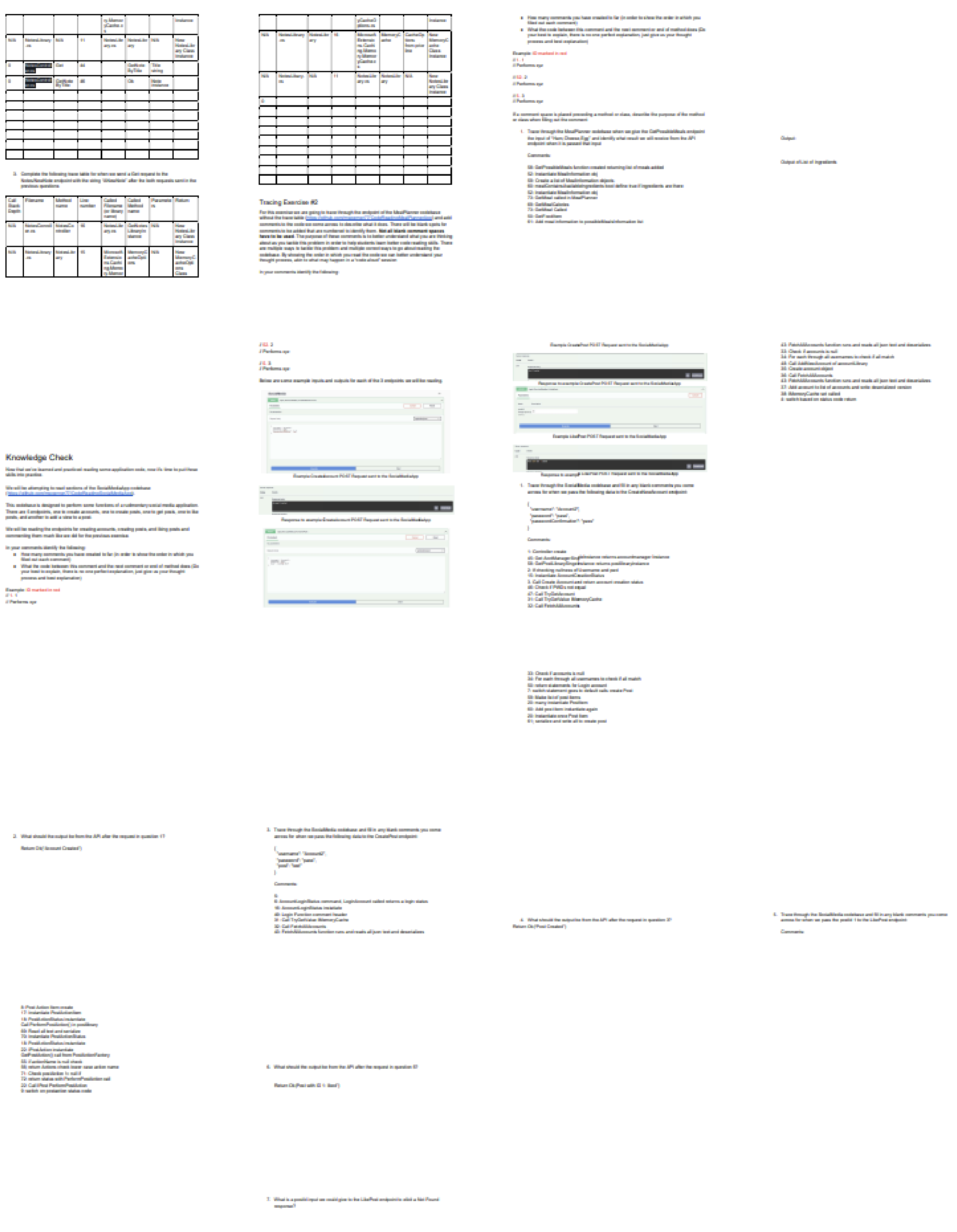


Figure 3.10.2. Condensed "shape" of the word document containing prototype lesson plan and student responses from usability testing.

The lessons and exercises were put into a word document and each student was given a copy of the document. The students were in the same room and worked through the exercise at the same time while I watched over zoom. The students were instructed not to collaborate with one another and to stop when they finished an exercise to wait for the other student. Once both students completed an exercise, we would stop and I would show them the answers for the exercise they completed. I would also collect feedback from both on how clear the lesson and exercise they worked through were, if any additions should be made, and if they felt they understood the material, this feedback is compiled in Figure 3.11.

This process of usability testing through the entire module ended up taking far longer than initially estimated. What I expected to be roughly a 90-minute process of working through the lessons and gathering feedback turned out to be over 150 minutes. This was due to the exercises taking longer than expected for the students to trace through, but also could be since both students had just finished a long workday before immediately joining the Zoom call to test these lessons while still in their office. This also served as an indirect benefit to the testing as I could see how the lessons impacted students already suffering from a bit of exhaustion and cognitive overload.

After the testing concluded I asked for some final feedback on the code reading module and if both students felt they were better equipped to read unfamiliar code following the completion of the prototype lessons. The answer was a resounding yes and it also showed in their results as the test went on. Both students were exhibiting many of the same anti-patterns they showed throughout their earlier interviews during the first several problems of the module. Once they had completed the first tracing exercise things appeared to change. In both the second tracing

- Namespace structure: add pictures, hierarchy (tree)
- Show the repo for 2-1 and revise wording.
- Revise 2-2 entirely. Very confusing
- Change 2-3 to Mapping question
- What is namespace of a file
- Teach structure a bit more
- Images are disconnected from codebase exercise
- Create your own namespace tree
- Introduce codebases more

Tracing

- Image text too small
- Bold mentions of headers and clarify them more
- Link it again and link the controller file specifically before table
- Hyperlink filenames
- Explain N/A earlier
- **Line number right after call stack depth**
- Colors for different columns
- Understand the constructors to understand state
- Bold things added each step
- Set apart code snippets
- Video see me fill out the table, or source code
- Recommend splitscreen code and table
- Dont jump into library functions
- Focus on the value of the parameters
- Mention cache in problem 1 so they don't get confused with move
- For 2 mention try get value will succeed and will get the note
- If we send a get request to the controller (2)
- Maybe provide videos of the solutions after the fact
- Explain questions better
- For question 2 give students a reminder about the common pitfalls
- Mention endpoint in the controller start get
- Output -> what controller returns, also give output example here too (be specific about what i want) (fill in the blank)
- Write down why exercises had the wrong answer, what assumptions? Make them think about it

Knowledge check:

Decide if json files will be populated (if so show them in NotesApp)
Clarify in question that past result remains

Overall it did help them
Just emphasize they take it slow
Beautify json
Break it up into lessons
Questions dont depend on one another
Ask for results other than output i.e. file changes

Figure 3.11. Feedback collected from first round of usability testing on prototype lessons.

exercise and the final knowledge check the students did not navigate to unused files and did not make any incorrect assumptions tied to their reading of the code. Based on this observation alone I considered the lessons to be a resounding success at this stage, but there were still several

problems that became apparent throughout this testing. Several exercise questions were described as poorly worded, and led to some confusion on behalf of the students that led to some incorrect answers. On the final knowledge check, I had designed the questions to each build off the response from the others. For example, the first endpoint the students traced through created an account, the second endpoint made a post from that account, and the third endpoint liked the post that was just created. The relationship between these questions was not made clear, which caused one student to operate on the assumption that the questions were independent of one another leading them to the answer that a post could not be created because the account did not exist, and that the post could not be liked because the post did not exist. Despite arriving at the wrong answer, the comments they wrote throughout the tracing process indicated that they had indeed performed a correct reading of the code given their incorrect input, which told me the question itself needed to be changed. Above all there was another very important piece of feedback that I received from these students, that being that video recorded examples would have helped them understand a lot better than some of the images and text I had provided. This was especially true in the table tracing example, where they had a hard time reading what had changed in the table at each step due to the changes not being bolded or standing out. They explained that seeing the table being filled out in real time with the code alongside it would have helped them understand the strategy a lot better. This bit of feedback was echoed for every other example as well.

Following all the incredible feedback I received from my usability testing, I decided to make several alterations to the exercises of the course as I converted the module from a word document to a Canvas module. These alterations are described in the next section.

3.7 CREATING THE FINAL MODULE

Following the usability tests, I decided to rework some of the codebase developed for the code reading module. Both the Notes and Social Media codebases originally had a component that cached results in addition to performing file I/O operations. Because I had committed to not having questions depend on one another's answers the caching did not really serve a purpose between questions without specifying what was inside the cache. I felt that adding additional clarifications to questions that were already dense with explanations would be unwise and chose to instead remove the caching elements from the codebases.

