– Independent Work Report Fall 2021 –

BUGCATCHER!: Teaching Debugging Techniques through Gamification

Monique Legaspi
Advisor: Robert Fish

Abstract

Students in COS 126, Princeton University's introductory computer science course, struggle with debugging on their own; there are no official or recommended course resources for learning or practicing debugging, and any resources a student could find on their own can be tedious and difficult to understand. Beyond that, the Java-specific nature of COS 126's curriculum necessitates a debugging resource focused on Java, which further limits the scope of what students may consider relevant, useful, and worth pursuing. This project seeks to teach COS 126 students to debug in Java by providing a gaming platform in which they may directly edit and run buggy code. This paper will examine the efficacy of this project at teaching debugging practices, as well as whether incorporating gamified elements (such as graphics and scoring systems) encourages students to engage more closely with the material. The utilization of this project hopes to facilitate more confidence and efficiency in COS 126 students' debugging skills.

1. Introduction

Learning something new is always difficult, and this is especially true when it comes to learning computer science. In order to code even the simplest of computer programs, one has to learn the proper syntax of the language they're using, the logic behind the function of the program, and the ability to debug the program to ensure it's working correctly. COS 126, Princeton University's introductory computer science course, takes care of teaching its students the former two aspects – it prides itself on a Java-focused curriculum that lavishes in learning not only the ins and outs of basic Java syntax, but also rudimentary data types, loops, recursion, and algorithms, among many more essential topics for the burgeoning computer programmer[8]. However, the course leaves no

room for teaching students to debug; there is no formal curriculum unit, and there are no provided resources that allow students to practice debugging on their own. I have observed, both as a COS 126 alum and as a grader for the course, that students struggle to find and fix bugs in their own programming assignments. Oftentimes, rather than step through their own code to look for bugs in an improperly-behaving program, they will defer to submitting their project right away, allowing the Autograder to catch their mistakes via its rigorous tests. This practice, while convenient in the short-term, is not sustainable. Particularly for students who go on to pursue careers in CS, the Autograder does not exist to check programs in the workplace, and even the next classes in the COS sequence, COS 226 (Data Structures and Algorithms) and COS 217 (Introduction to Programming Systems), place limits on how many times a student can submit their code for review, forcing them to be more diligent with debugging their own code despite never being taught the best practices for doing so. Here, the student may turn to the Internet for debugging wisdom – but resources are restricted due to Honor Code worries (they can't go to StackOverflow for fear of accidental plagiarism), programming knowledge limited to Java (tips for debugging in C or Python will be nonsensical to them), and time/attention-span restrictions (long articles and documentation take too long to parse for answers). Learning to debug as a COS 126 student should not have to be so laborious!

For my independent work, I wanted to design an application specifically for COS 126 students that allows them to learn and practice debugging alongside their course curriculum. I devised a list of desired traits for this project, based both on qualities observed in previous work and on my own preferences as a college student who also wished to learn debugging:

1. This application needs to be **intuitive**; it must be easy for the average COS 126 student to learn and understand, without requiring any prior knowledge outside of what is already taught in the course.
2. It must also be **engaging**; in keeping with the desire to make this learning process as easily digestible as possible, I turned to gamification for inspiration. As has been previously observed, adding game-like elements to educational material increases engagement with the material and

allows students to retain the learned information better[9], so I sought to add the same elements to my project.

3. It needs to be **age-appropriate**; many debugging resources online are meant for teaching basic problem-solving skills to primary- and middle-schoolers[10, 5, 6]. Most college students have already surpassed a middle-schooler's level of logical thinking, and as such would likely not get anything useful out of a resource aimed at younger minds.
4. Finally, it should be **Java-focused**. Researchers have observed that most programming bugs are due to syntax errors[10, 7], which are specific to the language being learned, and since COS 126 is taught exclusively in Java, it would be best to teach students how to debug in Java, too.

An application that closely adheres to the above four qualities would best achieve my goal of encouraging the thorough and independent learning of debugging skills in COS 126 students.

2. Related Work

Despite being a relatively niche topic, many researchers have already studied debugging education in the past, producing projects that serve a similar function to my own, with varying levels of success. In conducting my preliminary research, I followed three such cases – Gidget, G4D, and Ladebug – in order to determine which factors played into their success, as well as what other features I would require in order to adapt the same basic ideas to students undergoing the COS 126 curriculum.

2.1. Gidget

Developed by Michael Lee for his PhD thesis at the University of Washington, Gidget is a "social debugging game"[5] in which the player helps a robot, Gidget, clean up pockets of slime spread by a factory accident. However, during the accident, Gidget has been damaged, and the player must debug his code so that he may clean up properly. The game is partitioned into levels, each preceded by a robust tutorial that teaches the player new concepts necessary to completing that particular level. The levels build up naturally in difficulty, facilitating a smooth learning experience. The interface (see Figure 1) is simple and eye-catching – a visual of Gidget and his surrounding

environment takes front-and-center, with editable code to his left, and necessary level statistics to his right.

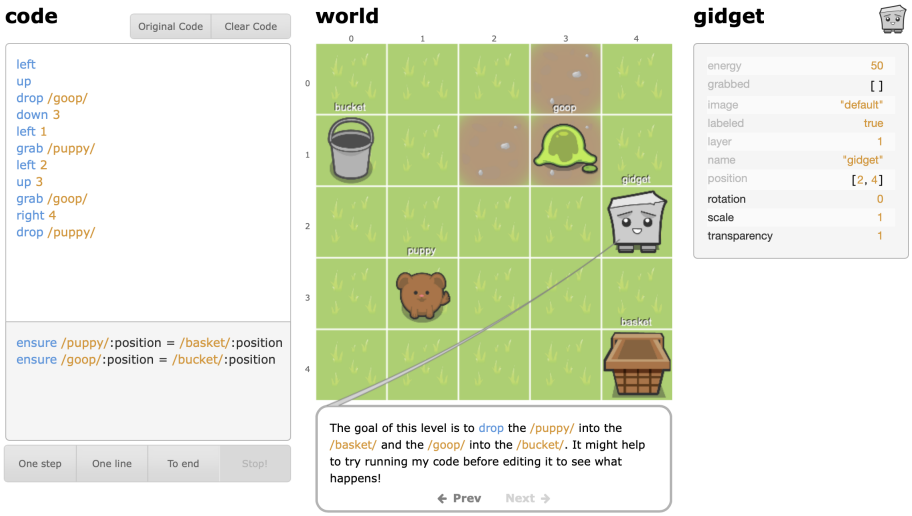

Figure 1: A snapshot of Gidget's interface.

