Turning Up the Heat!: Using Fault-Localizing Heat Maps to Help Students Improve Their Code

Kenneth R. Edmison, Jr.

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Stephen H. Edwards, Chair
Manuel A. Pérez-Quiñones
Na Meng
Francisco Servant
Rui Abreu

November 22, 2019
Blacksburg, Virginia

Keywords: Computer Science Education, Spectrum-based Fault Localization, Adaptive Feedback, Heat Map, Visualization, Debugging

Turning Up the Heat!: Using Fault-Localizing Heat Maps to Help Students Improve Their Code

Kenneth R. Edmison, Jr.

(ABSTRACT)

Automated grading systems provide feedback to computer science students in a variety of ways, but often focus on incorrect program behaviors. These tools will provide indications of test case failures or runtime errors, but without debugging skills, students often become frustrated when they don't know where to start. Drawing on the experiences of the software engineering community, we apply a technique called statistical fault location (SFL) to student program assignments. Using the GZoltar toolset, we applied this technique to a set of previously-submitted student assignments gathered from students in our introductory CS course. A manual inspection of the student code demonstrated that the SFL technique identifies the defective method in the first three most suspicious methods in the student's code 90% of the time. We then developed an automated process to apply the GZoltar SFL analysis to student submissions in the Web-CAT automated grading system. Additionally, we developed a heat map visualization to show the results of the SFL evaluation in context to the student's source code. After using this tool suite in the Fall 2017 semester, we surveyed the students about their perceptions of the utility of the visualization for helping them understand how to find and correct the defects in their code, versus not having access to the heat map. We also evaluated the performance of two semesters of students and discovered that having the heat maps led to more frequent incremental improvements in their code, as well as reaching their highest correctness score on instructor-provided tests more quickly than students that did not have access to the heat maps. Finally, we suggest several directions for future enhancements to the feedback interface.

Turning Up the Heat!: Using Fault-Localizing Heat Maps to Help Students Improve Their Code

Kenneth R. Edmison, Jr.

(GENERAL AUDIENCE ABSTRACT)

Automated grading systems provide feedback to computer science students in a variety of ways, but often focus on incorrect program behaviors. These tools will provide indications of test case failures or runtime errors, but without debugging skills, students often become frustrated when they don't know where to start. Drawing on the experiences of the software engineering community, we apply a technique called statistical fault location (SFL) to student program assignments. Using the GZoltar toolset, we applied this technique to a set of previously-submitted student assignments gathered from students in our introductory CS course. A manual inspection of the student code demonstrated that the SFL technique identifies the defective method in the first three most suspicious methods in the student's code 90% of the time. We then developed an automated process to apply the GZoltar SFL analysis to student submissions in the Web-CAT automated grading system. Additionally, we developed a heat map visualization to show the results of the SFL evaluation in context to the student's source code. After using this tool suite in the Fall 2017 semester, we surveyed the students about their perceptions of the utility of the visualization for helping them understand how to find and correct the defects in their code, versus not having access to the heat map. We also evaluated the performance of two semesters of students and discovered that having the heat maps led to more frequent incremental improvements in their code, as well as reaching their highest correctness score on instructor-provided tests more quickly than students that did not have access to the heat maps. Finally, we suggest several directions for future enhancements to the feedback interface.

Dedication

For Sondra and Rob

Acknowledgments

When I first started working on my Doctorate, I had it all planned out. I even had a project plan laid out of the courses I would take and when, and when I'd graduate, and I had it down to four years...which makes me only eight years late.

There are always a ton of people to acknowledge and thank when working on a project like this. This isn't the kind of thing you finish alone. First, my thanks to Steve Edwards and Manuel Pérez for sticking with me when there were times it looked like there was no end in sight, and for becoming friends in the process. Also thanks to Dr. Meng and Dr. Servant, who both made me consider things that were completely off my radar. This work is so much better for it. Special thanks to Dr. Abreu and his team for creating GZoltar, without which this work would not have been possible. Also, a tip of the hat to José Campos, who faithfully answered my questions, both smart and not-so-smart, about how GZoltar worked. He was a lifesaver.

It would not have been possible for me to get to this point without the support of my supervisors and co-workers, first Mark Howard and company at the Software Technologies Laboratory, in the Department of Industrial and Systems Engineering, and now Brian Broniak and team at Learning Systems Administration in TLOS in the Division of Information Technology. I'm sure Mark and Brian both got tired of signing the tuition waiver forms (which is one of the great, underused benefits of working at Virginia Tech) that came across their desks most every semester for the last 12 years. But signed them they did. I have been blessed by the support of my co-workers, and I am forever grateful.

I was fortunate as in undergraduate to fall into the orbit of Dr. Edward Weisband. Besides making me a better writer with his constant admonishments of being “clear, cogent and compelling”, his other great insight was to “get the question right.” I have tried to keep those two ideas in mind while writing. I think this work is better for them.

Last, but certainly never least, thank you to my wife Sondra and my son Rob. They have put up with me working on this for a long time. In fact, my son has never really known a time when I wasn’t in school in some fashion. They have encouraged me and kept me going through frustration and guilt. I would not have finished this without their constant encouragement. I love you both very much.