When creating the module in Canvas using the word document as a base, several minor steps had to be redone. Several images had to be retaken and resized to be more easily viewable. Important text was bolded. JSON was made more readable by spreading it out to multiple lines. Many explanations in lessons were reworded slightly to make them clearer to the reader. Some major changes occurred. I added new explanations on nullable types in C# and a reworked answer key. Due to the codebases for this module being written in .NET 6, my IDE, Visual Studio, prevented me from compiling without warning messages unless I included nullable types. Nullable types are not exclusive to C#, so I figured it was another useful thing to teach to students that may help them write better code in the future.

Answer keys were completely re-done from the ones in the usability tests. Instead of just providing the correct answer for every problem, the answer keys were broken into segments with large problems consisting of multiple segments. Each segment fully explained the tracing process and my route for approaching the problem using the lessons taught in the course alongside step-by-step guides and a video recording of myself solving the problem segment by segment and providing commentary on every decision made (Figure 3.12). The recordings

themselves total over 90 minutes for both answer keys and were designed so that students could learn from their mistakes after finishing an exercise.

Call Stack Depth	Line #	Filename	Method	Called Filename (or Called Library)	Called Method	Parameters	Return Value
0	24	NotesController.cs	Post	NotesLibrary.cs	CreateOrUpdateNote	Note creation request with title = "NewNote" and contents = "This is a brand new note"	**NoteCreationStatusCode. Created**
1	27	NotesLibrary.cs	CreateOrUpdateNote	NotesLibrary.cs	TryGetNote	Title: "NewNote" and out parameter note	false, sets out parameter to null
2	77	NotesLibrary.cs	TryGetNote	NotesLibrary.cs	FetchAllNotes	None	No return value, but fetches list of notes from json file and assigns them to this.notes
3	96	NotesLibrary.cs	FetchAllNotes	System.IO.File	ReadAllText	"Notes.json"	string version of Notes.json file content
3	97	NotesLibrary.cs	FetchAllNotes	System.Text.Json. JsonSerializer	Deserialize	string version of Notes.json file content	string version of Notes.json converted into a list of Note objects
1	43	NotesLibrary.cs	CreateOrUpdateNote	Note.cs	Note	Data needed to create a note object	**new note with filled in parameters**
1	53	NotesLibrary.cs	CreateOrUpdateNote	System.Text.Json .JsonSerializer	Add	new note with filled in parameters	**No return used, adds not to list of all notes**
1	54	NotesLibrary.cs	CreateOrUpdateNote	NotesLibrary.cs	WriteAllNotes	None	**N/A**
2	**102**	**NotesLibrary.cs**	**WriteAllNotes**	**Sytem.Text.Json. JsonSerializer**	**Serialize**	**List of all notes**	**string representation of all note objects in a list**
2	**103**	**NotesLibrary.cs**	**WriteAllNotes**	**System.IO.File**	**WriteAllText**	**"Notes.json" filename and string representation of all note objects in a list**	**No return, writes siring to json file Notes.json**

Figure 3.12. Example of a filled in trace table provided in the answer key. This table is filled in over 3 small videos with the progress from the final video in bold.

After the course structure and codebases were redesigned, the module was added to Canvas as a series of pages containing lessons and answer keys and quizzes containing exercises. The

module was designed to lock the tracing exercises behind reading the relevant material, and the answer keys behind completion of their associated quiz.

The module lessons were constructed in 2 parts that built on one another. The first part of the Canvas module focused on code architecture and explained the overall file structure of a typical codebase, how to identify files based on namespaces, and distinguishing between outside library functions and functions within the codebase among other lessons. The second module was focused on the act of tracing through larger codebases and highlighting various pitfalls that could lead to incorrect readings of code and other misinterpretations (see Figure 3.13).

Following the lessons, students were instructed to trace through the Notes App codebase's CreateNote and GetNote endpoints with different inputs and fill out a trace table (Figures 3.5, 3.12) while doing so. After revisions, the Notes App codebase consisted of 6 files and roughly 222 total lines of source code (Figure 3.14). Students should be able to understand how this codebase worked from their traces and needed to explain the effect different code paths within the same endpoint had on the note storage JSON file in an additional question.

Once students successfully completed the tracing table exercise, they were presented with an answer key accompanied by video demonstrations of the trace table being filled out for the exercise step by step. Students are not expected to perform a perfect reading of the code in the first tracing exercise; the answer key serves as part of the learning process wherein students come to realize any mistakes they may have made during their exercise and avoid repeating them in the future.

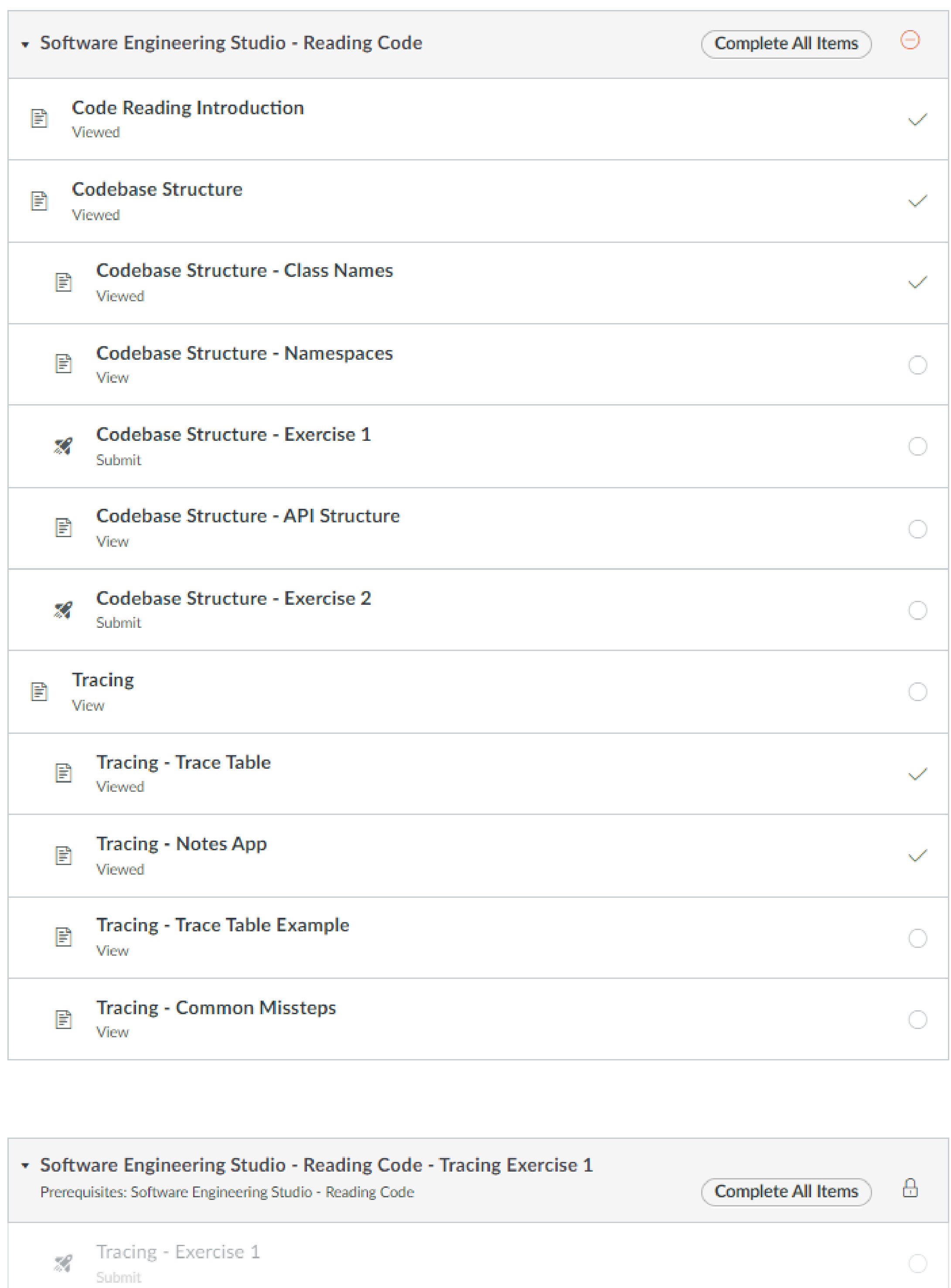


Figure 3.13. Partial layout of the module in Canvas, students needed to complete each section before accessing the next section of content.

