Debug Tutor: Automated Deliberate Debugging Practice for Undergraduate Programmers

by

Gabrielle E. Ecanow

B.S., Computer Science and Engineering
Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by:	Gabrielle E. Ecanow Department of Electrical Engineering and Computer Science May 12, 2023
Certified by:	Robert C. Miller Distinguished Professor of Computer Science Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Debug Tutor: Automated Deliberate Debugging Practice for Undergraduate Programmers

by

Gabrielle E. Ecanow

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Novice programmers struggle with debugging. Despite a rich literature of research on the effectiveness of teaching debugging, debugging is often not taught systematically in computer science curricula. This thesis presents the Debug Tutor, an automated debugging tutor for explicit debugging practice at the college level. The Debug Tutor's suite of exercises drill particular microskills essential for competent debugging, and it offers automated expert hints and feedback by observing students' debugging actions in real time. The Debug Tutor was incorporated into MIT's undergraduate Software Construction course (6.102, formerly 6.031) in the Spring 2023 term. The Debug Tutor's effectiveness at teaching important low-level debugging skills was investigated by analyzing exercise completion statistics and subsequent debugging-related quiz scores of the over 500 MIT students enrolled in the undergraduate course. The analysis revealed that completing Debug Tutor exercises was positively correlated with performance on debugger-related exam questions, regardless of students' prior comfort levels with using a debugger. Furthermore, the software design of the Debug Tutor as a tutoring architecture with event tracking support was shown to be robust in capturing specific student actions to compare against exercise event patterns, flexible enough to handle a wide range of unexpected action sequences and on-the-fly updates, and extensible to domains other than the use of the debugger.

Thesis Supervisor: Robert C. Miller
Title: Distinguished Professor of Computer Science

Acknowledgments

I would, first and foremost, like to thank the staff and students of the Spring 2023 semester of 6.102. To the staff, your patience and feedback helped shaped the Debug Tutor into what it is now. To the students, I only hope the Debug Tutor was a positive learning experience overall, despite the inevitable hiccups of a version 1 release!

I would especially like to thank Rob Miller, my thesis advisor and the incredible solo-flying professor of 6.102 (at least for Spring 2023—Max, you were missed!). Thank you for your guidance, quick feedback, and encouragement, and for letting me release the Debug Tutor to the students of 6.102. I am inspired by your dedication to 6.102, a course that has had a profound impact on my experience at MIT and, no doubt, on my software engineering career to come.

To Mario, the other half of the Praxis duo. Thank you for chatting with me every week about automated tutoring for the past year.

To Milka, my co-head 6.102 TA. Thank you for being my sounding board, for patiently listening to all of my rants, for being an early and continuous Debug Tutor tester, and for all of your invaluable feedback. Thank you for generally keeping me sane this past year.

I would like to thank my prior teachers: Herbert Lichtman, for introducing me to software engineering, and Tom Bredemeier, for encouraging me to teach software engineering. You showed me early on how great teachers inspire students to want to learn.

Last but not least, I would like to thank my family. To my siblings, Eli and Naomi (and Jeff!), and to my parents, Marci and Jacob: Thank you for being my biggest supporters and cheerleaders. I could not have gotten through these last five years at MIT without you.

Contents

1 Introduction 15

2 Related Work 19

2.1 Understanding Bugs 19

2.2 Understanding Debugging 20

2.3 The Debugging Process 20

2.4 On Teaching Debugging 22

2.5 Existing Tools for Teaching Debugging 24

2.6 Applying Deliberate Practice to Debugging 26

3 Design 29

3.1 A Debug Concept Map 29

3.2 Design of Exercises 34

3.3 User Interface 37

3.3.1 Exercises 38

3.4 It's Time for an Update 45

4 Implementation 47

4.1 History Event Patterns 49

4.1.1 Authoring Exercises with History Events 49

4.1.2 Adopting Event Patterns in the Tutor 55

4.2 The Server 57

4.2.1 Concrete Exercises 57

4.2.2 A Concrete Concept Map 62

4.2.3 Uploading to the Server 64

4.2.4 Answer Checking 2.0 64

4.3 The Web Application 65

4.4 The Extension 70

4.4.1 Event Tracking API 70

4.4.2 extension.ts 74

4.4.3 Concrete Trackers 77

4.5 Version 1 Complete 79

5 Deployment **81**

5.1 Debugging Reading 81

5.2 Staff Testing 82

5.3 Official Release 83

5.3.1 Grading 84

6 Evaluation **85**

6.1 Student Data 85

6.1.1 Self-Reported Data 85

6.1.2 Problem Reports 88

6.2 Tutor Exercise Completion 89

6.2.1 Time Spent & Attempts Per Exercise 93

6.3 Course Metrics 97

6.3.1 Class 9 Nanoquiz Results 97

6.3.2 Quiz 1 Results 101

6.3.3 Impact of the Debug Tutor 104

6.4 Summary & Main Takeaways 112

7 Conclusion **115**

7.1 Discussion 115

7.2 Future Work 116

7.2.1 Extension Updates 116

7.2.2 Concept Map Updates 118

7.2.3 Realtime Hints 118

7.2.4 A Hypothetical Case Study: Extending to Git 119

List of Figures

1-1 Full VS Code window in student view, with level 0 completed, level 1 unlocked, and levels 2-5 locked 17

1-2 A debugger exercise with a hint displayed after an incorrect attempt . 18

3-1 Debug Tutor Concept Map 30

3-2 Full VS Code window in student view, with level 0 completed, level 1 unlocked, and levels 2-5 locked 37

3-3 Example exercise when a concept group is first opened 38

3-4 Basic Workflow 40

3-5 Setup Required Workflow 41

3-5 Setup Required Workflow (cont.) 42

3-6 Start Over Required Workflow 43

3-7 On A Correct Submission 44

4-1 The Debug Tutor (Praxis) Architecture. 47

4-2 Regular Object Matching Algorithm 54

4-3 The Praxis Tutor Webapp Event Loop 69

4-4 The Editor Namespace in `event-tracking-api.ts` 72

4-5 The Tutor Tracker interface in `event-tracking-api.ts` 73

5-1 Clip of Reading 9 *Scientific Method* Subsection 82

6-1 Student Data 86

6-2 Prior Debugger Experience 87

6-3 Prior Debugger Skill Experience 87

6-4 Prior Debugger Comfort Level 88

6-5 Problem Report Analysis 88

6-6 Per-Exercise Pass Rate 90

6-8 Matrix-mix Approaches 91

6-7 Matrix-mix 92

6-9 First-time passer Matrix-mix students split by approach and binary prior debugger comfort level 92

6-10 Time Spent Per Exercise 94

6-11 Attempts Per Exercise 95

6-12 Student Data (without the top 0.01 quantile) 96

6-13 Clip of Class 9 Nanoquiz and Solutions 98

6-14 NQ Grade, broken down by concept 99

6-15 Nanoquiz average grade by concepts passed 100

6-16 Quiz 1 sub-question 4.2 and sample answers 101

6-17 Quiz q4.2 grade and answer distribution 101

6-18 Quiz q4.2 average by highest completed debugger level 102

6-19 Effect of Particular Exercise on Quiz Answer 103

6-20 Prior experience on combined quiz grade 104

6-21 Effect of Prior Comfort and Highest Level Completed on Combined Scores 106

6-22 Effect of Particular Exercise on Quiz Answer 108

6-23 Prior experience with breakpoints and Backgammon-breakpoint-continue-navigation on quiz answer 110

6-24 Prior experience with single stepping and Backgammon-step-in-out-navigation on quiz answer 111

List of Tables

3.1 Debug Tutor Concept Map 31

4.1 Example Debugger Event History Patterns 52

4.2 Concrete Pattern Streams & Events 53

4.3 Webapp-Server Routes 66

4.4 Webapp-Host Plugin API 67

4.5 Event Tracking API's `HostIDE` Exports 71

Chapter 1

Introduction

Debugging, the art of finding and fixing bugs in code, is an integral part of the software engineering process. Yet novice programmers struggle with debugging. More specifically, they lack the systematic approach to debugging that expert programmers exhibit. While debugging is a skill closely related to programming, research finds that the skills necessary for competent debugging are separate from those found in competent programming [1][17]. And, although research on teaching debugging finds that student programmers benefit from explicit debugging instruction, most high school and college curricula lack dedicated time towards debugging instruction. There is much work left to be done in crafting an appropriate method of debugging instruction for undergraduate computer science curricula.

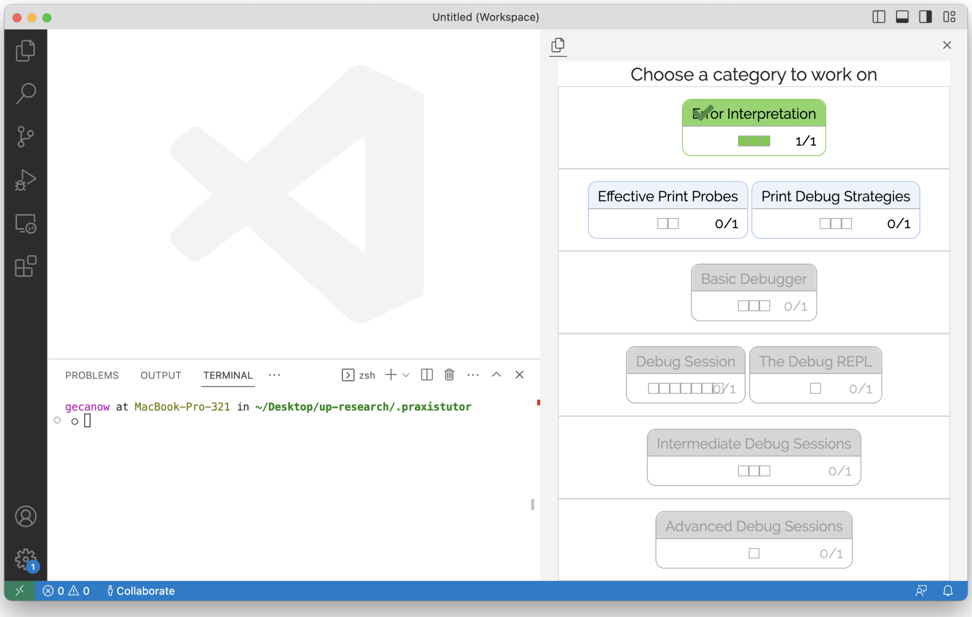
Like building competency in both the syntax and semantics of programming, building competency in debugging requires learning a wide range of skills across abstraction levels. A debugging curriculum should include instruction in high-level expert approaches, such as defensive programming and the scientific method for debugging, as well as practice in low-level skills like test-case reduction, print debugging, and use of the debugger, not to mention employing these together to build intuition in finding and fixing bug. While many undergraduate courses address at least some high-level debugging practices, most lack any explicit instruction in low-level tooling, nor do they include any sustained, feedback-oriented practice in debugging. The Debug Tutor, therefore, aims to offer such practice in low-level debugging microskills.

One generally effective method of instruction is deliberate practice [7]. Deliberate practice describes how to systematically master a new skill by performing intentional exercises with immediate and expert feedback, repeated regularly and for an extended period of time. Deliberate practice is the basis behind the MIT CSAIL Usable Programming Group's Praxis Tutor, an educational framework for automatically applying deliberate practice to any skill whose practice and feedback can be mechanized. For example, the Praxis framework drives the TypeScript Tutor, a Visual Studio Code (VS Code) plugin currently used in MIT's undergraduate 6.102 Software Construction course for learning the syntax of the TypeScript language.

The Debug Tutor is an extension of the Praxis framework applied to debugging. The Debug Tutor hosts a suite of debugging exercises that target concepts necessary for mastering low-level debugging skills, including interpretation of stack traces, methodical print debugging, and the use of the debugger. These concepts are organized into related concept groups, which are split into levels, each of which cannot be accessed until the previous level is completed. The Debug Tutor is accessed as a plugin to any supported code editor, such as VS Code. In the Debug Tutor, a student must first complete exercises in error interpretation. Next, the student must complete exercises in effective use of print statements and in print debug strategies. Finally, the student must work through a series of concept groups exercising use of the debugger. Fig. 1-1 displays the Debug Tutor with all concept groups visible in VS Code.

Within each concept group, students must complete a certain number of concrete exercises, each drilling one or more abstract concepts. For example, an exercise drilling the use of certain single step actions in the debugger might ask the student to watch as a breakpoint is automatically set in a code file. Then, the exercise might ask the student to begin a debug session and use some sequence of debugger actions to change and observe how the debug session state updates based on the actions taken.

The Debug Tutor provides expert feedback and hints based on observing debugging actions in real time. Exercise authors define each exercise's solution pattern as an ordered event list, where each event includes an action and its data, such as setting a breakpoint on a certain line. Exercise authors can use the same pattern format to

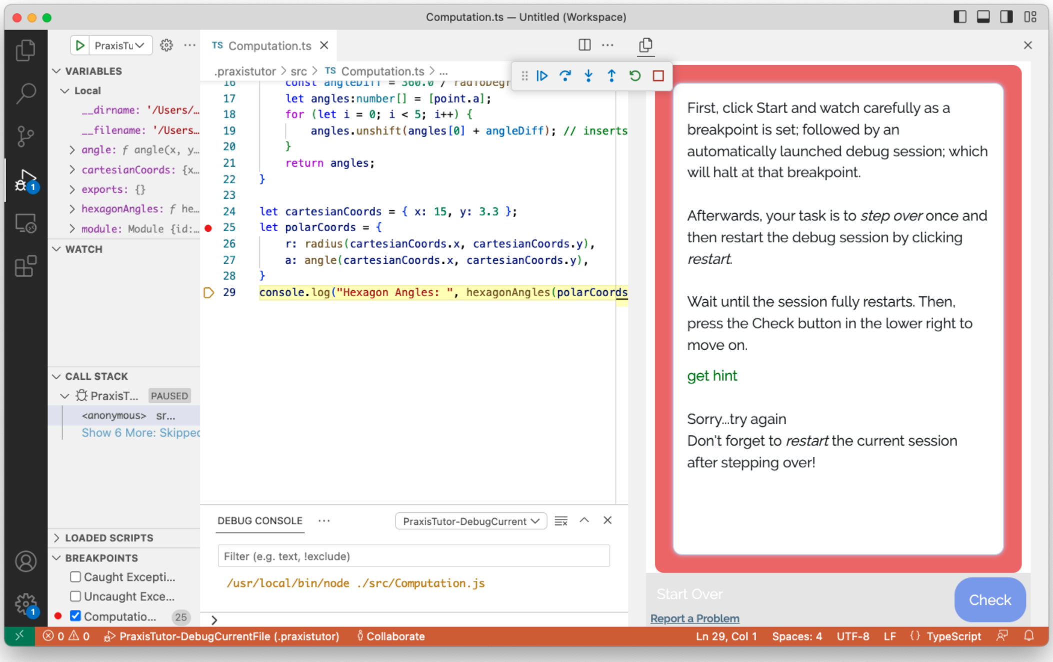


```
gecanow at MacBook-Pro-321 in ~/Desktop/up-research/.praxistutor
○ □
```

Figure 1-1: Full VS Code window in student view, with level 0 completed, level 1 unlocked, and levels 2-5 locked

define hints triggered by an incorrect action sequence and warnings triggered by a correct, but non-expert, sequence. As the student works through an exercise, the Debug Tutor observes and logs relevant actions in real time. To check a submission, the Debug Tutor uses a custom pattern matcher to compare the student event sequence and the solution event pattern. Fig. 1-2 displays an example exercise drilling the use of the step over action, with a hint displayed after an incorrect attempt was made.

This thesis evaluates the effectiveness of the Debug Tutor as used in MIT's 6.102 Software Construction Spring 2023 term, where it attempted to improve both the skill and self-efficacy in debugging for the over 500 students enrolled. Each Debug Tutor level was recommended to students at appropriate points throughout a reading assignment dedicated to teaching high-level debugging approaches, therefore blending the abstract high-level and hands-on low-level instruction into a single assignment. After completing the reading, students were also asked about their prior experience using a debugger. During the following lecture dedicated to debugging, students were evaluated on their knowledge of single stepping in the debugger on a nanoquiz. Students



```
__dirname: '/Users/…
__filename: '/Users…
> angle: f angle(x, y…
> cartesianCoords: {x…
> exports: {}
> hexagonAngles: f he…
> module: Module {id:…
```

```
    const angleDiff = 360.0 / radToDegr
    let angles:number[] = [point.a];
    for (let i = 0; i < 5; i++) {
        angles.unshift(angles[0] + angleDiff); // inserts
    }
    return angles;
}

let cartesianCoords = { x: 15, y: 3.3 };
let polarCoords = {
    r: radius(cartesianCoords.x, cartesianCoords.y),
    a: angle(cartesianCoords.x, cartesianCoords.y),
}
console.log("Hexagon Angles: ", hexagonAngles(polarCoords
```

□ Caught Excepti...
□ Uncaught Exce...
● ☑ Computatio... 25

```
/usr/local/bin/node ./src/Computation.js
```

First, click Start and watch carefully as a breakpoint is set; followed by an automatically launched debug session; which will halt at that breakpoint.

Afterwards, your task is to *step over* once and then restart the debug session by clicking *restart*.

Wait until the session fully restarts. Then, press the Check button in the lower right to move on.

Sorry...try again
Don't forget to *restart* the current session after stepping over!

Figure 1-2: A debugger exercise with a hint displayed after an incorrect attempt

were re-evaluated on similar content two weeks later on the first quiz. Comparing the quiz scores of the student population split by prior debugger comfort levels and number of Debug Tutor concept groups completed revealed that, regardless of prior comfort, completing more concept groups was correlated with higher quiz scores.

Furthermore, most Debug Tutor exercises had a pass rate well above 80%, but only a small minority passed each exercise on the first try. Therefore, the design of the event pattern language and event tracking API introduced into the Praxis architecture was robust enough to ensure that concepts were sufficiently practiced, yet flexible enough to account for a range of potential student approaches, even during multi-step exercises. The incorporation of event tracking in the Praxis architecture, including trackers for both the debugger and the terminal, opens the possibility of adding new Tutors for other event-based learning domains in the future.

Chapter 2

Related Work

In order to set the stage for the design and development of a debugging tutor, one must first understand bugs: What are bugs, and when do they occur?

2.1 Understanding Bugs

Bugs in computer programs come in a variety of flavors. The literature largely does not have a standard way of categorizing bugs [17], instead focusing on unique categorizations to support an end-goal, such as understanding what kinds of bugs typically plague novice programmers [1]. At a high level, bugs are typically divided between logical or semantic errors that do not affect the running of the program, per se, but instead arise from a discrepancy in what the programmer intended for the program to do versus what the program actually does; compile-time bugs, which arise at compile time; and runtime bugs, which arise during runtime. For the Debug Tutor, logical will refer to bugs that do not halt the program, runtime will refer to bugs that halt the program during runtime, and compile-time will refer to bugs that halt the program during compile time.

Spohrer & Soloway (1986), found that novices suffered primarily from non-language-construct bugs [20]. Ahmadzadeh et al. (2005) similarly found that a majority of over 100,000 recorded bugs in student code was a result of semantic errors, where they define semantic to be any inconsistency with the language, as opposed to only inconsistencies that do not halt the program [1]. Exercises for the Debug Tutor contain only logical or runtime errors.

2.2 Understanding Debugging

Debugging, the act of finding and fixing bugs in code, is an essential skill for a successful programmer, with some sources claiming that over 50% of work days are spent debugging [12][22]. As expected, novice programmers struggle more with debugging than professional software engineers, but perhaps surprisingly, studies have found that the root of this disparity does not lie in programming ability alone [1].

Ahmadzadeh et al. (2005) conducted a study asking students to debug logic errors and found that a majority of students who successfully found and corrected these errors were also competent programmers [1]. But the same was not true the other way around: while a good debugger was a clear sign of a good programmer, being good at programming did not imply a similar proficiency at debugging. In other words, a knowledge gap existed between what drove the successful programming versus the successful debugging. Others have since come to the same conclusion: that debugging is a distinctly separate skill from programming [17]. In order to teach debugging, therefore, just knowing the types of bugs that novices struggle with is not enough; understanding what is missing from novices' mental models when debugging is equally crucial. What are the building blocks for an expert debugging process, and where do novices fail?

2.3 The Debugging Process

Mccauley et al. consolidated and summarized the literature on the debugging process [16], finding that, in general, the debugging process consistently followed four steps: understanding the system, testing the system, locating the error, and fixing the error. Importantly, the first three stages were the biggest hurdles during the debugging process [16].

Several studies compared the process of expert and novice debuggers. One study found that experts use seven types of knowledge when locating bugs: (1) *knowledge of the intended program*; (2) *knowledge of the actual program*; (3) *an understanding of the implementation language*; (4) *general programming expertise*; (5) *knowledge of the application domain*; (6) *knowledge of bugs*; and (7) *knowledge of debugging methods* ([6] Table 2-1). The seventh type, debugging methods, includes knowledge about the availability and use of tools such as a *tracer*, an early form of a debugger that included step-by-step tracing and breakpoints [6]. Nanja & Cook (1987) similarly found that the expert debuggers in their study tended to use an online debugger tool, whereas neither the novice nor the intermediate debuggers took advantage of such tooling, sometimes even opting for hand-written notes tracing the program evolution instead [19]. In line with this research, the proposed Debug Tutor places a big emphasis on learning to use the debugger.

Many studies attributed the difference in expert and novice debuggers to a lack of overall program comprehension [1][19][21]. Vessey (1985) found that this lack in program comprehension caused the difference in the approach to debugging. While the experts used a *breadth-first approach* that began with general program comprehension before sourcing the bug, novices instead went with a *depth-first approach.* Likewise, Nanja & Cook (1987) found that experts spent more time simply reading the code before attempting to debug, and, perhaps more importantly, the experts read the code in execution order, while the novices read the code in line order [19].

Blending these different approaches and knowledge gaps into a single framework, Li et al. [14] adapted Jonassen & Hung (2006)'s categories of knowledge required for general troubleshooting to debugging:

- **Domain**: Understanding the programming language
- **System**: Understanding the program to be debugged, including both *topological knowledge* (the architecture diagram of the program) and *functional knowledge* (the causal relationship between components of the program—i.e., comprehension of the program)

- **Procedural**: Knowing how to use the IDE's debugger or how to set up a test suite

- **Strategic**: Knowing how to employ *global strategies* (such as use of the debugger for tracing, or, e.g., following a breadth-first search strategy) and *local strategies* (such as context-dependent placement of print or log statements)

- **Experience**: Being able to draw on prior experience debugging [8]

Jonassen & Hung (2006)'s categories are useful for identifying what gaps in knowledge the proposed Debug Tutor aims to drill [8]. The Debug Tutor aims to drill *procedural knowledge*—that is, the use of the debugger—as well as some aspects of *system knowledge*, such as interpretation of stack traces and call stacks.

Evidently, expert and novice debuggers differ significantly in their mental representations during debugging, their approach to debugging, and the tools used during debugging. Given these findings, how has teaching debugging evolved, and what approaches already exist for teaching debugging?

2.4 On Teaching Debugging

Many studies agree that explicit instruction of debugging helps students become better debuggers [1][3][4][5][14][17]. In general, most studies on teaching debugging fall into one of two categories: long-term learning, in which students spend an extended period of time on debugging exercises without the direct help or guidance of an expert, and short-term teaching, in which students spend a single session being taught a systematic approach to debugging.

Taking the long-term learning approach, Chmiel & Loui (2004) found that debugging exercises incorporated into a semester-long undergraduate course on assembly language improved debugging skill in students [4]. One exercise type asked students to perform *code review* on buggy code, i.e., identify errors without running the code. The second exercise type asked students to debug buggy code, with the option of using a debugger tool for single stepping, setting breakpoints, and observing memory or register contents. The authors found that students who completed these exercises debugged significantly faster than students who did not, regardless of aptitude, and they urged instructors to "integrate activities throughout an entire curriculum" because debugging "is an important, complicated skill that requires repeated practice" [4]. The Debug Tutor, following their suggested approach, is intended to be used repeatedly and can be integrated across a CS curriculum.

On the other hand, Carver & Risinger (1987) employed a short-term teaching approach, opting to teach a systematic method in the course of one session [3]. They gave sixth grade students a flow chart diagramming a procedure for debugging and found that, later, students did continue to use the systematic approach unlike the control group's brute force approach. Michaeli & Romeike (2019) ran a similar study teaching a *scientific method* (observe, hypothesize, experiment, repeat) approach, varied slightly for compile time, runtime, and logical bugs:

1. Compile: Compile-time errors
    (a) Is the program compiling successfully? [*Observe*]
        i. If necessary, revert changes.
    (b) Read and understand error messages.
    (c) Adjust your program. [*Experiment & Repeat*]
2. Run: Runtime errors
    (a) Does the program run without errors? [*Observe*]
        i. If necessary, revert changes.
    (b) Read and understand the **first** error message.
    (c) "What's the cause?"
        i. Modify your assumption or make a new one. [*Hypothesize*]
    (d) Determine the error and find the relevant lines of code.
    (e) Adjust your program. [*Experiment & Repeat*]

3. Compare: Logical errors
    (a) Do expected and actual behavior match?
        i. If necessary, revert changes.
    (b) "Why is this the case?"
        i. Modify your assumption or make a new one. [*Hypothesize*]
    (c) Determine the error and find the relevant lines of code.
    (d) Adjust your program. [*Experiment & Repeat*]
4. Done! (Systematic debugging process presented by [17])

They, too, found that teaching this approach over the course of only one session improved both self-efficacy (self-reliance) and skill (performance) in students. Interestingly, in order to circumvent the apparent lack of program comprehension and tool-use skills in these novice debuggers, they designed their exercises to be short program prototypes that successively build off previous prototypes, such that no large chunk of code was ever new for the students. The Debug Tutor borrows from that approach as well, opting to use shorter, easier-to-understand chunks of buggy code in the exercises, although it uses this approach precisely to hone in on building skills in the use of tools without the need for advanced program comprehension skills.

Previously, MIT's undergraduate Software Construction course (6.102, formerly 6.031) curriculum included one reading assignment and associated class session devoted to debugging with the scientific method [1]. By designing Tutor exercises devoted to aspects of the reading, the deployment of the Debug Tutor in 6.102 blends these two long-term and short-term teaching approaches.

2.5 Existing Tools for Teaching Debugging

Li et al. used their knowledge characterization (Domain, System, Procedural, Strategic, Experience) not just for categorizing novice versus experts, but also to categorize

[1] https://web.mit.edu/6.102/www/sp23/classes/09-debugging/

existing tools [14]. They found that most tools teach *domain knowledge* in the form of reading and writing code. They also mentioned that few tools are language-agnostic; the Debug Tutor indeed has this feature.

Other tools, like Gidget [13]) and RoboBUG [18], gamify the instruction of debugging. While both tools support some debugger tools like setting breakpoints, the Debug Tutor serves exercises inside Microsoft's Visual Studio Code editor, a popular IDE used by many professional software engineers [2], for the express purpose of exposing novices to readily available and widely used tools.

A handful of existing tools target *system knowledge*. For example, ViLLE [11]), a general-purpose education framework, has programming exercises with system architecture visualizations built up frame-by-frame from the call stack, which help students develop a mental model of the program. DebugIt [12] and Debug-ITS [2], tools that targets *system* and *experiential knowledge*, are similar to the proposed Debug Tutor in that they host a suite of exercises with short programs and automated feedback. However, unlike the Tutor, the exercises are not hosted in a general-purpose IDE, they ask the student to find *and fix* the error (rather than target a microskill, such as placement of breakpoints), and do not encourage use of a debugger tool. Intelligent Tutoring System [10], likewise, has a suite of exercises with automated feedback, but, unlike the Debug Tutor, is not aimed at drilling debugging-specific microskills.

LadeBug [15] is a novel debugging tutor that uses an *omniscient* debugger. Unlike a typical debugger, which only supports forward stepping, an omniscient debugger supports forwards and backwards stepping by recording the entire execution trace. Like the Debug Tutor, LadeBug hosts a suite of exercises that staff can easily add, edit, or delete from, encourages repeated practice, and provides automated feedback. Unlike the Debug Tutor (which only asks students to drill a microskill or locate a bug), though, LadeBug asks students to both find *and fix* errors, and it currently

[2] Ranked the top preferred IDE according to StackOverflow surveys in 2018 (https://insights.stackoverflow.com/survey/2018/#development-environments-and-tools), 2019 (https://insights.stackoverflow.com/survey/2019#development-environments-and-tools), 2021 (https://insights.stackoverflow.com/survey/2021#most-popular-technologies-new-collab-tools), and 2022 (https://survey.stackoverflow.co/2022/#section-most-popular-technologies-integrated-development-environment)

only supports Python code.

Furthermore, although omniscient debugging is growing more mainstream with companies such as Replay [3] making them more practical and widely available, omniscient debuggers are still far from the norm when it comes to general debugging approaches. Tools taught by the Debug Tutor should already be widely available and applicable in industry, which is why it is built on debuggers that are available through VS Code. However, were VS Code to ever support omniscient debugging, the Debug Tutor would be able to as well, since it can plug into any method exposed by the `vscode.debug` Application Programming Interface (API). [4]

IDE Embedded Tutorials [9] perhaps comes closest to the Debug Tutor, as it embeds debugging tutorials in the popular IDE IntelliJ IDEA. However, unlike the drill-based exercises of the Debug Tutor, the IDE Embedded Tutorials exercises are tutorial walkthroughs. The system is also not yet deployed, and is only an early prototype linked to planned research for the author's dissertation project.

Finally, one-off debugging assignments such Maze solver [5] and Binary Bomb[6] (from the SIGCSE Nifty Assignment session [7]) differ from the Debug Tutor in that they are not completed over a semi prolonged period of time. Some debugging textbooks do exist, such as *Why Programs Fail: A Guide to Systematic Debugging* [22] for learning systematic debugging, complete with exercise assignments at the end of each chapter. However, they are mostly aimed at advanced undergraduates and professions, and, of course, do not provide automatic feedback.