Gidget[1], which is freely available to play online, has gained thousands of users since its public release[6], and Lee demonstrated in an experiment that students who learned concepts from Gidget versus Codecademy (an online tutorial site) "showed similar learning gains, with Gidget players doing so in about half the time"[6], signifying how Gidget's gamified nature allows students to pick up new concepts more quickly. Additionally, by anthropomorphizing the to-be-debugged subject, Gidget, and having him interact with the player, Lee showed that "novice programmers are engaged with an education when ... [t]he computer compiler/interpreter is personified"[6], building a strong case for the inclusion of a non-playable character (or NPC) who actively recites important information to the player, rather than displaying that same information as plaintext.

While Gidget clearly showcases the merit of teaching debugging through gamified means, when placed in the context of teaching university students – particularly those enrolled in COS 126 – to debug, it falls short of our previously-stated goals. With its rudimentary gameplay and rehashing of basic programming logic, the game is aimed more at middle-school audiences who are beginning to build their own problem-solving skills in general, rather than at college students who have already

[1] https://www.helpgidget.org

displayed the capacity for complex thinking. The program also utilizes a pseudo-language with its own unique syntax, which is the biggest obstacle. While Lee designed the language to be English-like and easy to pick up, the hassle of having to learn the syntax of an entirely new language is unjust to place upon a COS 126 student, who already has so much to learn about Java, and who cannot apply their knowledge about syntactical errors in the pseudo-language to debugging their own Java syntax errors.

2.2. G4D

Developed by researchers Akhila Sri Manasa Venigalla and Sridhar Chimalakonda, G4D is a treasure-hunting game in which the player inhabits the role of Veda, a girl "whose spaceship crashed on an alien planet"[10]; she must travel to various cities, each of which comprises a level in the game, in order to find the tools she requires to repair her spaceship, which runs on buggy C code. The world of G4D entails a more immersive gaming style than previously displayed by Gidget; the player can move Veda across a 3-dimensional terrain and explore as they see fit. Once they have finished finding all of the clues, they may return to Veda's spaceship and use those clues to locate and fix bugs in snippets of C code (see Figure 2).

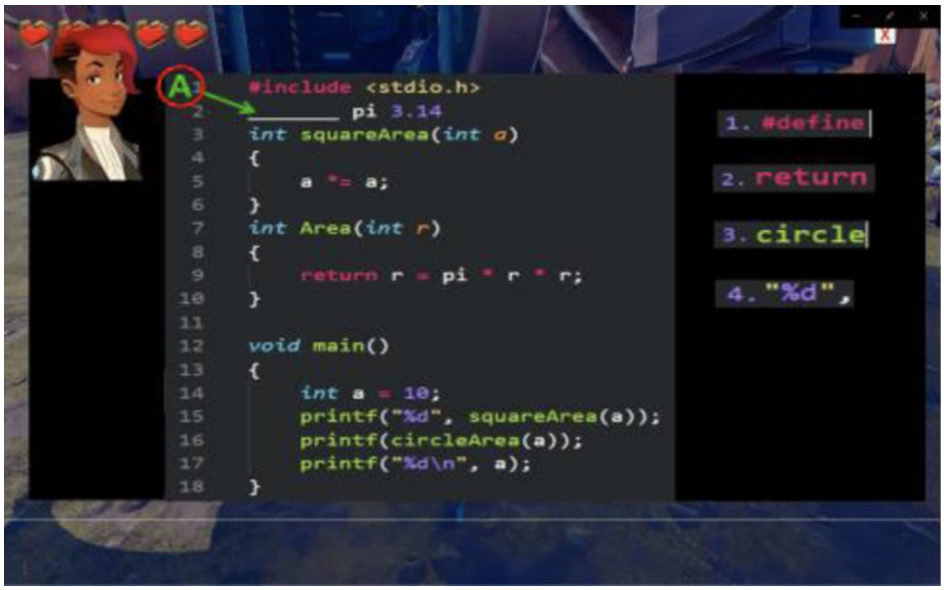

Figure 2: A snapshot of G4D gameplay[10].

G4D is different from Gidget in that its creators have a special focus on the syntactical aspect

of debugging. Venigalla and Chimalakonda have observed that syntax takes up the majority of concern for debugging, but most other debugging resources focus on logic errors; therefore, they wanted G4D to allow players to "isolate the chain" (aka, the line on which the bug occurs) and then "correct the defect on the isolated chain"[10]. Additionally, in noting that many debugging tools found online are designed for people in primary and middle school (one of which, as they cite in their paper, being Gidget), they sought to design a game specifically for first-year university students – one that is appropriately difficult and entertaining.

Based on user testing conducted using their prototype of G4D with 20 volunteers within their proposed age demographic (17-25 years old), all with varying levels of programming experience, the researchers found their game was easy to play, engaging, and appropriately challenging[10]. However, in selecting participants for testing, the researchers only briefly mention that the majority of participants already played video games at least regularly prior to testing[10], which may have skewed the data in terms of ease of play for the average computer science student. Although G4D is not available to play online, descriptions of the game allude to the storyline being quite expansive and involved, which, alongside the learning curve of playing video games in general, could additionally distract from actual learning learning of debugging techniques for which the game was originally designed. The researchers admit that the clues given for debugging are not as well-integrated into the story as they could be, which could place a "cognitive load" on players who need to absorb both the story and the clues at the same time[10]. Finally, as with Gidget, G4D is not fit for use with COS 126 students due to its focus on a non-Java programming language, C in this case. Although it has the same commitment to teaching syntax-based debugging, learning to debug C syntax is ultimately not helpful to someone who only knows, and is only coding in, Java – especially those students who are pursuing a major in the humanities and have no intention of coding after COS 126.

2.3. Ladebug

Developed by researchers Andrew Luxton-Reilly et al., Ladebug is an online debugging practice platform meant for use alongside the introductory computer science courses, CS1 and CS2, at the University of Auckland. Students are given debugging exercises to complete in Python, as well as a sandbox where they may experiment with their own Python code. The environment in which the student completes their exercises is designed to be simple and intuitive, with large buttons and basic, contrasting colors (see Figure 3). Exercises have methods of flagging errors, "inspecting" variables to view their properties (global vs. local, value stored, etc.), running code, and setting breakpoints. Upon completing an exercise, the student receives a number of stars out of 5, based on how well they did. Each level is also fuly customizable by course staff, so the content may evolve alongside the course.