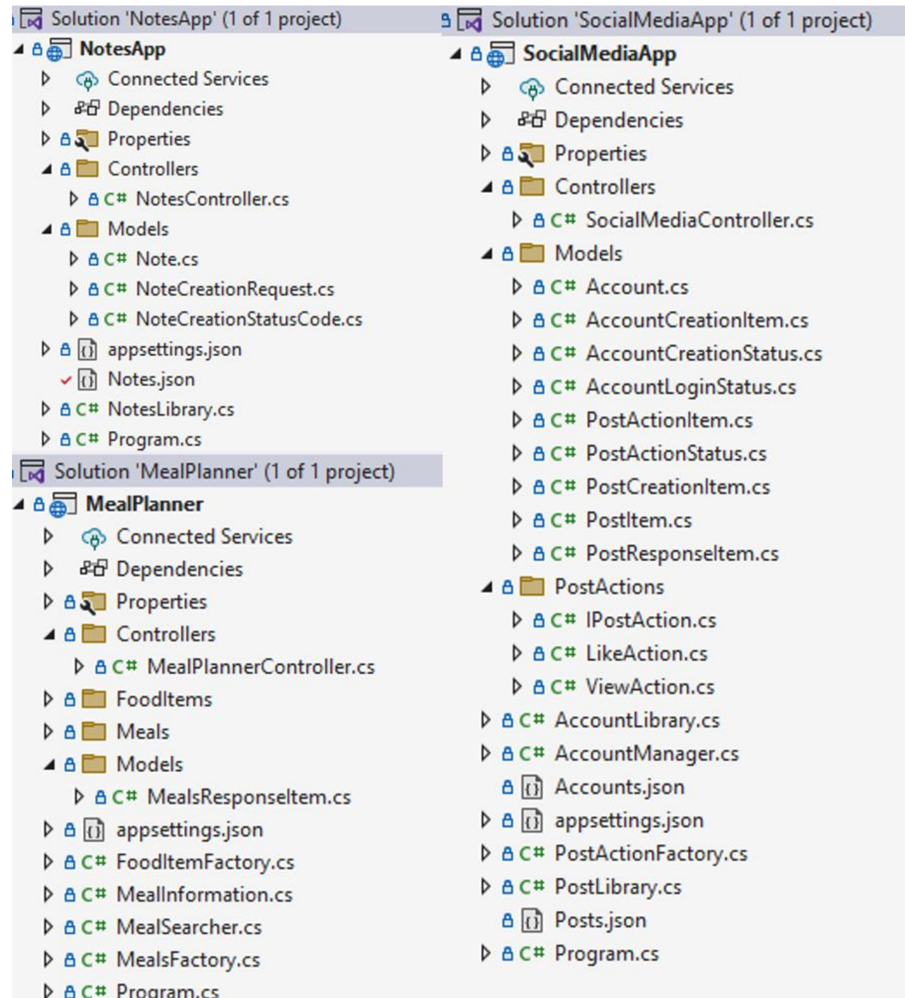


Figure 3.14. Layout of codebases used for the code reading module. Collapsed folders FoodItems and Meals in the Meal Planner codebases contain 15 and 29 files of source code respectively.

Once students progress past the trace table exercise and have learned the importance of keeping track of the call stack through various methods, they are presented with an explanation of the previously mentioned commenting system which they will use for the next exercise and the final knowledge check. I produced a video showcasing a section myself tracing in the Notes App codebase filling in added comment blocks that were not present in the student's version of the codebase. This video was accompanied by a list of written instructions (see Figure 3.15) for

commenting and submission instructions. Students were told to copy any comment blocks they came across in their trace along with an explanation of what each segment does to an essay space attached to each question; comments were separated by line and should be in the order that the students examined them. Students were informed that these exercises were to be graded for completion, not accuracy to avoid the possibility of students not including anti-patterns they correct themselves in their answers. With the commenting system set up the way it was if a student fell into using some anti-patterns but recognized it, they could show that in their comments. An example of this could be following an incorrect overload, realizing the mistake, and then progressing to the correct overload but leaving the incorrect one's comment block in the answer. Such a response demonstrates that a student knew they made a mistake and corrected themselves and is more valuable data than if a student omitted the mistake in their answer.

With the commenting system explained, students were introduced to the larger MealPlanner codebase which now contained 51 files containing 828 total lines of source code, which is deceptively large due to 44 of the files defining small classes with roughly 10 lines of code each (see Figure 3.14). This codebase was also designed to include various code reading pitfalls for students such as method overloads, similarly named methods, and a liberal use of outside functions that students may not have been previously exposed to. These pitfalls were included in this exercise to give students ample opportunity to demonstrate the anti-patterns that the module was designed to help correct. Like the first exercise, upon completion of this exercise students were presented with an answer key consisting of several short videos that broke down the problem step by step and walked through the solution.

Quiz Instructions

For this exercise we are going to trace through the endpoint of the MealPlanner codebase without a trace table (GitHub Link) and add comments to the code we come across to describe what it does.

In the code, there will be blank spots for comments to be added that are numbered to identify them. **Not all blank comment spaces have to be used.**

The purpose of these comments is to better understand what you are thinking about as you tackle this problem. This information will help us improve this material in order to help students learn better code reading skills. There are multiple ways to tackle this problem and multiple correct ways to go about reading the codebase. By showing the order in which you read the code we can better understand your thought process, akin to what may happen in a "code aloud" session.

In your comments identify the following:

- The order in which you came across each comment in the code path
- What the code between this comment and the next comment or end of method does (Do your best to explain, there is no one perfect explanation, just give us your thought process and best explanation)

Below is a brief video explaining the commenting process and how to submit this assignment:

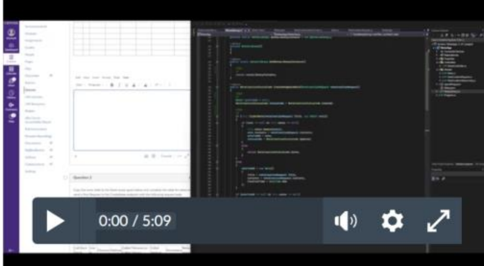


Figure 3.15. Submission instructions provided to students for comment block tracing problems.

Once students had gone through the required reading of the module and completed all attached exercises, they were presented with a final knowledge check designed to test them on every concept they learned. The Social Media App codebase was used for this exercise as it was designed to be denser with code reading pitfalls than the previous exercises and allowed for additional opportunities for students to exhibit anti-pattern behavior. The result was a codebase consisting of 18 files containing 734 lines of source code, resulting in the largest average lines of code per file of all codebases developed for the module (Figure 3.14). Students were expected to trace through the application starting at various endpoints in the controller with different provided inputs. Like the second exercise they were required to fill in any comment blocks they came across during their trace to showcase their path through the code and growing

understanding. After submitting their comments, students were also asked to provide the response from the endpoint they just examined given the same input they used in their trace.

3.8 Changes Made During Administration

Once the module was in the hands of the students, several issues became apparent and were addressed promptly. Minor issues such as questions not showing up properly did not impact the overall flow of the course and were addressed promptly with students able to re-examine any questions that had display issues. Two major changes needed to be made throughout the course's run. First, after 6 students had completed the final knowledge check, it became apparent that the question asking the response of the endpoint could be easily guessed by looking at an example response image that was posted for that endpoint earlier on the page with a different input as their outputs were the same. To remedy this, the response questions were revised to ask students for the specific line of the return call in the endpoint as well as an additional question for each endpoint that asked students about the various error responses the endpoints had.

The other major change to the module came about after some of the later submissions of the first and second tracing exercise came in. Some students had found a shortcut I overlooked where they would take advantage of the multiple submissions that I allowed for the tracing exercises. The students would submit a blank exercise which would unlock the answer key, then resubmit the exercise with the answer key responses. Only 4 students took this route, which did remove most ability to track their growth prior to the final knowledge check. One metric that remained intact was the length of time they spent reviewing the answer keys, which indicated some students who took this shortcut did spend a good bit time to learn from the responses in the answer key rather than just moving on after copying them. This shortcut was removed by

eliminating the multiple attempts students had after most students had already progressed past the tracing exercises.

Chapter 4. RESULTS & ANALYSIS

The code reading module was presented to students as part of their work for CSS 390 (Software Engineering Studio) Despite 19 students starting the module, only 17 completed the first tracing exercise and 12 completed the full module. 3 students of the 12 who completed the course were outliers from the rest of the class as they spent little time on the exercises or reviewing important material according to metrics collected by Canvas and/or turned in mostly empty or gibberish answers to exercises preventing the gathering of any meaningful data; as such these students, as well as the students who did not finish the module were excluded from evaluation of the module's effectiveness.