2.6 Applying Deliberate Practice to Debugging

No solution thus far has put deliberate practice at the heart of a debugging tutor, despite the fact that debugging is a clear candidate for this type of practice. With the core principles of long-term repeated drilling, automatic expert feedback, and

[3] https://www.replay.io

[4] https://code.visualstudio.com/docs/editor/debugging#_debugger-extensions

[5] http://nifty.stanford.edu/2008/blaheta-maze/

[6] http://csapp.cs.cmu.edu/public/labs.html

[7] http://nifty.stanford.edu

microskill-targeted exercises, the Debug Tutor meshes together several of the required techniques for successfully teaching debugging. Indeed, Li et al. notes that they "are not aware of any research that focuses on teaching students to acquire [procedural knowledge such as knowing how to use the features of an IDE] in a debugging context" [14]. The Debug Tutor aims to do just that.

Chapter 3

Design

The Debug Tutor extends the Praxis framework, which organizes teachable skills using a map of concepts. Each concept is described using a path to indicate its hierarchy of concepts. For example, in the Typescript Tutor, the string::length concept is the *length* sub-concept of the more general *string* concept in the Typescript language. Each concrete exercise given to the user drills a specific concept.

A concept *group* organizes concepts into a coherent set. Each concept group manifests as a set of exercises drilling concepts from the group. A student must complete a certain number of these exercises for the concept group to be marked as finished, i.e., sufficiently practiced. Concept groups are organized into levels, described by the concept *map*. Initially, the student has access to only the groups in level 0 and can choose to work through them in any order. Level i becomes *unlocked*—that is, accessible to the student—when all the groups in level $i-1$ are finished. The concept map is the primary method by which the Praxis framework organizes the skills being taught and measures student progress.

3.1 A Debug Concept Map

The backbone of the Debug Tutor as an educational tool is the design of its concept map. The debug concepts are intended to be specific microskills necessary for competent debugging. Similar to how Typescript Tutor exercises teach syntax and are incorporated into readings on software design, the Debug Tutor concepts drill the use of debugging tools and are embedded into a reading on systematic debugging. Fig. 3-1 shows the Debug Tutor's concept map.

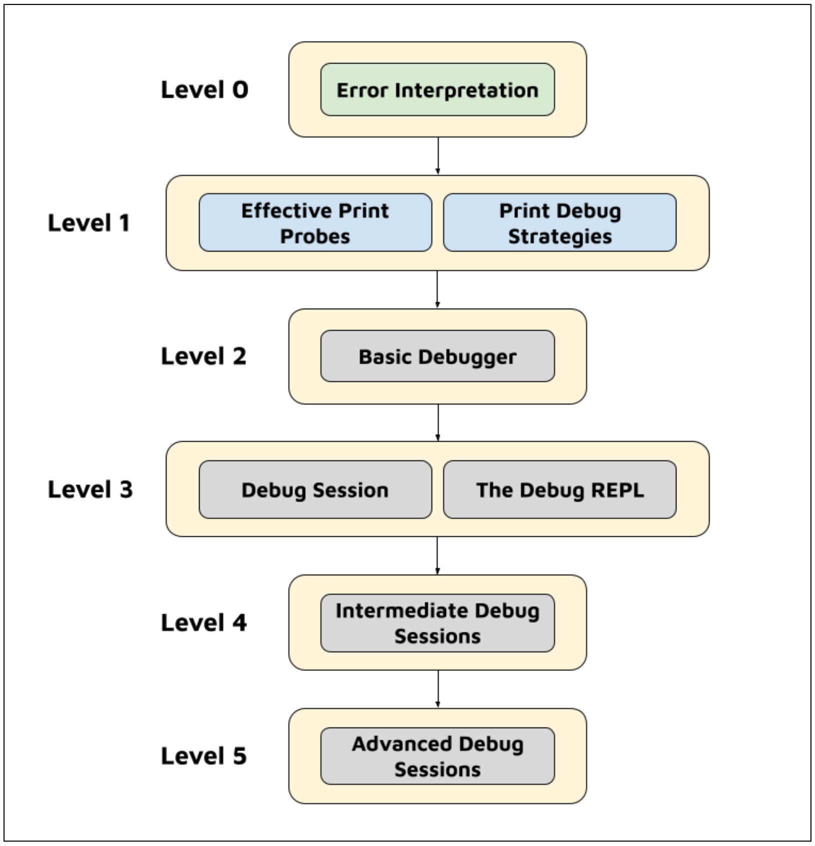


Figure 3-1: Debug Tutor Concept Map

Table 3.1 lists the Debug Tutor concept map in full, detailing the microskills associated with each concept, listed by concept group, and organized by level.

Level 0: Error Interpretation

Students are first drilled in interpreting runtime error messages. All runtime errors emit a stack trace, which includes the thrown error and a trace through code file locations that led to the thrown error. The trace includes the entry point, the last line executed, and, if applicable, the failed assert statement in buggy code. Experts in debugging are able to interpret these stack traces quickly, accurately identifying the flow of execution and most relevant lines. Most modern editors also support *terminal*

Table 3.1: Debug Tutor Concept Map

Level	Concept Group	Concept Skill	Exercised Microskill
0	**Error Interpretation** *Interpreting Runtime Error Messages*	runtime-error::last-line-executed	Navigate to the last line executed via a runtime error.
		runtime-error::entry-point	Navigate to the entry point via a runtime error.
		runtime-error::failed-assert	Navigate to a failed assert statement via a runtime error.
1	**Effective Print Probes** *Print Debugging*	print::probes::fstring	Print using an *fstring* template.
		print::probes::object-log	Print objects effectively.
	Print Debug Strategies *Strategic & safe printing*	print:strategy::state-before-return	State return values safely.
		print:strategy::confirm-called	Printing checkpoints.
		print:strategy::state-parameters	State parameters effectively.
2	**Basic Debugger** *Introduction to the Debugger*	warmup::set-breakpoint	Set breakpoints.
		warmup::remove-breakpoint	Remove breakpoints.
		warmup::begin-debug-session	Launch a debug session.
3	**Debug Session** *Introduction to single stepping.*	breakpoints::drive::step-into	Step in during a debug session.
		breakpoints::drive::step-over	Step over during a debug session.
		breakpoints::drive::step-out	Step out during a debug session.
		breakpoints::drive::continue	Continue during a debug session.
		breakpoints::drive::set-breakpoint	Set a breakpoint during a debug session.
		breakpoints::drive::stop	Stop a debug session.
		breakpoints::drive::restart	Restart a debug session.
	The Debug REPL *The read-eval-print-loop.*	breakpoints::drive::evaluate	Evaluate variables and expressions with the Debug REPL during a debug session.
4	**Intermediate Debug Sessions**	breakpoints::advanced-drive::step-in	Step in indefinitely to achieve a goal during a debug session.
		breakpoints::advanced-drive::continue	Continue indefinitely to achieve a goal during a debug session.
		breakpoints::advanced-drive::mixed-intro	Introduce mixed uses of debug tools to achieve a goal.
5	**Advanced Debug Sessions**	breakpoints::advanced-drive::mix	Mixed uses of debug tools to achieve a goal.

links within the stack trace, which are clickable paths that, when clicked, move the cursor to the specified file, line, and, sometimes, exact character.

Level 1: Effective Print Probes & Print Debug Strategies

The next two concept groups focus on print debugging, also known as *printf* debugging for historical reasons. Probably the most widely used debugging strategy, print statements are added to code as probes during a debug session. As the code executes, the print statements are printed to the console, thereby relaying information about state throughout code execution. Once the bug is found and fixed, the probes are then removed from the code.

Effective Print Probes covers effective uses of print statements. One such microskill is the use of template strings, or *fstrings*, which are strings that allow variable value insertions. The second microskill in this concept group, *print::probe:object-log*, is how to effectively use a language's built-in constructs to print objects meaningfully, rather than, say, printing a default `[Object object]` string.

Print Debug Strategies covers strategic and safe print debugging approaches. The *state-before-return* concept is the practice of saving the return value to print and return, rather than executing the return value twice. The *confirm-called* concept is checkpoint printing—that is, inserting print statements to declare the current execution trace, such as `print("inside while loop of computeProgress")`—throughout a logically buggy program to confirm that certain points in a program are indeed reached. Finally, the *state-parameters* concept is the strategic printing of parameters within the function call.

Level 2: Basic Debugger

The Basic Debugger concept group is an introduction to three fundamental skills necessary to run a debug session: setting a breakpoint (*warmup::set-breakpoint*), removing a breakpoint (*warmup::remove-breakpoint*), and launching a debug session (*warmup::launch-debug-session*).

Level 3: Debug Session & The Debug REPL

Debug Session concepts introduce single stepping, which include stepping into, stepping over, stepping out, continuing, setting a breakpoint and continuing to it, stopping, and restarting. The Debug REPL's concept, *breakpoints::drive::evaluate*, is evaluation with the debug read-eval-print-loop (REPL) during a debug session.

Level 4: Intermediate Debug Sessions & Level 5: Advanced Debug Sessions

The final two concept groups drill the integrated use of debugger tools to achieve an abstract goal. The Intermediate *breakpoints::advanced-drive::step-in* concept covers stepping in indefinitely to achieve a goal, such as understanding the depth of called functions. *breakpoints::advanced-drive::continue* covers continuing indefinitely to achieve a goal, such as reaching a certain loop state. Finally *breakpoints::advanced-drive::mixed-intro* introduces mixed use of single stepping to achieve an abstract goal.

Advanced Debug Session's *breakpoints::advanced-drive::mix* provide even less direction to the student and often allow freedom in how the student achieves a goal. The concept is the first effort towards open-ended debugging practice, but stops just short of covering actually finding and fixing bugs (which, as it stands, would be difficult for the Tutor to verify).

The Broad Application of Concepts

The concepts throughout the Debug Tutor concept map are agnostic to both the language of the exercise files and the specific features of the IDE in which it is embedded. Most widely-used languages support all of the runtime error and print concepts. As well, most standard debuggers across all IDEs and languages support all of the runtime error, print debugging, warmup, single stepping, and evaluation concepts. Each concept should therefore be general enough to be implemented by exercises in any widely-used language and deployed in any standard IDE. The following section describes the design behind the exercises implemented for each concept group.

3.2 Design of Exercises

At a high level, each exercise drills a specific concept by expressing an abstract goal. For example, when drilling *warmup::set-breakpoint*, an exercise prompt might read *Set a breakpoint where the variable `total` is defined* as opposed to *Set a breakpoint on line 5.* Although exercises do not necessarily ask the student to find a bug, they may nonetheless be drilled on buggy code.

Error Interpretation To drill both interpretation of stack traces and efficient use of the IDE terminal, each exercise in the Error Interpretation group asks the student to navigate to a logical location rather than a specific file and line (i.e. the *entry point* rather than *line 9 in `Computation.ts`*) and requires navigating there using the terminal link in the stack trace.

Each exercise begins with displaying the exercise file in the editor. The student is asked to click Start in the Tutor window and is directed to watch closely as the file is automatically compiled and executed on the command line. The buggy code then emits a stack trace for the student to interpret. The code itself is inspired by common bugs often faced in the wild, such as aliasing or off-by-one errors.

Print Debugging All exercises in the Effective Print Probes and Print Debug Strategies groups use a fill-in-the-blank style of exercise. These exercises bootstrap off of a student's purported knowledge of Python, a prerequisite of the 6.102. A student is presented a code file with one or more lines of code blanked out—that is, replaced with a series of periods. The exercise asks the student to fill in the blanks based on either a general end-goal (such as, *print the value of `x`*) or given a Python line that must be converted to TypeScript.

Exercises of the *print::probes::fstring* concept in the Effective Print Probes group ask students to practice using TypeScript template strings of the form `` `${x}$` ``. Exercises of the *print::probes::object-log* concept, on the other hand, use TypeScript's `console` library to effectively print objects by passing in comma-separated strings and objects to `console.log` values rather than a concatenated string and object-string.

Exercises for the *state-before-return* concept in Print Debug Strategies ask that the student practice saving the return value to a variable, printing the variable, and then returning the variable, as opposed to executing the return value both for printing and again for returning. Exercises drilling the *confirm-called* concept ask the student to plant checkpoint print statements—that is, print statements that declare the current state of the program, such as `print("inside while loop of computeProgress")`—throughout a logically buggy program. Finally, exercises for the *state-parameters* concept drill the strategic printing of parameters within the function call.

Basic Debugger Each concept in the Basic Debugger concept group is implemented by one concrete exercise. The first exercise shown to the student, one for *warmup::set-breakpoint*, asks students to set a breakpoint. Building on the set breakpoint exercise, the following exercise, one for *warmup::remove-breakpoint*, automatically adds a breakpoint and then asks the student to remove that breakpoint. The third and final exercise, one for *warmup::begin-debug-session*, again automatically adds a breakpoint, but then asks students to launch a debug session and observe as it pauses at that breakpoint.

Exercises that require launching a debug session come pre-packaged with a debugger launch configuration for the open file. Students are instructed to begin sessions using this configuration, entitled *PraxisTutor-DebugCurrentFile*. Pre-packaging the exercises with a launch configuration gives the exercise author control over the configuration of the debug sessions.

Debug Session & The Debug REPL Four of the seven concepts in the Debug Session group—*step-into*, *step-over*, *continue*, and *stop*—are implemented in exercises that automatically set breakpoints and launch a debug session, and then prompt the student to perform the single step action. *Step-out* is implemented by an exercise that automatically sets a breakpoint, launches a debug session, and then steps in once, so that students need only step out. Exercises for the final two concepts—*set-*

breakpoint and *restart*—automatically set breakpoints and launch a debug session, and then ask the student to perform two actions: For *set-breakpoint*, the student is asked to set another breakpoint and then continue to that breakpoint, to practice setting breakpoints mid-execution. For *restart*, the student is asked to step over and then restart.

Exercises implemented for the Debug REPL's one concept, *drive::evaluate*, each automatically set breakpoints and launch debug sessions and then ask the student to evaluate expressions with the Debug REPL. Some exercises practice evaluating whole objects and others practice evaluating object properties.

Intermediate & Advanced Debug Sessions Intermediate debug sessions drill concepts one level more abstract than single stepping. For example, an exercise may ask the student to set a breakpoint on a line specified by some condition in a loop (e.g. *where values are pushed to `deltas`*), and continue indefinitely until some condition is met (e.g., *when `value` becomes undefined*). Another exercise might ask the student to indefinitely step in until execution is paused inside a specific function.

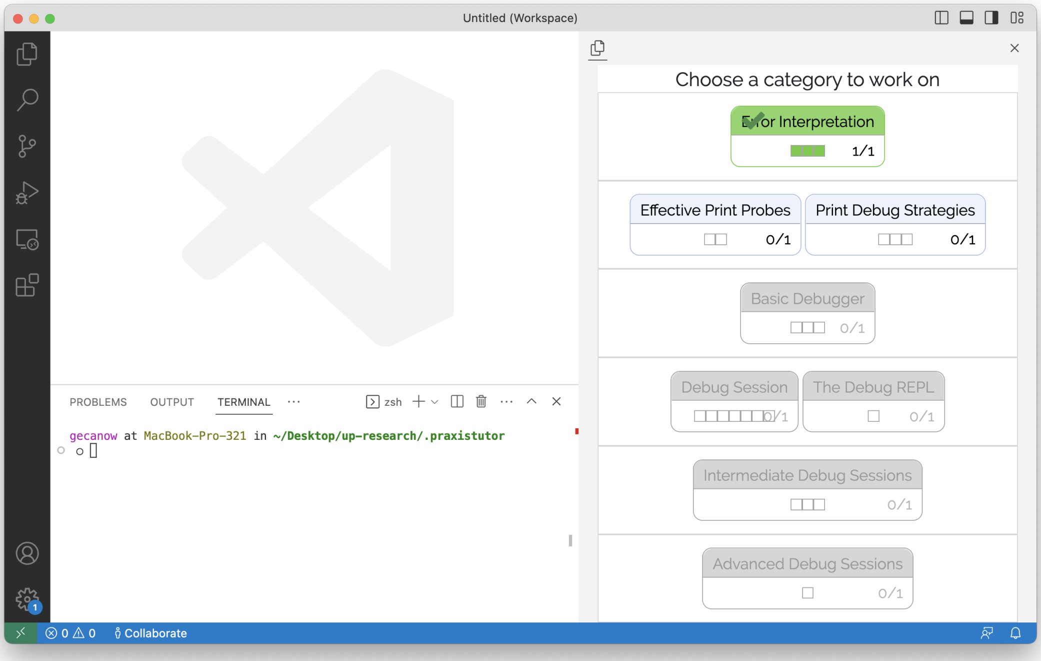
Exercises implemented for the Advanced concept group's *advanced-drive::mix* concept go one step further. For example, part of an exercise may ask the student mid-session to get the program execution paused on the first line of some function, without specifying which debugger tools to use or how. Although these exercises are certainly more elaborate than the typical Praxis Tutor exercise, they serve to drill practice in the use of the debugger as an integrated tool during a controlled session.

Automatic Setup As described, the design of many exercises includes an automated setup process to bring the code execution to a certain state, after which the student is asked to complete a task. For example, the setup may include automatically launching a debug session, at which point the student is asked to continue debugging to achieve a certain goal or state. The automated setup allows the student to focus completely on practicing a specific microskill, without worrying about compiling and executing code until the state at which the microskill is necessary.

3.3 User Interface

The Praxis Tutor and its exercises are implemented as an extension for VS Code, Microsoft's popular, free, and widely-used code editor.[1] The installation process is detailed for students on the 6.102 website.[2]

For the debugging exercises in particular, students are asked to move the Praxis Tutor window to the right sidebar so that it is visible with the Run and Debug pane is displayed on the left sidebar. The Tutor displays its Home UI. All concept groups are visible, but only unlocked concept groups are clickable. Fig. 3-2 shows the what the installed Debug Tutor would look like after a student dragged the window to the rightside bar and then completed the concept group in level 0.



```
gecanow at MacBook-Pro-321 in ~/Desktop/up-research/.praxistutor
○
```

Figure 3-2: Full VS Code window in student view, with level 0 completed, level 1 unlocked, and levels 2-5 locked

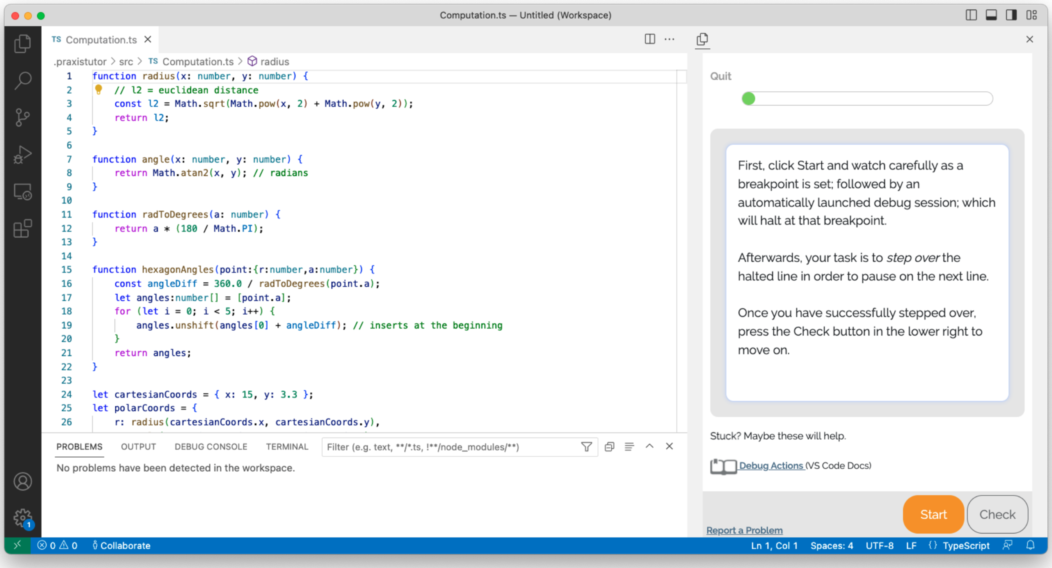
[1] Praxis Tutor also works on VSCodium, a "freely-licensed binary distribution of Microsoft's editor VS Code" (https://vscodium.com). Both are available for Mac, Windows, and Linux.

[2] The Spring 2023 installation guide can be found at https://web.mit.edu/6.102/www/sp23/tools/getting-started/#praxis-tutor

3.3.1 Exercises

When a student clicks on a concept group, they are shown an exercise for some concept in the group. Exercises are ordered by a priority level set by the exercise author, and exercises with the same priority are randomly ordered when the concept group is clicked. The exercise code file is automatically opened in the text editor alongside the Tutor window.

For a given exercise, along with the prompt, students have the option to Start (if the exercise has automated setup), Check, Start Over, Report A Problem, or Quit. Start, Check, and Start Over appear in the bottom grey bar below the prompt, Report A Problem appears below the bottom grey bar, and Quit appears above the prompt. After a first attempt is made, a hidden Show Answer button is enabled, which will display a textual description of the correct answer when clicked. Exercises can also include online tutorial links, displayed below the main prompt. Fig. 3-3 shows an example of an exercise displayed when the Debug Session concept group is first clicked.


Computation.ts — Untitled (Workspace)
TS Computation.ts
.praxistutor > src > TS Computation.ts > radius

```
function radius(x: number, y: number) {
    // l2 = euclidean distance
    const l2 = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
    return l2;
}

function angle(x: number, y: number) {
    return Math.atan2(x, y); // radians
}

function radToDegrees(a: number) {
    return a * (180 / Math.PI);
}

function hexagonAngles(point:{r:number,a:number}) {
    const angleDiff = 360.0 / radToDegrees(point.a);
    let angles:number[] = [point.a];
    for (let i = 0; i < 5; i++) {
        angles.unshift(angles[0] + angleDiff); // inserts at the beginning
    }
    return angles;
}

let cartesianCoords = { x: 15, y: 3.3 };
let polarCoords = {
    r: radius(cartesianCoords.x, cartesianCoords.y),
```

No problems have been detected in the workspace.

First, click Start and watch carefully as a breakpoint is set; followed by an automatically launched debug session; which will halt at that breakpoint.

Afterwards, your task is to *step over* the halted line in order to pause on the next line.

Once you have successfully stepped over, press the Check button in the lower right to move on.

Stuck? Maybe these will help.

Debug Actions (VS Code Docs)

Figure 3-3: Example exercise when a concept group is first opened

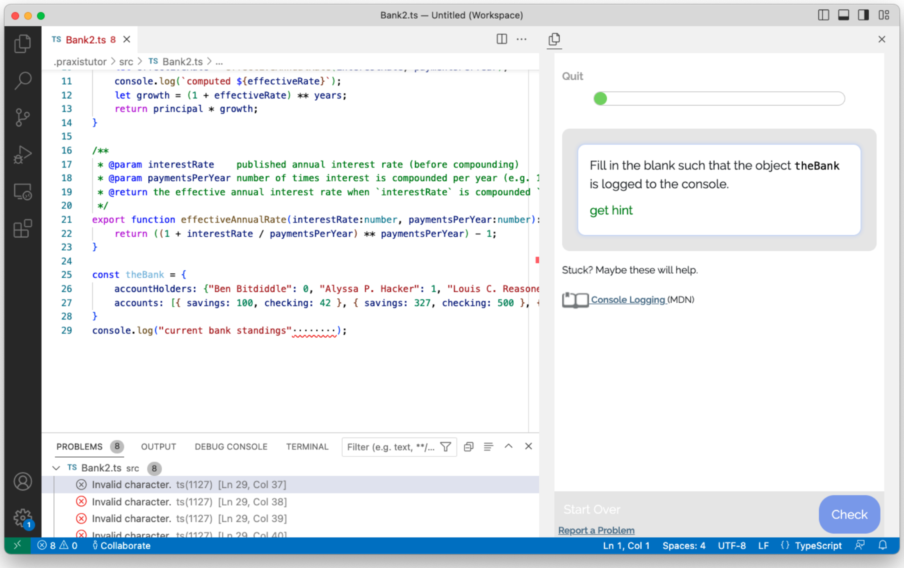
Initially, Praxis Tutor exercises were all of the save flavor: along with the exercise, the student was shown a code file with some fill-in-the-blanks, and the student had the ability to edit the file and click Check at any point to check their work. If incorrect, the Tutor exercise view would highlight in red and might show hints, and the student could continue editing clicking Check. If correct, the view would highlight in green and might show warnings, and the student could click Continue.

With the introduction of event tracking in the Debug Tutor, new exercises might instead require automated setup, such as automatically launching a code file or setting a breakpoint, and might require starting over, such as a simple debug action that, if done incorrectly, requires re-starting the exercise from the beginning. To support these new exercises, two new workflow options were introduced into the Tutor user interface, for a total of three potential user interface flows.

Basic Workflow: No Required Set Up, No Required Start Over An exercise that does not require set up immediately displays in a Ready for Interaction UI. There is no Start button, only an always-enabled Check button. The basic workflow does not require starting over, so on an incorrect attempt the UI will display hints (if any), and will otherwise remain ready for more submission attempts. Figure 3-4 shows a fill-in-the-blank Effective Print Probes prompt on first display and after an incorrect attempt.

The TypeScript Tutor exercises are all fill-in-the-blanks that follow this flow. In the Debug Tutor, Effective Print Probes and Print Debug Strategies, which consist of fill-in-the-blank exercises, as well as exercises that ask the student to perform all steps, such as some intermediate and advanced exercises, also follow this flow.

Setup Required Exercises that require setup first display with an enabled Start button next to a disabled Check button. Once the student clicks Start, both buttons are disabled until setup completes, at which point Check becomes enabled. On an incorrect attempt, if the exercise does not require starting over, it displays hints (if any) and otherwise remains ready for more submission attempts. Figure 3-5 shows a Debug Session prompt in four stages: on first display, during setup, immediately after setup, and after an incorrect attempt.



```
    console.log(`computed ${effectiveRate}`);
    let growth = (1 + effectiveRate) ** years;
    return principal * growth;
}

/**
 * @param interestRate    published annual interest rate (before compounding)
 * @param paymentsPerYear number of times interest is compounded per year (e.g. 1
 * @return the effective annual interest rate when `interestRate` is compounded `
 */
export function effectiveAnnualRate(interestRate:number, paymentsPerYear:number):
    return ((1 + interestRate / paymentsPerYear) ** paymentsPerYear) - 1;
}

const theBank = {
    accountHolders: {"Ben Bitdiddle": 0, "Alyssa P. Hacker": 1, "Louis C. Reasone
    accounts: [{ savings: 100, checking: 42 }, { savings: 327, checking: 500 }, {
}
console.log("current bank standings"········);
```

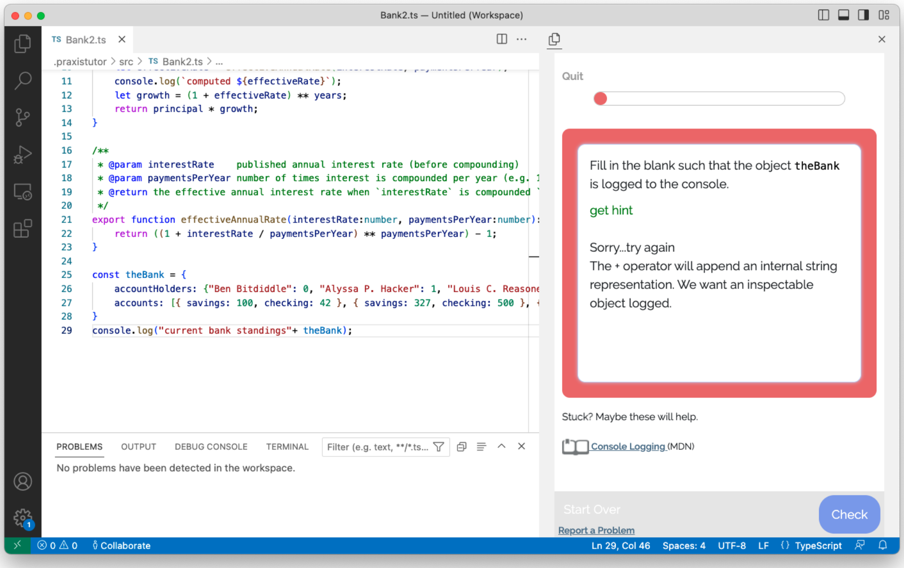
Fill in the blank such that the object `theBank` is logged to the console.

Stuck? Maybe these will help.

Console Logging (MDN)

```
Bank2.ts src 8
Invalid character. ts(1127) [Ln 29, Col 37]
Invalid character. ts(1127) [Ln 29, Col 38]
Invalid character. ts(1127) [Ln 29, Col 39]
Invalid character. ts(1127) [Ln 29, Col 40]
```

⊗ 8 ⚠ 0



```
    console.log(`computed ${effectiveRate}`);
    let growth = (1 + effectiveRate) ** years;
    return principal * growth;
}

/**
 * @param interestRate    published annual interest rate (before compounding)
 * @param paymentsPerYear number of times interest is compounded per year (e.g. 1
 * @return the effective annual interest rate when `interestRate` is compounded `
 */
export function effectiveAnnualRate(interestRate:number, paymentsPerYear:number):
    return ((1 + interestRate / paymentsPerYear) ** paymentsPerYear) - 1;
}

const theBank = {
    accountHolders: {"Ben Bitdiddle": 0, "Alyssa P. Hacker": 1, "Louis C. Reasone
    accounts: [{ savings: 100, checking: 42 }, { savings: 327, checking: 500 }, {
}
console.log("current bank standings"+ theBank);
```

Fill in the blank such that the object `theBank` is logged to the console.

Sorry...try again
The + operator will append an internal string representation. We want an inspectable object logged.