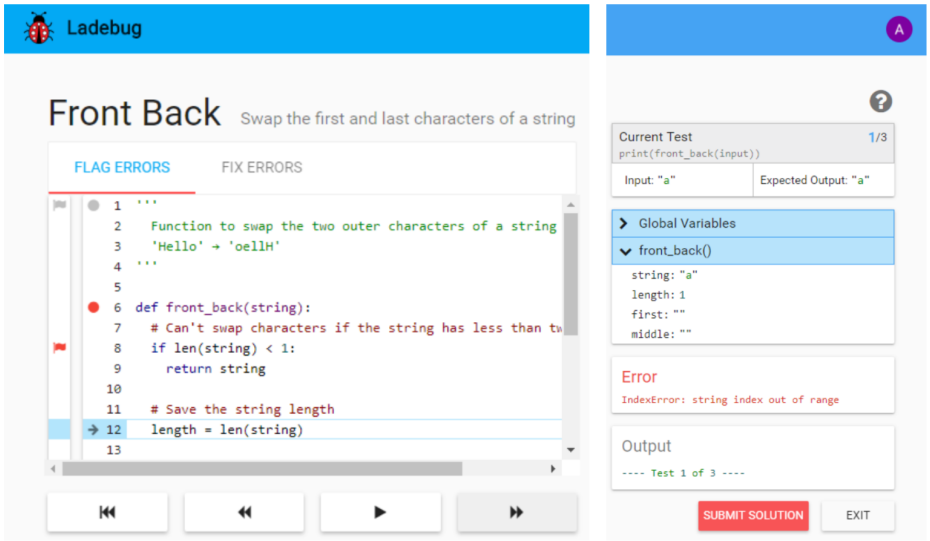

Figure 3: Snapshots of the left and right panels of a Ladebug exercise[7].

Upon testing with Auckland first-year computing students, Ladebug received generally positive reviews. The students found it easy to use and understand, and they indicated a desire to use the system frequently[7]. In particular, some students appreciated being able to step through the code and flag errors as they debugged[7].

Although Ladebug is more of a classroom-adjacent learning tool than a game, it displays some

gamified elements, such as the star system; however, users did not indicate whether the star system had any impact on their enjoyment of the experience, nor whether it influenced them to redo certain exercises[7]. As with the previous two cases, Ladebug does not teach debugging in Java, which again leaves it unfit for use in a COS 126 context. It is engineered in the same spirit as my own project, though, having been conceived for the purposes of use in one very specific context (namely, a particular college's introductory CS course), so this inability to adapt is understandable.

2.4. Takeaway

The three cases observed above each display promising qualities, partially fulfilling the criteria set forth in defining my goals for this project. Gidget is quite easy to understand and play along with; it warrants replaying through its charming game features, but its assumed skill level is not appropriate for university students. G4D is a more college-leveled game that zeroes in on syntactical debugging, but the mechanics may be difficult for users to learn if they are not already well-versed in video-gaming. Ladebug shows the most promise, in that it is easy to use, has a multitude of debugging aids, and is designed specifically for the course its niche audience is taking, but it lacks the level of engagement promised by a more gamified platform. All three cases do not offer debugging in Java. We may compare their fitness according to the goal criteria in the table below:

Application	Intuitive?	Engaging?	Age-appropriate?	Java-focused?
Gidget	Yes	Yes	No	No
G4D	Maybe	Yes	Yes	No
Ladebug	Yes	Maybe	Yes	No

Table 1: Related work, judged according to goal criteria.

In completing this project, I sought to synthesize the best aspects of these applications in a product that fulfills all criteria, and in a way that is best suited specifically to the needs of COS 126 students.

3. Approach

My novel approach involves the construction a web-based game where the player works to debug simple Java programs in levels, with each level centered around a specific type of bug one might encounter while constructing and debugging in Java. This focus on Java is the most important aspect; although some previous works shared the idea that syntax is incredibly important to hone in on while debugging, they solved this problem by tightening their curriculum to different specific languages, which are not well-suited to COS 126 students. Gamifying the website is crucial as well; rather than a plain exercise website where one trudges through lectures and quizzes, the environment must be enjoyable for the player. I decided to keep the gameplay as simple as possible, though, so as to minimize any learning curve associated with picking up the mechanics of a new game. While a more immersive game (à la G4D) would be exciting, engaging with an overly-detailed storyline could distract from the true purpose of the site, which is to learn and practice debugging. More energy would be expended on investing oneself in the game rather than on absorbing the debugging material, which then becomes peripheral. Finally, in keeping with the findings put forth by Lee as he designed and tested Gidget[6], I wanted to incorporate a personified tutorial guide, as it would allow the player to ease into the game without confusion, and it would introduce new concepts in a way that is more easily digestible.

4. Implementation

4.1. Overview

The final product of this project, BUGCATCHER!, is a web game that can be played from any laptop or desktop computer. Although it was designed with COS 126 students in mind, it can be accessed by anyone at https://falseaxiom.github.io/bugcatcher/, and its GitHub repository is also publicly accessible at https://github.com/falseaxiom/bugcatcher. Within the game, the player inhabits the role of a "bugcatcher" (hence the name of the site) who travels the world, seeking to

repair endangered “environments” (aka, Java programs) that are plagued by bugs. These bugs are damaging the environments’ ecosystems, preventing them from functioning properly.

The world of BUGCATCHER! is divided into levels, each with its own category of bug, for which the player must be on the lookout as they complete the levels. Each level (excepting the tutorial, Level 0) is divided into 4 uniquely buggy environments. The code in the environment is directly editable by the player; their goal is to edit the code in such a way that all bugs are removed, and the environment is able to run as intended. Once an environment has been sufficiently debugged, the player may move on to the next environment, and the next, until they complete the level.

At the end of the level, the player is awarded a score based on how efficiently they debugged. If it is their first time playing the level, the player unlocks the next level, which was previously inaccessible via the level select page. If they have already played the level, the application compares their current score to their high score (which is stored locally – see Section 4.2 below for details) and alerts the player to whether they have beaten this high score.

4.2. System Architecture

Due to the short timeline of the project and the limitations of my own abilities in realizing a full-fledged game website, the architecture of the project was kept relatively simple. See Figure 4 below for a full diagram.