Student responses for the 2 tracing exercises and the final knowledge check were almost entirely hand graded apart from a few auto-graded problems. Responses to tracing questions started off as perfect scores and had points subtracted when different mistakes were made with the total subtracted dependent on the severity of the mistake. Given the tendency for mistakes in tracing to snowball into a cascade of future mistakes, full points were only deducted for initial mistakes; errors that came about because of a previous mistake only deducted a fourth of the points usually deducted for that kind of error. In the first tracing exercise that relied on trace tables, there was only one correct solution per problem, so points were deducted for each deviation from the correct answer. These deviations were placed in a rubric with point penalties assigned to each. Deviations for the first tracing exercise could include missing return value columns, abandoning the correct code path, missing a row on the correct code path. The second

tracing exercise and final knowledge checks shared the same rubric of deviations for their tracing problems as they were in the same format. Figure 4.1. contains the rubric used to help grade the final knowledge check and shows several items students were penalized for including: examining an incorrect overload, not examining a constructor, skipping over a comment in the trace, not stepping into a method, or making an incorrect statement in a comment block. If a student made one of these mistakes, but corrected themselves later in their answer the penalty was reduced to a fourth of its original amount. Smaller essay questions were typically graded as all, half, or nothing, where a student would get half credit for an answer that was partially correct, but no credit for an answer completely off the mark. When grading of the tracing exercises started, students were assigned an identifier (S1-S17) based on the order in which they completed the first tracing exercise, with S1 completing the exercise on October 24th and S17 completing it on November 26th. These identifiers can be seen in subsequent figures and include all students, even students who did not finish the course and are not present in the figures.

4.1 STUDENT EXERCISE SCORES

Overall, the students performed quite well upon completion of the module compared to the baseline established in the semi structured interviews. Students tended to perform better on tracing exercises as they progressed further in the course despite the difficulty of the problems also increasing. The average and median scores among students all had significant jumps between exercises, starting at an average of 47% and median score of 48% for the first tracing exercise growing to an average of 61% and median score of 65% in the second exercise followed by an average of 81% and median score of 76% in the final knowledge check. The growth of individual students can be seen in their percentage scores from each assignment in Figure 4.2.

Knowledge Check:

Do not dock points for not examining a constructor if the constructor is not used in the question i.e. post library constructor in the create new account endpoint problem. Do not take points off if they DO have it either.

Do not take points off if they neglect to include a constructor that they read in a previous problem ONLY if they had comments for it in a previous problem.

Do not take points off for not including repeated comments as long as they are clear they understand something is called a second time by their comments

Compounding errors (i.e. errors that result from a previous one like constructors not being included because the constructor that invokes them is not included) have only lose a fourth of the points as the original error

More severe errors like no tracing in a file override smaller errors like missing a constructor. Penalties do not stack.

General Tracing score guidelines:
- .1 Examined incorrect overload
- .025 Examined incorrect overload but corrected self
- .5 Missing tracing in critical file (Any file that has methods we trace through)
- .5 Missing tracing in critical file (AccountManager)
- .5 Missing tracing in critical file (AccountLibrary)
- .5 Missing tracing in critical file (PostLibrary)
- .5 Missing tracing in critical file (PostActionFactory)
- .1 Did not trace into LikeActions PerformPostAction method
- .1 Ignored SocialMediaController constructor
- .1 Ignored AccountManager constructor
- .1 Ignored AccountLibrary constructor
- .1 Ignored PostLibrary constructor
- .1 Ignored PostActionFactory constructor
- .1 Ignored (insert class) constructor
- .1 Incorrect assumption about code made in comment
- .1 Incorrect stack trace followed/left the correct code path in trace
- .1 Skipped over a comment in the trace
- .025 Skipped over a comment in the trace (but was repeated code)
- .2 Did not follow stack trace into method (compounding errors as a result -.1)
- .5 Did not fill in comments
- .05 correct comment, but left critical information out such as mentioning loops or other operations
- .025 correct comment but typo in number
- .05 Skipped over multiple comments but comments were included in prior answer

Figure 4.1.1. Rubric for problems in the final knowledge check

Q1:
Follow guidelines above.
Expected to read AccountManager constructor -> AccountLibrary constructor
From endpoint go from CreateNewAccount -> CreateAccount -> TryGetAccount -> FetchAllAccounts -> TryGetAccount -> AddNewAccount -> FetchAllAccounts -> AddNewAccount -> CreateAccount -> CreateNewAccount

Return value question: autograded

Comprehension question: full credit if they give an actual correct username based on the accounts we have in the file. Half credit if they get the correct mechanism (i.e. username that already exists) but do not provide an actual username that triggers this.
- 0.5 Describes correct method, but does not provide an actual username to trigger this

Q2:
Follow guidelines above.
Expected to read PostLibrary constructor -> PostActionFactory constructor -> ViewAction Constructor and LikeAction constructor
From endpoint go from CreatePost -> LoginAccount -> TryGetAccount -> FetchAllAccounts -> TryGetAccount -> LoginAccount -> CreatePost -> CreatePost-> CreatePost

Return value question: autograded

Comprehension question: full credit if they give the correct answer (line 66 in AccountManager, the LoginAccount method). Half credit if they ignored the "To the Controller" part of the instructions and give us line 72 of the controller (NotFound)
- 0.5 Provides the not found return value from the controller. We were looking for what line in what function caused us to pass back the DoesNotExist status TO the controller.
- 0.25 Provides a function that would cause a NotFound response in the controller, but not one caused by a missing username

Q3:
Follow guidelines above.
From endpoint go from LikePost -> PerformPostAction -> GetPostAction-> PerformPostAction -> PerformPostAction -> PerformPostAction -> LikePost

Return value question: autograded

Comprehension question: full credit if they give an actual correct id based on the posts we have in the file. Half credit if they get the correct mechanism (i.e. call out that if we give an id not in the system) but do not provide an actual id that triggers this.
- 0.5 Describes correct method, but does not provide an ID to trigger this

Figure 4.1.2. Rubric for problems in the final knowledge check

S4 scored a 0% for both exercise 1 and 2 and was included in the average and medians calculated above. If their first two exercise scores are omitted, average for the first exercise jumps to 53% and the median jumps to 50%; the second exercise's average jumps to 69% and its median goes up to 68%. These changes still show a continuous improvement in average and median through every problem. From Figure 4.2 we can see that all but 1 student had a higher knowledge check score than exercise 1. The exception to this was S10 who performed well on the tracing section of the knowledge check but gave some incorrect answers on short answer problems that gave the appearance that they did not read the question fully.

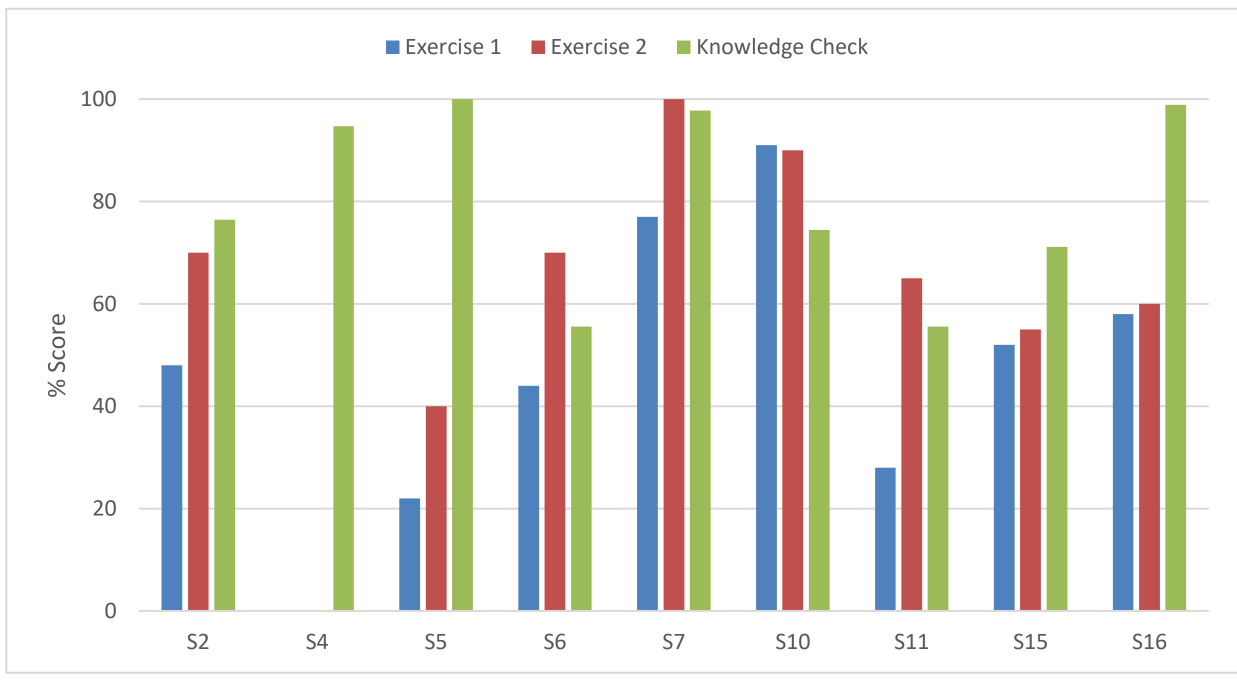


Figure 4.2. Percentage scores from graded student responses for the 2 tracing exercises and final knowledge check of the code reading module. Exercise 1 was 5 questions and 5 points; Exercise 2 was 2 questions & points and the final Knowledge Check was 9 questions & points.