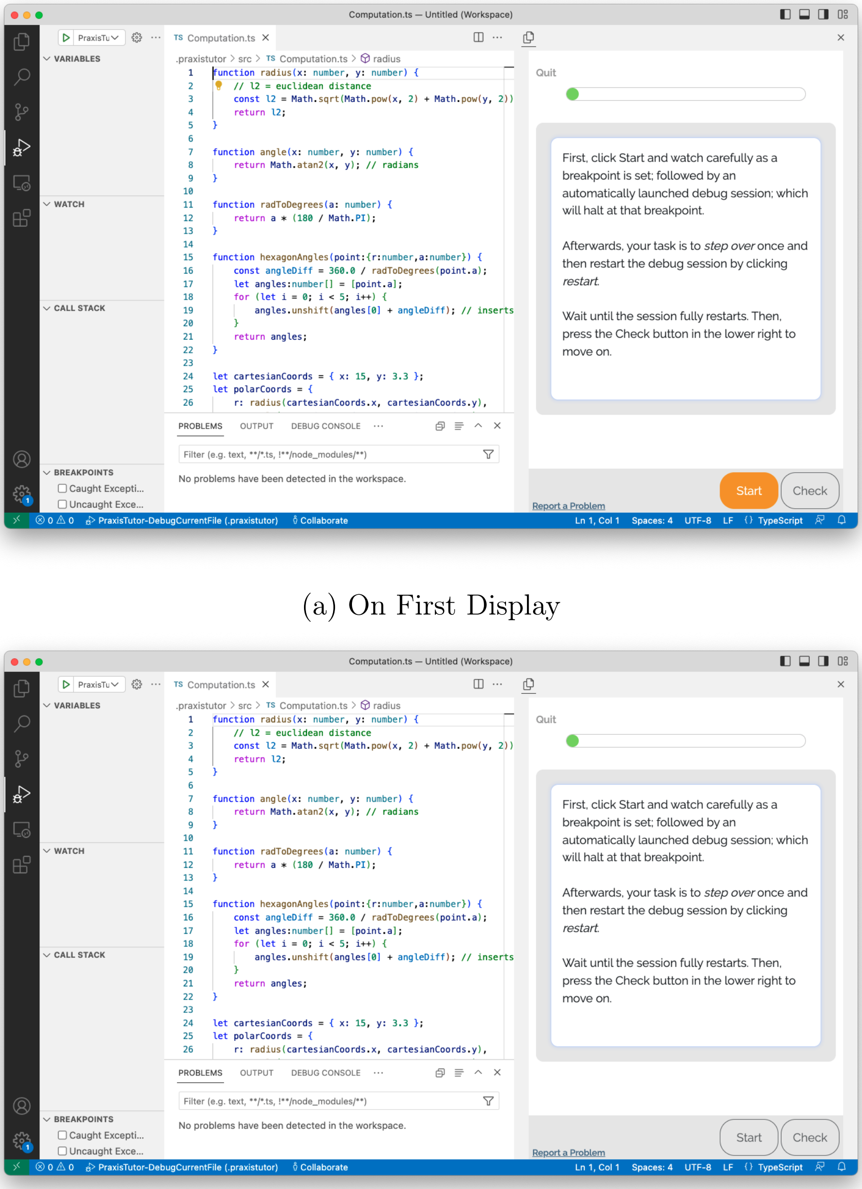
Stuck? Maybe these will help.

Console Logging (MDN)

No problems have been detected in the workspace.

⊗ 0 ⚠ 0

Figure 3-4: Basic Workflow



```
function radius(x: number, y: number) {
    // l2 = euclidean distance
    const l2 = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
    return l2;
}

function angle(x: number, y: number) {
    return Math.atan2(x, y); // radians
}

function radToDegrees(a: number) {
    return a * (180 / Math.PI);
}

function hexagonAngles(point:{r:number,a:number}) {
    const angleDiff = 360.0 / radToDegrees(point.a);
    let angles:number[] = [point.a];
    for (let i = 0; i < 5; i++) {
        angles.unshift(angles[0] + angleDiff); // inserts
    }
    return angles;
}

let cartesianCoords = { x: 15, y: 3.3 };
let polarCoords = {
    r: radius(cartesianCoords.x, cartesianCoords.y),
```

Filter (e.g. text, **/*.ts, !**/node_modules/**)

No problems have been detected in the workspace.

First, click Start and watch carefully as a breakpoint is set; followed by an automatically launched debug session; which will halt at that breakpoint.

Afterwards, your task is to *step over* once and then restart the debug session by clicking *restart*.

Wait until the session fully restarts. Then, press the Check button in the lower right to move on.

(a) On First Display

```
function radius(x: number, y: number) {
    // l2 = euclidean distance
    const l2 = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
    return l2;
}

function angle(x: number, y: number) {
    return Math.atan2(x, y); // radians
}

function radToDegrees(a: number) {
    return a * (180 / Math.PI);
}

function hexagonAngles(point:{r:number,a:number}) {
    const angleDiff = 360.0 / radToDegrees(point.a);
    let angles:number[] = [point.a];
    for (let i = 0; i < 5; i++) {
        angles.unshift(angles[0] + angleDiff); // inserts
    }
    return angles;
}

let cartesianCoords = { x: 15, y: 3.3 };
let polarCoords = {
    r: radius(cartesianCoords.x, cartesianCoords.y),
```

Filter (e.g. text, **/*.ts, !**/node_modules/**)

No problems have been detected in the workspace.

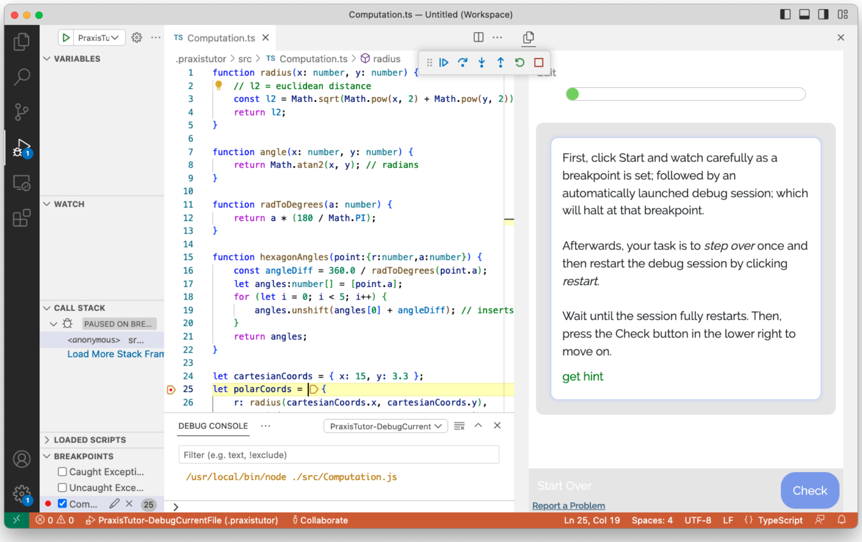
First, click Start and watch carefully as a breakpoint is set; followed by an automatically launched debug session; which will halt at that breakpoint.

Afterwards, your task is to *step over* once and then restart the debug session by clicking *restart*.

Wait until the session fully restarts. Then, press the Check button in the lower right to move on.

(b) During Setup

Figure 3-5: Setup Required Workflow



```
function radius(x: number, y: number) {
    // l2 = euclidean distance
    const l2 = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
    return l2;
}

function angle(x: number, y: number) {
    return Math.atan2(x, y); // radians
}

function radToDegrees(a: number) {
    return a * (180 / Math.PI);
}

function hexagonAngles(point:{r:number,a:number}) {
    const angleDiff = 360.0 / radToDegrees(point.a);
    let angles:number[] = [point.a];
    for (let i = 0; i < 5; i++) {
        angles.unshift(angles[0] + angleDiff); // inserts
    }
    return angles;
}

let cartesianCoords = { x: 15, y: 3.3 };
let polarCoords = {
    r: radius(cartesianCoords.x, cartesianCoords.y),
```

```
/usr/local/bin/node ./src/Computation.js
```

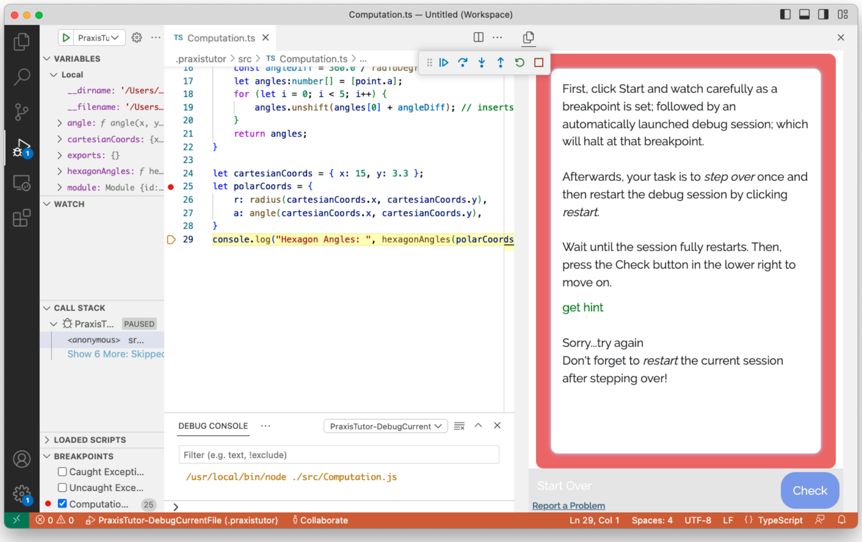
First, click Start and watch carefully as a breakpoint is set; followed by an automatically launched debug session; which will halt at that breakpoint.

Afterwards, your task is to *step over* once and then restart the debug session by clicking *restart*.

Wait until the session fully restarts. Then, press the Check button in the lower right to move on.

get hint

(c) After Setup Completes



```
    const angleDiff = 360.0 / radToDegr
    let angles:number[] = [point.a];
    for (let i = 0; i < 5; i++) {
        angles.unshift(angles[0] + angleDiff); // inserts
    }
    return angles;
}

let cartesianCoords = { x: 15, y: 3.3 };
let polarCoords = {
    r: radius(cartesianCoords.x, cartesianCoords.y),
    a: angle(cartesianCoords.x, cartesianCoords.y),
}
console.log("Hexagon Angles: ", hexagonAngles(polarCoords
```

```
/usr/local/bin/node ./src/Computation.js
```

First, click Start and watch carefully as a breakpoint is set; followed by an automatically launched debug session; which will halt at that breakpoint.

Afterwards, your task is to *step over* once and then restart the debug session by clicking *restart*.

Wait until the session fully restarts. Then, press the Check button in the lower right to move on.

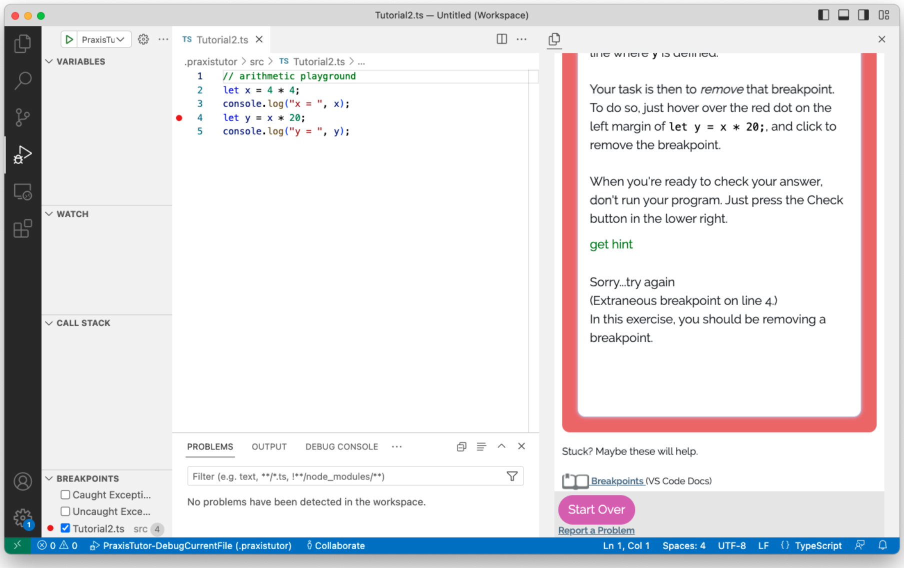
get hint

Sorry...try again
Don't forget to *restart* the current session after stepping over!

(d) An Incorrect Attempt

Figure 3-5: Setup Required Workflow (cont.)

Start Over Required Some Debug Tutor exercises explicitly require starting over. Forced start over may be enabled regardless of whether the exercise requires setup or not. In the case of forced start over, on an incorrect attempt, the Start Over button is made prominent by highlighting it in pink. The Check button is disabled and hidden, as well as the Start button if it was displayed, thus forcing the student to start over. On start over, the exercise will appear in its initial state, with a clean exercise file and either requiring setup or ready for submission. Figure 3-6 shows a Basic Debugger exercise with required start over prompt after an incorrect attempt. Importantly, students have the option to start over at any time during the course of attempting an exercise; The forced start over UI simply makes the button prominent.



```
1  // arithmetic playground
2  let x = 4 * 4;
3  console.log("x = ", x);
4  let y = x * 20;
5  console.log("y = ", y);
```

No problems have been detected in the workspace.

line where y is defined.

Your task is then to *remove* that breakpoint. To do so, just hover over the red dot on the left margin of `let y = x * 20;`, and click to remove the breakpoint.

When you're ready to check your answer, don't run your program. Just press the Check button in the lower right.

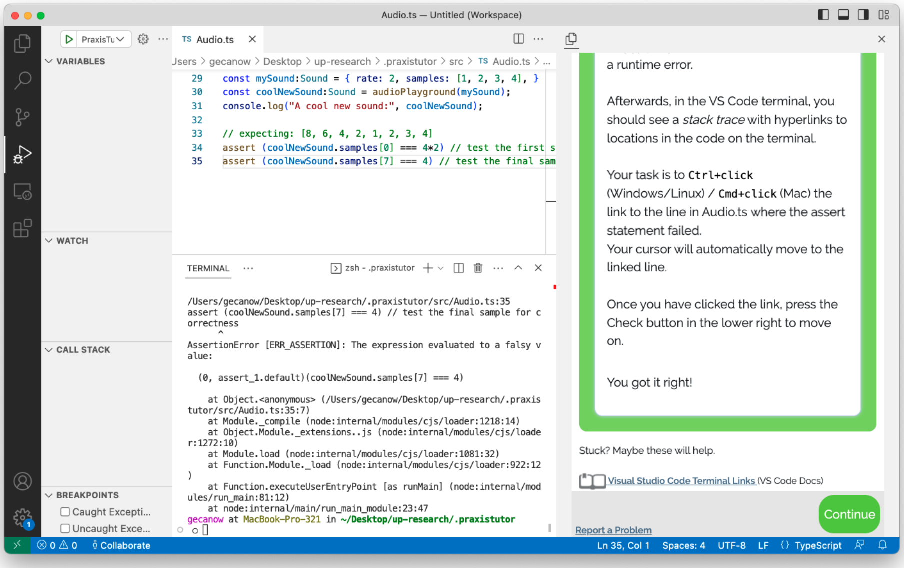
Sorry...try again
(Extraneous breakpoint on line 4.)
In this exercise, you should be removing a breakpoint.

Stuck? Maybe these will help.

Breakpoints (VS Code Docs)

Figure 3-6: Start Over Required Workflow

On a Correct Submission On a correct submission, any relevant warnings are displayed under the prompt and the bottom grey bar only displays a Continue button. If there are more exercises left in the concept group, clicking Continue displays the next exercise. Otherwise, clicking Continue displays a minimal congratulatory UI with a Go Home button to return to the full concept map view UI. Figure 3-7 shows a completed final Error Interpretation exercise and the congratulatory UI that follows.


Audio.ts — Untitled (Workspace)
PraxisTu
VARIABLES
WATCH
CALL STACK
BREAKPOINTS
□ Caught Excepti...
□ Uncaught Exce...
TS Audio.ts
Jsers > gecanow > Desktop > up-research > .praxistutor > src > TS Audio.ts > ...
TERMINAL
zsh - .praxistutor
Ln 35, Col 1 Spaces: 4 UTF-8 LF {} TypeScript
⊗ 0 ⚠ 0 Collaborate

```
const mySound:Sound = { rate: 2, samples: [1, 2, 3, 4], }
const coolNewSound:Sound = audioPlayground(mySound);
console.log("A cool new sound:", coolNewSound);

// expecting: [8, 6, 4, 2, 1, 2, 3, 4]
assert (coolNewSound.samples[0] === 4*2) // test the first s
assert (coolNewSound.samples[7] === 4) // test the final sam
```

```
/Users/gecanow/Desktop/up-research/.praxistutor/src/Audio.ts:35
assert (coolNewSound.samples[7] === 4) // test the final sample for c
orrectness
      ^
AssertionError [ERR_ASSERTION]: The expression evaluated to a falsy v
alue:

  (0, assert_1.default)(coolNewSound.samples[7] === 4)

    at Object.<anonymous> (/Users/gecanow/Desktop/up-research/.praxis
tutor/src/Audio.ts:35:7)
    at Module._compile (node:internal/modules/cjs/loader:1218:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loade
r:1272:10)
    at Module.load (node:internal/modules/cjs/loader:1081:32)
    at Function.Module._load (node:internal/modules/cjs/loader:922:12
)
    at Function.executeUserEntryPoint [as runMain] (node:internal/mod
ules/run_main:81:12)
    at node:internal/main/run_main_module:23:47
gecanow at MacBook-Pro-321 in ~/Desktop/up-research/.praxistutor
○ ○ ▯
```

a runtime error.

Afterwards, in the VS Code terminal, you should see a *stack trace* with hyperlinks to locations in the code on the terminal.

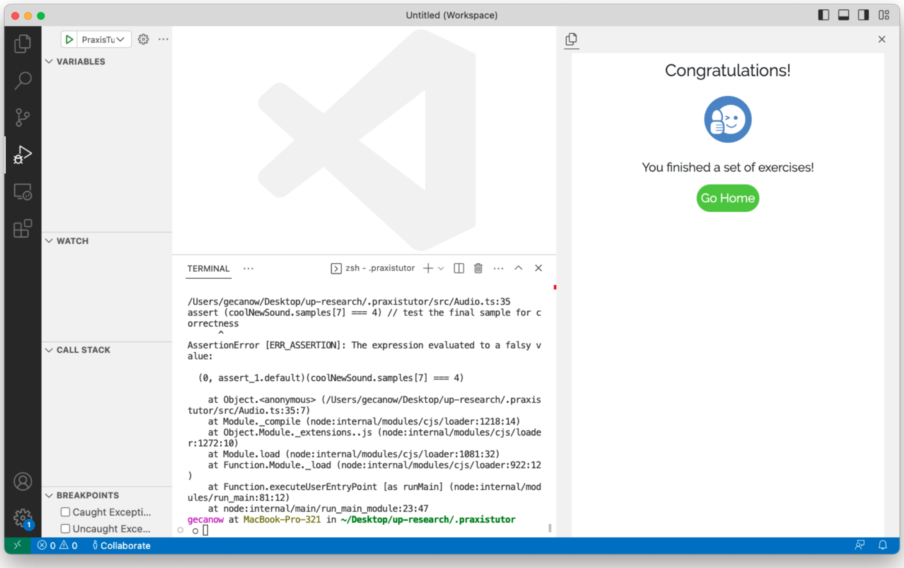
Your task is to `Ctrl+click` (Windows/Linux) / `Cmd+click` (Mac) the link to the line in Audio.ts where the assert statement failed.
Your cursor will automatically move to the linked line.

Once you have clicked the link, press the Check button in the lower right to move on.

You got it right!

Stuck? Maybe these will help.

Visual Studio Code Terminal Links (VS Code Docs)



```
/Users/gecanow/Desktop/up-research/.praxistutor/src/Audio.ts:35
assert (coolNewSound.samples[7] === 4) // test the final sample for c
orrectness
      ^
AssertionError [ERR_ASSERTION]: The expression evaluated to a falsy v
alue:

  (0, assert_1.default)(coolNewSound.samples[7] === 4)

    at Object.<anonymous> (/Users/gecanow/Desktop/up-research/.praxis
tutor/src/Audio.ts:35:7)
    at Module._compile (node:internal/modules/cjs/loader:1218:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loade
r:1272:10)
    at Module.load (node:internal/modules/cjs/loader:1081:32)
    at Function.Module._load (node:internal/modules/cjs/loader:922:12
)
    at Function.executeUserEntryPoint [as runMain] (node:internal/mod
ules/run_main:81:12)
    at node:internal/main/run_main_module:23:47
gecanow at MacBook-Pro-321 in ~/Desktop/up-research/.praxistutor
○ ○ ▯
```

Congratulations!

You finished a set of exercises!

Figure 3-7: On A Correct Submission

3.4 It's Time for an Update

Aside from the Effective Print Probes and Print Debug Strategies concept groups, all exercises necessitated significant updates to the Praxis Tutor framework. The new debug exercises require compiling and executing files, launching debug sessions, and checking ordered user events rather than just a final state. And, unlike language tutors, the Debug Tutor needs access to event sources such as the terminal and the debugger. Exercise authors, meanwhile, need new and flexible models to describe exercise configurations and solution event patterns. The web user interface needs to enable automatic set up for some exercises, as well as forced start over for others. In all, to support the debug concept map and updated user interface, the Praxis architecture required significant augmentations across its software stack. The following chapter details such implementation updates.

Chapter 4

Implementation

The Praxis Tutor framework has three main parts: (1) The host extension, (2) the Tutor web application (the webapp), and (3) the server. Figure 4-1 shows the Tutor architecture.

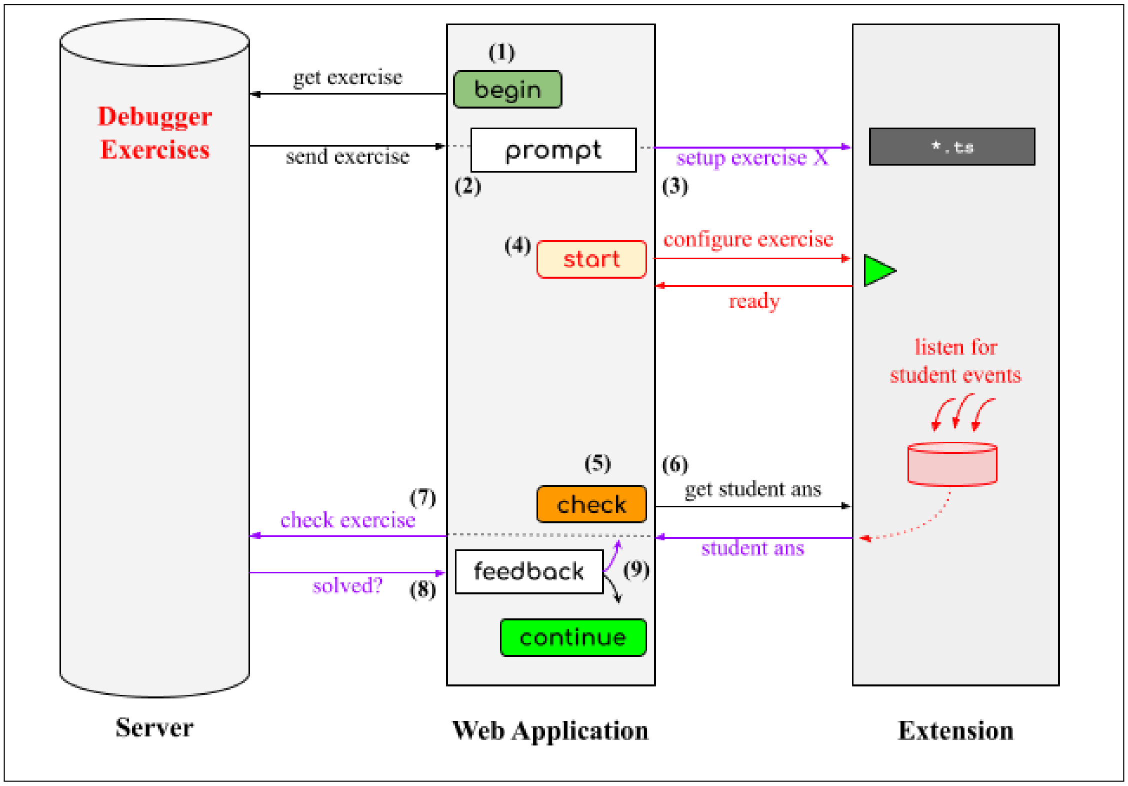


Figure 4-1: The Debug Tutor (Praxis) Architecture.

In Figure 4-1 time progresses downwards. (1) When a student clicks on a concept group in the webapp, the webapp requests an exercise from the database. (2) Upon receiving the exercise, the webapp displays the prompt and (3) requests that the extension perform any necessary setup for the webapp. (4) If an exercise requires automatic configuration, such as launching a debug session, the webapp displays a Start button to trigger setup. The host extension informs the webapp when it is done setting up, at which point (5) the webapp enables the Check button. If the exercise includes event tracking, the extension observes and logs relevant student actions. When the student presses Check, (6) the webapp requests the student's answer from the extension to (7) compare with the solution pattern stored on the database. (8) The webapp computes the results and displays the feedback, at which point (9) if incorrect, the student must try again. Otherwise, the student may continue on to the next exercise.

The exercises are described in the Praxis codebase using commented code files that exercise authors upload to the database. The server stores the uploaded exercises in a Mongo[1] database and also performs the solution checking.

The web application is the main driver of the Praxis framework. It communicates with the server and extension host to request exercises for display and to check if a student's answer is correct. It is displayed inside a window of the extension host. Through this web view, the student can interact with the web app directly, reading exercise descriptions, requesting hints, clicking buttons, receiving feedback, and generally progressing through exercises.

The extension is a lightweight plugin to a host development environment, such as the VS Code editor. The extension displays the webapp in a web view and makes changes to the host on behalf of the webapp displaying code files in the editor and collecting answer submissions. New for the Debug Tutor, the extension also listens to debugger events (such as a student setting a breakpoint, starting a debug session, or stepping through code) and terminal events (such as a student clicking on a terminal link) and forwards those events to the web app for handling.

In figure 4-1, elements drawn in red describe new functionality added for the

[1] https://www.mongodb.com

Debug Tutor. Elements drawn in purple, on the other hand, were modified to a large extent to support new interactions required for the debugging exercises. The following subsections discuss these implementation changes and updates to the exercises, the server, the webapp, and the extension.

4.1 History Event Patterns

Perhaps the biggest change to the Tutor architecture was the new requirement of tracking events during exercises rather than just capturing state. To do so, exercise authors needed a way to describe solution event patterns which could be parsed and stored in the server. That same scheme would then serve as the format by which the extension could log observed events to be sent to the server for checking via the webapp. Indeed, the format used to describe event patterns motivated many of the changes made across the Praxis architecture for the Debug Tutor.

4.1.1 Authoring Exercises with History Events

The design of the event pattern format had to be flexible enough to allow for a range of acceptable behaviors from students and on-the-fly updates by exercise authors, yet granular enough to ensure the skill being taught is sufficiently practiced. Because exercise concepts span a range of event sources, such as the terminal and the debugger, the pattern format needed to account for different streams from which events might be emitted. Within a stream, individual events can be identified by a name, but may also require matching against specific data values. For example, within a solution pattern, an exercise author may want to use a set breakpoint event on a specific line.

To capture all of the necessary information in a way still adaptable to any potential event, the history patterns are described using a flexible structure of the form `[rs]/stream/ [{event1} {event2}...{eventN}] ({eventA} {eventB} ...)`.

Streams The `stream` is the source of metadata to look at. Current concrete options include debugger, terminal, and editor. A preceding `s` marks an exact match, whereas a preceding `r` marks a regular expression (regex) that may match to multiple streams. For example, `r/debugger|terminal/` will pattern match with events in both the debugger and terminal streams.

Then, an ordered event list is demarcated with square brackets. These events define the history solution pattern that a student event log must match against to complete the exercise successfully. A second optional unordered event list is demarcated with parentheses. These events are individually allowable at any time. For example, a debugger exercise author may want to allow a student to evaluate anything at any point during the run of the exercise, in which case the pattern might look something like `s/debugger/ [{debugEvent1}...{debugEventN}] ({evaluateEvent})`, where *debugEvent* and *evaluateEvent* are patterns that match to some debug event and some evaluate event, respectively.

Events Each `{event}` takes the form `{[rs]/eventName/ [dataKey1:dataValue1, dataKey2:dataValue2, ...]}`. `eventName` is the concrete event to match, such as `setBreakpoint`. Like streams, events can also be described with a more general regex, such as `r/step.*/` to match `stepIn`, `stepOut`, or `stepOver`.

`dataKey`s can be strings (e.g. `s/f/` or `f` to match “f”), regexes (e.g. `r/f|g/`), or a number (e.g. `46`). Any of these can be negated—that is, matched to anything *but* the value—by sticking a tilde in front (e.g. `∼r/f|g/` to match anything but “f” or “g”, or `∼/46/` to match any number but 46).

`dataValue`s, like data keys, can be strings, regexes, or numbers, as well as code segments marked with a c (e.g. `c/let x =/`) or expressions marked with an e (e.g. `e/{radius:10, x:5, y:3}/`). Code segments are converted to their corresponding line number which may be useful, in, say, a set breakpoint event. Expressions are unique in that string transformers, such as replacing all double quotes with single quotes to standardize quote usage, are applied to both it and the submitted answer before comparison. For example, `s/debugger/ [{s/continue/ [stopReason: hitBreakpoint], line:c/let x =/}]` matches any single `continue` event that halts due to hitting a breakpoint on the line where `x` is initialized. Code segments and expressions may also be negated with a tilde (`~c/../` or `~e/../`).

Each event may entirely be negated with a tilde before the curly braces (i.e. `~{event}`). In that case, any event *except* those matching `{event}` will match `~{event}` . Events can also be repeated a certain number of times: `{event}!` must be matched exactly once, `{event}?` can be matched zero or one time, `{event}*` can be repeatedly matched zero or more times, and `{event}+` can be repeatedly matched one or more times. Events without a suffix default to the exclamation, so are matched exactly once. Exercise authors can use `{}` to match any event, or `{}*` to match any sequence of any events. Finally, if data is left out of an event description, it is assumed to match any values. For example, `{s/didEdit/ []}` without an `onFile` key matches a `didEdit` event on any file.

Anchoring An event stream is assumed to be *unanchored*, that is, a match to any subsequence of a student's answer is considered a successful match. Exercise authors can anchor the least recent event with a caret in front (e.g. `^[...]`) and can anchor the most recent event with a dollar sign in back (e.g. `[...]$`).

Examples Table 4.1 shows two example history streams taken from deployed debugger exercises, while Table 4.2 for a full list of supported events, along with their respective data keys, per each event stream.