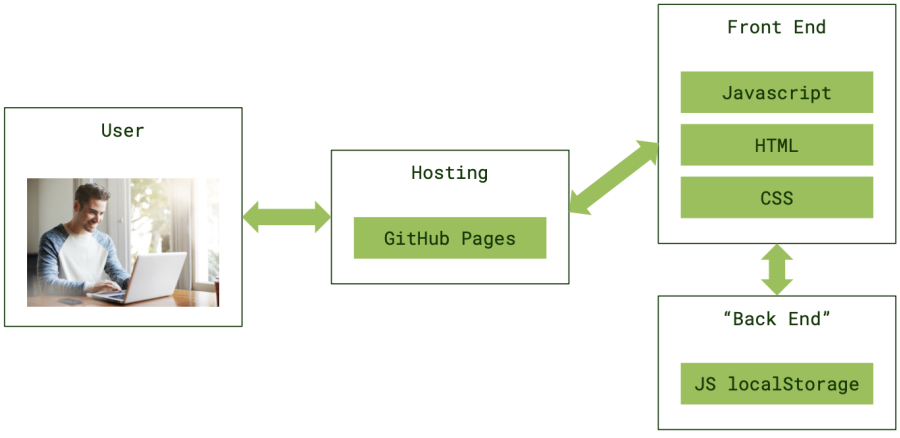

Figure 4: System architecture of the BUGCATCHER! website.

BUGCATCHER! is a static website hosted on GitHub Pages. All design components were created using modular HTML/CSS code, where each page of the website has its own HTML and CSS styling files. Pages utilize JQuery to display common elements, such as the header and footer. They also use JavaScript to populate level content, to display and hide certain conditional components (such as the tutorial dialog), and to allow buttons to perform functions such as "submit" code for review and toggle between levels.

Rather than using a true backend that stores scores and level progress in a separate database, this application instead makes use of localStorage, which creates and keeps track of variables within the user's browser. This allows for variables to be passed easily between pages, which in turn means that levels may be locked and unlocked according to the user's continued progress across multiple sessions, and high scores of levels can be kept track of and displayed whenever the user attempts to beat them. Additionally, if the user closes the tab in which they have the site opened, then returns to the site later on, all of their progress is saved in localStorage. So long as the user does not reset their browser data or attempt to open the site in another browser, they retain their progress on the site for as long as they wish. By using localStorage, one also does not have to create an account and log into it to access their own personal data. Since there is no sensitive information being handled over this site (localStorage is used only to track level progress and high scores), I did not see this as a security concern – although I will discuss this and other limitations of using localStorage in Section 6 later on.

4.3. Levels

4.3.1. Composition & Flow. Level design began with thorough research into what bugs are most common amongst beginning programmers. I wrote and distributed a Google Form amongst current and past COS 126 students to get a sense of what bugs they encountered in their own time as a student of the course. I provided a list of errors I felt might be common – some based on my own experience in the course, others based on lists I'd found online[3]. Surprisingly, indexing errors were the most common by a landslide; I had not anticipated this, because IntelliJ (the integrated

development environment, or IDE, which COS 126 students use to complete their programming assignments) automatically throws an error when you have an indexing bug, so I had assumed this would be easily fixable for most people. Among the other top 5 errors most commonly cited were typos, static vs. non-static, global vs. local variables, and public vs. private. See Figure 5 for the full graph.

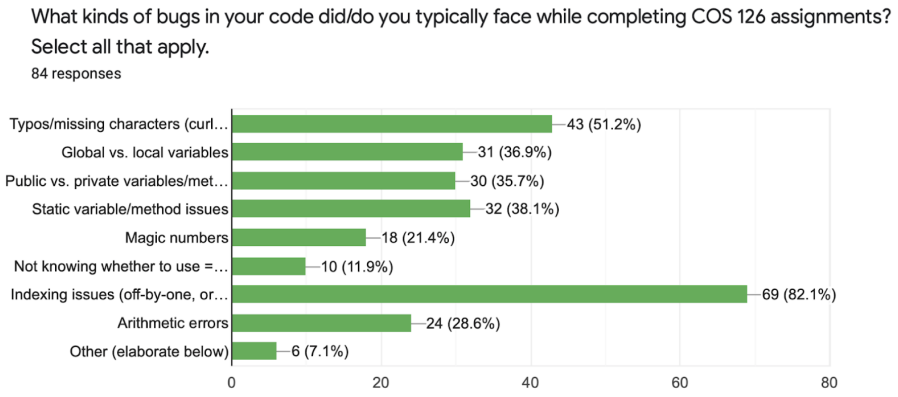

Figure 5: Commonly-encountered bugs, as indicated by current and past COS 126 students.

In a later section, where students could express their own bug findings, a good number of them cited logic errors. Although I went into the project intending to focus only on syntax, this concern about logic bugs inspired me to include a level focused specifically on at least the basics of logic debugging.

Beyond feedback given by COS 126 students and alums, I also turned to online databases and articles that listed common errors, Java or not, experienced by programmers of all levels, to get a sense of the kinds of syntactical and logical errors that this debugging game would have to focus on[3, 4]. From here, I discovered that, indeed, the same problems that were encountered by COS 126 students were also encountered by beginning Java programmers at large.

In its current state, the project has 1 tutorial level (focused on teaching the basics of how to use the site) and 4 real levels, each focusing on a different topic. I organized the topics by what I felt was most- to least-intuitive – the player starts by isolating and fixing typos, then moves on to determining variable scope, then to indexing errors, then to basic logic errors. Each level focuses only on the current topic; from intermediate trials and feedback with colleagues, I learned that

making the levels cumulative would ultimately distract from being able to fully learn the topic at hand, and a more cumulative test would be administered in the surveys at the testing stage, anyhow.

4.3.2. Tutorial Dialog. Upon entering a level for the first time, the player encounters the level page with an extra overlay on top, displaying a box which contains the tutorial dialog (see Figure 6). Following Gidget's eponymous, personified computer guide, I inserted an NPC named Thimis Hintz[2] to deliver important information in a fun and engaging manner. Thimis is a more seasoned entomologist in the player's field who informs the player about the various bugs that inhabit each level. His dialog mentions each of the specific species of bugs the player will encounter (for example, in Level 1: Syntax, he tells the player that semicolons are typically found at the end of most lines, but can sometimes appear in other places they don't belong), and gives explanations of why they appear and how they may be fixed. Some of Thimis's dialog is accompanied by graphics, which better illuminate the concepts he is speaking on. The player advances the dialog by clicking anywhere on the screen. Each level's tutorial dialog script is housed in a separate .html file, which each dialog bit inhabiting its own , called upon by level.js whenever the user advances the dialog. The player may also choose to skip this tutorial dialog via the "skip tutorial" button underneath the dialog box if they already know the topic. Additionally, Thimis's dialog is enacted only on the first play of a level; once a level has been completed, it is assumed that the player has adequately learned its contents, and thus does not need to reread Thimis's dialog upon replaying.