4.2 Anti-Pattern Analysis

When grading the final knowledge check, code reading anti-patterns within student responses were evaluated and tracked. The specific anti-patterns that were identified in the original series of semi-structured interviews of students [1] were counted for each of the 3 tracing problems present in the knowledge check. With these results it was found that despite the knowledge check requiring students to trace through more code than the semi structured interviews, they exhibited fewer anti-patterns than students from the semi-structured interviews in 2 of the 3 anti-pattern categories (uncorrected misinterpretations of code and irrelevant files examined).

Category (b), the number or percentage of call stacks not followed, was an improvement over the semi-structured interviews in terms of percentage of call stacks not followed. During the semi-structured interviews, students were given one of two different code reading problems. The first problem required 2 call stack traces and the average number of traces missed was 1.2 (60%), the second problem required 3 call stack traces and the average number of traces missed was 1.4 (46%). 8 of the 9 students who completed the module had their percentage of call stacks not traced fall below both averages from the semi-structured interviews. As seen in Figure 4.4, 7 of the 9 students missed 25% or fewer of the call stack traces required for a complete reading of the code segments they were asked to examine, compared to 2 of the 10 semi-structured interview students.

The improvement shown between the students in the semi-structured interviews as seen in Figure 4.3 and the students who completed the code reading module shown in Figure 4.4

provides evidence that the module was successful in reducing the number of anti-patterns students engage in whilst reading code.

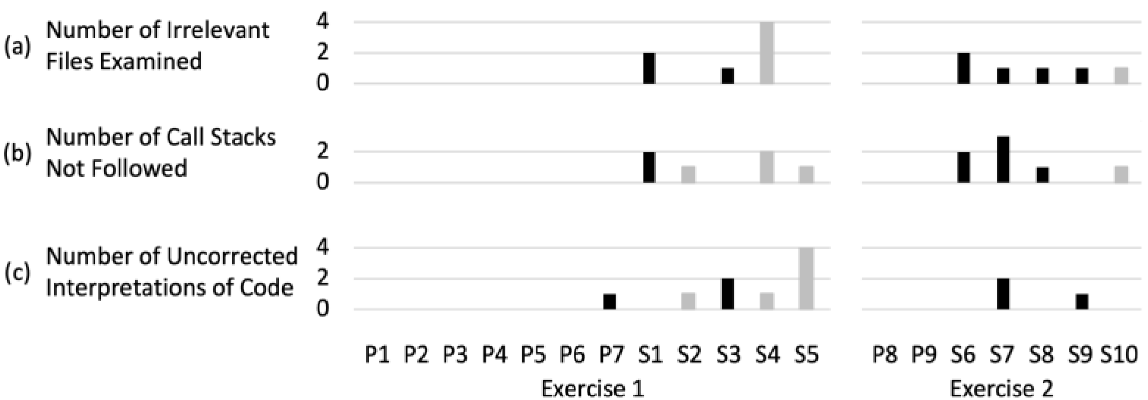


Figure 4.3. Quantity of code reading anti-patterns by students found in semi-structured interviews.

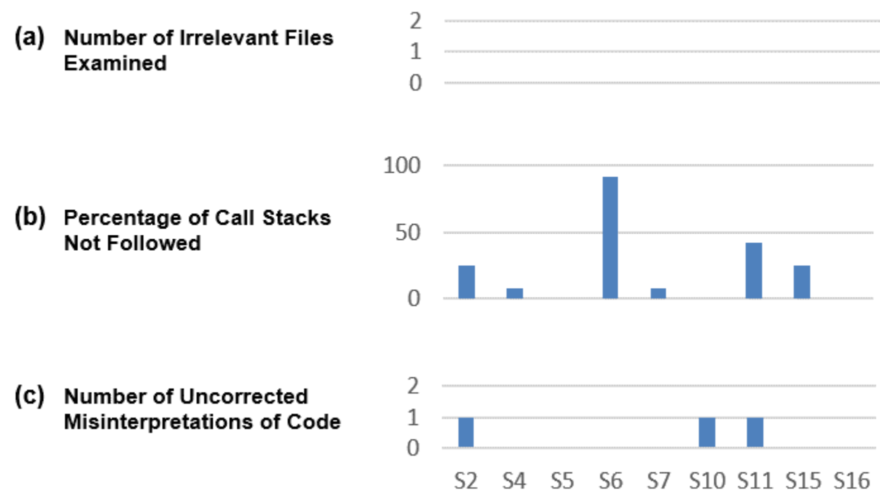


Figure 4.4. Quantity of code reading anti-patterns by students found in graded responses to the knowledge check at the end of the code reading module. Percentage of call stacks not followed is out of 12 possible call stacks to trace.

4.3 STUDENT FEEDBACK AND SENTIMENT

After students completed the module, they were asked to complete a small evaluation form with two Likert-scale questions that asked: “Do you think progressing through this module has improved your ability to read code that is unfamiliar to you?” and “Do you feel that the code reading strategies taught in these modules will be useful in guiding your approach to code reading?” Students had the opportunity to express strong disagreement up to strong agreement to both statements. 7 of the 9 students who completed the module completed this optional evaluation and all marked the same answer for both questions, leaving a single distribution for both questions which can be seen in Figure 4.5.

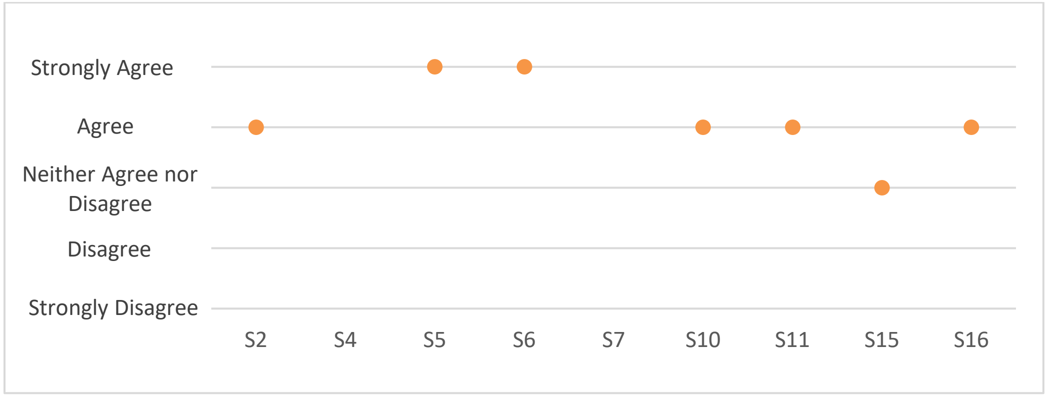


Figure 4.5. Student responses to optional feedback questions: “Do you think progressing through this module has improved your ability to read code that is unfamiliar to you?” and “Do you feel that the code reading strategies taught in these modules will be useful in guiding your approach to code reading?” Students S4 and S7 did not participate.

When asked at the end of the quarter, both professors running Software Engineering Studio expressed anecdotally that the students taking the class in the Autumn quarter tended to perform better in code reading challenges and required less assistance than in prior quarters following the administration of the code reading module. Both this anecdotal evidence from the professors

running the class and the feedback from the students themselves suggests the module was effective.

Chapter 5. FUTURE WORK & CONCLUSION

5.1 FUTURE WORK OF THE MODULE

With the conclusion of the module's administration and the grading work that followed, it became clear that the current design of the module would make it difficult to administer at a larger scale. For every student that took the module, it took roughly 90 minutes to grade and evaluate their responses to all exercises for anti-patterns. As such, injecting such a module into a class that already has an established curriculum and manual grading that already needs to be done for other activities would be challenging for the professors and teaching assistants running the class. The current form of the module is focused on a summative assessment to understand if the content and exercises helped students learn to read code. With the results of the module appearing to show that it did help students learn to read code, the next steps could also see it shift towards a focus on formative assessments only, which would also help it slot into a formal course and potentially provide more benefit to students.

To make the module fit in more seamlessly with existing courses, the module should be redesigned to not add extra strain on course-runners through excessive grading. As such, preliminary work has been done to convert the module to an auto-graded format while still attempting to preserve the workflow students go through to complete it. The three tracing exercises are the only components of the module that would require changes as they use manual grading. Each exercise had a specific workflow that students were expected to follow, be it filling in trace tables in a specific order or tracing through a specific path and filling in comments

along the way. Converting exercises to auto-grading while retaining the established benefit of the course would require maintaining these workflows for students.

Given the requirement of maintaining the workflows, I determined that the likely best approach is to still ask students to perform the work they would have done for the current exercises and to ask them questions afterward that can be auto-graded and that students should only know if they completed the work. To minimize the possibility of guesswork being used to circumvent the workflows, question banks would be used to randomize the questions given to students and each exercise would require students to achieve a specific score (e.g. 80% correct) before allowing them to advance to the next one. With this approach, multiple quiz attempts could be re-introduced without allowing for students to succeed. A wide variety of questions will also be used from matching to numerical answers, making it much harder to succeed without completing the associated work.