This work is supported in part by the National Science Foundation under grant DUE-1625425. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Motivation

Troubleshooting is a skill that is taught to people learning technical trades. When a defect occurs, the natural tendency is to start looking for the issue near where the symptom manifests. However, the actual root cause is often far away from the place where the defect is seen. I work as a theatre stagehand from time to time, and when I was first learning to troubleshoot problems, I was taught to start at one end of the signal path, never in the middle. When troubleshooting an issue with an audio signal, for example a crackling speaker, I would start at the input, whether it be a microphone or CD player, and verify the input was clean. Then, I'd check the output of the the mixing desk, then the effects rack, and so on, down to the amplifier rack and finally to the speakers. Somewhere in that chain, the signal went from clean to crackling. If it changed between pieces of equipment, it was often in a failed cable, but it would have been very difficult, or impossible, to identify the cable without starting the troubleshooting at one end or the other.

Debugging is troubleshooting in software. It requires a person to be able to identify defects, and then trace the execution of the program to identify the root cause of the defect. Unlike troubleshooting in the physical world, debugging software can become very complicated much more quickly, especially when advanced concepts such as recursion and threading are introduced. Additionally, whereas most technicians are trained to figure out how their equip-

ment can be corrected when it doesn't work, very little time is devoted to teaching novice programmers how to debug [76]. Indeed, McCauley et al. [76] identified only one introductory textbook that had a dedicated chapter to problem-solving and debugging recalcitrant programs.

Computer science as a discipline is under intense pressure to accommodate increased student enrollment with resource growth that is negligible or constrained at a slower growth rate than demand. Also, computer science programs are facing increasing scrutiny by accreditation bodies to insure the rigor of their programs [6] [9]. The importance of software testing is well understood [87][79], and many CS programs have adopted Test-Driven Development [19] as a paradigm for teaching students about development and the importance of testing their work as they progress. However, it is difficult to see a path that also includes additional time to teach students how to debug. Our work attempts to provide a scaffold for student debugging efforts, balancing the desire to provide guidance but not too much guidance, with a technique that does not, in an era of increasing enrollments, add to the workload for instructors.

There are several techniques that can be used to automate the process of identifying defects within a program. These techniques provide varying degrees of success. Typically these tools have been used in software engineering research to identify ways to help experienced developers become more productive. Applying these techniques to novice student developers allows us to provide a more focused, automated feedback process to encourage students to develop their debugging skills. Additionally, we can provide this feedback within the context of their source code, allowing them to focus their debugging efforts without telling them exactly where the issue is located. We can *guide* them toward the defect, without telling them where the defect is allowing them to practice their debugging skills.

1.2 Research Goals

We address this main question: **How can computer science education use automated fault localization tools to guide and assist students' debugging efforts?** To answer this question, we will take a multi-step approach: to validate that an automated defect identification process will reliably find defects in student code, that we can develop a tool that can be used to apply this technique at scale, and then provide that feedback to students in a contextualized manner that guides the student's debugging without telling them exactly where their errors are located, which will allow them to improve their code faster:

1. **Identify an automated defect localization technique and validate its ability to reliably identify defects in student code submissions.**
2. **Design and develop a tool to apply this defect localization technique to student submissions at scale, within the context of an automated grader.**
3. **Create a feedback method that provides contextualized feedback to students based on the results of the defect localization process**
4. **Assess the feedback provided based on students' perceptions of its utility in helping them understand the defects in their code and in correcting those defects.**
5. **Assess the feedback provided based on students' performance on programming assignments where the feedback is provided, versus when the feedback is not provided.**

1.3 Organization

The remainder of this work will be divided thusly:

In Chapter 2, we discuss the foundation of test-driven development, and the reasons for using it as the basis for computer science curricula in higher education. Additionally, we review the state of the art with regards to debugging and automated defect detection and localization. Also, we will discuss the importance of providing feedback to students and how providing feedback in context to the student's work is critical to the learning process.

Chapter 3 describes a study of applying spectrum-based fault localization to a set of student program assignments. We discuss the process of identifying a process for applying SFL techniques to localizing defects in student code. Also, we discuss the process of validating the accuracy of the SFL analysis. We discovered that in most cases (90%) it was quite accurate, but that left a non-trivial set of student defects that were not identified. As such, we conclude this chapter with a discussion of possible approaches to address these "edge" cases.

Chapter 4 focuses on the development of a visualization process that provides the results of the SFL feedback within the context of the student's source code. This visualization process employs concepts similar to heat maps used in numeric data visualizations, but in the context of student source code.

Chapter 5 focuses on the deployment of the complete SFL analysis and visualization process during two semesters of our CS 1114 course. We assess the students' perceptions of the systems utility to their learning, as well as investigate the impacts that alternative approaches would have to the feedback students would receive.

Chapter 6 explores the impact that providing the heat map feedback had on the performance of the students in two semesters of the CS 1114 class, Fall 2017 and Spring 2019, versus the performance of students in Fall 2015, when the heat map feedback was not available. We see that the students in both semesters who had access to the heat maps made more consistent incremental improvements to their code, as well as achieving their highest correctness scores faster than the Fall 2015 students who did not have access to the heat maps.

Chapter 7 provides a summary evaluation of the heat map feedback, and it's overall impact on student performance.

The final chapter of this work provides a summary of our findings, the contributions those findings make to the state of art, and a discussion of additional directions for future investigation.

Chapter 2

Literature Review

2.1 Test-Driven Development

Test-driven development (TDD) was first introduced by Kent Beck in his description of a development practice called "extreme programming"[18], and later in expanded upon in [19] and by Astels in [15]. TDD provides a framework for developers to focus on testing at two levels. First, it encourages writing test cases to test atomic portions of their code. This is "unit testing". Additionally, TDD encourages the practice of "test first" development. Developers are encouraged to develop their test cases before writing any of their functional code. Implicit in this is the encouragement to use a testing framework such as JUnit [1]. By writing tests first, and also using a framework to automate the testing process, the developer can easily test the functional code as it is being written rather than waiting until the end of the development process. This provides the developer immediate feedback at a point much earlier in the development cycle, when it is easier and less expensive to correct defects. TDD is a significant shift in testing philosophy from the the traditional development models that had programmers complete their development tasks, based on a set of requirements, and then throw their code "over the wall" to "testers" who wrote manual test cases to exercise the code and would then return the code back to the developers when a defect was encountered.