Table 4.1: Example Debugger Event History Patterns

Example #1

```
s/debugger/ [
     {r/setBreakpoint/ [line:c/let x/]}!
     ~{r/(set|remove)Breakpoint/ [line:c/let x/]}*
]$
```

Searches in the debugger stream and matches when a student sets a breakpoint on the line containing the substring let x and then does anything but set or remove a breakpoint on that line for the rest of the stream.

Example #2

```
r/debugger/ [
     {s/launchSession/ [configurationName:r/PraxisTutor/,
                        stopReason:s/breakpoint/,
                        stopsOnLine:c/dotProduct/]},
     {}*,
     {r/step*|continue/ [stopsOnLine:c//*[1]*/assert/]},
     {s/setBreakpoint/ [line:c/return dotProd/]}?,
     {s/continue/ [stopsOnLine:c/return dotProd/]},
     {s/evaluateOnDebugConsole/ [expression:dotProd, resultingIn:9]}
]$ ({r/evaluate.*/ []})
```

Searches in the debugger stream and matches when a student launches a `PraxisTutor` debug session that pauses on the line with `dotProduct`, then does anything, eventually stepping or continuing to land on the assert statement with `/*[1]*/`[a], then optionally sets a breakpoint on `return dotProd`, continues to that breakpoint, and evaluates `dotProd` with the debug console. At any point, the student may evaluate any expression (this is specified by the final evaluate event in the parenthesis)—these extraneous evaluations will be ignored.

[a] Concretely, each pattern is parsed in multiple rounds of regular expression matching in order to recognize data values that contain forward slashes. First, all key, value pairs are captured by detecting space- or comma-separated sequences of the form key:value. Then, each value is parsed by either removing the leading character plus the immediate following forward slash and final trailing forward slash if *both* exist, or else is kept in whole.

Table 4.2: Concrete Pattern Streams & Events

Debugger	
Events	**Data**
setBreakpoint, removeBreakpoint	**line**:number **character**:number
launchSession	**configurationName**:string **stopReason**:step|hitBreakpoint|disconnected|ongoing[a] **stopsOnLine**:number
disconnect	
stepIn, stepOver, stepOut, continue	**stopReason**:step|hitBreakpoint|disconnected|ongoing[a] **stopsOnLine**:number **variable-**X[b]:string
evaluateOnDebugConsole, evaluateWatch	**expression**:string **resultingIn**:string

Editor	
Events	**Data**
didEdit	**onFile**:string
movedCursor	**onFile**:string **anchorLine**:number **anchorCharacter**:number **activeLine**:number **activeCharacter**:number

Terminal	
Events	**Data**
clickCodeLine	**inFile**:string **line**:number **character**:number

[a] The default **stopReason** is *ongoing*, meaning the action is still ongoing.

[b] Captures a variable in the scope after the event, e.g. `s/stepOver/ [variable-i:10]` matches when, after stepping over, the variable `i` equals 10.

[c] Note: If data is left out of the history stream, it is assumed to match any value. For example, `{s/didEdit/ []}` will match a didEdit event on any file.

History Pattern Matching

For those familiar with regular expressions, the history stream expression language hopefully felt similar. In fact, the design borrows many of the basic operators and features from regular expressions, such as the ?, *, and + repetition characters, ∼ negation, and anchoring. Viewing the history stream language as regular language, then, the alphabet of this language, rather than mere characters, are objects with a type and data dictionary. As such, the Tutor's history pattern format necessitated a custom regular *object* matching algorithm, or RegObj matcher for short, to compare the collected student event logs against the exercise solution patterns.

The RegObj matcher uses a recursive backtracking search algorithm to match the sequence in full. To compare single event objects, the matcher uses a heavily modified version of the `node-matchr`[2] library. Algorithm 4-2 describes the RegObj matching algorithm at a high level. `executeMatchHistory` is the entry point, with `continueMatching` and `compare` defined as closures inside.

Figure 4-2: Regular Object Matching Algorithm

Algorithm 4-2: Regular Object Matching

```
procedure EXECUTEMATCHHISTORY(value:any[],
                              pattern:ModelHistoryEvent[])
    $candidate \leftarrow value[0]$
    $event \leftarrow pattern[0]$

    if candidate & event are null then return NullMatch
    else if event is null then
        if anchorLeastRecentEvent then return FailedMatch
        else return NullMatch
        end if
    else if candidate is null then return continueMatching(False)
    else return continueMatching(compare(candidate, event))
    end if
end procedure
```

[2] Allows free commercial and private use, modification, and distribution under the MIT Software License. See: https://github.com/moeriki/node-matchr.

Clients access the history matcher via a static public method that consumes a history pattern and returns a matcher on it to use against value arrays, e.g.:

```
// for pattern:HistoryEvent[],
// and options:{anchorMostRecentEvent:boolean,
//              anchorLeastRecentEvent:boolean,
//              transformers: Transformer[],
//              allowAnywere: HistoryEvent[]}
const matcher = RegObj.compileHistory(pattern, options);

// with value: any[]
const matchResult = matcher(value);
```

Both the history pattern and the value arrays are assumed to be ordered least recent event to most recent event. If *anchorMostRecentEvent* is true, the matching begins at the value's tail and returns false if no match is found. Otherwise, the matching tries successively popping off the tail until there are no more events to try.

4.1.2 Adopting Event Patterns in the Tutor

With the event history format defined and a complementary pattern checker implemented, the next step was integrating them into the Praxis Tutor architecture. Since the server hosts both the exercise models and the answer checker, the server, perhaps ironically, was the main client adopting these two updates.

Algorithm 4-2, Continued: Regular Object Matching

```
procedure COMPARE($candidate$:any, $event$:HistoryEvent)
    $typeMatch \leftarrow$ regexMatch($candidate.type, event.type$)
    $dataMatch \leftarrow$ objMatch($candidate.data, event.data$)
     return $typeMatch$ && $dataMatch$
end procedure

procedure CONTINUEMATCHING($headsMatch$:boolean)
    if invert then
        $headsMatch \leftarrow !headsMatch$
    end if

    if headsMatch then
        $matchRest \leftarrow$ executeMatchHistory($value[1:], pattern[1:]$)
        if $matchRest$ is a match then return Match
        end if
    end if

    if $any$(compare($candidate, event$)$\forall$event $\in$ allowAnywhere) then
        $matchSkip \leftarrow$ executeMatchHistory($value[1:], pattern$)
        if $matchSkip$ is a match then return Match
        end if
    end if

    if repeat is ! then return FailedMatch
    else if repeat is ? then return executeMatchHistory($value, pattern[1:]$)
    else if repeat is + then
        if ! headsMatch then return FailedMatch
        else  return executeMatchHistory($value, [event] + pattern$)
        end if
    else                                                        ▷ repeat is $*$
        if headsMatch then
            $consumeMore \leftarrow$ executeMatchHistory($value[1:], pattern$)
            if $consumeMore$ then return Match
            end if
        end if
     return executeMatchHistory($value, pattern[1:]$)
    end if
end procedure
```

4.2 The Server

Before the Tutor can be deployed to students, an exercise author must first write the concrete concept map and exercises to load into the server. The concept map and exercises are defined by YAML[3] descriptions and parsed into object models for storage.

4.2.1 Concrete Exercises

Each exercise is uniquely defined by a YAML description written in a comment at the top of a code file, within which history event patterns can be specified. For example, a debugger exercise YAML description at the top of a code file might look like:

```
//<yaml>
//  - id: Matrix-step-in
//    conceptIds:
//      - breakpoints::advanced-drive::step-in
//    prompts:
//      - >-
//        First, click Start and watch carefully as a breakpoint is
//        set at <code>column(j,m2)</code> in <code>multiply</code>.
//        <br><br>
//        Afterwards, your task is to:
//        <br>(1) Launch a PraxisTutor debug session;
//        <br>(2) Use a sequence of steps <i>in</i> to pause the
//                sdebug session inside <code>makeMatrix</code>.
//        <br> At that point, press Check to continue.
//
//    priority: 0
//    explanation: ""
//    configuration:
//          readonly: true
//          debugger:
//              breakpoints:
//                  - code: column(j,m2)
```

[3] YAML is a data serialization language similar to JSON

```
//    solutionMetadata:
//          history:
//              - r/debugger/ [
//                 {s/launchSession/ [
//                       configurationName:r/PraxisTutor/,
//                       stopReason:s/breakpoint/,
//                       stopsOnLine:c/column(j,m2)/
//                  ]},
//                 {s/stepIn/ []}*,
//                 {s/stepIn/ [stopReason:s/step/,
//                             stopsOnLine:c/const mat/*[1]*//]}
//                ]$
//          hints:
//              - pattern: r/debugger/ [{s/stepOver|continue/ []}*]$
//                hint: >-
//                      In this exercise, do not step over or
//                      continue.  Instead, <i>step in</i> by
//                      clicking the step into button on the debug
//                      toolbar
//                      (<i class="codicon codicon-debug-step-into"
//                       style="color:cornflowerblue"></i>).
//</yaml>
[exercise code omitted]
```

Because different exercises may use the same code file, multiple exercises can be described within a single YAML block. On upload, each unique exercise in every YAML comment is parsed into an `Exercise` object for storage in the server. The Praxis framework previously supported the following YAML keys:

- **id**: A unique exercise ID.
- **conceptIds**: One or more concept IDs to which this exercise belongs.
- **prompts**: HTML to display in the webapp, e.g. the exercise task. The first prompt describes the exercise task, while the remaining prompts are hints.
- **priority**: Index where the exercise is displayed.

- **explanation**: An explanation of the solution.
- **blanks**: One or more parts of the code file to replace with blanks (e.g., for `x = 5 + y;`, a blank at `5 + y` would result in the code file displaying `x = ......;`).

For the debugging exercises, the following keys were added to the YAML:

- **configuration**: Describes file setup, including:
  - **readonly**: If true, requires that no edits are made to the file during the duration of the exercise, else the attempt is marked automatically incorrect.
  - **autoreset**: If true, requires that the exercise be started over after an incorrect answer is submitted.
  - **debugger**: Describes debugger-related setup, including:
    - **breakpoints**: A list of key,value pairs specifying either the line and character or matching code in the exercise to set a breakpoint on, e.g.:
      - `line: 3` will place a breakpoint on line 3, after the YAML comment has been removed.
      - `code: let x =` will place a breakpoint on the first line in the code containing the substring "let x =".
    - **preLaunchedSession**: Describes a debug session to auto launch, with keys:
      - **name**: The name of the launch configuration, e.g. `PraxisTutor-DebugCurrentFile`. The named launch configuration must be present in the exercise directory's `.vscode/launch.json`.
      - **requests**: Optional, a list of requests to run, in order, such as `stepIn`.
  - **terminal**: Describes terminal-related setup, with keys:
    - **preRunCommand**: A command to run on the terminal, e.g. `npm run Rainfalls`. The Tutor will first attempt to `cd` into the Praxis

tutor code file directory. The run command must be present in the exercise directory's `package.json`.

- **solutionMetadata**: Describes the solution event history and state of the code file, including:
  - **configuration**: The same model as described above, except used instead to verify the solution configuration. Using the same model allows the **solutionMetadata** to be ready for updates as new post-exercise solution configurations are defined, such as if a new terminal state was added. However, when parsing **configuration** for checking exercises, keys relevant only to setup are ignored (e.g., **readonly**, **autoreset**, and **preLaunchedSession**).
  - **history**: A list of event patterns to match the student answer against.
  - **historyAnswer**: Optional, a list of strings describing how to perform the solution histories. If not provided, an automatic solution answer string will be generated from each history.
  - **hints**: A list of potential triggered hints to display when the student checks an incorrect answer, each with:
    - **pattern**: An event history pattern.
    - **hint**: An HTML hint to display when the pattern is matched.
  - **warnings**: A list of potential triggered warnings to display when the student checks a correct but non-expert answer. Similar to **hints**, each contains an event history **pattern** and an HTML **hint** string.
  - **transformers**: A list of string transforms of the form `r/find/replace/`, specific to this exercise.

All new keys are optional for backwards compatibility. The original **blanks** key requirement was relaxed for forward compatibility. Additionally, Microsoft's Codicon library was added to the Tutor webapp resources, which exercise authors can make use of in any HTML block.

An Updated Exercise Model

To support the new **configuration** and **solutionMetadata** keys, the `Exercise` model stores two new properties: a `Configuration` object and a `SolutionMetadata` object. The `Configuration` object includes `Debugger` and `Terminal` objects, among other properties, and the `SolutionMetadata` object includes a list of `History` objects (for the **history** key) and two lists of `HistoryHint` objects (for **hints** and **warnings**). Most importantly, the `History` object includes an **eventstream** property, an **events** property and an **allowAnywhere** property each listing `Event` objects, and boolean flags for anchoring. Finally, the `Event` object stores a **type**, **data**, **repeat**, **invert**, and **id**.

The `Exercise` model also has new *computed* properties—properties computed based on the YAML description but not explicitly listed therein—including:

- **solutionHistoryTemplates**: A list of all solution, hint, and warning history patterns stripped of solution data. For example, the pattern `s/debugger/ [{r/setBreakpoint/ [line:c/let x/, character:0]}]` would have the template `s/debugger/ [{r/setBreakpoint/ [line, character]}]`.

- **requiresSetup**: A boolean set to true if and only if the `Configuration` object has any keys matching an event stream (concretely, debugger, terminal, or editor), therefore implying that the exercise has requires automatic setup, such as an automatically set breakpoint.

- **requiresLaunchConfiguration**: A boolean set to true if and only if the exercise requires the launch configuration boilerplate file. This is determined by if the *Configuration* explicitly calls for automatically pre-launching a debug session, or if any history pattern in the `SolutionMetadata` includes any events that require debugger functionality (single stepping, evaluation, etc.).

4.2.2 A Concrete Concept Map

Like individual exercises, Praxis Tutor concept maps are also realized with YAML description files and boilerplate project configuration files. The entry point into the concept map is the file `exercises.yaml`, which describes the layout of the concepts and where to find the exercises. Because the `exercises.yaml` file keys already in use for the TypeScript Tutor were fully capable of incorporating the debug exercises, the `ConceptMap`, `ConceptGroup`, and `Concept` models required no format updates.

The `exercises.yaml` file includes the following keys:

- **exerciseFileFormat**: Describes where to find exercise descriptions. In the case of the Debug Tutor TypeScript files, `src/*.ts` (i.e., files in the `src` subdirectory with a `.ts` file format)

- **boilerplateFiles**: Lists project boilerplate files, such as a `package.json`.

- **languages**: Lists each supported language (such as `java`). For each language, lists global **transformers** (of the form `r/find/replace/`) to apply when checking student answer code against solution code (e.g. `r/’/”/` to standardize all quotes).

- **conceptMap**: Defines the concept map. Relevant keys include:
  - **id**: Concept map identifier.
  - **numberToMaster**: The number of exercises required to pass for each concept difficulty.
  - **levels**: Lists the concept groups in each level. Each level includes:
    - **id**: Concept group id.
    - **uiLabel**: Concept group label to display in the webapp.
    - **conceptIds**: Lists each concept id in this group.

- **concepts**: Lists all concept ids. Under each concept id, a **uiLabel** is displayed in the webapp and an optional **numberToMaster** gives the number of concrete exercises required to complete the concept (defaults to the conceptMap's **numberToMaster** key).

- **tutorials**: List of online tutorials. Each tutorial includes a **url**, **title**, a list of **conceptIds** for which this tutorial is helpful, a **type** (e.g. webpage), a **source**, and an ok/good/great **rating**.

Boilerplate files are specific to the extension host and language used. For TypeScript, the Tutor includes a `package.json` file and a `tsconfig.json` file, two standard configuration files. Aside from some minor updates to these files to support debugging, the Debug Tutor also required the addition of `.vscode` workspace setting files, for storing a VS Code debugger `launch.json` configuration file and a `settings.json` file to control what is displayed in VS Code's file Explorer pane.

Debug Boilerplate Files

To launch debug sessions in VS Code, an opened workspace needs a launch configuration.[4] Launch configurations are kept in a `launch.json` file inside a `.vscode` directory in the workspace. Exercise authors can customize launch configurations for different languages and with different settings. To support the Debug Tutor on TypeScript files, `launch.json` lists a *PraxisTutor-DebugCurrentFile* configuration for launching sessions with the standard JavaScript node debugger. VS Code also supports a `settings.json` file in the `.vscode` directory. The Debug Tutor boilerplate `settings.json` file is used to exclude the `.praxistutor` folder in the Open Editors pane of the Explorer tab. For VS Code to recognize the `.vscode` configuration files, the parent directory (`.praxistutor`) must be opened as a workspace. The webapp and extension handle inferring when the `.praxistutor` folder is required and automatically opening it as a workspace.

[4] https://code.visualstudio.com/docs/editor/debugging#_launch-configurations

4.2.3 Uploading to the Server

The concept map and suite of exercises are written locally and are uploaded to the server by the exercise author. When uploaded, the concept map is interpreted as object models representing a `ConceptMap` of `ConceptGroups`, each with `Concepts`, and the exercises are interpretted as `Exercise` models. It is these objects specifically that are stored on the server in the Mongo database.

At the start of each exercise, the server then is responsible for sending the exercise object models to the webapp upon request, which then parses the relevant information to share with the extension. Like the `Exercise` model, the server's answer checker, responsible for checking student submissions, required major updates to support the new solution models and submission structure of the Debug Tutor, specifically by adopting the new history pattern RegObj matcher.

4.2.4 Answer Checking 2.0

Originally, the Praxis Tutor framework's solution checking algorithm only compared filled-in blanks in the student's code file at the time of submission to the corresponding code segments in the exercise description. The checker first repeatedly applied any global and local string transformers to both the student's submission segments and the original code segments until the transformations were no longer productive and then compared the final canonical versions. On a match, the server marked the submission as correct and then compared the answer to each warning, appending the warning's hint to the output if its pattern matched the answer submission. On a mismatch, the server marked the submission as incorrect and then compared the answer to each triggered hint, likewise appending the triggered hint's hint to the output if its pattern matched the answer submission.

Beyond comparing the state of code, many Debug Tutor exercises require comparing the state of the IDE. *Are there breakpoints on the intended lines?* Other exercises require comparing students' event histories to the solution event history. *Did the student execute the right events in the right order?*

Adding in IDE state comparisons was somewhat trivial; The solution checker now accepts an optional final **config** submission object. If the `Exercise`'s `SolutionMetadata` contains a `Configuration` object, it is compared to the final **config** submission. If there's a mismatch, an automatic triggered hint is generated based on the difference between the two configuration objects.

Comparing event histories, on the other hand, required adopting the RegObj matcher. With the new history matcher, the answer checking proceeds with checking blanks (if applicable), then checking configurations (if applicable), and finally checking histories (if applicable), and only returns that the submission is correct if all relevant checks pass, thus completing the server updates to support the Debug Tutor.

Checking answers on the server is not only useful because it is where the exercise descriptions are stored, but it also keeps the solutions private. Because solutions are never sent to the webapp or extension, a crafty student cannot observe solution data in the communication between these modules, nor on any student-facing frontend code. The server acts as a firewall, protecting the solutions, ingesting student submissions, and only outputting relevant, user-facing feedback.

4.3 The Web Application

The webapp is the main interface through which the student interacts with the Tutor. It drives Praxis Tutor's *event loop*—the driver of change during the flow of a user interaction—and acts as the link between the server and the host extension. As such, the webapp was updated to accommodate the updates to the event loop and in communication with the server and extension.

The webapp communicates with the server through POST and GET requests. To support the updates to the stored exercise model, the webapp was updated to unwrap optional exercise configuration properties on an exercise request. To support the new answer checker, the webapp was likewise updated to fold the new optional configuration properties into the check answer request sent to the server. The new properties are optional for backwards compatibility. Table 4.3 lists the webapp-to-server routes updated for the Debug Tutor.

Table 4.3: Webapp-Server Routes

Route	Specification
`GET /exercise/:exerciseID`	Returns a JSON describing the exercise object with ID `exerciseID`. Updated to include the *configuration*, the booleans *requiresSetup* and *requiresLaunchConfiguration*, *solutionHistoryTemplates*, and *realtimehintEventTriggers*.
`POST /exercise/:exerciseID`	Checks a student's submission to the exercise with ID `exerciseID`. The request body was updated to include *studentAnswerMetadata*, which includes the file configuration and event histories, among other things.

The webapp communicates with the host extension via a *plugin API*. In the case of VS Code, the API is activated by VS Code's Webview API.[5] Table 4.4 lists the updates and additions made to the plugin API, with specific changes highlighted in orange.

[5] https://code.visualstudio.com/api/extension-guides/webview

Table 4.4: Webapp-Host Plugin API

Plugin API Method	Specification

`hostStartExercise(`
`exerciseID:string,`
`zipFile:string,`
`blankChar?:string,`
`solutionHistoryTemplates?:`
`HistoryTemplateType,`
`requiresLaunchConfiguration?:`
`boolean,`
`clientState?:any,`
`):void;`
	Asks the extension host to begin the exercise with ID `exerciseID`.
	Updated to include *solutionHistoryTemplates*, an optional list of all event history patterns stripped of solution data, *requiresLaunchConfiguration*, a boolean specifying if the exercise requires a launch configuration (e.g. for debugging), and *clientState*, a package of the necessary information to restart from this point, e.g. if the extensions needs to reload.

`hostGetStudentAnswer(`
`exerciseID:string,`
`callback:(result:string[])`
`->void`
`):void;`
	Fetches the student answer submission from the extension host and passes it to `callback`.
	Updated to include solution metadata in the `result` array.

`hostSetupExercise(`
`exerciseID:string,`
`exerciseConfig:Configuration,`
`callback:(void)->void`
`):void;`
	New; Requests the extension host to set up the exercise with ID `exerciseID` according to the `exerciseConfig`. The `callback` is invoked when the set up completes.

Continuation of Table 4.4

Plugin API Method	Specification

`hostCloseProject(`
`  didQuit:boolean=false,`
`):void;`
	Request the extension host to close the displayed exercise.
	Updated to include the `didQuit` parameter to differentiate between progressing to the next exercise and quitting. If true, removes event-tracking listeners, cleans up file configurations, and stops exercise-related event tracking.

`hostGetRecoverExerciseState(`
`  callback:(clientState:any)`
`        ->void`
`):void;`
	New; Requests from the host any saved exercise state, e.g., to restore the Tutor from that state.

Figure 4-3 presents a state diagram of the webapp event loop. Changes to the webapp are drawn in orange. The webapp first displays the Home UI with the full concept map in view (label A). When an unlocked concept group is clicked, the webapp requests exercises from the server and asks the host to display the first one (label B). Originally, the first state was always Ready for Interaction UI (following the purple arrow). But new for debugging, if the exercise requires automatic configuration, the webapp will instead progress to the Setup Required UI (following the orange arrow). The student must then click Start to progress to the Ready for Interaction UI. The student can then attempt the exercise. At any point, if Start Over is clicked, the webapp will ask the host to reset the exercise to the original state. When ready to submit, the student will click Check, and the webapp sends the submission to the server for checking. If incorrect (label C), the original webapp added informative text but stayed in the same Submission Ready UI. New for debugging, if the exercise requires resetting, the webapp will instead display the Start Over Required

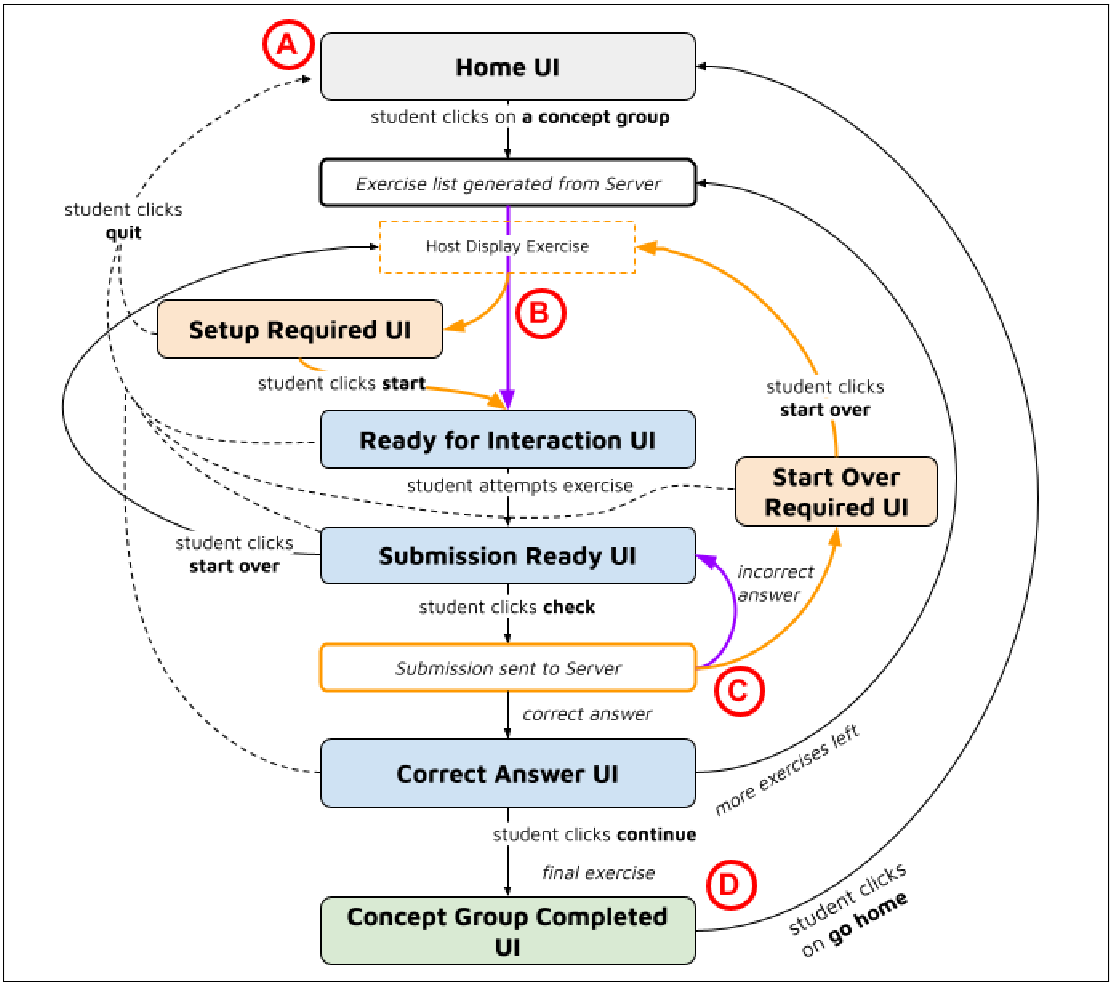


Figure 4-3: The Praxis Tutor Webapp Event Loop

UI, in which the only option is to click start over. If correct, the only offered option is to click Continue. At that point, if there are still exercises left in the group, the webapp will ask the host to display the next exercise. Otherwise, the webapp enters the Concept Group Complete UI (the green box at label D), and offers a Go Home button to return to the Home UI.

`index.js`, the web entry point that implements the event loop and makes calls to the server routes and plugin API, was expanded significantly to support the major changes described and displayed in orange on the state diagram.

Keeping Extensibility In Mind The updates to the exercise description models, server answer checker, and webapp were purposely implemented with extensibility in mind. The exercise Configuration model is intended to be able to support new and unforeseen configuration states beyond just the debugger and the terminal. Event history descriptions and submission checking is likewise ready to extend to any potential event stream, with a variety of potential data keys and values. The webapp simply acts as the event loop driver, irrespective of the exercise details. The changes introduced to the extension, detailed in the next section, were designed with the same extensibility in mind.

4.4 The Extension

Finally, the extension plugin required major updates to support event tracking. Not lost on a debugging education system, the updates take full advantage of static checking by introducing an API for standardizing event tracking across the Praxis software stack. All in all, these implementation updates were built with the three pillars of 6.102—safety from bugs, ease of understanding, and readiness for change—at heart.

4.4.1 Event Tracking API

Inside the event tracking API, individual concrete streams must declare a namespace exporting an enumeration of its concrete events and a type describing an event object. For example, Figure 4-4 shows the editor namespace.

The core of the event tracking API is the HostIDE namespace, which consolidates the information in each tracker namespace and holds other general event tracking types for use in the extension. The HostIDE is exposed to the webapp and server, which can type inputs and outputs accordingly. Table 4.5 lists the exports from the HostIDE namespace.

The API exports a few other useful functions for use across the extension and server and finally defines an interface that every concrete tracker must implement. Finally, Table 4.2, presented during the introduction of the history event pattern language, lists the concrete tracked streams, events, and data values currently defined in and exported by the HostIDE.

Table 4.5: Event Tracking API's `HostIDE` Exports

Enumeration	**Description**
`IDEEventSource`	An enum identifying the source of the event, which is either `User`, i.e. student-initiated, or `System`.
`IDEEventStream`	An enum of trackable stream types that an event may belong to, appearing as `metatypes` in each stream namespace. All history pattern streams and configuration object keys must match to an `IDEEventStream`, else the extension will have no way to relate those to their concrete counterparts. Currently includes *debugger*, *editor*, and *terminal*.

Constant	**Description**
`ideEventStreams`	A string array of all `IDEEventStreams`. `EXTENSION_EVENTS`, a constant object mapping each event stream type to a list of its concrete enum event types.
`EXTENSION_EVENTS`	A constant object mapping each event stream type to a list of its concrete enum event types.
`allEventTypes`	A string array concatenation of all concrete events across all streams.

Type	**Description**
`TutorEvent`	A union event type of all stream-specific trackable event types.
`MetadataType`	For storing tracking information about an exercise, with properties `eventLog`, `fileVersions`, `historyTemplates`, and `didOpenTutorWorkspace`.
`freshMetadata`	A function that generates an fresh object of type `MetadataType`.