As for the questions themselves, students will be asked various questions regarding the output of programs given specific inputs, as well as the purpose of various methods. To maintain the benefits that the commenting exercise from the current module provided by identifying anti-patterns, these empty comment blocks will be repurposed into labels used to identify “code segments” which students will sequence to identify the order of execution for various questions. With this approach, it would still be possible to identify if students trace into an incorrect overload or did not trace further down the stack into a called method. The answer keys in the current modules would also need repurposing as they would no longer be answers to the exercises we give to students. Despite this, they could still act as examples of tracing through various codebases for the students to help them learn from after meeting the grade thresholds for a specific exercise.

With the revisions to the module not expected to impact the workflow of students completing it while also easing work on instructors it is hoped that the revisions will reduce the benefits of the module. Further testing would be required to establish this by having a new class run through this module and collect student feedback. Evaluating the number of attempts at exercises would also serve as a metric to evaluate student learning because if students are taking fewer attempts to meet thresholds as the module progresses, there is a higher likelihood that they have learned. True evaluation of the new module's effectiveness would likely take much longer without a manually graded assessment, but the potential positive impact by allowing the module to roll out to a larger audience ensures these results will be much more impactful than the relatively small-scale test that the current module was able to go through.

5.2 Conclusion

The performance of students throughout the code reading course module demonstrates that such a course can substantially improve students' ability to read and comprehend unfamiliar code. Integration of such modules into computer science courses early in the curriculum may serve to benefit students throughout their undergraduate studies and into their futures.

The steady improvement students tended to exhibit with each exercise suggests that both trace table visualization and direct commenting are both effective strategies at improving students' ability to complete code reading challenges. These results further support the claims made by Xie et al. [2] in their study that tracing sketches, like the trace table present in our module, are effective at improving student performance with code reading problems.

Student improvement may also be an indication that the modules were successful in lowering the cognitive load novice programmers may experience when presented with code reading

problems for complex code [3,4]. Future neurological studies conducted with students before and after such a module could help bolster the findings of studies like Peitek et al. [3] and Siegmund et al. [4] if student performance improved post-administration of the module and markers for cognitive load also decreased. Such an outcome would lend further support to the idea that increased cognitive load is a key driver for lowered performance in reading unfamiliar complex code for novice programmers.

Larger scale studies of the current code reading module would help to verify the results found by this small-scale test of the modules positive benefits. If this module was administered to a much larger beginner computer science class, the performance of those students in future courses throughout their curriculum could be compared to previous years for any sign of tangible benefit brought by the addition of the module. For large-scale adoption to take place, several changes to make the module easier to administer such as implementing an auto-grading feature would also likely need to be implemented first. These adjustments along with rolling out the module to more students remain key areas to tackle for future work on this project.

BIBLIOGRAPHY

[1] Woerner, M., Socha, D., & Kochanski, M. (2023). Code Reading: How Students and Professionals Differ. J. Comput. Sci. Coll., 39(1), 28–37. https://dl.acm.org/doi/10.5555/3636517.3636521

[2] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, August 14, 2017, Tacoma Washington USA. ACM, Tacoma Washington USA, 164–172. https://doi.org/10.1145/3105726.3106190

[3] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *Proceedings of the 43rd International Conference on Software Engineering* (*ICSE '21*), November 05, 2021, Madrid, Spain. IEEE Press, Madrid, Spain, 524–536. https://doi.org/10.1109/ICSE43902.2021.00056

[4] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. 2017. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (*ESEC/FSE 2017*), August 21, 2017, New York, NY, USA. Association for Computing Machinery, New York, NY, USA, 140–150. https://doi.org/10.1145/3106237.3106268

[5] William C. Adams. "Conducting Semi-Structured Interviews". In: Handbook of Practical Program Evaluation. John Wiley & Sons, Ltd, 2015, pp. 492–505. https://doi.org/10.1002/9781119171386.ch19

[6] Leelakrishna Yenigalla, Vinayak Sinha, Bonita Sharif, and Martha Crosby. 2016. How Novices Read Source Code in Introductory Courses on Programming: An Eye-Tracking Experiment. In *Foundations of Augmented Cognition: Neuroergonomics and Operational Neuroscience* (*Lecture Notes in Computer Science*), 2016, Cham. Springer International Publishing, Cham, 120–131. https://doi.org/10.1007/978-3-319-39952-2_13

[7] SeolHwa Lee, Andrew Matteson, Danial Hooshyar, SongHyun Kim, JaeBum Jung, GiChun Nam, and HeuiSeok Lim. 2016. Comparing Programming Language Comprehension between Novice and Expert Programmers Using EEG Analysis. In *2016 IEEE 16th International Conference on Bioinformatics and Bioengineering (BIBE)*, October 2016. 350–355. https://doi.org/10.1109/BIBE.2016.30

[8] Sarah Jessup, Sasha Willis, Gene Alarcon, and Michael Lee. 2021. Using Eye-Tracking Data to Compare Differences in Code Comprehension and Code Perceptions between Expert and Novice Programmers. In *Proceedings of the 54th Hawaii International Conference on System Sciences*, January 5, 2021. 114-123. https://doi.org/10.24251/HICSS.2021.013

[9] dotnet-bot. System.Reflection Namespace. Retrieved January 20, 2024 from https://learn.microsoft.com/en-us/dotnet/api/system.reflection?view=net-8.0

APPENDIX A: CCSC-NW 2023 PAPER

Code Reading: How Students and Professionals Differ[1]

Matthew Woerner, David Socha, Mark Kochanski
Computing & Software Systems
School of Science, Technology, Engineering & Math
University of Washington Bothell Bothell, WA 98011
{woerner,socha,markk}@uw.edu

Abstract

This paper reports on a series of semi-structured interviews and code reading exercises conducted with 10 undergraduate students and 11 professional software engineers in order to better understand how students and professionals differ in how they come to understand a codebase that is novel to them. The goal was to uncover distinctions to help us design teaching activities to help students know how to read source code whose call stack spans multiple methods and files. Students had a more difficult time correctly reading source code than professionals and tended to lack a clear process for tackling code reading exercises. Professionals tended to tackle the code reading exercises more methodically, checked their assumptions, and avoided spending time reading code irrelevant to the exercise presented to them.

Chapter 1. INTRODUCTION

This paper reports on results of a qualitative study exploring the following research questions: How do undergraduate students and professional software developers differ in how they come to understand a codebase that is novel to them? This question came to our attention while teaching a new course, Software Engineering Studio, designed to reduce the time and effort students take to become effective in their first software engineering internship or job. The course provides an

[1] Copyright ©2023 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

on-campus internship-like experience for students to learn key aspects of working in teams in an organization evolving more complex codebases than they have encountered before. We taught this course to 90 undergraduate students for seven consecutive quarters from autumn 2021 through spring 2023.

In this course, students work in teams as they progress through a sequence of four onboarding badges that progressively a) increase in size and complexity, b) provide fewer instructions, c) move from individual to team-based work, and
d) require students and teams to take on more ownership of the work. After doing the core work of a badge, each student creates a portfolio demonstrating their badge work, and then schedules a 30-minute "badge challenge" Zoom meeting with one or both instructors. The badge challenge is an interactive semi-structured interview [1] in which the instructors' notes from their review of the portfolio form starting points for a conversation with the student about their work, processes, mindsets, assumptions, knowledge, and so on. Each conversation is unique as instructors meet the students where they are at, probing to understand precisely what they know, their assumptions, and their current habits—all in the service of uncovering the particular guidance or information suitable for that student. To help stay grounded in facts, students share their screens showing the exact code, tests, documents, or other work they had done. These badge challenges sometimes lead into unexpected territory and sometimes reveal dynamics about our students and curriculum that were opaque to us instructors despite our decades of teaching in our department. Over the course of several hundred challenges, some particularly interesting issues about student preparation have emerged and been the seeds for further research, such as this paper.

This paper emerged from the second badge students do, whose core work required students to modify a C# codebase of five files containing 221 lines of production code, two files of test code, and one JavaScript Object Notation (JSON) data file. When asked during the badge challenge to explain what a particular set of lines of code did, only a handful of the 90 students followed the call stack to see and comprehend what the lines of source code in the called methods actually did. The rest simply made assumptions about what the called code did, despite our repeated questions about their assumptions. They only navigated to the called code when we asked them to, and almost all needed us to tell them how to do that. Looking at the actual lines of code implementing a called method appeared to be a novel idea to all but a handful of the students. We became curious about how to help our students learn how to read and understand code, and decided to see how professionals would perform in this same exercise.