The TDD process is straightforward and easy to understand. As Beck describes it [19]:

1. **Write a test:** Create the test necessary to exercise the code. This is the unit test.

2. **Make it run:** Write the code.

3. **Make it right:** Run the tests. If the test fails, adjust the code until it passes. If it passes, clean up the code. Regardless, it is vital to retest after each change of the code.

Also, as enumerated by Edwards [38], TDD translates easily to the teaching environment. Students have less difficulty applying TDD to their programming assignments than some of the more traditional testing strategies. It also encourages, by design, incremental development, and the idea of having a running version of the program. By focusing on developing tests as features are being constructed, defects are identified and corrected earlier in the development process, and the automation of the test cases means regression defects are more likely to be detected quickly. Writing tests first also gives students a sense that they are making progress. Finally, TDD is widely supported in the the development tools that are available. Test automation suites exist for a variety of languages. JUnit is widely used, because of the popularity of Java as a teaching language and a number of popular CS educational tools, including BlueJ [17] provide support for a JUnit-like testing format, allowing testing to be introduced at the CS1 or even CS0 level.

While TDD encourages students to code to the test cases they build, it also encourages them to write tests for features that can be constructed incrementally. However, not all test cases are created equally. Some researchers have looked into the quality of test cases that students write, as well as techniques for evaluating test quality [5][83]. This work has shown that while a test case may cause a code defect to be executed, the test itself does not always contain enough information about what the expected outcome should be to adequately detect the defect. Also, a student's submission may not include a complete implementation of the required program functionality, but the test cases may miss these omissions. Additionally,

research has been done into how good the test cases written by students are at uncovering defects. Edwards and Shams [40] discovered that students often practicing "happy-path testing," testing typical cases without testing edge or unusual cases, where bugs were more likely to be revealed. Test case quality is often a point of assessment in core CS courses, and students are graded on both the quality of their tests, as well as the amount of code coverage their own tests complete. But, if a test fails, how do students learn to identify and correct the error?

2.2 Debugging

2.2.1 Debugging as a Skill

Debugging is a skill that is, in large part, learned by doing. Previous work [30][46][91] describes debugging as a task targeted at exploring a program's behavior, within the context of models of what is believed to be the program's correct running state. In order to debug effectively, a programmer needs to complete a number of inter-related tasks[30]:

- Hypothesize which actions caused the program's failure
- Observe data about the program's state
- Change data into different representations
- Compare the new data representations with what was expected
- Diagnose the cause of the failure
- Correct the defect to prevent the failure from recurring.

For a novice programmer, these tasks taken individually can be challenging. These same tasks can quickly become overwhelming when taken as an integrated whole.

Some defects are easier to locate than others and there are a variety of existing tools to assist programmers. Integrated development environments (IDE), such as Eclipse [41] and IntelliJ [4], provide many static code analysis tools that locate certain kinds of errors, such as syntax errors, or unintended side effects, like using the assignment operator in an `if/else` instead of the equality operator. IDEs also provide debugging support for interactively exploring the behavior of a buggy program. They will also identify some potential execution issues, such as unreachable statements. Many runtime errors, such as off-by-one errors in control loops are easily found, while some defects can only be found with deep knowledge of the program and underlying libraries.

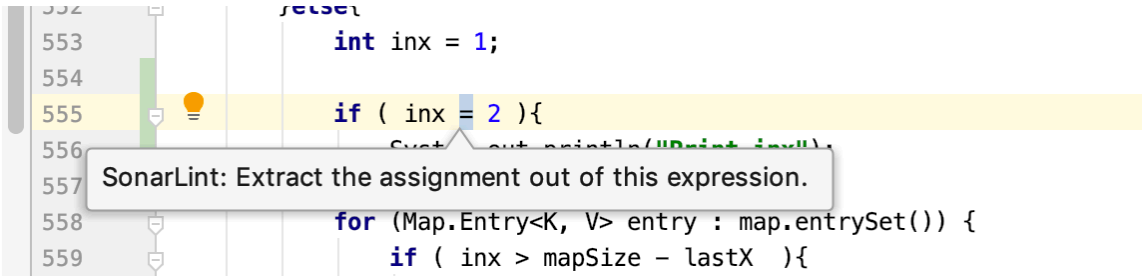


Figure 2.1: An example from the IntelliJ IDE where the static analysis tools identify the erroneous usage of the assignment (=) operator in place of the equality (==) operator.

However, even with the plethora of tools and approaches that are available, debugging is not typically included as a specific learning outcome in most CS curricula[9]. Debugging as a skill is not specifically taught. Students pick up debugging as a secondary skill necessary to complete course assignments. When a bug occurs, or a test case fails, the actual error is often not at the location where the incorrect output or behavior is observed. Instead, the bug is often in an "upstream" location, in a portion of the code executed before the point where the failure is observed, perhaps even in a different method. For example, in an assignment where the student must implement an iterator for some form of list, an error in the iteration

may cause a failure to be observed in one of the other list methods, not because the error is in the other method, but because the other method uses the buggy iterator.