4.3.3. Environment. As stated in Section 4.1, each level (excepting the tutorial) has 4 environments to debug. The first 3 environments of a level are identical in function, so as to allow the player to focus on finding bugs, rather than being distracted by learning a new program's intended function. The program that populates the first three environments is intentionally simple in function – for example, in Level 1, the program merely prints the sum and product of the contants of an int[] array – and the bugs are relatively easy to find. The last environment is slightly more complex, and it contains a few more bugs than the first three; this is intended as a "check" to ensure that the player

[2]Named after the protagonist of a story told by a friend, Hank Ingham '23, who derived the name from that of another friend, Thomas Hontz '22.

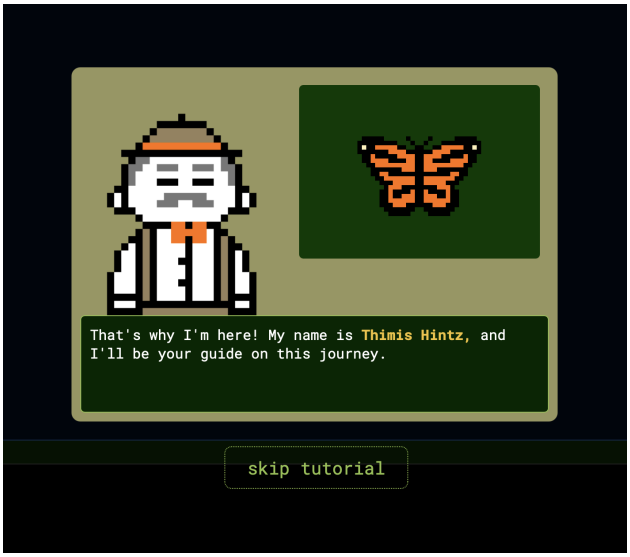

Figure 6: Thimis Hintz's tutorial dialog box.

understands what they learned from the first three environments, and can readily apply those skills to more complicated programs.

The basic layout of an environment can be observed in Figure 7. A typical environment consists of a fixed-size <textarea>, where the buggy code is on display. This code is pulled from a separate .txt file, labeled according to the level and environment. To the right of the <textarea> is a box containing a brief description of what the code should do (for example, print a specified statement), a button for asking for "Hintz" (elaborated upon in the next section), a button to "run" the code, and a button to proceed to the next environment. This last button starts greyed-out and becomes green once the player has properly debugged and run their code to receive the correct output.

When the player runs their code, the value displayed in the <textarea> is compared to the contents of another .txt file containing the correct code; if these values are an exact match, the program will "output" either the appropriate value (which is just a in a separate .html file containing the right answer, as the current version of the website is unable to run Java code locally), or a generic error of when the earliest bug occurs in the code, to aid a bit in debugging. Below the <textarea> is the aforementioned pseudo-terminal where these outputs are displayed. When the player is finally done debugging, they may click the green "next" button to continue to the next environment, or to the final score page if they are on the last environment of the level.

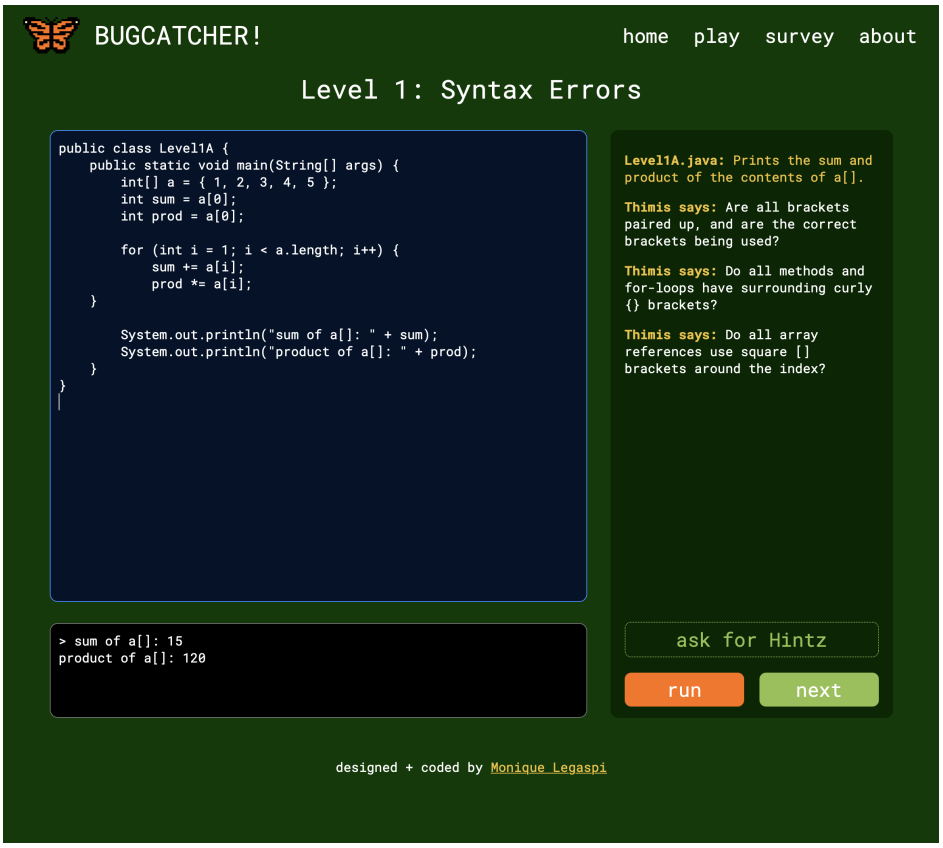

Figure 7: Layout of a completed level environment, with Hintz and a correct output.