The remainder of this paper is organized as follows. Section 2 briefly reviews some of the related literature. Section 3 describes our methodology for gathering and analyzing data from 21 research participants. Section 4 presents and analyzes the results. Section 5 summarizes our findings and mentions future work.

Chapter 2. BACKGROUND LITERATURE

Reading and understanding each statement of software code that one has not written is an essential skill for developers. Studies indicate this code reading can take anywhere from 35-70 percent of a developer's work time. One study of 78 professionals across 7 projects over 3,148 working hours found that "on average program comprehension takes up 58 percent of developers' time" [9]. Despite code reading's importance, there appears to be a relative lack of resources dedicated to helping growing developers practice their code reading skills [11]. This lack of formal education often places new graduates into a situation where they must quickly try

to adapt to the code reading abilities of their colleagues in the high-pressure environment of a workplace. A large-scale study at Microsoft found that there is a gap in tool support for reading code, which puts more pressure on the engineer [8].

Previous studies of code reading revealed that professionals are more proficient than students when it comes to identifying and retaining relevant information needed to solve code reading problems [4, 6]. Another study used functional magnetic resonance imaging (fMRI) to show that the size of the codebase being examined by a programmer increased cognitive load, perhaps due to the increased number of identifiers and symbols needed to be tracked for comprehension [5]. These studies suggest that professionals can process code more efficiently than students and at a lower cognitive load [6]. Some hypothesize that more experienced programmers can focus on relevant information needed to read the code in front of them in order to accomplish these tasks with a lower level of cognitive load [6, 5]. Other studies suggest techniques like sketching (the act of creating a visualization of a programming state) are effective at improving novice programmer's ability to complete code reading problems by helping to manage cognitive load [2].

Chapter 3. METHODOLOGY

Our study used semi-structured interviews and code reading exercises to gather data from 10 undergraduate students and 11 professional software engineers (see Table 1) during spring of 2023. The first author did all interviews, using two weekly meetings with the other authors to discuss and adapt the research. Student participants were recruited from the University of Maryland: College Park and the University of Washington through social media outreach and the interviewer's personal networks. Nine of the ten students had prior software engineering internship experience, and most students were in their third year of their undergraduate program.

The professionals were recruited via the interviewer's personal network, and came from a variety of large software companies such as Amazon and Microsoft. All but one professional (P2) had been working professionally as software engineers for at least 3 years, and all but two (P3, P7) had a formal computer science education. All interviews and code reading exercises were conducted remotely via Zoom, and video recorded for later analysis. Our university's Human Subjects Division determined that this research qualified for exempt status, and thus did not need to obtain Institutional Review Board (IRB) approval.

Job Title[1]	S	S	S	S	SS	SS	SS	S2	SS						S2	SS					
Professional Experience[2]	3	4	3	1	24	8	15	6	6	1/2	1/2	1/2	0	1/2	8	20	1/4	1/4	1/4	3/4	1/2
University Attending[3]										M	M	M	M	M			M	M	M	UW	M
Year in University										3	3	3	2	3			2	2	3	4	4
CS/CE degree[4]	G	G	G	G		G	G	G		E	E	E	E	E	G	G	E	E	E	E	E
CS internship	✓	✓	✓	✓			✓	✓		✓	✓	✓		✓	✓		✓	✓	✓	✓	✓
Identifier			P1	P2	P3	P4	P5	P6	P7	S1	S2	S3	S4	S5	P8	P9	S6	S7	S8	S9	S10
Exercise Given			E1	E1	E1	E1	E1	E1	E1	E1	E1	E1	E1	E1	E2	E2	E2	E2	E2	E2	E2

Figure 1: Interviewee demographics. [1]Software Engineer (S); Software Engineer II (S2); Senior Software Engineer (SS). [2]Years. [3]University of Maryland (M); University of Washington (UW). [4]Has graduated from (G) or is enrolled in (E) a computer science or computer engineering degree program.

3.1 Semi-Structured Interviews

The semi-structured interview [1] consisted of 14 open-ended questions such as "Can you tell me about how you come to understand code you have never seen before?" and "What do you find makes it easier or more difficult to onboard to a new codebase?" These questions were designed to try and understand how the participant thought about and did code reading, what made the process easier or more challenging, how participants viewed their own approaches to code reading in different contexts (e.g. debugging an issue, onboarding to a new codebase), how those

approaches changed over time, and what caused those changes to occur. As usual for semi-structured interviews, the 14 questions served as a set of example questions to guide the interview. The particular questions asked was decided at the time of the interview, and questions were rearranged, re-worded, or not used depending on the direction of the conversation and thus varied for each interview.

3.2 Code Reading Exercises

After doing two interviews, we added a code reading exercise (E1) to the end of each interview. The exercise used the code repository from the Software Engineering Studio course. Participants were tasked with tracing through a particular unit test to identify what was causing the test to fail, and were allowed to search the internet during their time, and to ask clarifying questions. Participants could solve this problem by following the execution of the unit test's code from the test data file to the class that the test was calling. From there participants could examine the class' initialization in the class constructor, as well as the method within it that was subsequently called, to see that the code read the data file into a global variable of a list of quotes and then searched the list for the first entry that matched the GUID (globally unique ID) provided in the test. All a participant would then need to do in order to find the issue would be to examine the data file itself and see that the information in the data file tied to the GUID used in the test did not match the result that the test was expecting, causing the unit test to fail. If a participant following the path of code calls they would navigate down two separate call stacks before encountering the problem.

To help us understand what they were thinking, participants were asked to talk aloud through their thought process. When participants stopped talking, they were prompted with questions about their thought process. If a participant was not making progress after a significant amount of

time, the interviewer provided guiding hints to steer them towards the answer. Participants that appeared unable to solve the exercise and thus were given specific guidance towards the answer had their times logged, but were marked as not completing the exercise unassisted. The interviewer also used the interviewee's cursor movement as a supplementary factor to give hints about the participant's thought process. The interviewer took notes on all participant spoken thoughts and behaviors deemed relevant to the exercise. Most notes ended up focusing on similarities to and deviations from a direct route to the faulty code.

After the 14th interview, we redesigned the code reading exercise to see if the differences between students and professionals revealed from E1 became more extreme with a more difficult exercise. Exercise 2 (E2) added another layer of complexity by asking participants to examine a series of nearly identical passing tests and to determine how to make exactly one test fail by adding lines to a file in the codebase. The starting point for E2 was an additional layer up the call stack from the starting point for E1, and following the code calls would require navigating three separate call stacks, which also added to the difficulty of the exercise. A correct reading of the code would have participants tracing down to the area where the test data file is read and realizing that the code searches for the first element in the data file with a matching GUID. The participant could suggest adding to the start of the data file a new element with the same GUID as the test they wanted to fail but with different data; this would cause data to be populated from the new entry, causing the test to fail while all others continue to pass.

Chapter 4. INSIGHTS FROM THE INTERVIEWS

The semi-structured interviews revealed several differences and some similarities between the students and professionals. Both spoke of the importance of documentation and how it helps improve their ability to read code. However, students and professionals tended to place greater

emphasis on different types of documentation as being the most helpful. Students tended to emphasize the importance of "in-method comments" which explain what the next few lines in a method are designed to do, similar to pseudocode. Professionals more often used higher level documentation such as official library/package documentation, one-pager feature design documents, or Swagger pages which were used to provide context for their reading of lower level code. Professionals also emphasized the importance of method headers and good method and variable naming as the key low-level components to their understanding of code.

Students and professionals described different approaches to debugging an issue in unfamiliar code. Professionals tended to discuss the importance of narrowing down the bug's location by examining logs and metrics, and would only step through the code if the bug cannot be found with logs alone. Students highlighted a similar high-level approach of narrowing down the search area, but tended to rely on inserting print statements or making minor code changes to see their effect on the output, rather than examining error logs. Roughly half of the students interviewed made no mention of stepping through code with a debugger, whereas all professionals did so.

Chapter 5. RESULTS FROM CODE READING EXERCISES

Figure 2 charts (a)-(c) show the incidences of three particularly interesting anti-patterns that we noticed during the code reading exercises: 1) examining irrelevant files, sometimes apparently randomly choosing different source, data, and configuration files; 2) not following the call stack through files; and 3) making uncorrected misinterpretations of the code. The three anti-patterns were identified during the first few interviews, discussed by all authors who also noted that the same anti-patterns were common during student challenge meetings, and then quantified by the interviewer reviewing the interview videos. Chart (d) shows the total number of anti-patterns

each participant exhibited. All but one of the 37 anti-patterns instances were from students, and each student exhibited between 2 and 7 instances.