2.2.2 Automated Debugging

Several different approaches to systematic debugging have been proposed since the late 1970s and early 1980s. Most of this work has focused on helping professional developers through automation because of the costs associated with defect detection and correction [20]. Whereas in the early days of computing, the hardware resources were the expensive part of the development equation, now the labor cost for programmers is the largest component. Techniques to reduce the amount of time developers have to spend on debugging have received more attention, especially those that automate debugging tasks.

Weiser [94][95] described a technique called program slicing whereby a slice was defined as the set of all statements within a program that altered the value of a specified variable. To determine the cause of a defect for a specific variable, one need only review the lines of code in the slice where that variable was touched. In practice, this technique is unwieldy in most cases because, while the relevant statements can be captured in a slice, the number of lines in the slice is too large to be usable to debug the program. Others have attempted to apply different methods based on Weiser's program slicing technique [64][32][50][100][101], but the resulting slices remain too large to be practically used. Additionally, while these techniques narrow the scope of the program that the developer has to review, they still rely on detailed understanding of the underlying context of the program, a context that most novice programmers lack.

Another approach that has been proposed is the idea of *interrogative debugging* [63]. The idea behind this technique is to allow a developer to ask specific questions about the state of

the runtime of the program, based on hypotheses about the failures. Ko and Myers developed a tool called *Whyline* to support these types of interrogations. The *Whyline* tool enabled the developers to focus their attention by testing various hypotheses about the defect. Based on the answer of the query, the programmer could continue with that line of inquiry, or discard it in favor of a different hypothesis. Their initial experiments showed that this type of debugging could reduce time to correct defects, but the experiments again focused on experienced programmers, and the feedback was outside the context of the program itself.

Zeller asked a question in a paper title students often pose: "Yesterday my program worked. Today, it does not. Why?" [99]. This is a scenario that is very frustrating to novice programmers. Regression defects occur when changes in the codebase break program behaviors that had previously tested correctly. Zeller describes a technique called *delta debugging*, which identifies the source of the defects based on the differences (the deltas) in the code. The process can be automated to identify changes across a large set of code deltas. But, as Zeller points out, this technique is very resource intensive, and is not as quick to find defects as conventional debugging.

These approaches, and variations on them, all rely on automation to some extent to facilitate bug detection and correction. Parnin and Orso [80] ask the question: Does automation really help programmers? Their research suggests that automated debugging techniques can provide novice programmers helpful support. They observed that automated debugging tools are more likely to help identify and correct the root cause of a defect, rather than just correcting the symptom. Additionally, they observed that programmers benefit from overviews of debugging results that include information about the failed cases:

> For developers exploring unfamiliar code, an automated debugging tool can suggest many promising starting places. Even if the tool did not pinpoint the exact

> location of the fault, displaying relevant code entrance points could help program understanding considerably. This benefit strongly resonated with the developers in our study. One potential way to build on this strength would be to explore alternative interfaces for clustering and summarizing results. For an initial overview, for instance, developers may want to see suspiciousness aggregated by methods or files [80]

However, Parnin and Orso also observed that only looking at the rankings alone might not be enough. For novice developers, we do not want to provide specific direction to the defects in their code. We want to provide a scaffold for their debugging efforts, with a level of guidance that provides some focus within the context of their code, without telling them "look at line 148 in file Node.java. That line is the one causing this test case to fail." In fact, as we will see, that level of specificity is not typically possible, using the currently available spectrum-based fault location approaches.

2.2.3 Spectrum–based Fault Location and Defect Visualization

Spectrum-based fault localization (SFL) is one approach for automating the localizing of faults within a software system[98]. Program spectra [51] are profiles of which parts of a program are executed in a particular run. SFL works by comparing the spectra of software test runs that pass against the spectra for tests that fail, in order to identify the portions of the program that are uniquely involved when failures are observed. Research has shown [60][7][8] that SFL provides a more accurate approach to defect localization than other techniques, including set union and set intersection, nearest neighbor and cause transitions. A key feature of SFL is that no prior knowledge of the system being tested is required before analysis can take place.

Several systems have been developed that implement this technique, each focusing on a different type of software system. Pinpoint [26] was developed to identify the sources of problems in Internet service environments. It maps the fault log from user requests to the trace log of the execution of the components to service those requests. It then passes both of those logs to an analysis process to identify faults in components. The result is a grouping that shows which components were associated with specific failures.

The Tarantula system, introduced in [61], utilizes information that is provided by most testing suites, such as whether a test case passes or fails, which parts of the program were executed as part of the test, and the underlying source code for the tested program. The underlying idea is that entities (which can include program statements, methods, and branches) which are executed by failed test cases are more likely to be defective, that those elements predominantly executed in passing test cases. For each statement in the resulting evaluation, a suspiciousness score can be assigned based on the following equation:

$$Suspiciousness_{\text{Tarantula}} = \frac{\frac{failed}{totalfailed}}{\frac{passed}{totalpassed} + \frac{failed}{totalfailed}} \tag{2.1}$$

Thus, the more failed tests the line is involved in results in a higher ratio of failures to passes for that line, and thus a higher suspiciousness score. They showed that this approach is more accurate in more situations than the approached described in [100],[99],[71], or [26].

An additional feature of the Tarantula system is that provides a visualization tool for the fault location analysis it generates [61]. This visualization displays the entirety of the program's source code, with the suspiciousness score for each line overlayed in color on the code. Lines that only executed in passing tests and are thus not suspicious at all are colored green. Lines that executed in only failed tests are colored red. Lines that executed in a mix of passing failing tests are colored yellow. The limitation of this system for applying to student

assignments, especially where an auto-grading system is used, is that the statistical analysis and the visualization tool are combined into a single application.