Log Collection Setting	**Description**
`LOG_COLLECTION_OPTIONS`	For storing options related to collecting and storing logs, such as `shouldKeep`, listing data properties that should be kept regardless of if it appears in any templates.
`IS_STAFF`	A boolean marking if this user is staff or not.

```typescript
/**
 * A namespace for tracking events in the editor.
 */
export namespace IDEEditorTracking {

    export enum IDEEditorEvent {
        didEdit = "didEdit",
        movedCursor = "movedCursor"
    }

    export type TutorEditorEvent = {
        timestamp: number,
        metatype: "editor",
        type: string,
        data: any,
        source: HostIDE.IDEEventSource,
    }
}
```

Figure 4-4: The Editor Namespace in `event-tracking-api.ts`

The Tutor Tracker Interface The Tutor Tracker interface has five methods: `configure`, to set up the state according to a configuration object, `clean`, to clear out any configuration that may have been set, `compileConfiguration`, to package the current configuration, `register`, to set up and begin tracking, and `teardown`, to end tracking and dispose of internal listeners. Figure 4-5 displays the interface in full. With the Tutor Tracker interface, plugging new trackers into the extension is seamless, because new trackers need only implement the interface and be declared in the `HostIDE` to be included in the extension's tracking.

```
// An interface for defining an event tracker that can be
// automatically incorporated into the tutor.
export interface TutorTracker {

    /**
     * Configures the file according to a tracker-specific
     * configuration. Resolves when finished configuring.
     */
    configure(config: any, file?: vscode.Uri, args?: any):
    Promise<void>;

    /**
     * Cleans this file of any auto configuration
     * that may have been set.
     */
    clean(file?: vscode.Uri, args?: any): Promise<void>;

    /**
     * Compiles a tracker-specific configuration.
     */
    compileConfiguration(file?: vscode.Uri, args?: any): any;

    /**
     * Registers (starts) the tracker with a listener,
     * optionally on a file.
     */
    register(listener: (event: HostIDE.TutorEvent) => void,
    onFile?: vscode.Uri): void;

    /**
     * Tears down this tracker & disposes of any disposables.
     */
    teardown(file?: vscode.Uri, args?: any): Promise<void>;
}
```

Figure 4-5: The Tutor Tracker interface in `event-tracking-api.ts`

4.4.2 extension.ts

VS Code Extensions are defined inside a `extension.ts` file. Updates to the extension file involved reacting to the new Setup Required UI and Start Over Required UI webapp states, as well as coordinating the tracking of events, and the collection and preparation of the logs for sending to the webapp.

The extension includes a private, read-only map `EVENT_TRACKERS` mapping each `HostIDE.IDEEventStream` to its respective concrete `TutorTracker`. The map's type, `{ [key in HostIDE.IDEEventStream]: TutorTracker }`, enforces two important conditions: First, if a new tracker is added to the enum `HostIDE.IDEEventStream` but not to the `EVENT_TRACKERS`, a compile time error will complain that not all event types are represented in the map. Second, concrete trackers are forced to implement the `TutorTracker` interface. This is the *only* place in `extension.ts` where concrete trackers are listed. Everywhere else, relevant trackers are filter from this map using `Exercise` model object properties and by invoking `TutorTracker` operations. In short, the use of `EVENT_TRACKERS` and its typing ensures that, as new event trackers are introduced by new Praxis Tutor developers, those trackers are guaranteed to be included in `extension.ts`.

Reacting to the Webapp Event Loop

As students progress through the webapp's event loop described in Figure 4-3, the extension implements a sort of event sequence of its own, triggered by actions in the webapp. Within the extension's event sequence, operations are driven by delays in concurrent code. For example, on the Host Display Exercise transition triggered by the webapp, the extension proceeds as follows, with steps highlighted in orange denoting new actions added to support event tracking exercises:

1. `await` any potential cleanup still in effect from a previous exercise.

2. `await` tearing down of listeners still in effect from a previous exercise.

3. Unzip the exercise file contents inside the `.praxistutor` folder.

4. Open a background process to `npm install` the necessary dependencies.

5. If the exercise is marked as `requiresLaunchConfiguration` but the `.praxistutor` workspace is not yet installed, insert the `.praxistutor` workspace as the root (i.e. first) workspace. Mark in the global context to automatically recover with this exercise, as the installation will cause the VS Code window to reload.

6. Set this exercise with `freshMetadata` (including a new, empty event log).

7. `await` opening the exercise files in VS Code.

8. `await` cleaning the file from any prior configurations.

9. `await` registering any relevant event trackers.

10. Inform the webapp that *Host Display Exercise* is complete.

In step (9), the extension filters for relevant trackers by looping through the keys of `EVENT_TRACKERS` and registering the respective tracker of each one if and only if the key matches any listed history stream in the exercise's solution history templates. For example, if the only history template listed began with `r/debugger|terminal/`, the extension would find that *debugger* and *terminal*, but not *editor*, match to the event stream and would register both of their respective trackers. The listener callback is the same for each tracker: consume a `TutorEvent` object, filter the event for its relevant data according to the solution templates, and append it to its `metatype`'s event log.

In the Setup Required UI state (Fig. 4-3), when a student clicks Start, the webapp forwards the request the extension, which filters the trackers in much the same way as registering listeners: for each key of `EVENT_TRACKERS`, call `configure` on its tracker and `await` its result if and only if that key appears in the exercise configuration description, passing in the appropriate `configuration[key]` config. When all configurations complete, inform the webapp that host setup is done so that it completes the transition to the Ready for Interaction UI.

The exercise's `MetadataType` object maps each tracker type name to an array of `TutorEvents`. When a student clicks Check in the webapp, the extension packages the exercise's entire event log plus the outputs of the relevant tracker's `compileConfiguration`, along with the file source.

If the student clicks Quit, or, on a correct submission, clicks Continue with no more exercises in the set (part D in Fig. 4-3), the extension will perform one final round of cleanup followed by tearing down all event tracking, before closing the exercise and returning to the Home UI.

The Event Log

Compared to computation, user interface rendering can take a long time. Therefore, for performance reasons, updates to the UI are often handled in sequential parts. For example, stepping over in the debugger triggers multiple changes in the UI, including the paused session pointer location (arguably the most important state), local variable values, and plenty of other minor changes. Were VS Code to halt the debugger until all changes were rendered, debugging would be aggravatingly slow. Instead, the VS Code debugger triggers piece-wise updates (*step-in-event*, *update-variable-values-event*, ...) which can each be thrown away if its output is no longer up-to-date. This allows a user to repeatedly and aggressively step in, because perhaps only the session pointer location is updated to reflect each step, while the local variable values lag behind until the final step's UI updates propagate through fully.

To build the event log, then, trackers must distinguish between user-initiated events—that is, student actions such as a step in event—and system update events in response to those actions. That way, the extension can determine when to append a new student event to the log versus when to update the most recent event on the log with new data. To that end, `TutorEvents` are marked with a `IDEEventSource` source: `User` events are appended to the log, while `System` events replace the most recent event on the log. It is up to the trackers themselves to format `System` events as replacements for the most recent student-initiated event. Defining the event log protocol in this way has the benefit of always being up-to-date with UI events.

4.4.3 Concrete Trackers

Currently, Praxis Tutor supports an editor tracker, a terminal tracker, and a debugger tracker.

Editor Tracker The editor tracker listens for just two events: the `didEdit` event, which plugs in to VS Code's `workspace.onDidChangeTextDocument` listener, and the *movedCursor* event, which plugs in to VS Code's `window.onDidChangeTextEditorSelection` listener. The former is used to detect editor changes on readonly exercises, while the latter is used in triggered hints for Error Parsing exercises.

Terminal Tracker The terminal tracker only listens for one event—the `clickCodeLink` event—but is considerably more complicated than the editor tracker because it overrides VS Code's default `TerminalLinkProvider`.[6]

For the duration of an exercise with terminal tracking, the tracker implements its own `TerminalLinkProvider`, reading in terminal input, interpreting links, and routing link clicks. Overriding with its own link provider allows the tracker to emit a `clickCodeLink` event at the same time it routes link clicks.

The tracker uses the following regular expression to detect links:

$$\texttt{/}\underbrace{\texttt{([\^{}\textbackslash s(]*\textbackslash .praxistutor\textbackslash Ssrc\textbackslash S\textbackslash w*\textbackslash .ts)}}_{\text{Capturing Group 1}}\texttt{:}\underbrace{\texttt{(\textbackslash d+)}}_{2}\texttt{(:}\underbrace{\texttt{(\textbackslash d+)}}_{3}\texttt{)?/g}$$

where each underlined portion represents a distinct capturing group. The first capturing group matches the file path. `[^\s(]*` excludes starting with whitespace or an open parenthesis, followed by ".praxistutor", followed by a single non-whitespace separator `\S`, followed by "src", followed again by a non-whitespace separator, followed by any character string `\w*`, followed by ".ts". The second and third groups capture the line number and optional character number, respectively. The trailing flag "g" instructs the regex to match all occurrences. All together, the regular expression can match Windows paths such as "C:\Users\gecanow\.praxistutor\src\Buggy.ts:5" or Mac and Linux paths such as "/Users/gecanow/.praxistutor/src/Computation.ts:2:46".

[6] https://code.visualstudio.com/api/references/vscode-api#TerminalLinkProvider

When a matching filepath is clicked in the terminal, the tracker emits an event containing the file name, line, and character and then moves the student's cursor to the clicked location.

Debugger Tracker Unlike for the editor and terminal, the VS Code extension API does not provide explicit event listeners for debugger single step and evaluate actions. While there exist listeners for setting breakpoints, removing breakpoints, and launching debug sessions,[7] the value of Debug Tutor exercises comes from observing *all* debug events, including single stepping and evaluation with the Debug REPL.

Internally, VS Code communicates with debuggers via the Debug Adaptor Protocol (commonly abbreviated to DAP).[8] Debug Adaptors powering the debug functionality for a specific language can implement the protocol and plug in to the VS Code runtime environment, automatically making use of VS Code's generic debug UI. At its core, VS Code debugging boils down to sequences of well-formed messages sent back and forth between the VS Code runtime environment and a Debug Adaptor. VS Code comes with many preexisting Debug Adaptors[9], so users can readily start a debug session for, say, their JavaScript, TypeScript, or Python project. But, it is worth noting that *any* obscure language can in theory be debugged in VS Code, so long as someone authors and publishes a Debug Adaptor for it. Even more exciting, DAP is supported by a variety of IDEs, not just VS Code.[10]

These two points—that any language could in theory have a Debug Adaptor and that DAP is recongized by a variety of IDEs—are particular exciting for the Debug Tutor, because it means that its debugging exercises can be fully IDE- and language-agnostic. So long as the editor recognizes DAP and the language has a Debug Adaptor, the Tutor extension will be able to observe, parse, and interpret the standard debug event messages passed between it and the VS Code runtime environment.

[7] From the `vscode.debug` API (https://code.visualstudio.com/api/references/vscode-api#debug), `debug.onDidChangeBreakpoints` and `debug.onDidStartDebugSession`

[8] https://microsoft.github.io/debug-adapter-protocol/

[9] https://microsoft.github.io/debug-adapter-protocol/implementors/adapters/

[10] For a list of IDEs recognizing DAP, see https://microsoft.github.io/debug-adapter-protocol/implementors/tools/

To observe the message sent between the Debug Adaptor and VS Code runtime environment, the debugger tracker implements VS Code's `DebugAdapterTracker`[11]. The debug tracker has an `InterceptedDebugger` namespace for organizing its interception and interpretation of messages. All messages are classified as a `Request`, `Response`, or `Event`, with corresponding classes for interpreting the string message. The debug tracker takes advantage of these classes by defining three corresponding classes of its own implementing `InterceptedDebugger`, each responsible for interpreting messages of its type and returning well-formed `TutorDebugEvents` marked as `User` or `System`. Then, messages can be passed to `InterceptedDebugger`'s `interpret` method, for appropriate hand-off to the correct subclass.

```
/**
 * Helper function for intercepting and handling DAP messages.
 * @param message, the DAP message.
 */
private interpret(message: any): void {
    const event = InterceptedDebugger.interpret(message,
                                                this.lastUserEvent);
    if (event) this.post(event);
}
```

Finally, to ensure that TypeScript files are debuggable by the Node.js debugger, the `clean` method forks a background process to compile the exercise files with `tsc`.[12] Put together, the debug tracker is able to observe all messages between the debugger and the runtime environment in real time, parse them for relevant information, and log them into an organized student event history.

4.5 Version 1 Complete

With the changes implemented in the exercise models, server, webapp, and extension, the Praxis architecture could now support the concrete debugging concept map and

[11] https://code.visualstudio.com/api/references/vscode-api#DebugAdapterTracker

[12] For more on TypeScript compiling in VS Code, see https://code.visualstudio.com/docs/typescript/typescript-compiling

all the concrete exercises designed for that concept map. Just under 30 exercises were authored across 11 code files spanning the 23 concepts in the Debug Tutor concept map for the version 1 release.

Chapter 5

Deployment

The Debug Tutor was deployed as part of the 6.102 Spring 2023 curriculum. 6.102 has a dedicated reading and lecture on debugging,[1] and because the Tutor is intended to be incorporated into a larger curriculum, 6.102 provided a suitable environment for evaluating the effectiveness of the Debug Tutor.

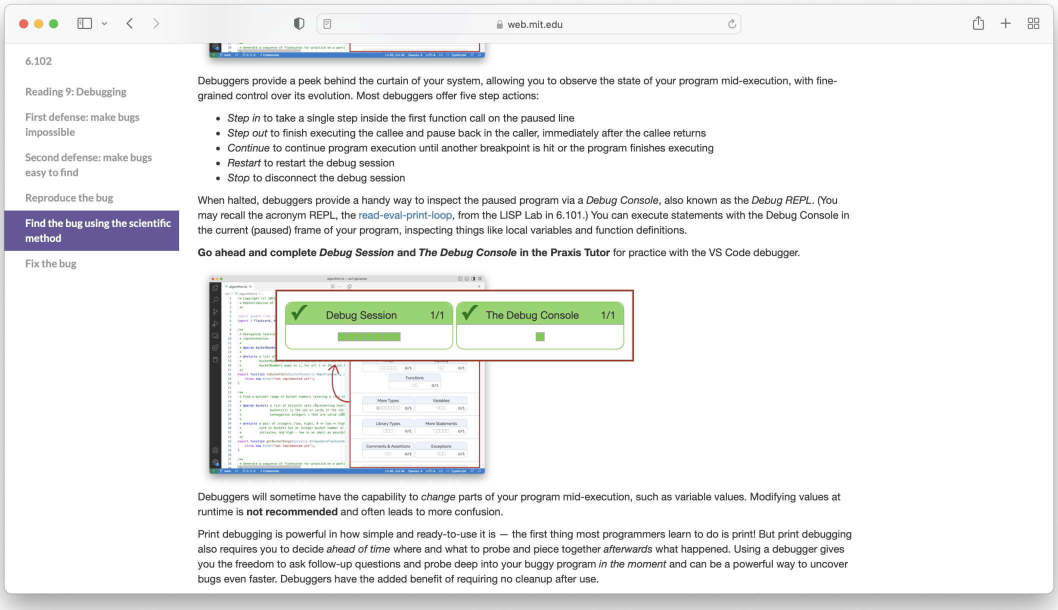
5.1 Debugging Reading

The debugging reading was the ninth in a series of nineteen readings assigned throughout 6.102's Spring semester. Every reading accompanies a lecture on the same topic, during which students get hands-on practice.

In the debugging reading, each concept group in the Debug Tutor is recommended to be completed following its relevant introductory section. The reading first discusses how to prevent bugs with techniques like static typing and immutable design. The reading then describes how to localize bugs by failing fast and with incremental development, and how to minimize scope through modularity and encapsulation. In the third section, the reading details the steps in systematic debugging: reproducing the bug, finding the bug with the scientific method, and fixing it carefully. The Debug Tutor concept groups are recommended at appropriate points throughout the

[1] https://web.mit.edu/6.102/www/sp23/classes/09-debugging

scientific method section[2]. Finally, the reading dives deeper into ensuring the bug is fixed thoroughly and gives some final suggestions on what to do when stuck.


web.mit.edu
6.102
Reading 9: Debugging
First defense: make bugs impossible
Second defense: make bugs easy to find
Reproduce the bug
Find the bug using the scientific method
Fix the bug

Debuggers provide a peek behind the curtain of your system, allowing you to observe the state of your program mid-execution, with fine-grained control over its evolution. Most debuggers offer five step actions:

- *Step in* to take a single step inside the first function call on the paused line
- *Step out* to finish executing the callee and pause back in the caller, immediately after the callee returns
- *Continue* to continue program execution until another breakpoint is hit or the program finishes executing
- *Restart* to restart the debug session
- *Stop* to disconnect the debug session

When halted, debuggers provide a handy way to inspect the paused program via a *Debug Console*, also known as the *Debug REPL*. (You may recall the acronym REPL, the read-eval-print-loop, from the LISP Lab in 6.101.) You can execute statements with the Debug Console in the current (paused) frame of your program, inspecting things like local variables and function definitions.

Go ahead and complete *Debug Session* and *The Debug Console* in the Praxis Tutor for practice with the VS Code debugger.

Debuggers will sometime have the capability to *change* parts of your program mid-execution, such as variable values. Modifying values at runtime is **not recommended** and often leads to more confusion.

Print debugging is powerful in how simple and ready-to-use it is — the first thing most programmers learn to do is print! But print debugging also requires you to decide *ahead of time* where and what to probe and piece together *afterwards* what happened. Using a debugger gives you the freedom to ask follow-up questions and probe deep into your buggy program *in the moment* and can be a powerful way to uncover bugs even faster. Debuggers have the added benefit of requiring no cleanup after use.

Figure 5-1: Clip of Reading 9 *Scientific Method* Subsection

The release deadline for the Debug Tutor was therefore the release date of the ninth reading.[3] Before the official release, though, it went through several rounds of staff testing. Staff were first asked to work through the exercises as an honest student might, followed by another round of trying to break the Tutor for stress-testing.

5.2 Staff Testing

The Debug Tutor went through three rounds of staff testing before being released. Due to a more than 50% increase in the 6.102 enrollment from the previous Spring semester, the Spring 2023 term's staff was particularly large, with two head TAs, ten graduate TAs, 13 undergraduate TAs, and 20 undergraduate lab assistants (LAs). This fortunately allowed for plenty of staff testing before release to students.

[2] https://web.mit.edu/6.102/www/sp23/classes/09-debugging/#find_the_bug_using_the_scientific_method

[3] Tuesday, February 28th, 2023

Round 1: Some Staff In the first round of staff testing, five TAs were asked to work through the Debug Exercises. These TAs were chosen because they spanned different operating systems and level of familiarity with the ideas behind the Debug Tutor. All testers completed the exercises on their own time and without supervision. They were asked to give specific feedback about their experience working through the exercises. The first round led to refinements in the concept map, updates to individual exercise prompts and hints, additional suggested tutorials, and critical bug fixes, particularly those only evident on the Windows operating system.

Round 2: All Staff After the first round of testing and subsequent updates, and once the Debug Tutor had been fully incorporated into the debugging reading, the reading was released to all staff for another round of playtesting. The staff were therefore completing the exercises in the same condition as the students would: working on their own time, without supervision, as properly contextualized in the course reading on debugging. Releasing to the wider staff not only led to more critical bug fixes and exercise refinement, but it also confirmed that the Debug Tutor worked across a variety of development environments and its concept map logically followed in the course reading.

Round 3: Final Push In the last push, staff were asked to playtest the final iteration and sign off on releasing to students. 18 staff members responded (two on Linux, three on Windows 10, two on Windows 11, seven on macOS Monterrey, and four more on various other macOS releases) and all gave an *lgtm*.[4]

5.3 Official Release

The debugging reading was released to students about a week before the scheduled lecture on debugging and was due at 10pm the night before the lecture.[5] Around 400 students completed all Debug Tutor concept groups by the 10pm deadline. 75

[4] *Looks good to me!*

[5] Thursday, March 9th, 2023

additional students completed at least one concept by the start of class (45 of which completed all concept groups). At the data collection cutoff on Sunday of that week, of the 542 students on the roster at that time, a total of 496 students had attempted at least one Debug Tutor exercise.

5.3.1 Grading

About 10% of a student's grade in 6.102 is their classwork grade. Classwork grades consist of before-class work (e.g. reading exercises) plus in-class collaborative work and a nanoquiz, for each of the 19 in-person lectures and accompanied readings.

Like the TypeScript Tutor exercises incorporated into earlier readings, the Debug Tutor exercises were part of the before-class grade for class 9. Completion was graded out of 7 points, accounting for a little over 10% of class 9's before-class grade (the other 90% was from multiple-choice conceptual questions embedded in the reading itself). In all, the Debug Tutor accounted for less than 0.1% of the class 9 grade.

To test understanding of the debugger, on the class 9 nanoquiz, students were given two debugger questions (randomly drawn from three options), worth 6 out of the 12 nanoquiz points. Students were also encouraged to use the debugger during the in-class collaborative debugging exercises. About two weeks later, one sub-question on the first quiz tested understanding of the debugger. So, while the actual Debug Tutor exercises were not worth enough to affect a student's grade in any meaningful way, the knowledge it is designed to teach should certainly have aided during class 9's nanoquiz as well as the subsequent quiz 1 debugger sub-question, not to mention during problem set debugging and in all future programming pursuits. The following chapter examines the collected data in an effort to answer these questions.

Chapter 6

Evaluation

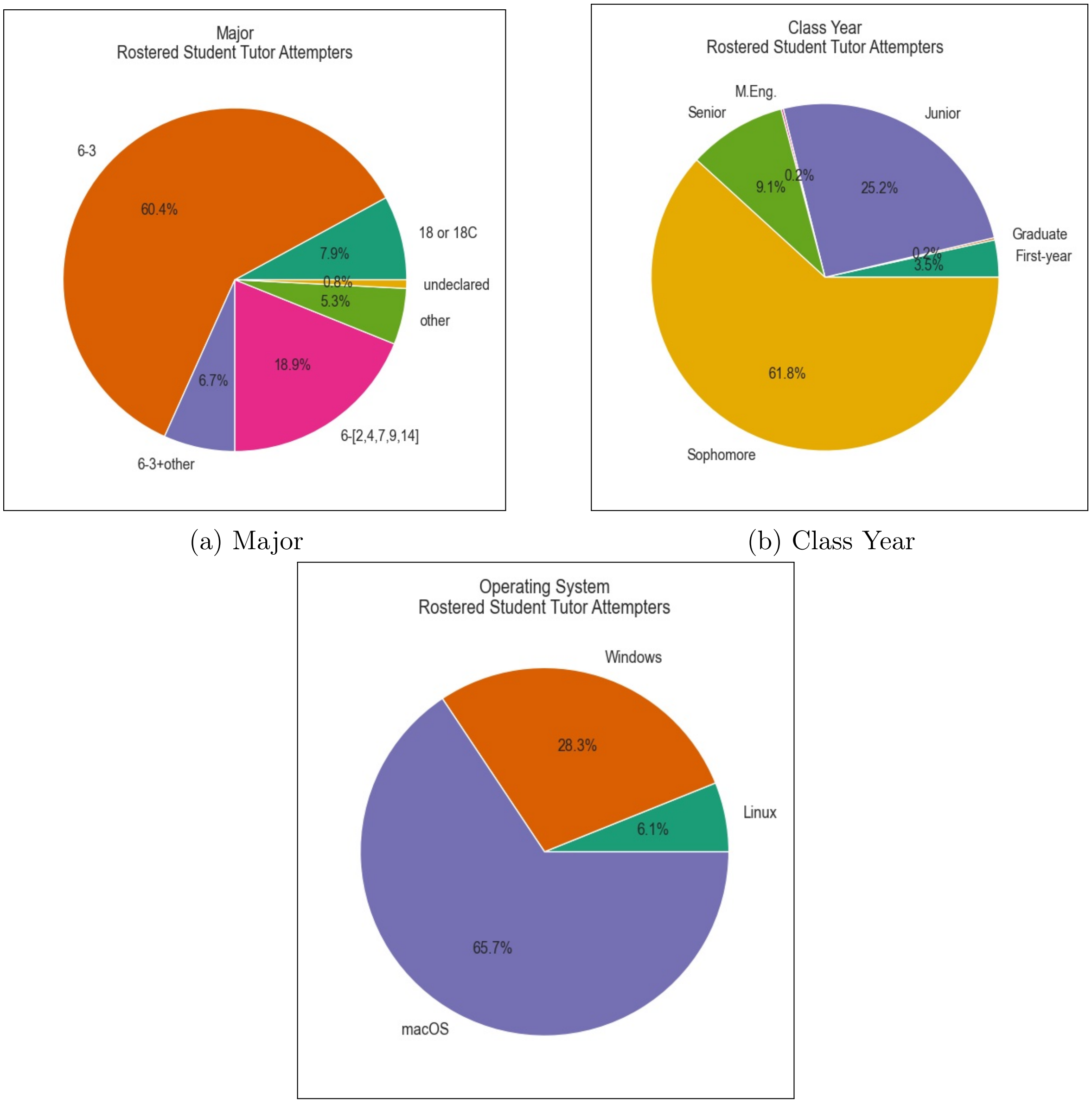
Following the release of the Debug Tutor in the 6.102 curriculum, several metrics were collected to evaluate its effectiveness. The release also uncovered a handful of fixable bugs in the system and in the design of the concept map, which will be updated for next term. On the whole, the release was a success.

6.1 Student Data

At the beginning of the semester, students were asked to report their class year, major, and laptop operating system. Of the 492 students who attempted at least one Debug Tutor exercise, a majority were majoring in course 6-3 (Fig. 6-1a), in their sophomore year (Fig. 6-1b), and using a laptop running macOS (Fig. 6-1c).

6.1.1 Self-Reported Data

Embedded in reading 9 were several questions asking students to reflect on their previous debugging experience before using the Debug Tutor. Of the students who attempted at least one Debug Tutor exercise, about 65% reported prior experience with any debugging concept (setting and removing breakpoints, single stepping, or evaluating with Debug REPL), and about 33% (165 students) noted that they had had prior experience with the VS Code debugger (Fig. 6-2).



(a) Major

(b) Class Year

(c) Operating System

Figure 6-1: Student Data

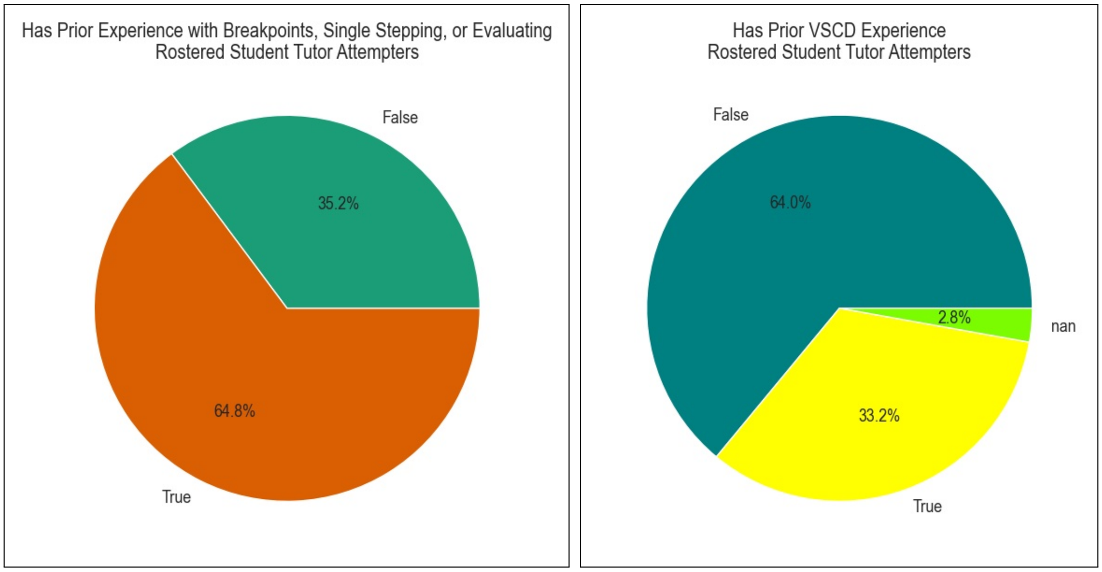


Figure 6-2: Prior Debugger Experience

Of the 322 students who reported any prior experience, the characterization of their prior experience is described in Fig. 6-3.

About half of the students who attempted at least one Debug Tutor exercise felt either "not comfortable" using a debugger or had never used a debugger before, a quarter felt "somewhat comfortable", and the remaining students felt either "comfortable" or "very comfortable" using a debugger, prior to using the Debug Tutor (Fig. 6-4).

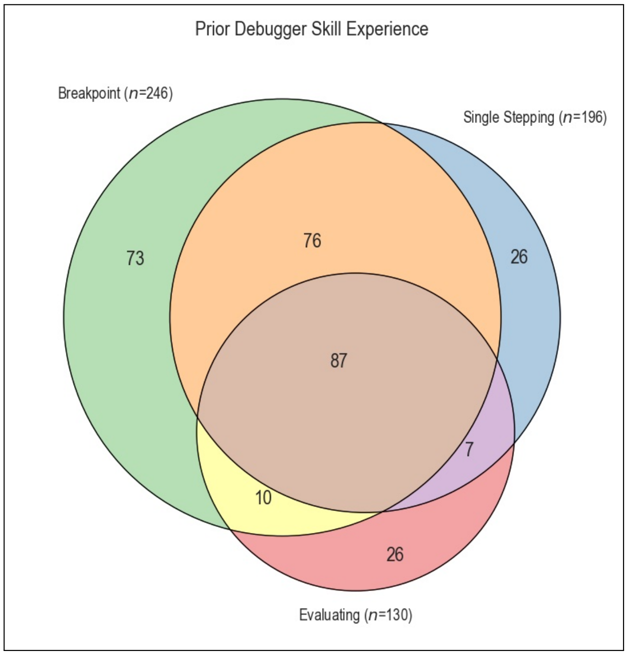


Figure 6-3: Prior Debugger Skill Experience

The self-reported data suggests that, although most students have some experience with the debugger prior to the 6.102 dedicated lecture on the topic, these students are not an overwhelming majority and most need practice. Thus, the sophomore-level 6.102 curriculum can indeed benefit from debugger-specific exercises, as the Debug Tutor aims to provide.