4.4. Hintz

Should the player get stuck on an environment, they are able to ask for "Hintz" from Thimis, which may be accessed by clicking the "ask for Hintz" button. There are 3 Hintz per environment, with each hint taking the form of a leading question and becoming more and more specific about the problem; this is meant to mimic the leading questions one might encounter when asking a Lab TA or preceptor for help with a programming assignment. Accessing Hintz does not affect the game in any way, other than providing guidance towards the correct answer for the player.

4.5. Scoring

At the end of a level, the player receives a score that indicates how efficiently they've debugged. The scoring method for each level is relatively simple: as the player progresses through the level, the amount of time they have spent working on each environment, as well as the number of times they have run each environment using the "run" button, are stored in variables t and r, respectively, and

incremented appropriately. (Time spent, in particular, utilizes Jason Zissman's TimeMe.js library, which keeps track of how long a user has spent on a webpage[11].) The formula used to calculate score S is as follows:

$$S = 1000 - \text{Math.min}(900, (t + 10r)) \tag{1}$$

4.6. Graphics

To give the game a homebrew, vintage feel, all sprites in BUGCATCHER! are done in a classic 8-bit pixel style. In keeping with the bug theme, code bugs in the game are represented as actual bugs, and Thimis Hintz's sprite dons an explorer costume (See Figure 8). The site's logo is an orange butterfly with tiger stripes, signifying the project's Princeton affiliation. All pixels were created in Piskel, an open-source, web-based paint program which allows the user to create both static and animated pixel art, then export these works as .jpgs, .pngs, or .gifs for use elsewhere[2].

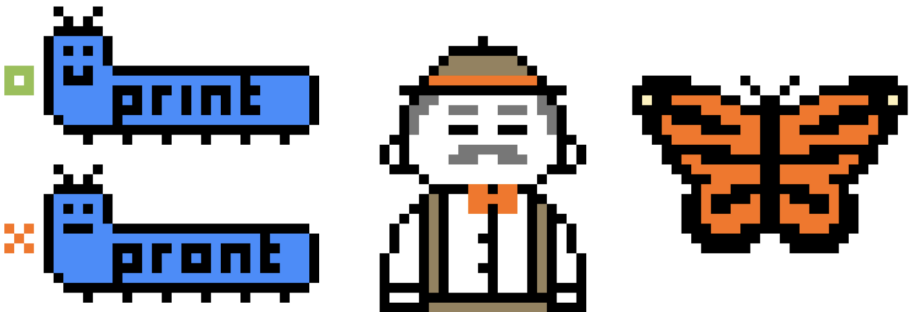

Figure 8: Various sprites created for BUGCATCHER!.

5. Evaluation

5.1. Experiment Design

In order to test the efficacy of BUGCATCHER!, I employed a week-long user test consisting of a pre-survey, regular gameplay, and a post-survey. I collected participants via an email blast to all residential college listservs, asking for current and past COS 126 students who had or retained at least some working knowledge of Java. The primary audience of my project would be students enrolled in COS 126, using the site as they learn Java through the course, so it would be unrealistic

to test on people who either have not undergone COS 126 or no longer remember its contents, as they do not know anything about Java and would have to (re-)learn it just to participate.

The testing process is as follows:

1. Participants complete a pre-survey consisting of 5 pre-set buggy Java programs which they must debug. After debugging, they are prompted to complete a Google Form which asks them for demographic information (class year, major, previous programming experience), as well as questions specifically about the difficulty of the debugging portion of the survey.
2. Participants explore the website and play the levels at their own pace over 5 days.
3. Participants complete a post-survey consisting of 5 pre-set buggy Java programs which are similar to, but not the same as, the code from the pre-survey. After debugging, they are once again prompted to complete a Google Form about debugging difficulty, as well as to provide feedback on their experience using the site (how enjoyable it was over time, efficacy at teaching debugging skills, allure of gamified elements, etc.).

5.2. Survey

The survey is built into the website and was designed to be similar in both function and appearance to the game levels. This was done for two reasons: first, so that mechanics learned from gameplay would easily translate to the survey, and second, so that metrics from the survey (namely, time and runs, which were previously used to calculate level score) could be tracked in the same manner, allowing for easier data collection and analyzation. (See Figure 9; compare to Figure 7.)

5.3. Results

Of the 21 students who completed user testing, the most common class year that participated was the Class of 2025 (freshmen), comprising 42.9% (9/21) of the group, with even participation from each of the other undergraduate class years, as well as one graduate student. The most common major by a wide margin was Computer Science (COS) at 38.1% (8/21), followed by Operations Research and Financial Engineering (ORF) at 14.3% (3/21), then a smattering of other engineering majors tied at 2 or fewer students. The most common semester in which students took COS 126 was Fall 2021

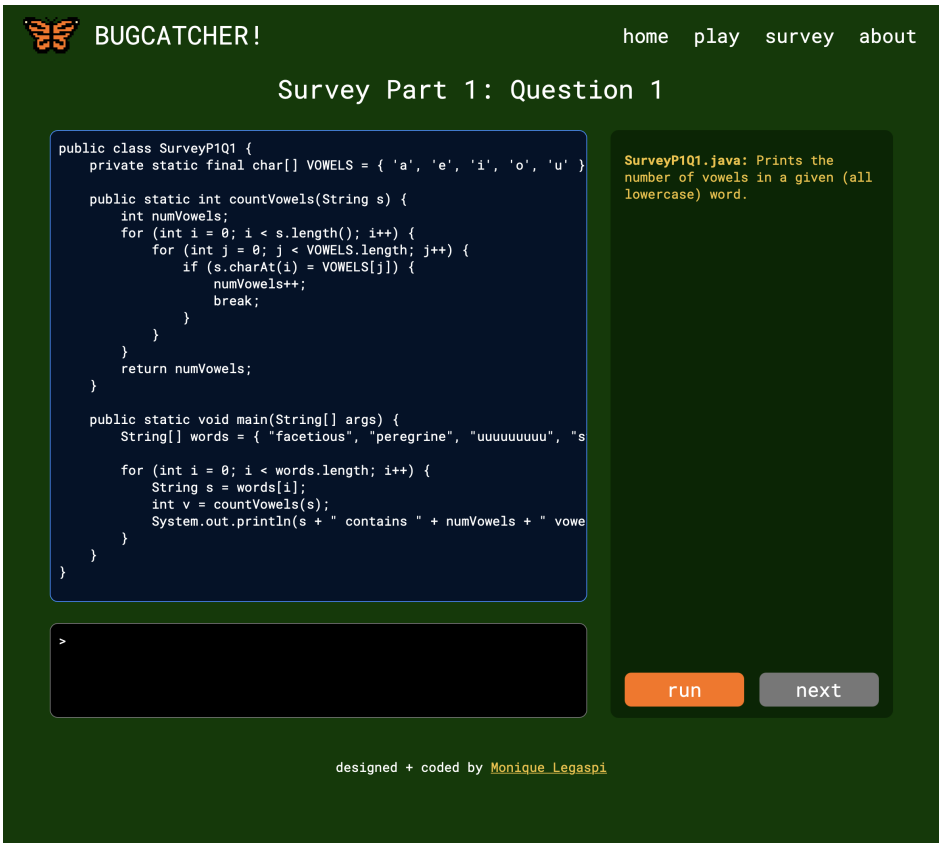

Figure 9: Survey interface.