GZOLTAR [25] is a Java toolkit that implements a version of SFL. GZOLTAR instruments a program's Java bytecode, and then executes a set of JUnit [1] test classes associated with the program. GZOLTAR records the success or failure of each test case, as well as tracking which parts of the bytecode were executed in each test run. GZOLTAR then use an SFL algorithm to generate a score for each line of source code that was executed in any failed test. As with other SFL tools like Tarantula, GZOLTAR generates a suspiciousness score, reflecting the likelihood that a given line of source code was involved in producing the observed failure. However, it uses the Ochiai method [8] for calculating this values. This method has its roots in computational molecular biology, specifically in calculating genetic similarity. The equation used in the Ochiai method to calculate the suspiciousness score for any statement is [8]:

$$Suspiciousness_{\text{Ochiai}}(line) = \frac{\#failedTests_{\text{line}}}{\sqrt{(totalFailedTests) * (\#failedTests_{\text{line}} + \#passedTests_{\text{line}})}} \tag{2.2}$$

In the Ochiai calculation, shown in Equation 2.2, the suspiciousness score for a line of code is calculated as the number of failed tests in which the line of code is code is involved, divided by the square root of the total number of tests that failed (regardless of the lines involvement), multiplied by the sum of the number of tests that passed and failed that the line was involved in.

GZOLTAR was chosen for our work for two reasons. First, based on an evaluation of the Ochiai method versus other SFL implementations [8], the Ochiai method provides more accurate predictions of defects because the Ochiai method is more sensitive to the potential locations

identified in failed test runs rather than those of passed runs. Secondly, while there is an Eclipse IDE plugin that encapsulates the GZoltar analysis and provides a visualization tool within Eclipse, the GZoltar tool is also available as a Java library, which is the language of choice for our core computer science courses. This allowed us to create a plug-in to automatically perform an SFL analysis on student code, as they are being submitted for evaluation by the Web-CAT[39] auto-grading system. GZoltar decouples the analysis from the visualization, where as other SFL tools have the analysis and the visualization tightly integrated. As we will show, this is important for adding the SFL analysis to the Web-CAT tool chain used to evaluate student submissions.

Other systems have implemented variations on SFL approaches, but these approaches are not appropriate for the problems we are trying to address with novice developers. Wang et al. [92] described a process for using information retrieval techniques on bug reports to provide programmers with direction on where to focus their debugging efforts, but student projects rarely result in bug reports, and most students don't have the knowledge necessary to create the type of bug report that would provide meaningful help in Wang et al.'s approach. Liblit et al. [71] created a process that can help distinguish the effect of multiple defects, and can also identify program attributes that act as predictors of defects within the program. However, as they point out, their technique is best applied at a scale much larger than the small (>1 KLOCS) programs students are typically assigned.

In addition to providing an analysis of the location of defects within the software program, SFL techniques lend themselves to visualization techniques that allow the results of the SFL analysis to be merged with the source code of the program being analyzed. There are a number of ways authors have attempted to do this. The GZoltar provides a visualization tool to visualize the results of the SFL analysis it performs. The combination has been deployed as an Eclipse IDE plug-in [47]. The GZoltar plug-in provides flags next to each

code line for one of four different levels: “low” - green, “medium” - yellow, “high” - orange, and “very high” - red. Additionally, it provides three different diagrams that show the entire body of code in a hierarchical representation of the program structure. Each of the different diagrams allows the programmer to zoom into the code down to the code line level. These diagrams, while tied to the hierarchy of the program, are not directly tied to the context of the program code. The flags do provide a certain level of context, but they are still separate from the code.

Wong et al. [97] developed a system for defect localization based on code coverage analysis, called χDebug. After conducting the defect localization analysis, the suspicious code is visualized as a code listing with each line being assigned one of seven color codings, based on the results of the analysis. Each color is associated with a range of suspiciousness, ranging from light blue for the least suspicious to red for the most suspicious. Unlike other visualization systems we have encountered, χDebug seems to be unique in that the values associated with each range is variable depending on the results of the analysis, where as most of the other approaches use a color scheme that is fixed and the scores are assigned to the color scheme based on some known conversion calculation. The implication here is that regardless of the suspiciousness of the code line, the most suspicious lines will always be red, whereas in other tools those lines might be indicated as something other than red.

As described earlier, the Tarantula system [61] shows the source code as either red, yellow or green lines, depending on the results of each lines passing or failing the tests applied to it. In this case, the tool differentiates between the suspiciousness of each line of code by changing the hue from red to green, based on the ratio of failed to passed tests, with red indicating more failures that passes, and green indicated more passes than fails, with yellow indicating a mixture of both passing and failing tests impacting the line. Additionally, the Tarantula tool also adjusts the brightness component for each line, setting the value to the maximum

percentage of passing or failing tests. The practical result is that more failures a line has, the more red and the brighter the red it will be. The visualization model developed for the Tarantula system maps the most closely to the model we have adopted for our work here.

2.3 Contextual Feedback and Educational Theory

Contextualized feedback is the notion that feedback provided to students is not simply the text that a grader or an instructor writes to a student about some aspect of an assignment the student submits. It is also situated within the assignment itself, communicating information about the student's performance based upon its position with the document. In traditional paper-based systems, the "red pen" was used to make notations on student submissions. With the increased use of auto-grading systems such as Web-CAT and ASSYST [56], as well as course management systems such as Canvas [54] and Blackboard [3], there is now a disconnect between the submission and the feedback. The feedback is now often reported back to the student outside the context of the assignment. This removes some of the effectiveness of the feedback.