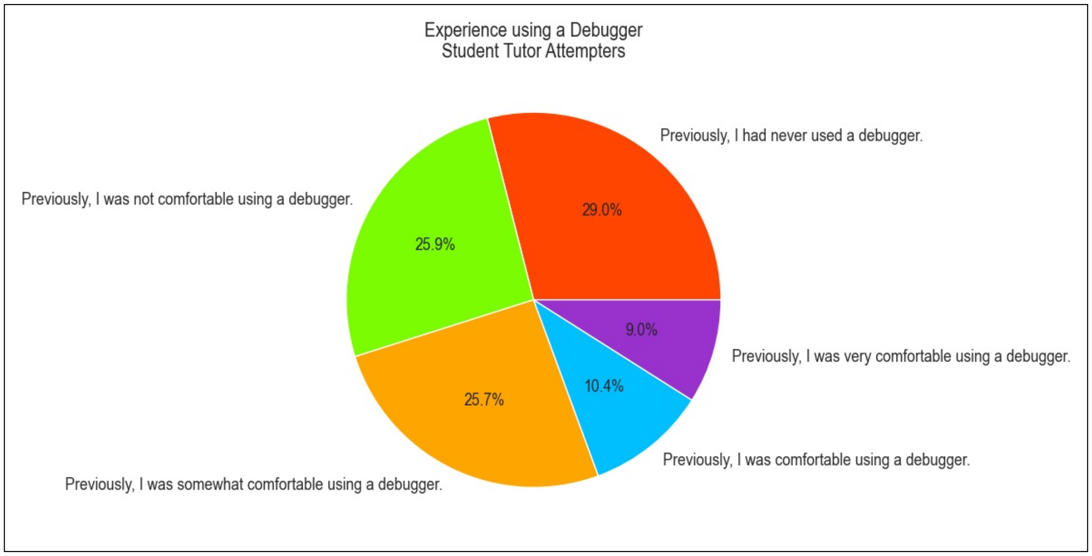


Figure 6-4: Prior Debugger Comfort Level

6.1.2 Problem Reports

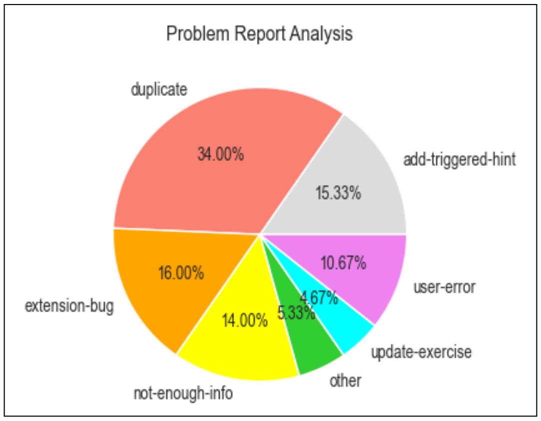


Figure 6-5: Problem Report Analysis

As students worked through the Debug Tutor exercises, they were able to report bugs or issues with an exercise directly inside the Tutor web view via a Report A Problem form. 150 problems were reported, across 51 unique student reporters. About a third were duplicate reports. Others warranted new triggered hints, an update to either the exercise solution or the exercise prompt, or were due to user error or a mis-configured setup.

Despite explicit written directions in the reading and inside some early exercises prompts, the most common user error was students not dragging the Tutor window to the right sidebar. Luckily, those students who submitted problem reports with enough time before the deadline were told the quick and easy fix, and were able to continue working through exercises. Other mis-configured setups included one-off VS Code issues such as a mis-installed Node package.

24 of the problem reports revealed three separate bugs in the extension:

1. A handful of setups did not comply with the automated compiling of the TypeScript exercise files before launching the Node debugger. Luckily, a manual workaround allowed the affected students to continue progressing through the exercises.

2. File paths with spaces were not being properly handled by the Tutor in the Error Interpretation exercises. The three students who reported having spaces in their file paths were unable to complete the exercises in that concept group.

3. Due to the nature of decoupling UI actions from resulting internal changes, some required data values in exercise descriptions were marked as missing even though the student performed the correct actions. In such cases, students were recommended they work through each exercise slowly as a temporary workaround.

All three extension bugs will be fixed for future use of the Debug Tutor. The relatively few problem reporters and problem reports in relation to the number of student attempters (just under 500) and total attempts (around 24,000) indicate that, on the whole, the Debug Tutor was usable.

6.2 Tutor Exercise Completion

At the time of data collection, three days after the completion deadline, over 90% of the 542 registered students had attempted at least one exercise. Of those, about 86% earned all 7 Debug Tutor completion points.

The Debug Tutor exercise pass rates were fairly high. Only a small minority were unable to pass at all per exercise, while a large majority failed on the first attempt but passed on a later attempt.

Rainfalls-mix, the conceptually hardest and most involved exercise, had the lowest pass rate (78.5%). Backgammon-breakpoint-continue-navigation had the second lowest pass rate (83.0%), followed by Taxes-evaluate (84.8%) and Matrix-mix (87.3%),

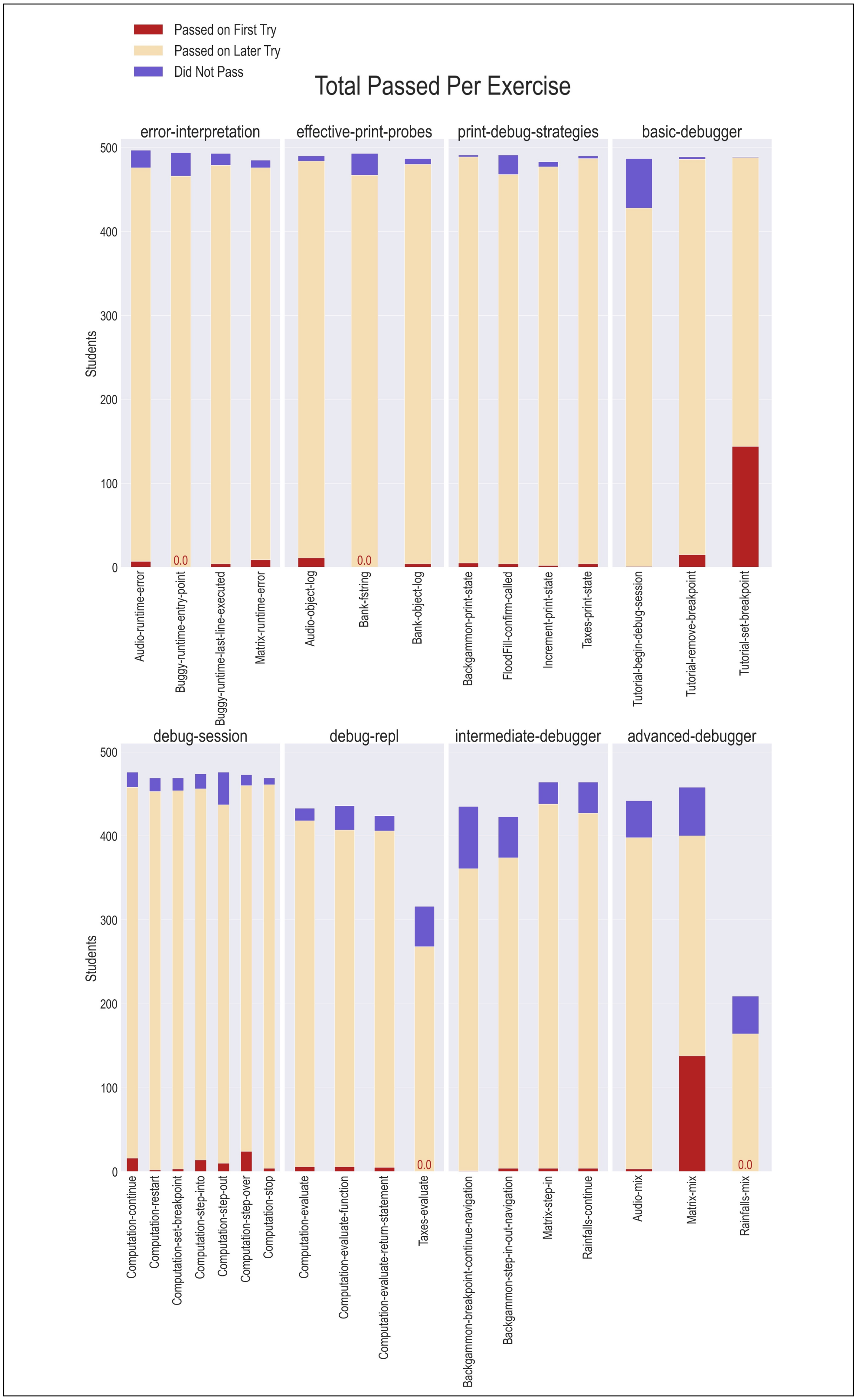


Figure 6-6: Per-Exercise Pass Rate

all more involved exercises that perhaps frustrated students. Surprisingly, Tutorial-begin-debug-session had the fifth lowest pass rate (87.9%). However, given it is the first exercise that requires launching a debug session, and extrapolating from the problem reports, the high failure rate is likely due to a combination of unreported user errors and unreported cases of the first identified extension bug relating to the TypeScript file compilation.

Of the two exercises with the highest pass-on-first-try rates, Tutorial-set-breakpoint and Matrix-mix, the latter is most surprising, while the former is as expected given the nature of the exercise requiring just one click. Matrix-mix only asks that students "[use] the debugger" in any way available to "get the debug session paused at the first line in `dotProduct`", followed by setting a breakpoint, continuing to that breakpoint, and evaluating `dotProd` with the debug REPL. Its first step is unusually lenient, implying that students perhaps had an easier time comprehending goal-oriented instructions and struggled on more detail-oriented exercise prompts.

Analyzing Student Approaches

To drill in to Matrix-mix's high pass-on-first-try rate, the solution approaches of the 138 student submissions in that category were analysed. The Matrix-mix exercise is shown in full in Fig. 6-7.

The two most common approaches to reach the first line of `dotProduct` were to set a breakpoint on the first line of `dotProduct` and continue to it or to step into and out of `column`, then step into and out of `row`, and then step into `dotProduct`. Only 10 students who passed on the first try used a significantly more complex sequence of steps to reach the first line in `dotProduct`. The split of approaches used is shown in Fig. 6-8.

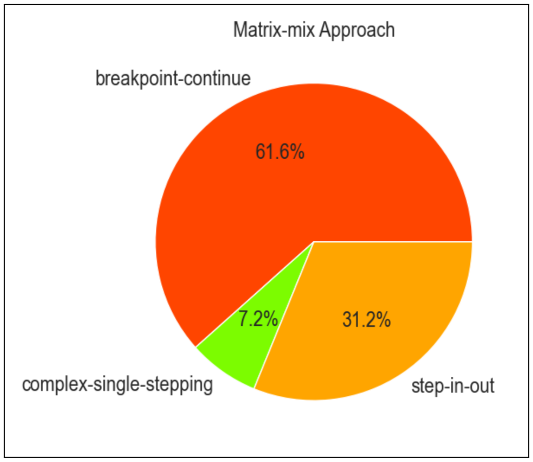
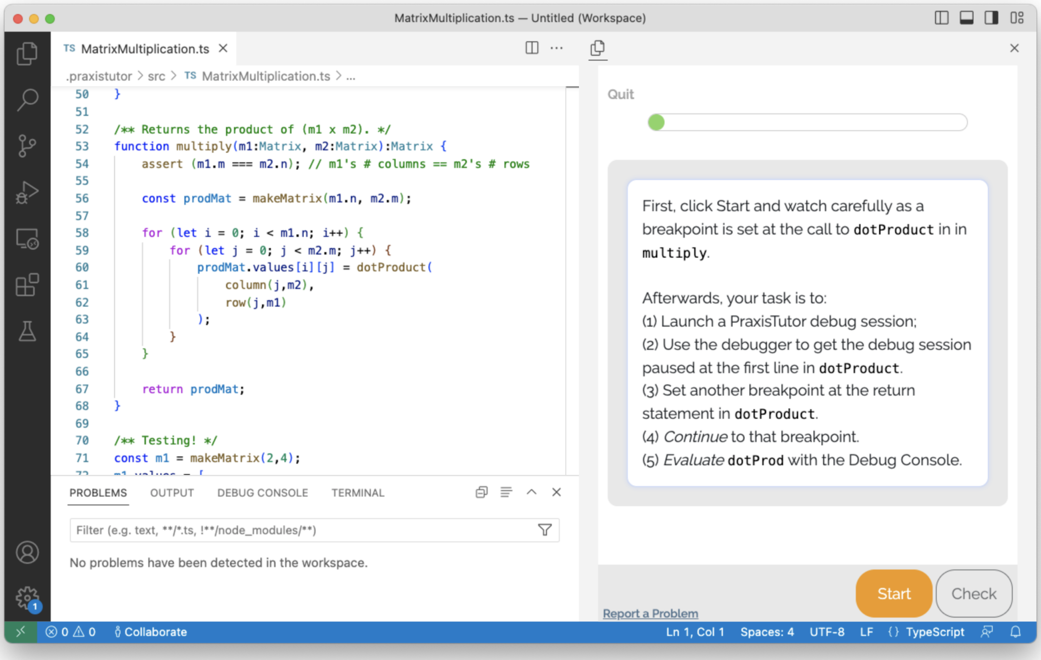


Figure 6-8: Matrix-mix Approaches


MatrixMultiplication.ts — Untitled (Workspace)
TS MatrixMultiplication.ts ×
.praxistutor > src > TS MatrixMultiplication.ts > ...

```
}

/** Returns the product of (m1 x m2). */
function multiply(m1:Matrix, m2:Matrix):Matrix {
    assert (m1.m === m2.n); // m1's # columns == m2's # rows

    const prodMat = makeMatrix(m1.n, m2.m);

    for (let i = 0; i < m1.n; i++) {
        for (let j = 0; j < m2.m; j++) {
            prodMat.values[i][j] = dotProduct(
                column(j,m2),
                row(j,m1)
            );
        }
    }

    return prodMat;
}

/** Testing! */
const m1 = makeMatrix(2,4);
m1.values = [
```

PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
Filter (e.g. text, **/*.ts, !**/node_modules/**)

No problems have been detected in the workspace.

First, click Start and watch carefully as a breakpoint is set at the call to `dotProduct` in in `multiply`.

Afterwards, your task is to:
(1) Launch a PraxisTutor debug session;
(2) Use the debugger to get the debug session paused at the first line in `dotProduct`.
(3) Set another breakpoint at the return statement in `dotProduct`.
(4) *Continue* to that breakpoint.
(5) *Evaluate* `dotProd` with the Debug Console.

Figure 6-7: Matrix-mix

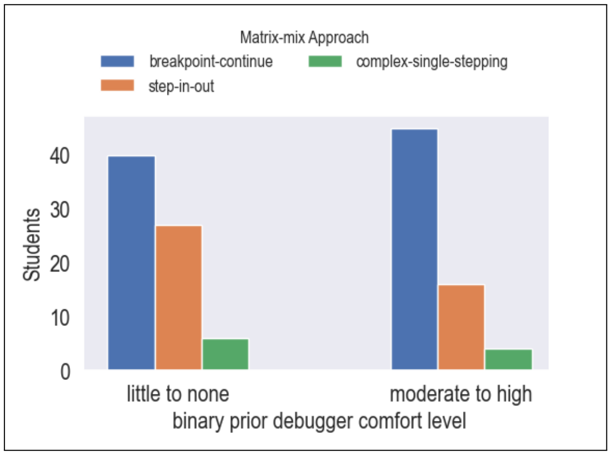


Figure 6-9: First-time passer Matrix-mix students split by approach and binary prior debugger comfort level

Interestingly, prior debugger comfort had no significant effect on the approach used. When the population was split by Matrix-mix approach and little to no versus moderate to high prior comfort level (Fig. 6-9), a chi-squared test[1] yielded a χ^2 value of 3.05 with 2 degrees of freedom and a p-value greater than 0.05. Therefore, the null hypothesis that the two factors were independent cannot be rejected.

The solution pattern for Matrix-mix was `r/debugger/ [{s/launchSession/ [configurationName:r/PraxisTutor/, stopReason:s/breakpoint/,`

[1] Results were calculated with Python package `scipy.chi2_contingency`

`stopsOnLine:c//*[1]*/dotProduct/]}, {}*, {r/step.*|continue/ [stopsOnLine:c//*[1]*/assert/]}, {s/setBreakpoint/ [line:c/return dotProd/] }?, {s/continue/ [stopsOnLine:c/return dotProd/]}, {s/evaluateOnDebugConsole/ [expression:dotProd, resultingIn:9]} ]$ ({r/evaluate.*/ []}, {r/.*Breakpoint/ []})`. One key takeaway from this analysis is that the Tutor's event history pattern matcher was indeed robust enough to handle matching a variety of correct student action sequences against the pattern in the exercise description.

6.2.1 Time Spent & Attempts Per Exercise

Time spent on each exercise was measured from the time the first GET request was received until the final POST request was received by the server. Fig. 6-10 displays box-and-whiskers plots of total time spent per exercise, with the top 5% quantile removed. The plots are separated by those who passed the exercise either on the first try or a later try (in white) and those who eventually gave up (in purple).

As expected, students on average spent slightly more time on the more involved exercises. Across exercises, there is no significant differences between time spent before giving up and time spent before passing, implying that students most likely have a set amount of time set aside for at-home Tutor exercises, and either pass or give up after that amount of time.

Attempts per exercise were measured by the total number of POST requests. Fig. 6-11 displays box-and-whiskers plots of attempts per exercise, again with the top 5% quantile removed. Similar to the time plots, these are also separated by those who passed the exercise either on the first try or a later try (in white) and those who eventually gave up (in purple). The most evident trend appears to be that students who gave up tended to use more attempts than those who succeeded.

Interestingly, prior comfort level seems to have little effect on the total time spent, the average time spent per exercise, the number of exercises passed, and the attempts taken per exercise, as shown by the relative histograms in Fig. 6-12. Evidently, on average, students spent roughly half an hour working through about 30 exercises.

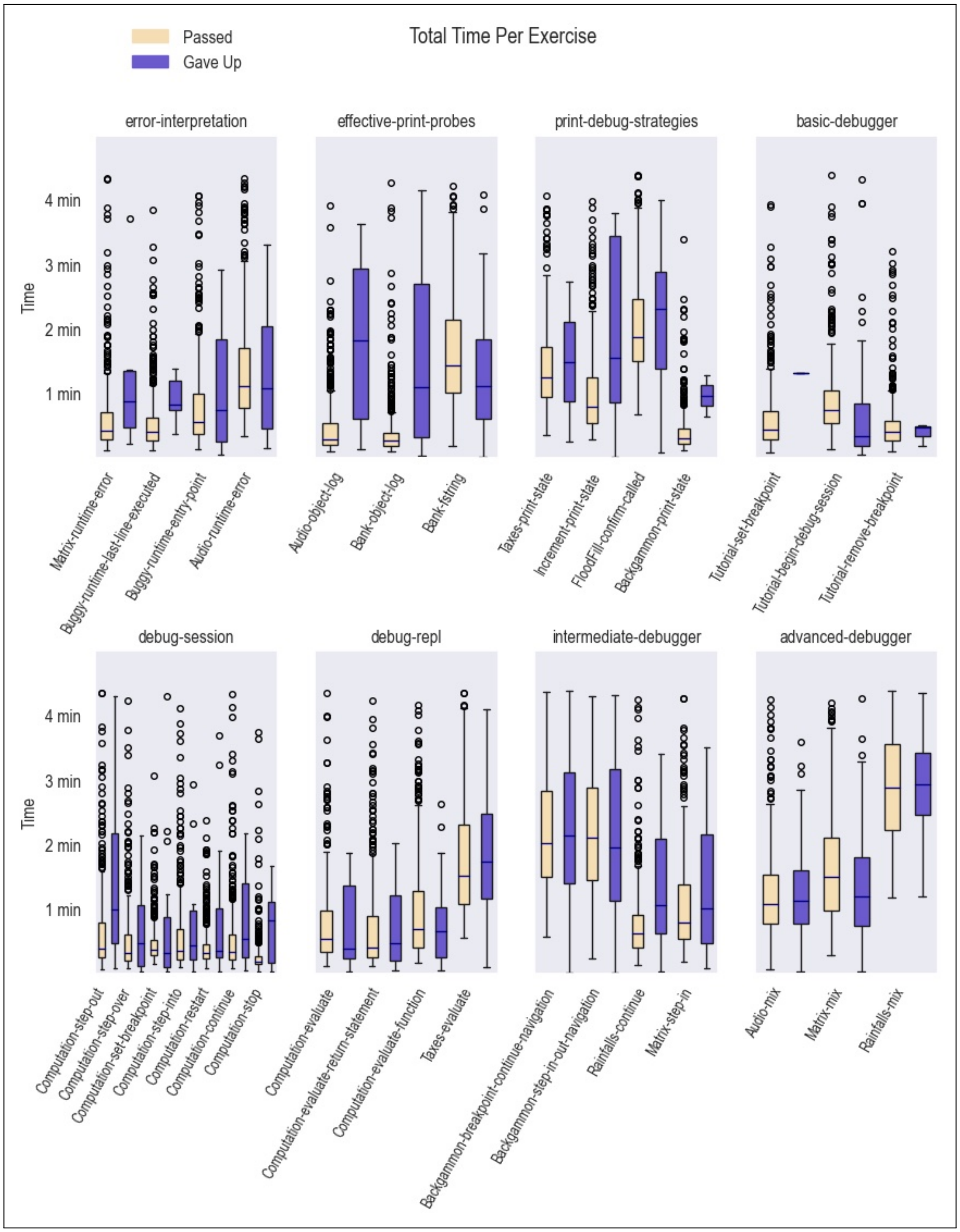


Figure 6-10: Time Spent Per Exercise

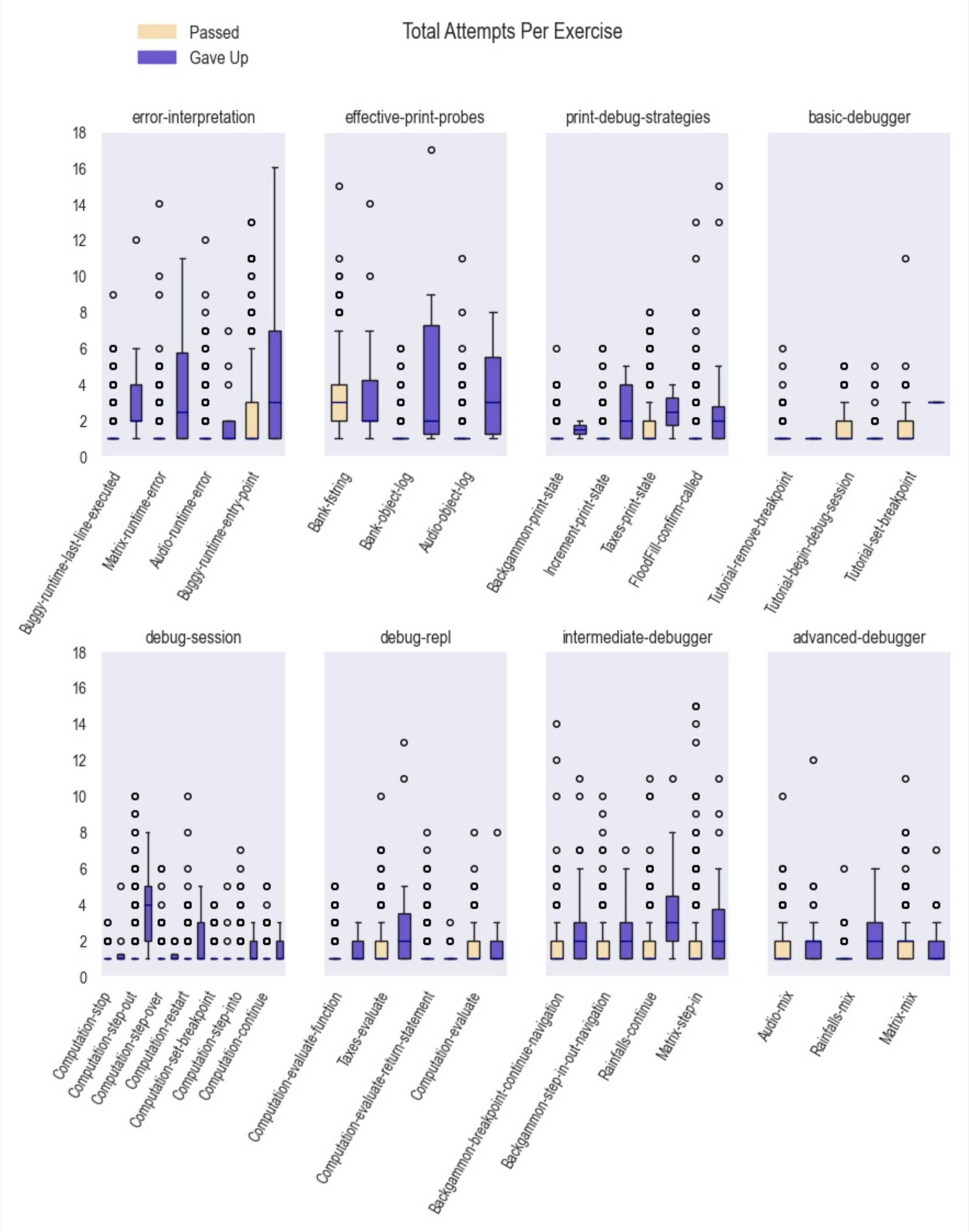
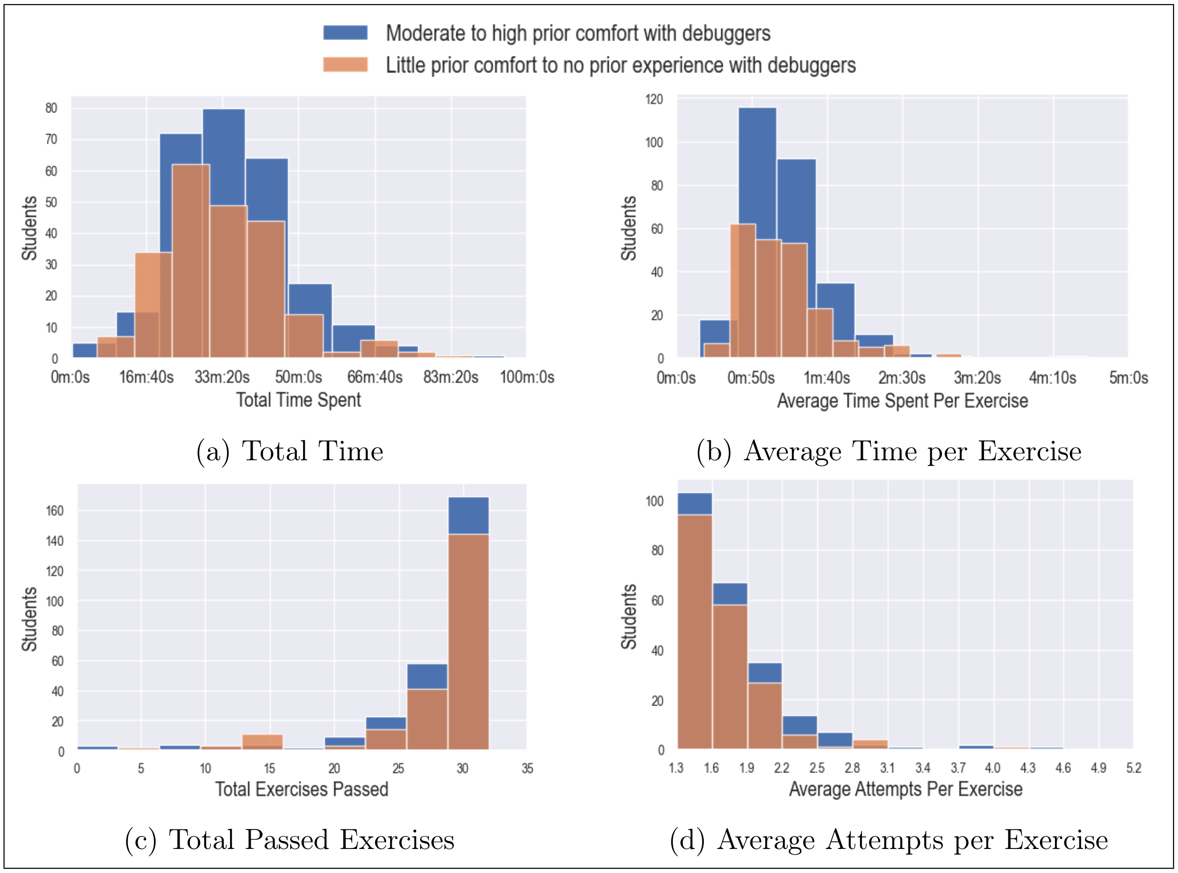


Figure 6-11: Attempts Per Exercise



(a) Total Time

(b) Average Time per Exercise

(c) Total Passed Exercises

(d) Average Attempts per Exercise

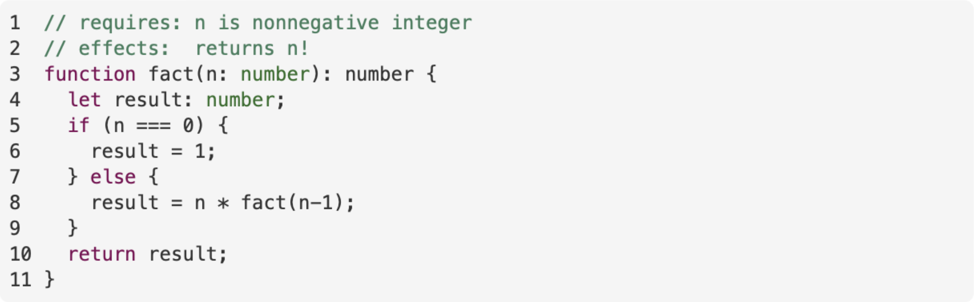
Figure 6-12: Student Data (without the top 0.01 quantile)

6.3 Course Metrics

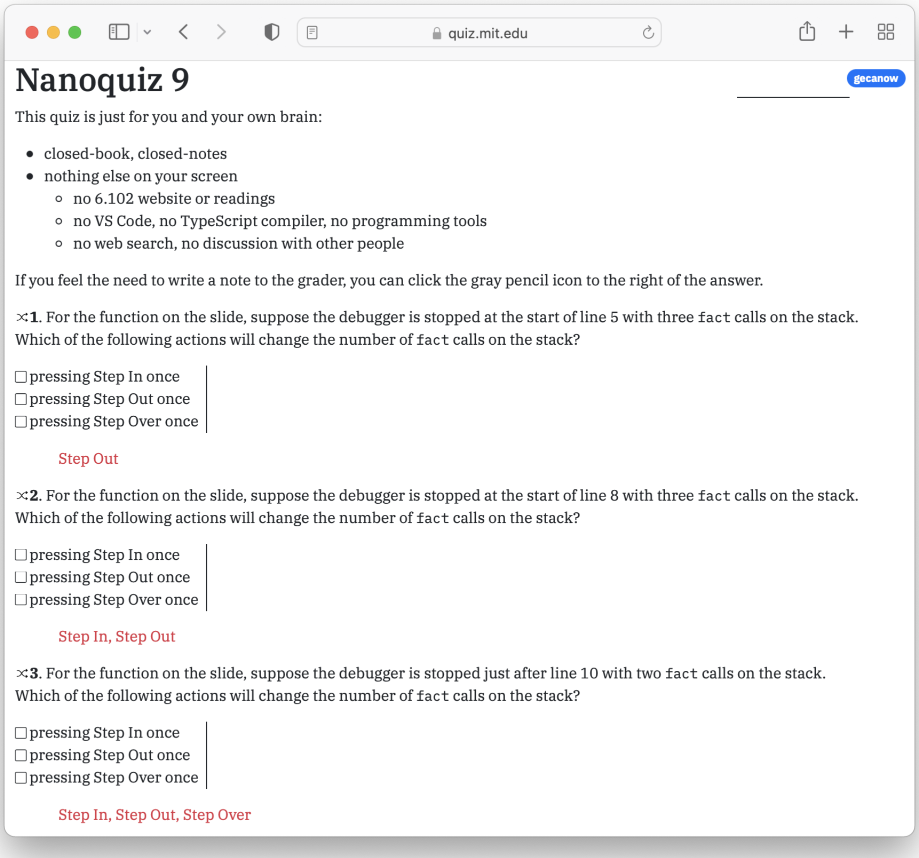
Two course metrics were used to gauge the effectiveness of the Tutor on students' knowledge of the debugger: part of nanoquiz nine and quiz 1 sub-question 4.2.