(the same semester in which this independent work was conducted), again comprising 42.9% of the group, as these were likely all of the freshmen. Only 2 participants indicated having worked for the course as either a Lab TA, Grader, or Facilitator. Programming experience prior to entering the course ranged from none at all, to several years of AP CS and coding competitions throughout middle and high school. When asked in the pre-survey, the average confidence in Java programming abilities ranked by students was 2.91 out of 5, while the average confidence in debugging abilities was 2.77 out of 5. Suffice to say, the participant pool exhibited a wide range of backgrounds and abilities, as one might find typical in the student roster of COS 126.

In the pre-survey, the average time taken to debug all 5 Java programs was 981.673 seconds, with a standard deviation of 665.664 seconds. This wide range of completion times mirrored the varied experience levels of the participants coming into the experiment. After playing BUGCATCHER! for 5 days and completing the post-survey, however, the average debugging time decreased to 640.015 seconds, with a similarly decreased standard deviation of 290.763 seconds. Figure 10

below represents the data visually, with error bars indicating 1 standard deviation away from the mean in either direction.

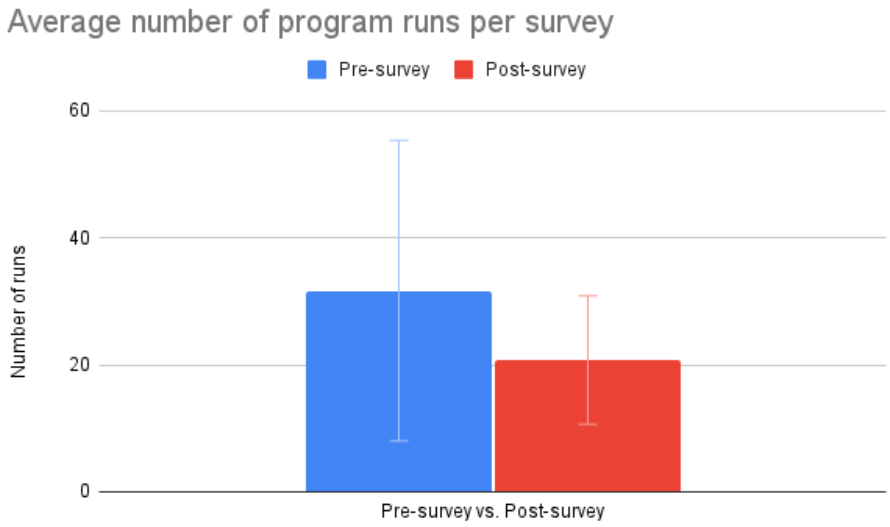

Figure 10: Bar graph of average time taken to complete each survey, in seconds.

In a similar manner, the pre-survey indicated that the average number of times each participant ran the programs within the survey was 31.682 times, with a standard deviation of 23.654. After playing BUGCATCHER! for 5 days and completing the post-survey, the average number of runs decreased to 20.762, with a standard deviation of 10.113. Figure 11 below represents the data visually, with error bars again indicating 1 standard deviation away from the mean in either direction.

In addition, participants ranked the difficulty of the debugging part of the pre-survey as, on average, a 2.5 out of 5, and ranked the difficulty of the debugging part of the post-survey as a 2.43 out of 5.

Participants spent, on average, 13.225 minutes per day playing the game over the span of 5 days. The majority cited their experience with the game as helping them to complete the debugging post-survey (ranking its helpfulness as 3/5 or higher). The majority also felt BUGCATCHER! would help the average COS 126 student learn debugging, and also enjoyed the experience overall. 19 out of 21 participants indicated their willingness to play the game either stayed the same or increased throughout the week. Surprisingly, the scoring function did not play into most participants'

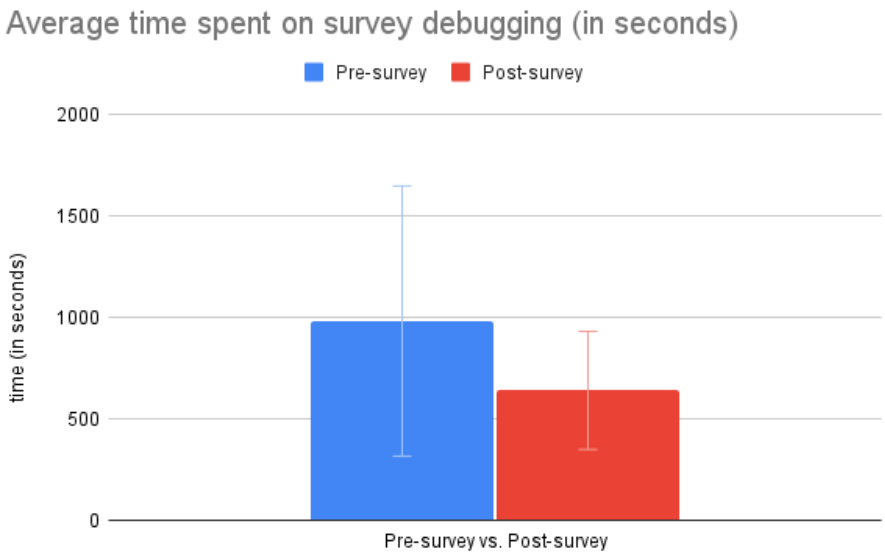

Figure 11: Bar graph of average number of times participants ran every program in each survey.

willingness to replay levels, with 13 participants ranking its effectiveness as 1/5. Participants stated that the graphics, website aesthetic, and Thimis tutorials added to their enjoyment of the game, while the scoring and simplified IDE (<textarea> with basic error messages) detracted from their enjoyment. When asked to give custom feedback, many expressed generally positive sentiments, but also cited the overly sensitive code-checking system and fact that there are no line numbers as being their foremost frustration with the game.