The importance of providing feedback to students has been well established in educational research since the early 1960s (see [12][45][78]. Feedback is defined generally as any device used to tell a learner whether their response was correct or not [65]. However, as Kulhavy points out, the mechanism for this feedback has been open to debate. Many behaviorists argued the position of feedback-as-reinforcement. They argued that providing the correct response would reinforce the correct knowledge to the student, thereby improving performance [84][24]. Alternatively, Kulhavy cites a significant body of work that suggests, for material that is very unfamiliar to the student, that there is an advantage to waiting for some period of time to elapse before providing the feedback to the student. This delay ac-

tually increases the retention of the provided guidance by allowing the student to "forget" their incorrect response. Thus, waiting to tell the student why their answer is or is not correct greatly increases the likelihood that the information is correctly remembered in a later assessment[66]. This is directly counter to the typical reinforcement framework, which posits that the impact of the reinforcement decreases as it is removed from the initial event. It is also counter to most of the auto-grading mechanisms in computer science education, which typically provide feedback in timeframes of less than five minutes. Later research [85] suggests that this phenomena, known as delay-retention effect, manifests in assessments like multiple-choice tests, rather than iterative assessments such as programming assignments.

While Kulhavy provided a definition for feedback, Hattie and Timperley [52] examined what feedback means in terms of classroom interactions between teacher and student. They argue, "[F]eedback is conceptualized as information provided by an agent (e.g., teacher, peer, book, parent, self, experience) regarding aspects of one's performance or understanding." This new definition indicates feedback should be focused towards a student's performance of a task, or understanding of some concept. Hattie and Timperley [52] go on to say that for feedback to have the most impact, it must occur within in an active learning context. Their model for effective feedback identifies three specific questions that must be answered to insure that the feedback is effective, but, they note that the most important aspect of this framework is ensuring that the feedback presented is situated and in context to the learner's current level and in concert with the learner's goals. Thus, for programming assignments, positioning the feedback within the student's submission, is the optimum location.

Well-crafted feedback situates the learning within the knowledge creation process. The student knows precisely why they missed a question, and that feedback situates the student in the learning context so that they can better retain the new information later. If the feedback is designed to provide guidance rather than specific answers, that guidance should

be structured around the activity the student is attempting to complete, and should provide direction to facilitate knowledge creation. Additionally, as Wang and Wu have shown [93], feedback that provides more elaboration increases the student's motivation, leading to increased student learning effectiveness. Without this situated feedback, the students are less motivated, and the learning less effective. Feedback becomes an activity in an active learning experience. The student receives feedback on an assignment, within the context of the assignment, and makes the connections between the feedback and the concepts. Also, the feedback can serve to provide insight to the instructor, allowing a view into the performance of the students beyond just the single data point encapsulated by the grade value. This is the motivation for one of our goals of making instructional feedback easier to provide for electronic submissions.

For the instructor, feedback is also important. Because learning is a process of knowledge construction, and because active learners are more successful at this than passive ones [52][65], feedback provides another way for instructors to engage students. With the automation provided by learning systems, it is possible to collect data at a very low level, allowing the instructor to map submissions, to feedback, to follow-on submissions and so on, providing a way to evaluate the feedback being provided. In the past, the instructor would have been able to get this type of information only by hand-collating the comments they were making. Thus, the instructional design of the class can be modified based on data about student performance.

Merriam-Webster's dictionary defines "learning" as "modification of a behavioral tendency by experience (as exposure to conditioning)". An alternate definition is "knowledge or skill acquired by instruction or study". Both of these definitions imply active communication between the instructor and the learner. Current theories of learning make five assumptions, which will be familiar to those who study HCI[44][67]:

- Learning is a process of knowledge construction, and it requires learners to be active designers rather than passive consumers.
- Learning is situated. People act until a breakdown occurs, at which point a coach can provide feedback to the learner identify the situation that caused the breakdown.
- Learning is knowledge-dependent. A person's existing knowledge is used as the basis for creating new knowledge.
- Learning is influenced by distributed cognition, whereby people learn that the skills and knowledge required to address a problem does not reside in one person, but is spread across may people.
- Learning is affected by motivation as much as by cognition. Learners must understand why it is important for them to learn and contribute.

Löwgren and Stolterman [73] argued that "designers", whether they be interaction designers or learning systems designers, must identify the "situation" of the design, as it indicates to the designer many of the constraints that will need to be addressed. For them, learning is a complex undertaking that must account for the situations and motivations of the learner and the environment in which the learning is to take place [88]. Learning is influenced both by the internal motivations of the learner as well as the external situation wherein the learning occurs, as research has shown [93]. Feedback is situated within the context of the content to which it references. Without that context, the utility of the feedback is diminished[89]. As we have discussed above, many of the SFL analysis tools include a visualization component as well, because the authors realized that having a list of suspicious line numbers and their suspiciousness scores was not enough. Showing the suspiciousness of the lines involved with a failed test along with all of the other lines involved within the context of the overall program is a more powerful tool for locating the actual defect.

However, there is a challenge when providing feedback, especially when providing feedback in the middle of an assignment: knowing how much feedback is enough to scaffold discovery and learning, without providing so much guidance the instructor is "giving away" the answer. Another range of tools that have faced this problem, besides auto-graders, is that of intelligent tutoring systems (ITS)[11]. An ITS provides interactive feedback to students based on the actions the student has taken, with the interaction based on a model of student learning behavior. The challenge for developing intelligent tutors for programming is the "open-ended" aspect of programming, where many possible paths lead to the same result [82]. When a student is following his or her own path and constructs a buggy solution, it can be difficult to provide intelligent coaching on how to address the situation.