6.3.1 Class 9 Nanoquiz Results

455 students took the class nine nanoquiz. Students were shown two of three possible questions related to the use of the debugger, each worth three points. All three options and their solutions are shown in Fig. 6-13.



```
1  // requires: n is nonnegative integer
2  // effects:  returns n!
3  function fact(n: number): number {
4    let result: number;
5    if (n === 0) {
6      result = 1;
7    } else {
8      result = n * fact(n-1);
9    }
10   return result;
11 }
```



Nanoquiz 9

This quiz is just for you and your own brain:

- closed-book, closed-notes
- nothing else on your screen
  - no 6.102 website or readings
  - no VS Code, no TypeScript compiler, no programming tools
  - no web search, no discussion with other people

If you feel the need to write a note to the grader, you can click the gray pencil icon to the right of the answer.

⋊**1**. For the function on the slide, suppose the debugger is stopped at the start of line 5 with three `fact` calls on the stack. Which of the following actions will change the number of `fact` calls on the stack?

□ pressing Step In once
□ pressing Step Out once
□ pressing Step Over once

Step Out

⋊**2**. For the function on the slide, suppose the debugger is stopped at the start of line 8 with three `fact` calls on the stack. Which of the following actions will change the number of `fact` calls on the stack?

□ pressing Step In once
□ pressing Step Out once
□ pressing Step Over once

Step In, Step Out

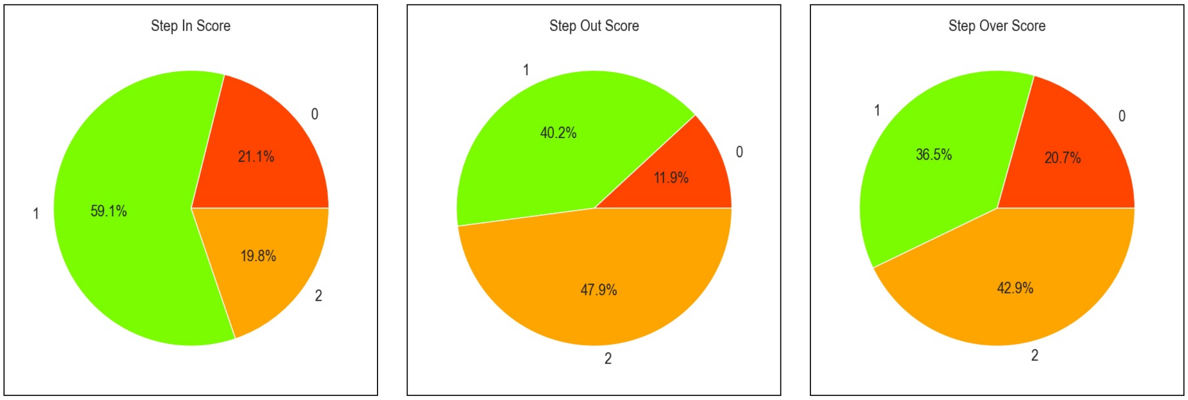
⋊**3**. For the function on the slide, suppose the debugger is stopped just after line 10 with two `fact` calls on the stack. Which of the following actions will change the number of `fact` calls on the stack?

□ pressing Step In once
□ pressing Step Out once
□ pressing Step Over once

Step In, Step Out, Step Over

Figure 6-13: Clip of Class 9 Nanoquiz and Solutions

Each question quizzed multiple concepts. A correct answer may have involved stepping in or not stepping in, stepping over or not stepping over, and stepping out or not stepping out. Fig. 6-14 displays the percentages of points earned from each checkbox grouped by concept. Evidently, the hardest concept was knowing the effect of stepping in, with over half of the 455 students only earning 1/2 points. However, looking deeper into the answer submissions of each individual check box, the low average was due specifically to the third question's step in checkbox, which only 28% of students got correct. The other two step in checkboxes had similar pass rates to those of the step out and step over. Therefore, students most struggled to understand that stepping in at the end of a function call, like stepping out, will return.



(a) Step In

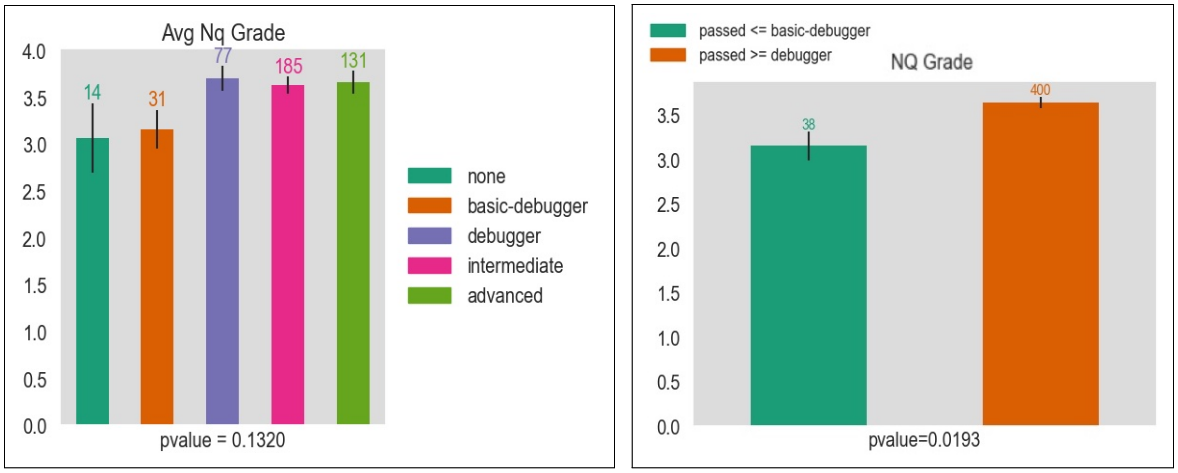
(b) Step Out

(c) Step Over

Figure 6-14: NQ Grade, broken down by concept

To investigate the effect of exercise completion on the nanoquiz grade, the students were bucketed into five groups: *advanced*, if the highest completed concept group was advanced-debug-sessions concept group; else *intermediate*, if the highest completed concept group was intermediate-debugger-sessions concept group; else *debugger*, if the highest completed concept group was debug-session (the debug-repl concept group was not included, given that it appeared in the same level as debug-session but its concepts were not quizzed); *basic-debugger*, if the highest completed concept group was basic-debugger; otherwise, *none*, indicating no debugger-specific levels were completed.

Despite showing a visual bump in average grade from *basic-debugger* to *debugger*,



(a) Nanoquiz grade by highest debugger level completed

(b) Nanoquiz grade by binary concepts passed

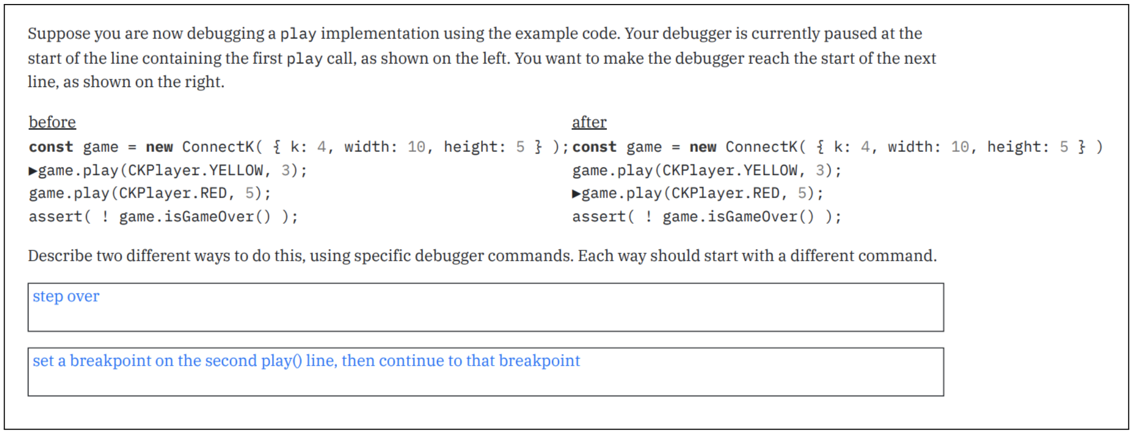
Figure 6-15: Nanoquiz average grade by concepts passed

a one-way ANOVA test returned a p-value > 0.05, implying that the groups' averages were not sufficiently different from each other (Fig 6-15-a). But, grouping the students further into binary buckets by combining the first two groups and last three groups and then running a t-test yielded a p-value less than 0.05, revealing that passing at least all single step concepts is correlated with a higher nanoquiz grade (Fig 6-15-b). However, this correlation may also be due to the fact that average-to-high achieving students are both likely to complete assignments fully and perform well on quizzes.

Takeaway While the nanoquiz asked to consider which step function could be used to achieve a specific state, the debugger exercises mostly focused on specific steps to achieve an abstract goal (e.g., step in until X state occurs). Therefore, the moderate performance on the debugger nanoquiz questions implies that several exercises exploring how certain step functions may or may not achieve specific states should be incorporated into the concept map. For example, one new exercise might explore what happens when stepping in on a return statement.

6.3.2 Quiz 1 Results

On the first quiz, one sub-question worth four points quizzed students on single stepping, asking for two different sequences of steps that would pause the debug session at the next line. Possible answers included a single step over (by far the most popular answer), a step in then out, and setting a breakpoint followed by a continue (Fig. 6-16). 529 students took quiz 1, with about 68% earning all four points on sub-question 4.2.



Suppose you are now debugging a play implementation using the example code. Your debugger is currently paused at the start of the line containing the first play call, as shown on the left. You want to make the debugger reach the start of the next line, as shown on the right.

before

```
const game = new ConnectK( { k: 4, width: 10, height: 5 } );
▶game.play(CKPlayer.YELLOW, 3);
game.play(CKPlayer.RED, 5);
assert( ! game.isGameOver() );
```

after

```
const game = new ConnectK( { k: 4, width: 10, height: 5 } )
game.play(CKPlayer.YELLOW, 3);
▶game.play(CKPlayer.RED, 5);
assert( ! game.isGameOver() );
```

Describe two different ways to do this, using specific debugger commands. Each way should start with a different command.

step over

set a breakpoint on the second play() line, then continue to that breakpoint

Figure 6-16: Quiz 1 sub-question 4.2 and sample answers

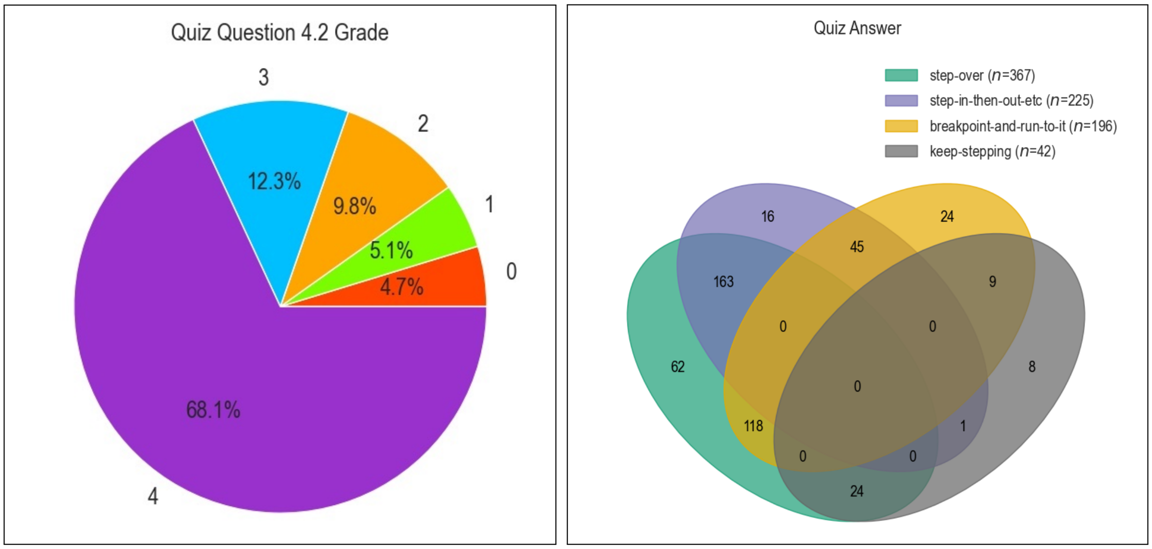


Figure 6-17: Quiz q4.2 grade and answer distribution

Bucketing the students by highest completed debugger level and running a one-way ANOVA resulted in a p-value < 0.05, implying that the population means were

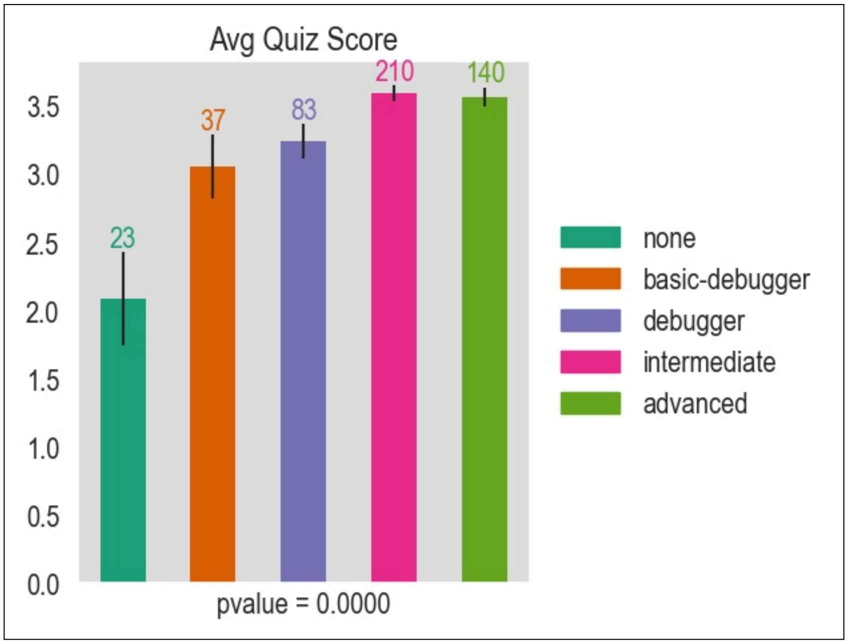


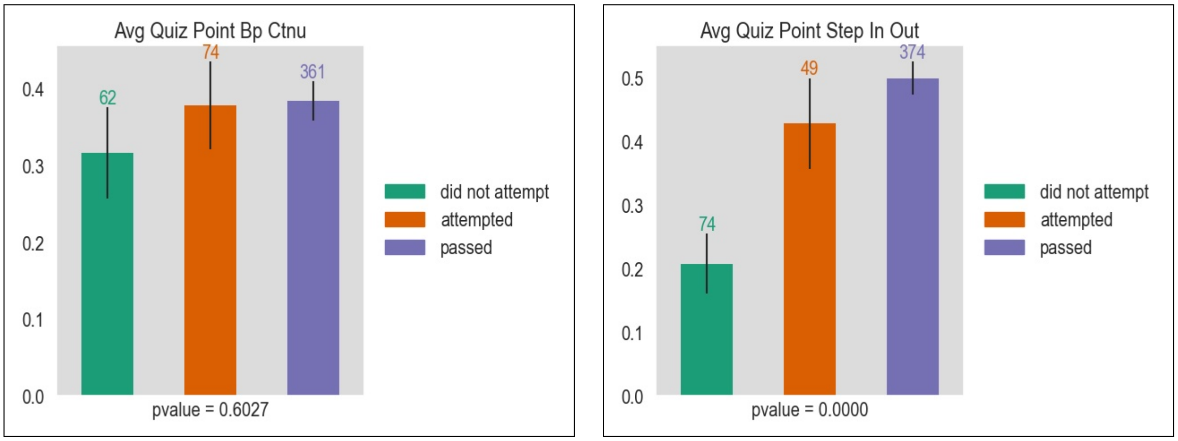
Figure 6-18: Quiz q4.2 average by highest completed debugger level

different enough to reject the null hypothesis. That is, more completed levels was correlated with a higher quiz grade. However, observing Fig. 6-18, it is clear that completing the final advanced concept group provided no additional bump in grade.

One potential reason why the concept group buckets were correlated with quiz grade but not the nanoquiz grade is because the quiz question was posed in a more similar fashion to tasks given by the Tutor: given a paused session, how might you get to the next line?

To that end, one interesting consideration is whether completing some particular Debug Tutor exercise that stressed some specific step sequence affects whether that step sequence was given as an answer on the quiz. In particular, Backgammon-step-in-out-navigation and Backgammon-breakpoint-continue-navigation attempt to teach the skill of manipulating a debug session state via stepping in and out and via continuing to breakpoints, respectively.

To compare the populations, a one-way ANOVA test was conducted to determine if the population means differed significantly for each exercises, as shown in Fig. 6-19. Setting up the experiment with a null hypothesis H_0 that attempting or passing the exercise had no effect on quiz answer, the Backgammon-step-in-out-navigation test (6-19-a) returned a p-value less than 0.05, implying that attempting or passing Backgammon-step-in-out-navigation did have an effect on the quiz answer. In other words, exposure to and passing the step in, step out exercise did have some correlation on likeliness of answering with step in, step out on the quiz.



(a) Effect of Step In-Out Exercise on Quiz Answer

(b) Effect Breakpoint-Continue Exercise on Quiz Answer

Figure 6-19: Effect of Particular Exercise on Quiz Answer

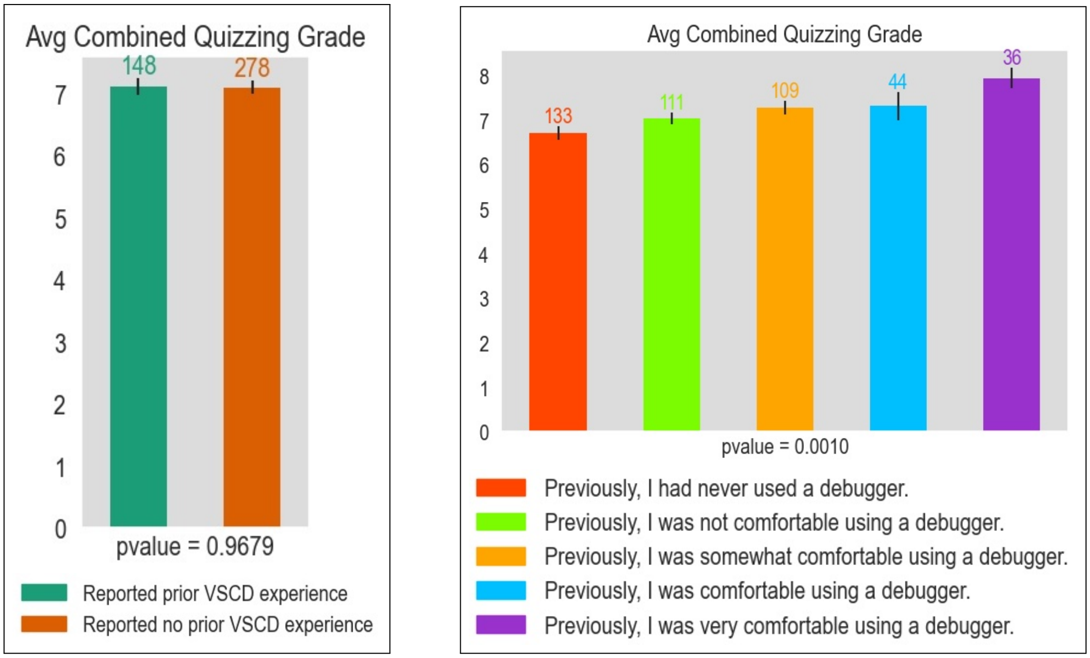
One potential reason Backgammon-breakpoint-continue-navigation had no significant effect on the quiz answer is that the sequence was simply answered by fewer students overall. In rereading the quiz question, although the quiz prompt was open-ended and asked for any "specific debugger commands" to reach the next line, it is possible that students were simply more inclined to think of step functions as the only debugger commands to use.

Takeaway Overall, performance on the quiz debugger question exceeded that of the nanoquiz debugger question, and passing more Debug Tutor concept groups was correlated with higher quiz grade. Furthermore, particular debugger exercises may have influenced students' ability to recall the sequences those exercises taught in the new setting on the exam.

6.3.3 Impact of the Debug Tutor

While the previous two sections investigate the effect of the Debug Tutor on nano- and quiz grades, they ignore an essential potential impact factor: prior debugging experience.

Although prior experience with the VS Code debugger has no impact on combined nanoquiz and quiz score, prior comfort level does have an effect (Fig. 6-12). An ANOVA test yielded a p-value less than 0.05, implying that the average population means are sufficiently different between comfort levels. As expected, as each population grows in comfort level, so do the combined quiz scores.



(a) Prior VSCD experience on combined quiz grades

(b) Prior debugger comfort level on combined quiz grades

Figure 6-20: Prior experience on combined quiz grade

To properly assess the impact of the Debug Tutor, multiple two-way ANOVA experiments were conducted across a variety of population slices.

Prior Debugger Comfort and Highest Level Completed

First, the population was grouped into two debugger comfort levels: those who reported little comfort to no experience and those who reported somewhat comfortable,

comfortable, or very comfortable. These populations were then split again by highest completed debugger concept group. Fig. 6-21 describes the calculated results from running a two-way ANOVA on the populations.[2] Because both *none* groups had less than 10 individuals, they were dropped from the ANOVA, though they are still included in the graph. Each individual factor—prior comfort level and highest concept group completed—has a significant impact on combined quiz score, as both p-values were significant enough to reject the null hypothesis. However, the interaction of the two factors had a p-value greater than 0.05, so the effect of the factors with each other was not significant.

Discussion Analyzing the interaction plot, with binary debugger comfort on the x-axis and average combined quiz score on the y-axis, and each level bucket graph on its own line, it is evident that prior comfort does have *some* positive relationship with quiz score. However, the highest completed level seems to have far more of an impact, with all three highest levels (debugger, intermediate, and advanced) appearing significantly above the two lowest levels (none and basic-debugger), regardless of comfort level. Then, for students who had little prior debugger comfort, completing beyond the *debugger* level gave almost no significant bump in combined score. On the other hand, for students who had moderate to high prior comfort, each higher level completed bumped the grade a non-negligible amount.

One conclusion to draw from this analysis is that, while the debug-session concept group indeed helped all students learn how to use the debugger regardless of prior comfort, the highest two Debug Tutor levels—*intermediate* and *advanced*—were only helpful for those who had prior comfort using a debugger. The concept map should therefore be updated with more exercises at the debugger level, while the last two concept groups might be simply left as extra practice for motivated students.

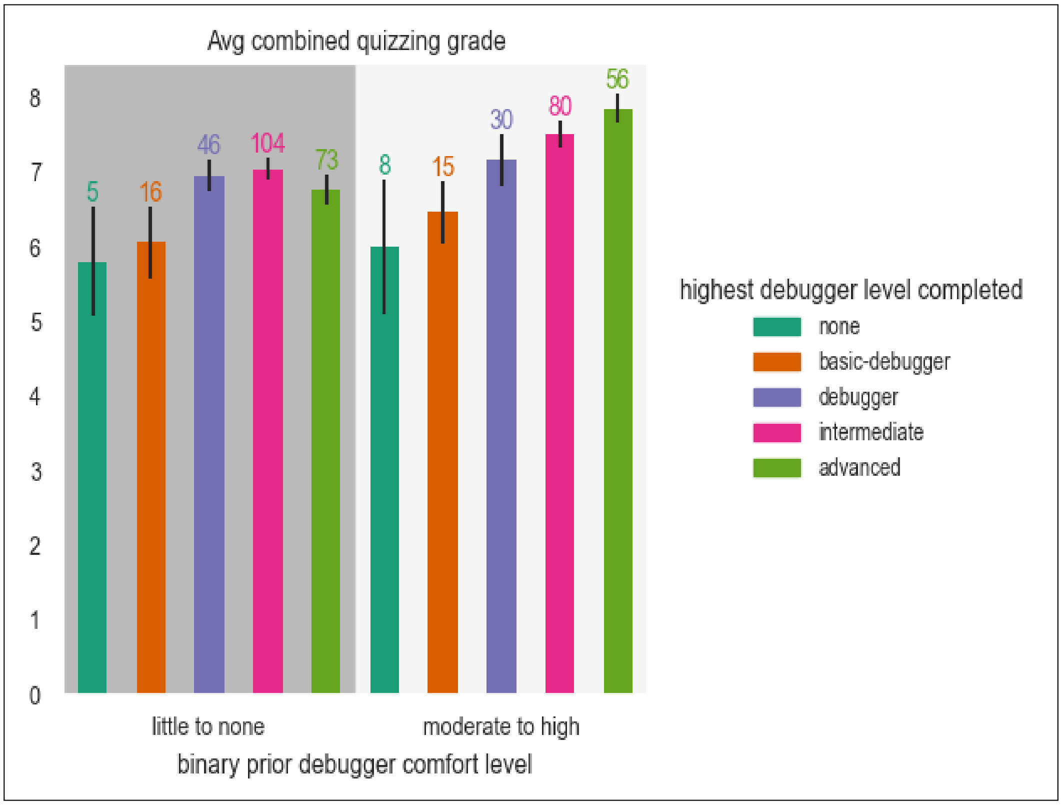
[2] Results were calculated with Python package `sm.stats.anova_lm`

population	sum_sq	df	F	PR(>F)
C(binary_prior_debugger_comfort_level)	37.980652	1.0	14.523935	0.000160
C(highest_debugger_level_completed)	29.732494	3.0	3.789937	0.010548

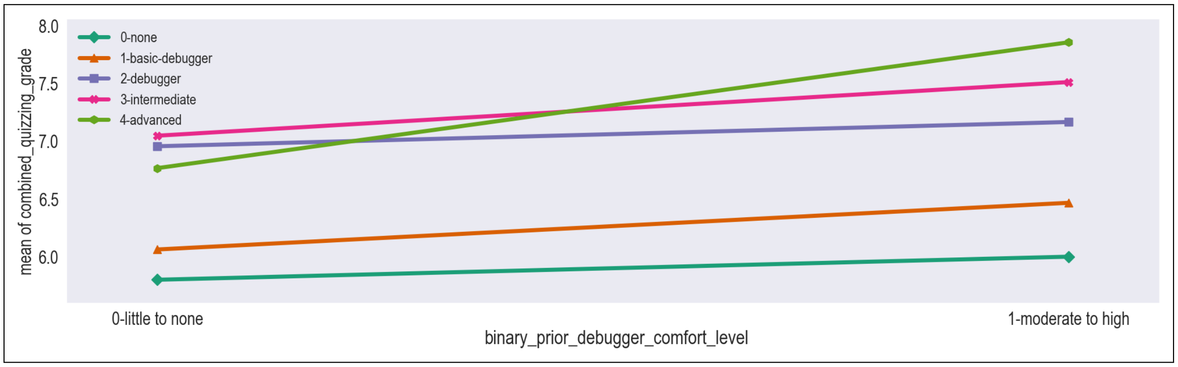
C(binary_prior_debugger_comfort_level):
C(highest_debugger_level_completed)
	11.490901	3.0	1.464720	0.223603

Residual	1077.395898	412.0	NaN	NaN

(a) 2-way ANOVA, with statistically significant result highlighted in yellow



(b) Prior comfort level and highest level completed on combined quiz grade



(c) Interaction plot of prior comfort level and highest level completed on combined quiz grade

Figure 6-21: Effect of Prior Comfort and Highest Level Completed on Combined Scores

Prior Debugger Comfort and Total Time Spent

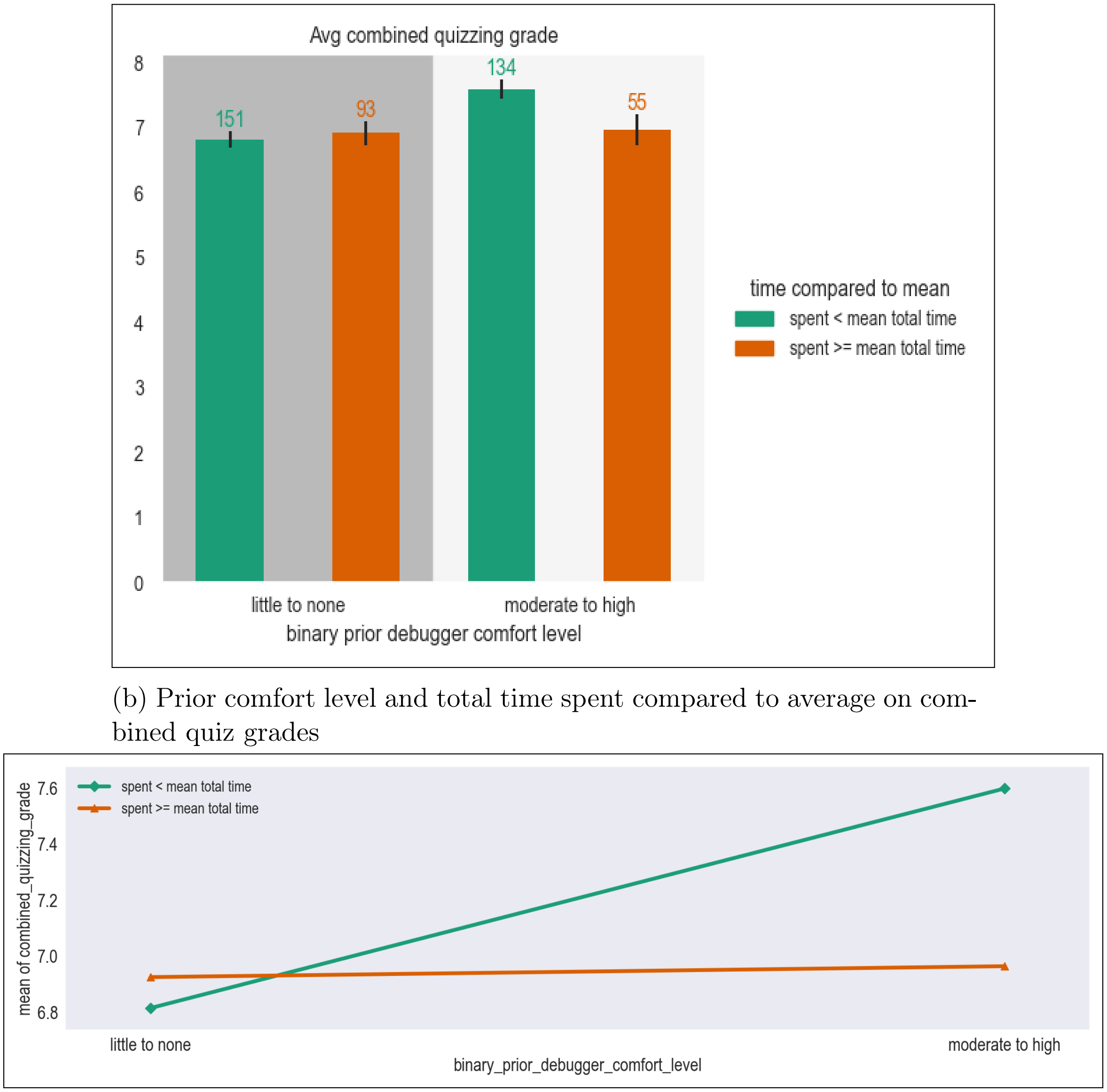
Next, another two-way ANOVA compared the effect of comfort level and time spent on the combined quiz score. This time, the sub-populations were split by if they spent less than the average time (42 minutes and 40 seconds) and more. Fig. 6-22 displays the results.