6. Summary

Based on quantitative evidence collected during user testing, it can be reasonably inferred that playing BUGCATCHER! improved participants' debugging skill across the board. Comparing their survey times and runs from after the game to before the game revealed both decreased averages and standard deviations, indicating that, alongside growing debugging skills, the gap between less- and more-experienced students shrank significantly. Perceived difficulty of debugging also decreased, showing that, perhaps, the participants' experience playing BUGCATCHER! inspired confidence in their own debugging abilities.

Similarly, qualitative feedback indicated generally positive feelings from participants about the game. They found BUGCATCHER! mostly pleasant and easy to play, citing some gamified

elements as enhancing their experience. However, the scoring method, lack of line numbers, and sensitive code-checker were among participants' main grievances.

In implementing a scoring function based on time and number of runs, I had hoped that, rather than spamming the "run" button in search of bugs, as a typical COS 126 student might submit their programming assignments to the Autograder over and over to check their work, a lower score might encourage the user to step through the code themselves, discovering bugs on their own, and then get faster at it as they replay the levels. However, as users pointed out, since they did not know how the scores were calculated, they were not encouraged to do either of these things – and a lack of knowledge about the minimum/maximum possible scores or how other people were scoring (for example, via a leaderboard) meant there was no reference against which they could compare their own score, and thus did not encourage replaying. Additionally, since the levels retain the same bugs every time (as they are hard-coded into the level), there is no real value in replaying in the first place, since a player can simply memories where the bugs are and replay until they receive their desired score. In the future, I hope to improve upon this gap in user knowledge by implementing a leaderboard – although this would require either a true backend or a more solid knowledge of game frameworks such as Phaser.js[1], which I had initially considered using for this project, but eventually scrapped due to its steep learning curve.

In terms of line numbers, I had originally intended to display line numbers alongside the <textarea>, but found the task to be unexpectedly difficult; there is no way to add line numbers inside the <textarea> such that the numbers cannot be accidentally deleted by users, as they will inevitably become part of the value that the <textarea> exports upon submission, and displaying them in a outside of the <textarea> becomes similarly tricky with hard-coding the numbers to be the same exact spacing as the lines within the <textarea>. In an attempt circumvent the problems that this omission would inevitably create, I kept the Java programs relatively short, so that users could count the lines themselves – but ultimately forcing them to count, especially when error messages only give the line number of the error, becomes frustrating quite quickly. In the future, I would like to find a better way of displaying line numbers; perhaps this would require using an

embedded IDE instead, although I have yet to find one that supports Java.

The method of code-checking, as stated previously, is unusually sensitive; because it essentially relies on a === equivalence check between two Strings, any change in spacing or newlines is regarded as an error, even though Java would not normally cite these as errors. This resulted in many users taking longer than necessary to debug their code. I initially tried stripping spaces and newlines from the Strings to allow for variant spacing to pass through the check, but quickly discovered that this would take away from the ability for the error message to state the lines in which the bugs occur. Again, I believe this would be solved if I could find an embedded IDE that had the ability to run Java locally, but this will require more future research.

The application's use of localStorage can also be problematic at times; if the user were to clear their browser history, or even simply open the website on Safari when they'd been playing on Chrome the whole time, they will lose their progress. If the user is tech-savvy enough, they can easily edit their localStorage variables to reflect dishonest scores, or to access levels they have not yet earned the right to play. Future work in this area once again entails the implementation of a proper backend, as well as an account system, so that all data may be secure and selectively editable by the user.

Finally, in the future, I would like to change the level-creation method so that there is less hard-coding; as it stands, every level has hard-coded bugs, and changing them requires poring carefully over multiple .txt files. If levels are easier to create and edit, it allows for more customization, such that BUGCATCHER! (much like Ladebug) may evolve alongside COS 126 – and, to add excitement, the bugs could someday be randomized!

When compared to my original goal criteria, BUGCATCHER! manages to fulfill each quality, somewhat – it is easy to play (save for counting line numbers), engaging (beyond a disregard for scoring), age-appropriate, and Java-focused – but it is far from perfect. With time and work, though, I believe it has the potential to become a mainstay in the COS 126 classroom, providing future computer science students with the support and confidence they need to debug their own assignments.

Application	Intuitive?	Engaging?	Age-appropriate?	Java-focused?
Gidget	Yes	Yes	No	No
G4D	Maybe	Yes	Yes	No
Ladebug	Yes	Maybe	Yes	No
BUGCATCHER!	Maybe	Maybe	Yes	Yes

Table 2: BUGCATCHER! and related work, judged according to goal criteria.

7. Honor Code

I pledge my honour that this paper represents my own work in accordance with University regulations. /s/ Monique Legaspi

References

[1] "Phaser - a fast, fun and free open source html5 game framework." [Online]. Available: https://phaser.io

[2] "Piskel." [Online]. Available: https://www.piskelapp.com

[3] "Top java software errors: 50 common java errors and how to avoid them," 2021. [Online]. Available: https://stackify.com/top-java-software-errors/

[4] A. Ettles, A. Luxton-Reilly, and P. Denny, "Common logic errors made by novice programmers," 01 2018, pp. 83–89.

[5] M. J. Lee, "How can a social debugging game effectively teach computer programming concepts?" in *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ser. ICER '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 181–182.

[6] M. J. Lee, "Teaching and engaging with debugging puzzles," Ph.D. dissertation, University of Washington, Seattle, WA, 2015.

[7] A. Luxton-Reilly *et al.*, "Ladebug: an online tool to help novice programmers improve their debugging skills," 07 2018, pp. 159–164.

[8] R. Sedgwick and K. Wayne, "Introduction to programming in java • computer science." Available: https://introcs.cs.princeton.edu/java/home/

[9] R. Smiderle *et al.*, "The impact of gamification on students' learning, engagement and behavior based on their personality traits," *Annals of Mathematics*, vol. 7, no. 3, 2020.

[10] A. S. M. Venigalla and S. Chimalakonda, "G4d - a treasure hunt game for novice programmers to learn debugging," *Smart Learning Environments*, vol. 7, no. 21, 2020.

[11] J. Zissman, "Timeme.js." Available: https://github.com/jasonzissman/TimeMe.js