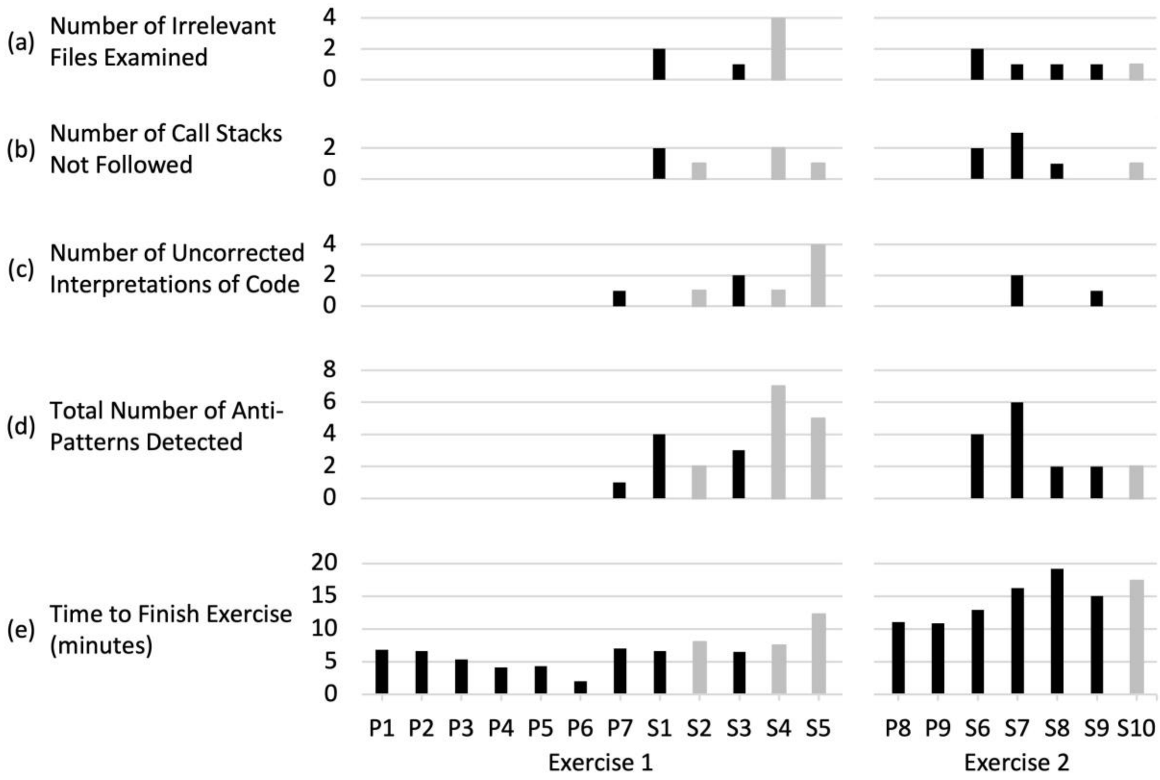


Figure 2: Differences exhibited between students ("S") and professionals ("P") during code reading exercises as compared to the ideal solution. Gray bars denote participants who did not finish.

Figure 2 chart (e) shows the time taken to solve the code reading exercises. For the simpler E1, students on average took longer than the professionals; only two students (S1, S3) took a few seconds less than the slowest professionals (P1, P2). For the more complex E2, all five students took longer than the two professionals. As predicted the more complex E2 took longer to solve.

For E1, participants that appeared more confident in their approach to the exercise tended to speak up about their thought process unprompted, as opposed to participants who showed signs of having difficulty with the exercise. All E1 professionals solved the exercise with minimal deviations from the most direct path to the solution, including four who had no C# experience.

Students had varying degrees of difficulty, and all requested assistance. Four out of the five students did not demonstrate a good ability to trace calls through multiple files; many scrolled though irrelevant files outside of the code path looking for a method, or becoming lost and needed assistance finding a method in a different file. This issue could be for a number of reasons, but we hypothesize that the likeliest causes are lack of understanding of how the code was structured in files, and a lack of knowledge of code navigation tools.

Once students found the relevant method the test was calling, four out of the five students assumed that the global variable holding the list of quotes had nothing to do with the error and ignored it until given a suggestion to look deeper into it. One professional also assumed it was irrelevant to the exercise at hand, but self-corrected within 2 minutes without prompts or hints.

Students demonstrated difficulty in finding relevant information within a file they were examining if that information was located in another method or in the constructor. Part of this could be due to the use of a singleton design pattern to instantiate the class, but it still demonstrates a difference and an area where education could be useful.

All five E2 students abandoned the code path to explore files irrelevant to the exercise. E2 students were less likely to misinterpret the code than were E1 students, despite most of the code being largely the same. This could be due to the fact that E2 students more frequently searched the internet for information or asked questions than E1 students, with all but one E2 student seeking additional information on critical library functions they were unfamiliar with, even though the function in question was a part of both exercises. On average, the E2 professionals took 2.1 times longer than E1 professionals, but never got stuck and always kept track of relevant information throughout the process.

Professionals appeared more confident in solving code reading exercises than students and appeared to use a clear methodology. Professionals rarely had to be prompted to speak what they were thinking, while students were far more likely to have long periods of silence (e.g., 30 seconds) during which the interviewer had to prompt them to think aloud. Furthermore, while we have not analyzed this in detail, professionals tended to use key terms indicating that they had a conceptual model of how to read code. Professionals stuck to the flow of code execution and checked any assumptions made for correctness before proceeding. Professionals that were unsure of a language feature or outside function always asked questions or looked up documentation rather than assuming what the thing did. Many of the deviations from this manner of thinking by students appeared to be a result of lack of knowledge, and to students appearing to not use a defined methodology to help guide them through the process.

Chapter 6. CONCLUSION AND FUTURE WORK

This paper reports on the results of a qualitative inquiry into differences between how undergraduate students and professional software developers come to understand a codebase that is novel to them. While our data come from only 21 interviews (10 students, 11 professionals), Figure 2 demonstrates noticeable differences between how students and professionals solved the two code reading exercises. Our hypothesis is that key reasons professionals performed better include that they checked their assumptions about what code did, and understood how to trace code through the execution stack.

Our next step is to design and test a set of teaching activities, sample codebases, and tools to make these qualities salient to students and to give them practice avoiding the anti-patterns shown in Figure 2. Unlike much of the prior work on teaching code reading that mainly focuses on smaller snippets of code [10, 3, 7], we aim to create teaching activities to help students read

code in larger codebases like those they will encounter in the workplace. If appropriate, we may include exercises that rely on a form of trace sketching [2] in order to help students understand specific patterns in code.

We believe that reading code is a skill that can be effectively taught to undergraduates by providing them with key simple models of code reading techniques based upon what professionals do, exercises in which to practice those techniques, and a summative assessment exercise to see how well the models and exercises worked to improve students coding abilities. We continue to mine our interview information to inform the design of this and other work which we look forward to reporting on in the future.

REFERENCES

[1] William C. Adams. "Conducting Semi-Structured Interviews". In: *Handbook of Practical Program Evaluation*. John Wiley & Sons, Ltd, 2015, pp. 492–505.

[2] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. "Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw". In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ICER '17. New York, NY, USA: Association for Computing Machinery, Aug. 2017, pp. 164–172.

[3] Cruz Izu and Claudio Mirolo. "Comparing Small Programs for Equivalence: A Code Comprehension Task for Novice Programmers". In: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. Trondheim Norway: ACM, June 2020, pp. 466–472.

[4] Sarah Jessup, Sasha M. Willis, Gene Alarcon, and Michael Lee. "Using Eye-Tracking Data to Compare Differences in Code Comprehension and Code Perceptions between Expert and Novice Programmers". In: *Proceedings of the 54th Hawaii International Conference on System Sciences*. 2021.

[5] Norman Peitek, Sven Apel, Chris Parnin, Andre Brechmann, and Janet Siegmund. "Program Comprehension and Code Complexity Metrics: An fMRI Study". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, ES: IEEE, May 2021, pp. 524–536.

[6] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. "Measuring neural

efficiency of program comprehension”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Paderborn Germany: ACM, Aug. 2017, pp. 140–150.

[7] Leigh Ann Sudol-DeLyser, Mark Stehlik, and Sharon Carver. “Code comprehension problems as learning events”. In: *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. Haifa Israel: ACM, July 2012, pp. 81–86.

[8] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. “How do software engineers understand code changes? an exploratory study in industry”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE ’12. New York, NY, USA: Association for Computing Machinery, Nov. 2012, pp. 1–11.

[9] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. “Measuring Program Comprehension: A Large-Scale Field Study with Professionals”. In: *IEEE Transactions on Software Engineering* 44.10 (Oct. 2018), pp. 951–976.

[10] Benjamin Xie, Greg L. Nelson, and Amy J. Ko. “An Explicit Strategy to Scaffold Novice Program Tracing”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. Baltimore Maryland USA: ACM, Feb. 2018, pp. 344–349.

[11] Tammy Xu. *Reading Code Is an Important Skill. Here’s Why. | Built In*. Nov. 2021