Evidently, time spent compared to the mean yielded a p-value greater than 0.05 and thus did not have a significant impact on quiz score. However, both prior comfort with the debugger and the interaction between prior comfort with time spent compared to the average yield a p-value less than 0.05 and thus have some correlation with quiz score.

Takeaway Interestingly, the interaction plot shows that, for students who have moderate to high prior comfort using a debugger, spending *less* time spent overall predicts scoring higher. However, this may be due to the fact that the highest speed is indicative of those who were previously *most* comfortable using a debugger above all others. For students with little prior comfort, on the other hand, spending more time indeed was correlated with a slightly higher combined quiz score, implying that for those students, more time spent with the Debug Tutor was helpful.

population	sum_sq	df	F	PR(>F)
C(binary_prior_debugger_comfort_level)	30.666706	1.0	11.135510	0.000921
C(time_compared_to_mean)	3.490754	1.0	1.267542	0.260858
C(binary_prior_debugger_comfort_level): C(time_compared_to_mean)	12.851134	1.0	4.666427	0.031311
Residual	1181.447144	429.0	NaN	NaN

(a) 2-way ANOVA, with statistically significant result highlighted in yellow



(b) Prior comfort level and total time spent compared to average on combined quiz grades

(c) Interaction plot of prior comfort level and total time spent compared to average on combined quiz grades

Figure 6-22: Effect of Particular Exercise on Quiz Answer

Prior Debugger Experience and Specific Exercise Completion on Quiz Answer

Prior experience with specific debugger skills may have also impact quiz results.

Breakpoint, Continue In revisiting the effect Backgammon-breakpoint-continue-navigation had on the quiz answer, a two-way ANOVA with prior experience with setting and removing breakpoints revealed that neither sub-population had an identifiable effect on the answer given in the quiz (Fig. 6-23).

Step In, Step Out In revisiting the effect Backgammon-step-in-out-navigation had on the quiz answer, a two-way ANOVA with prior experience with single stepping revealed that performance on the exercise still had an identifiable effect (yielding a p-value < 0.05) on the answer given in the quiz (Fig. 6-24).

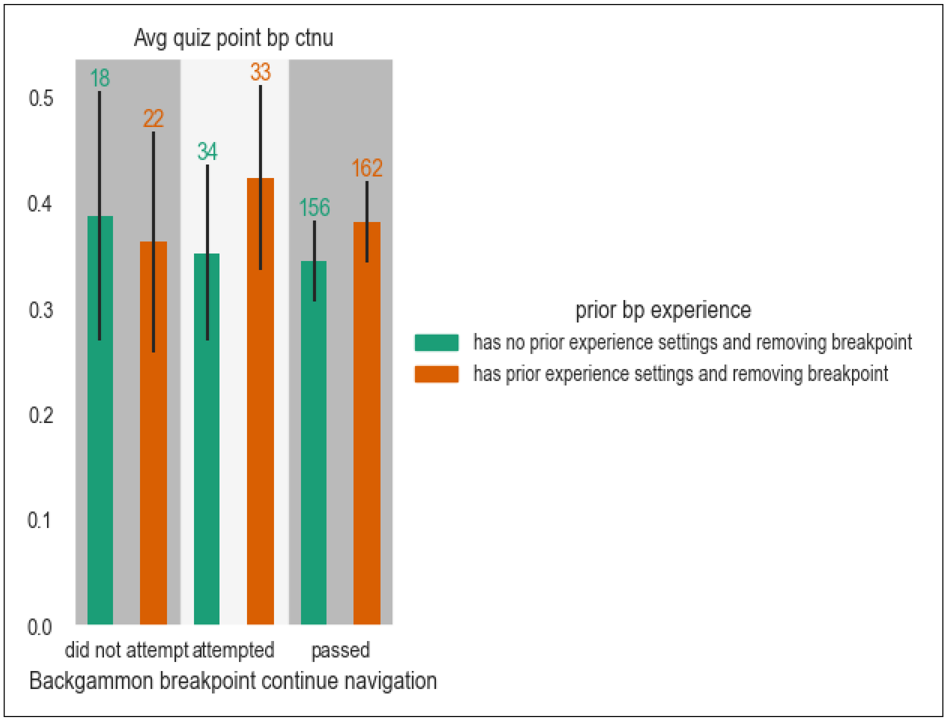
Takeaway Only the exercise on stepping in and out had a noticeable effect on the quiz answer. But it is encouraging that, regardless, the nature of the Debug Tutor's emphasis on micro-skills had some effect on students' ability to recall such microskill on the quiz.

population	sum_sq	df	F	PR(>F)
C(Backgammon_breakpoint_continue_navigation)	0.032353	2.0	0.068622	0.933690
C(prior_bp_experience)	0.139657	1.0	0.592430	0.441915

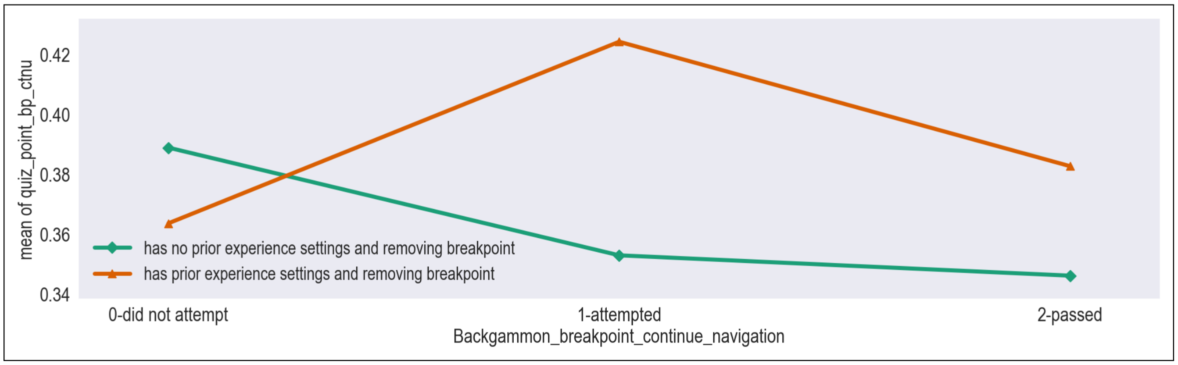
C(Backgammon_breakpoint_continue_navigation):
C(prior_bp_experience)
	0.058029	2.0	0.123081	0.884224

Residual	98.773296	419.0	NaN	NaN

(a) 2-way ANOVA, no statistically significant results



(b) Average of listing "breakpoint, continue" as an answer on Quiz 4.2



(c) Two-way Interaction Plot

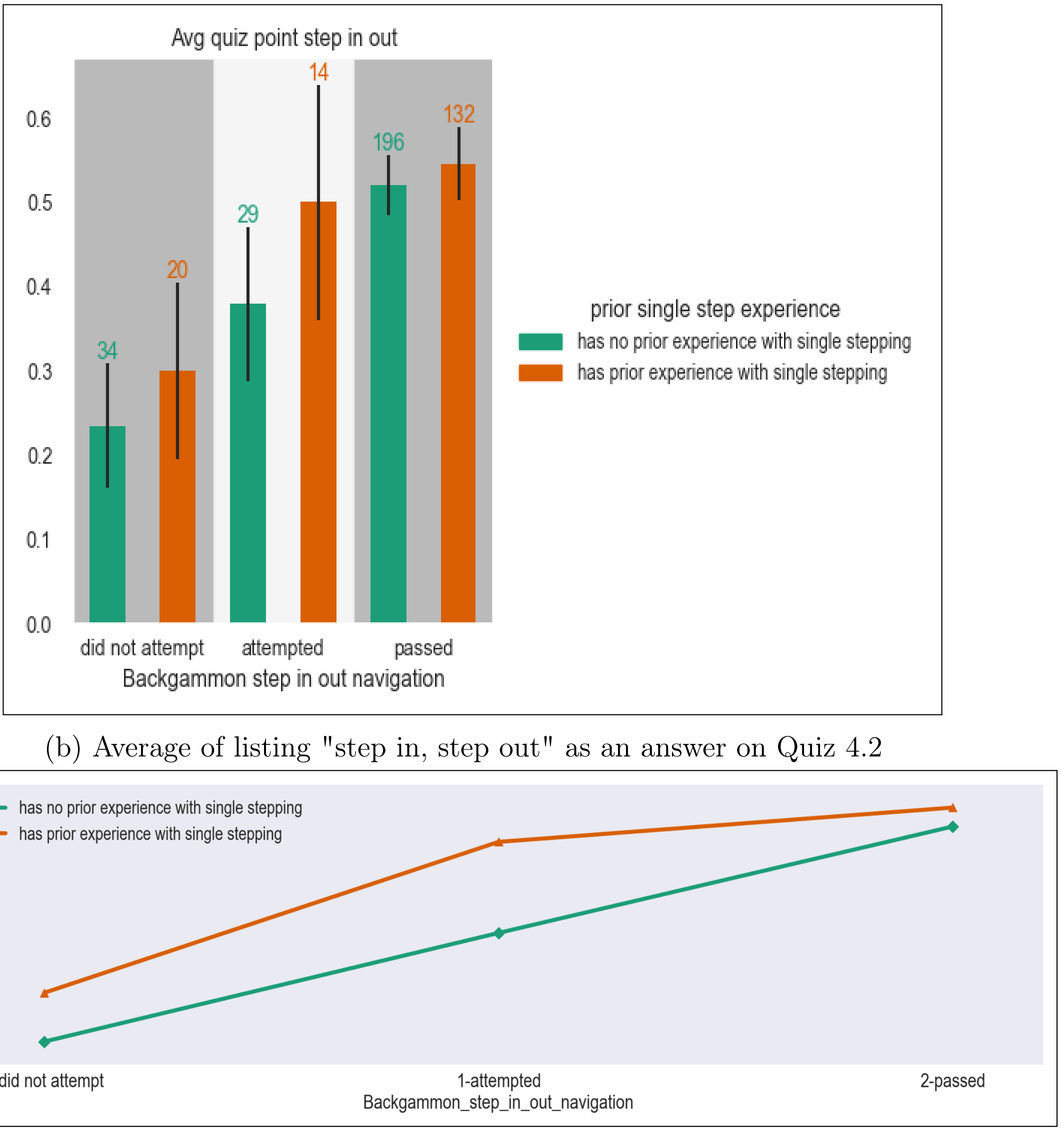
Figure 6-23: Prior experience with breakpoints and Backgammon-breakpoint-continue-navigation on quiz answer

population	sum_sq	df	F	PR(>F)
C(Backgammon_step_in_out_navigation)	2.214286	2.0	4.759843	0.011209
C(prior_single_step_experience)	0.190476	1.0	0.818898	0.368289

C(Backgammon_step_in_out_navigation):
C(prior_single_step_experience)
	0.023810	2.0	0.051181	0.950138

Residual	18.142857	78.0	NaN	NaN

(a) 2-way ANOVA, with statistically significant result highlighted in yellow



(b) Average of listing "step in, step out" as an answer on Quiz 4.2

(c) Two-way Interaction Plot

Figure 6-24: Prior experience with single stepping and Backgammon-step-in-out-navigation on quiz answer

6.4 Summary & Main Takeaways

On the whole, while most students had some form of debugger-related experience before 6.102 (Fig. 6-2), less than 25% of the class reported being comfortable with using the debugger prior to trying the Debug Tutor (Fig. 6-4). Regardless of the Debug Tutor's effect, given that most students in 6.102 are computer science majors and already in their Sophomore Spring semesters (Fig. 6-1), the self-reported comfort level certainly highlights the need for more debugging related training in the computer science curriculum.

The Debug Tutor exercises had a fairly high pass rate (Fig. 6-6), with the lowest pass rate still well above 75%. Although most students did not pass on their first try, almost all were able to pass within five attempts. Whether passing or giving up, most students spent less than four minutes per exercise. Whether previously comfortable with debuggers or not, students spent around 33 minutes completing around 30 Debug Tutor exercises. Therefore, the number of exercises and per-exercise difficulty, on the whole, were doable as recommended within a single reading assignment.

Three critical bugs were identified through problem report analysis. While frustrating for students, two of the bugs at least have workarounds, and all three will be fixed in the next release of the Tutor. The problem reports also confirmed that, for exercise authors, the collected data and the way exercise descriptions are formatted allow for quick and easy updating of exercise prompts and triggered hints based on unforeseen student answers.

Students' understanding of breakpoint setting and single stepping was tested on the ninth nanoquiz and on question 4.2 of the first quiz. While completing at least all concept groups through the single step group was correlated with a slightly higher score (Fig. 6-15), overall, performance on the nanoquiz revealed serious holes in the current concept map that should be addressed in future exercises: namely, the result of single step actions across various paused session states, such as stepping in when paused on a line without a function call, or stepping out when at the top of the call stack.

Students performed relatively well on quiz question 4.2, with over 75% giving at least one correct step sequence (Fig. 6-17). Unlike the nanoquiz, there was a stronger positive correlation between the quiz score and completing more concept groups (Fig. 6-18), though completing the final concept group provided no additional benefit. Perhaps more interestingly, performance on exercises that stressed particular step sequences, such as Backgammon-breakpoint-continue-navigation and Backgammon-step-in-out-navigation, was correlated with whether that step sequence given as an answer on the quiz.

The later exercises were among the most difficult (Fig. 6-6) and most time-consuming (Fig. 6-10), but did not provide much additional benefit to students when quizzed (Fig. 6-15, 6-18), which suggests that the concept map should include more exercises focused on basic skills and introductory skill combinations, instead of longer, integrated exercises such as those in the advanced concept group.

Not surprisingly, previous comfort level was also shown to be correlated with combined quiz score (Fig. 6-20). To investigate the effect of the Debug Tutor within each comfort population, the students were grouped into binary comfort levels (high or low), and then re-examined for effect of highest level completed and total time spent. Highest level completed was still shown to be correlated with high quiz scores, but time spent had an inverse relationship for students with moderate to high prior comfort. This perhaps was due to wasted extra time spent on advanced exercises that did not boost scores, or confused students gaining no new help from continuous attempts, perhaps also indicating that additional triggered hints are needed.

All in all, the Debug Tutor showed promising results in helping students understand microskills essential to competent debugging, such as use of the debugger, and its extensible design proved robust, flexible, and supportive for making updates in real time.

Chapter 7

Conclusion

This thesis presented the Debug Tutor, an automated tutor for explicit practice in debugging aimed for novice computer programmers. The Debug Tutor is an improvement on existing debug tutors, designed to drill essential debugging microskills agnostic to coding environments, such as single stepping in a debugger, yet is realized as an extension embedded in a widely-used professional code editor, and is therefore directly applicable beyond the classroom. Because the Debug Tutor gives automatic feedback and is completed on students' own time, it can be seamlessly integrated into the curriculum of classrooms ranging from small interactive discussions to large lecture-style computer science courses.

7.1 Discussion

The essence of the Debug Tutor is its concept map, which organizes debugger-related concepts, grouped by similarity, into levels. Each concept is realized by one or more concrete exercises. Each concept level can then be recommended to students at appropriate points in the curriculum.

The Debug Tutor was deployed in the Spring 2023 semester of MIT's 6.102 Software Construction class, a required computer science course at the sophomore/junior level. Students were asked to complete each concept group at applicable points throughout the course's ninth reading on debugging, due the night before the corresponding lecture on the same topic. Of the 542 students on the roster at the time, 86% had earned all seven Debug Tutor completion points by the start of class. Students spent around 35 minutes on around 30 exercises total, with even the hardest exercise yielding an above 75% pass rate. Students were also asked to report their prior experience with debuggers before use of the Debug Tutor. Although most students reported some previous experience using a debugger, under 25% reported moderate to high prior comfort using a debugger. Furthermore, the design of the Debug Tutor concept map and exercise description format proved flexible enough for exercise authors to add exercises and update existing exercises on-the-fly.

The class nine nanoquiz and part of the first quiz tested students' knowledge of single stepping in the debugger. Analysis of the prior comfort level and performance on the debugger exercises showed that completing more concept groups was positively correlated with higher quiz scores regardless of prior comfort level. However, the highest levels only provided additional benefit for those who were previously comfortable with using a debugger. Therefore, the Debug Tutor concept map should put more of an emphasis on shorter step sequences rather than advanced integration exercises.

The Debug Tutor was built on top of the existing Praxis Tutor framework. The framework's extension, webapp client, and server were extended significantly to support event tracking. The extended design was robust enough to handle hundreds of student attempters of different class years, majors, and operating systems. Furthermore, the event tracking API was implemented in such a way as to readily extended to new tutoring domains that hinge on event histories.

7.2 Future Work

Despite the overall success of the first Debug Tutor release, several improvements should be made moving forward.

7.2.1 Extension Updates

Most pressing are the three extension bugs identified in the first release.

Bug #1 To patch the automatic TypeScript compilation bug, the extension should instead compile all TypeScript files *before* uploading them to the server and should automatically install the `.js` files along with the `.ts` ones. Because debug exercises are read-only, pre-compiling them will avoid the need to compile in real time.

One way to implement pre-compilation in a language agnostic manner is to define a new `compilation` YAML key for each of the `languages` listed in `exercises.yaml`, with a `command` string and an `outFiles` list of file patterns. To avoid running the compilation on exercises that do not need it, a new computed value in the exercise model (or, in the case of the Tutor, the `requiresLaunchConfiguration`) can determine whether to pre-compile the exercise directory or not. Finally, when uploading exercises, the Tutor can automatically run the compilation command on those exercise directories and include any files matching the listed `outFiles` in the zipped folder after the compilation completes.

Bug #2 The terminal tracker should insert quotes if a path has spaces to avoid space-related bugs, and must update its link detection code to detect paths with spaces in them.

Bug #3 Similar to how VS Code decouples UI events from system updates, the event tracking answer checker should automatically consider all data values optional, only failing if data is mismatched.

This change would be localized to the RegObj matcher's individual object matching algorithm. If an exercise author would like to ensure that a data value is captured, they can do so by introducing latency into the exercise directions, for example by crafting the exercise such that the event requiring data values occurs at the end of the pattern, or perhaps even as a directive to the user to pause for a few moments to reflect on some part of the exercise.

7.2.2 Concept Map Updates

Based on performance analysis and quiz evaluation, the concept map should be updated with more single step exercises across a variety of paused session states and scale back on the more advanced, integrated exercices. These updates will both address the missing concepts revealed by the nanoquiz as well as the difficulty of the final two levels.

As 6.102 turns over new staff, it will also be important to investigate if and how easily new exercise authors are able to understand event tracking exercise descriptions and add to the Debug Tutor concept map.

7.2.3 Realtime Hints

With the introduction of event tracking into the Praxis architecture, it was hypothesized that feedback given in real time would greatly help improve the user learning experience. To that end, in addition to the exercise model's `triggeredHints` to display to the user after an incorrect submission attempt, the updated exercise model also includes `realtimeHints`, using same YAML format as `triggeredHints`, to be displayed in popups presented to the user at the exact moment the hint is relevant.

The extension was updated to ping the webapp on every relevant event action, which would in turn ping the server to compare the `realtimeHints` patterns against the user event log. Finally, the webapp-extension communication protocol was updated with a `showPopup` request, which the VS Code extension was updated to recognize and respond to with a modal or toast notification.

However, pinging the server on every student action is infeasible in a large system. So, although tested and functional, real-time hints were disabled to reduce the load on the server, with the exception of edit events during a read-only exercise triggering a real-time modal notification warning the student that edits are not allowed. In a future iteration, perhaps some version of an answer checker, at least for real-time hint checking, can be made available on the client side, thereby removing the need to ping the server on every action and allowing real-time hints to be re-enabled.

Multi-Step Exercises

As discussed earlier, based on the evaluation, the Debug Tutor's concept map should pare down the more difficult intermediate and advanced debugger concepts. The multi-step nature of the exercises implementing these concepts would likely have benefited the most from real-time hint popups. Perhaps, when the improved client-only real-time hint design is implemented, it would be worth re-introducing the more complicated concepts back into the Debug Tutor's concept map, as the multi-step exercises would become more accessible to all students.

Another option might be to introduce a multi-check flow into the webapp's event loop, to support exercises that require multiple check steps before passing. For example, a student might be shown the first half of an exercise prompt and would then have to click Check and pass before being shown the second half of the exercise prompt. A final Check and pass would then mark the exercise as complete. However, muli-check exercises may get complicated if they also require starting over. For example, if a student fails in the second half, would the exercise start over from half-way or from the beginning?

7.2.4 A Hypothetical Case Study: Extending to Git

As a final exercise testing the extensibility of the event tracking API, this section presents a walk through of a hypothetical case study: developing a Git Tutor.

Step 1: Is VS Code's Git API accessible?

Does VS Code exposes Git state and events? Good news—it does! VS Code comes pre-packaged with a git extension with an accessible API[1], which exposes the workspace's open repositories. Each repository object contains state information like the latest commit, status, and head branch, as well as event listeners such as `onDidRunGitStatus` and `onDidRunOperation`. Perfect!

[1] Via `vscode.extensions.getExtension('vscode.git')?.exports`; See https://code.visualstudio.com/docs/sourcecontrol/overview and https://github.com/Microsoft/vscode/blob/main/extensions/git/src/api/git.d.ts

Step 2: Update the Event Tracking API.

Define an IDEGitTracker in event-tracking-api.ts with an IDEGitEvent enum (it may have, for example, *getStatus*, *add*, *commit*, *push*, etc.) and event type TutorGitEvent. Once the namespace is defined, HostIDE's IDEEventType, EXTENSION_EVENTS, and TutorEvent can be updated accordingly.

Step 3: Add a Git property to the Configuration model.

Next, add a Git property to the Configuration model. It is likely that a configuration setup will consist of pre-running git commands on a repo (each could be represented by a GitCommand, similar to the Debugger's SessionData). Potential final states might be the branch HEAD or number of commits (which could be represented by a RepoState, similar to the Debugger's BreakpointData). So, an initial pass at a Git model to store inside the Configuration might have a list of GitCommand preRunCommands and a RepoState repoState.

Step 4: Write a GitTracker.

With the git data organization established, tie the pieces together in a GitTracker implementing the TutorTracker interface. The Git configuration is defined, so implement configure and compileConfiguration accordingly. It is unclear whether git requires explicit cleaning, so leave that empty. Implement register using the appropriate callbacks and pushing TutorGitEvent of IDEGitEvent types. An initial pass at teardown should dispose of those listeners. Finally, fix the compile-time error in extension.ts by adding git:TutorGitTracker.tracker to the EVENT_TRACKERS.

Step 5: Author some exercises.

Last but not least, author some exercises using the new git configuration and stream events. For Git in particular, one final change may be explicitly running git init and git remote add origin in the background (similar to how npm install is run) for a git exercise—perhaps the exercise model can have a computed property marking if git is required, similar to requiresLaunchConfiguration.

And of course, the final step: Design the concept map and write some exercises!

Bibliography

[1] M. Ahmadzadeh, D. Elliman, and C. Higgins. Novice programmers: An analysis of patterns of debugging among novice computer science students. *Inroads*, (37):84–88, 2005.

[2] E. Carter and G. D. Blank. An intelligent tutoring system to teach debugging. In *P. Pavlik, K. Yacef, J. Mostow, H. C. Lane (Eds.), Artificial Intelligence in Education*, 16th International Conference, Memphis, TN, USA, 2013. Springer Berlin Heidelberg.

[3] S. Carver and S. Risinger. Improving children's debugging skills. In *In G. Olson, S. Sheppard E. Soloway (Eds.), Empirical studies of programmers*, Second Workshop. Ablex, 1987.

[4] R. Chmiel and M. C. Loui. Debugging: from novice to expert. In Proceedings of the 35th SIGCSE technical symposium on Computer science education (SIGCSE '04), pages 17–21, Memphis, TN, USA, 2004.

[5] C. Du. Empirical study on college students' debugging abilities in computer programming. *First International Conference on Information Science and Engineering*, pages 3319–3322, 2009.

[6] M. Ducassé and A. M. Emde. Empirical study on college students' debugging abilities in computer programming. Proceedings of the 10th international conference on Software engineering, page 162–171. IEEE Computer Society Press, 1988.

[7] Anders Ericsson and Robert Pool. *Peak: Secrets from the New Science of Expertise*. Houghton Mifflin Harcourt, 2016.

[8] D. H. Jonassen and W. Hung. Learning to troubleshoot: A new theory-based design architecture. *Educ Psychol Rev*, (18):77–114, 2006.

[9] O. Kiljunen. Teaching students to fix programming errors with tutorials embedded in an ide. *21st Koli Calling International Conference on Computing Education Research*, pages 1–3, November 2021.

[10] A. N. Kumar. Model-based reasoning for domain modeling in a web-based intelligent tutoring system to help students learn to debug c++ programs. In *In*

S. A. Cerri, G. Gouarderes, F. Paraguacu (Eds.), Intelligent Tutoring Systems, 6th International Conference, page 162–171, Biarritz, France and San Sebastian, Spain, June 2002. Springer Berlin Heidelberg.

[11] M. Laakso, E. Kaila, and T. Rajala. Ville – collaborative education tool: Designing and utilizing an exercise-based learning environment. *Educ Inf Technol*, 23:1655–1676, 2018.

[12] G. C. Lee and J. C. Wu. Debug it: A debugging practicing system. *Computers Education*, 32(2):165–179, 1999.

[13] M. J. Lee. *Teaching and engaging with debugging puzzles*. PhD thesis, University of Washington, 2015.

[14] C. Li, E. Chan, P. Denny, A. Luxton-Reilly, and E. Tempero. Towards a framework for teaching debugging. In *In Proceedings of the Twenty-First Australasian Computing Education Conference*, ACE '19, page 79–86, 2019.

[15] A. Luxton-Reilly, E. McMillan, E. Stevenson, E. Tempero, and P. Denny. Ladebug: an online tool to help novice programmers improve their debugging skills. In *In Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2018, page 159–164, 2018.

[16] R. Mccauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: A review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92, 2008.

[17] T. Michaeli and R. Romeike. Improving debugging skills in the classroom: The effects of teaching a systematic debugging process. In *In Proceedings of the 14th Workshop in Primary and Secondary Computing Education*, WiPSCE'19, page 1–7, 2019.

[18] M. A. Miljanovic and J. S. Bradbury. Robobug: A serious game for learning debugging techniques. In *In Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, page 93–100, 2017.

[19] M. Nanja and C. R. Cook. An analysis of the on-line debugging process. In *In G. Olson, S. Sheppard E. Soloway (Eds.), Empirical studies of programmers*, Second workshop, page 172– 184, 1987.

[20] J. C. Spohrer and E. Soloway. Novice mistakes: are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632, 1986.

[21] I. Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[22] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Elsevier Science, 2009.