In the context of ITS, approaches that have been explored include comparison with reference programs [43] or with a set of constraints imposed on the program [69]. In either case, some additional work is required from the instructor to develop the hints to be shown to the student. Recent work [68] has focused on analyzing the edits made by students between submissions, and then using those edits to attempt to find a correct solution for the program. Those edits could then be used as a source for hints to be supplied to the student.

Striking the balance between enough feedback and too much feedback, and providing that feedback within the context of the student's code is central to fostering learning while still providing the student the information they need to learn where to look for the defects in their code.

Chapter 3

SFL and Locating Defects in Student Code

3.1 Background

The original concept seemed to be straightforward: using SFL techniques, apply instructor-supplied reference tests to each student submission, and for buggy submissions, identify the source line(s) most likely to contain the bug. However, moving from this fairly simple idea to a feasible implementation was challenging.

The work described in this section was presented at the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) in the article "Applying spectrum-based fault localization to generate debugging suggestions for student programmers"[35].

3.1.1 Implementing an Automated SFL–based Evaluation Tool

The implementation of an automated SFL-based evaluation tool for student submissions had several goals and constraints that needed to be met in order to be feasible within our context. First, most basically, it needed to be able to analyze Java-based programs. Also, it had to work within the context of our existing automated grading system. Finally, the analysis needed to be completed quickly, so that the overall time from the point where the student

submits their assignment to the point where they get their feedback is not appreciably impacted.

We decided on the the GZoltar library [25]. It turned out not to be an "out of the box" solution. Several hurdles had to be cleared in order to use the toolset. GZoltar comes in two forms, an Eclipse [41] plug-in and a stand-alone Java library. The plug-in provides both a textual display (called the RZoltar view) as well as visualizations of the output that allows the developer to drill down into the methods that fail a test to see the scores associated with individual source lines.

Initially, we looked at using the Eclipse plug-in to evaluate student assignments. We provide a pre-configured Eclipse build, as well as our own Eclipse plug-in for accessing Web-CAT. We considered including the GZoltar Eclipse plug-in as part of this distribution. This proved to be problematic for several reasons. First, the plug-in assumes that only one project will be evaluated at a time. It uses the currently open Eclipse project as the project to be evaluated, and would not run consistently if multiple Eclipse projects were open. In order to process multiple projects, we would have to open each project individually in Eclipse, run the GZoltar analysis, and then collect the data that GZoltar generated. This would be impractical for evaluating many student projects. Also, it would make the long-term goal of adding this functionality to an automated testing system more complicated.

As a result, we turned instead to using the Java library version of GZoltar. This library implements the core test execution and statistical analysis engine used by the Eclipse plug-in. To use this library, we created an Apache Ant [2] build script that takes a student project reference as an input, builds the GZoltar program as well as the student project, and then execute the GZoltar program, which then runs the JUnit tests on the assignment.

Unfortunately, the GZoltar library does not provide a simple, clean output stream for use

by other tools. Instead, the library produces two separate types of data that are not directly integrated. First, it captures the results of all of the test cases it executes on the student project. This includes the signature of each test case, as well as whether it passed or failed. If the test failed, GZOLTAR also records the JUnit output trace from the failed test. The second type of data GZOLTAR collects is the program spectrum from each failed test, in the form of a list of source lines that were executed as part of that failed test. Source lines are identified by line number, signature of the enclosing method, and name of the defining class.

As noted earlier, GZOLTAR generates "suspicion" scores for each line that is involved in a failed test. These scores, with values between 0 and 1, are associated with each line that participates in the execution of a failed test. However, the suspicion scores are not returned in the same method call as the listing of the code covered in the failed test cases. Therefore, it was necessary to construct a driver program that collected these separate results and then collated them in a single output format that was more convenient for analysis. Our program created two comma-separated values files: one containing the JUnit output for all failed tests for a submission, and one containing the lines executed in the failed files, along with the corresponding suspiciousness scores.

While this approach was sufficient to show the feasibility of using GZOLTAR, and to permit validation of GZOLTAR's accuracy on student programs, additional work was required to provide a capability that could be employed automatically at scale. Specifically, the initial GZOLTAR driver program included hard-coded information about an assignment's dependencies, and the names of the reference test classes being used for evaluation. In later versions, these values were appropriately parameterized so that references to student assignments and project reference tests could be input dynamically without altering the driver program. Subsequent versions were deployed as an Apache Ant[2] plug-in to the Web-CAT build process for student submissions.

3.1.2 From Lines to Methods

Once we began using GZoltar to analyze student programs, we quickly discovered the practical difficulties of using SFL to localize bugs down to individual source lines. In most cases, GZoltar produced the same suspicion score for many lines in a program. Figure 3.1 show an example where this occurs, in the buggy constructor for a class used in a Minesweeper game. One task of this constructor is to randomly place a series of mines in the two-dimensional array representing the board. As is apparent from the very high suspicion scores, GZoltar did identify the location of the bug as some line in this constructor. However, all of the lines have the same score, meaning that the bug could not be localized down to an individual source line.

```
1   public MineSweeperBoard( int row, int col, int mine )   GZoltar Score
2   {
3       rows = row;                                          0.922958207
4       columns = col;                                       0.922958207
5       grid = new int[row][col];                            0.922958207
6       mines = mine;                                        0.922958207
7       TestableRandom random = new TestableRandom();        0.922958207
8
9       int i = 0;                                           0.922958207
10      while ( i <= mine )                                  0.922958207
11      {
12          int y = random.nextInt( row );                   0.922958207
13          int x = random.nextInt( col );                   0.922958207
14          if ( grid[y][x] != -2 )                          0.922958207
15          {
16              grid[y][x] = -2;                             0.922958207
17              i++;                                         0.922958207
18          }
19      }                                                    0.922958207
20  }                                                        0.922958207
```

Figure 3.1: A method where all of the lines receive the same suspiciousness score from GZoltar. The defect is an "off by one" error in line 10.

In Figure 3.1, the bug is on line 10, where the loop condition should use < instead of <=. The wrong comparison operator causes the loop to repeat one more time than necessary. There are two consequences to this: first, this bug is evident on every invocation of the constructor,

assuming the number of mines to plant is never negative. Second, every single line in the entire constructor is always executed each time the bug is observed. This means that all spectra from failed tests for this bug include all of the lines in the entire constructor. As a result, it makes sense that GZOLTAR's suspicion scores are identical for all of its lines.

```
1   public String toString()                                    
2   {                                                           
3       String st = "";                                 0.26222648
4       st += "<";                                      0.26222648
5       if (size > 0)                                   0.26222648
6       {                                                       
7           Node<Item> temp = new Node<Item>(null);             
8           temp = front.getNext();                             
9           while (temp != rear)                                
10          {                                                   
11              if (temp == rear.getPrevious())                 
12              {                                               
13                  st += temp.getElement() + ">";               
14                  return st;                                  
15              }                                               
16              st += temp.getElement() + ", ";                 
17              temp = temp.getNext();                          
18          }                                                   
19      }                                           0.26222648 //Error is here.
20                                                  //Need the else branch so that
21                                                  //if size<=0, attach ">"
22                                                              
23      return st;                                  0.26222648
24  }
```

Figure 3.2: A method where the error location is not directly scored by GZOLTAR. In this case, the error is a missing `else` condition at line 19 and the associated code

Figure 3.2 shows another method containing a different bug, and where many lines again have the same score. This method generates the string representation of a linked queue's contents by walking its chain of linked nodes. Here, the bug is a missing `else` condition for the `if-else`, misplacing code to provide the closing `>` at the end of the string being generated. When the queue is empty, the wrong string is returned. The lines inside the `if` statement have no scores because they are not suspicious—they only occur on spectra for passing tests. However, all failing tests associated with this bug involve executing all of the scored lines in the method, which results in those lines all receiving the same suspicion score. In this case, the scores are lower, because all of those lines also participate in all passing

tests.

While GZoltar did not always produce uniform scores throughout a method, it quickly became apparent that all of the lines that were consistently executed together were going to participate in the same spectra, and thus receive the same suspicion score. Because of this, it is unreasonable to expect SFL to pinpoint defects down to the line level. However, GZoltar was much more accurate at identifying the method that contained a bug. As a result, we switched our attention from using SFL to identify lines containing bugs to using SFL to identify methods containing bugs. This may potentially lead to better guidance for students, if this approach can point students to one or more methods in their own programs that are likely to contain bugs, and do so with reasonable accuracy. The next step was to validate the accuracy of this approach across a number of student programs.

3.2 Manual Evaluation of GZoltar Results

3.2.1 Methods

To validate the approach of using SFL on student assignments, we applied our GZoltar-based analysis to existing programming assignments that were submitted in a previous academic term. The assignments were implemented in Java, and were submitted as part of the CS2 class offered by our department in the Spring 2011 semester. Our analysis included two assignments. One assignment required students to implement the classic "Minesweeper" game, complete with the ability to mark mine locations, calculate the number of adjacent mines, flag cells, and reveal cells. While the basic data structure was simple—a two-dimensional array —this assignment stressed the logic used in implementing the game's features. The second assignment included in our analysis required students to implement two separate versions

of a queue, one using a primitive array with circular indices, and one using a doubly-linked chain of nodes. Each version of the queue was required to provide iteration support, as well as implementing `hashCode()`, `toString()`, and `equals()`. Unlike the minesweeper assignment, the queue assignment focused more on manipulation of data structure state, rather than control logic.

A total of 110 students in the course produced 110 programs for the minesweeper assignment and 102 for the queue assignment. Of these 212 programs, many passed all reference tests. Only 135 contained bugs revealed by the instructor-provided reference tests, so our validation study focused on these. We applied GZOLTAR using our custom driver program to all 135 programs containing bugs, generating lists of suspiciousness scores for all suspect lines in each student's program. We then used the maximum suspicion score assigned to any line in a given method to create a list of method-level suspicion scores. For each student program, methods receiving suspicion scores were ranked in decreasing order by score.

After GZOLTAR processed the assignments, a manual inspection of the programs that had failed test cases was conducted. In the inspection, each program was reviewed to verify that the methods that GZOLTAR had identified with errors actually contained errors. Additionally, the programs were reviewed to determine if there were methods with errors that did not receive a score by GZOLTAR. From this information, it was also possible to determine which methods most frequently contained true bugs in each assignment.

3.2.2 Common Bug Locations

Figure 3.3 shows all methods in the minesweeper assignment that were found through manual inspection to contain bugs, arranged by frequency. By far the most common bug location was the method called `numAdjacentMines()`, which calculated the number of mines present

surrounding a given board location. This method had the greatest logical complexity in most student submissions. Figure 3.4 shows the methods where bugs were found in the queue assignment, where `dequeue()` (for the array-based implementation) was the most error-prone method. Following that, the `remove()`, `next()`, and `hasNext()` methods provided by queue iterators were also common bug locations.

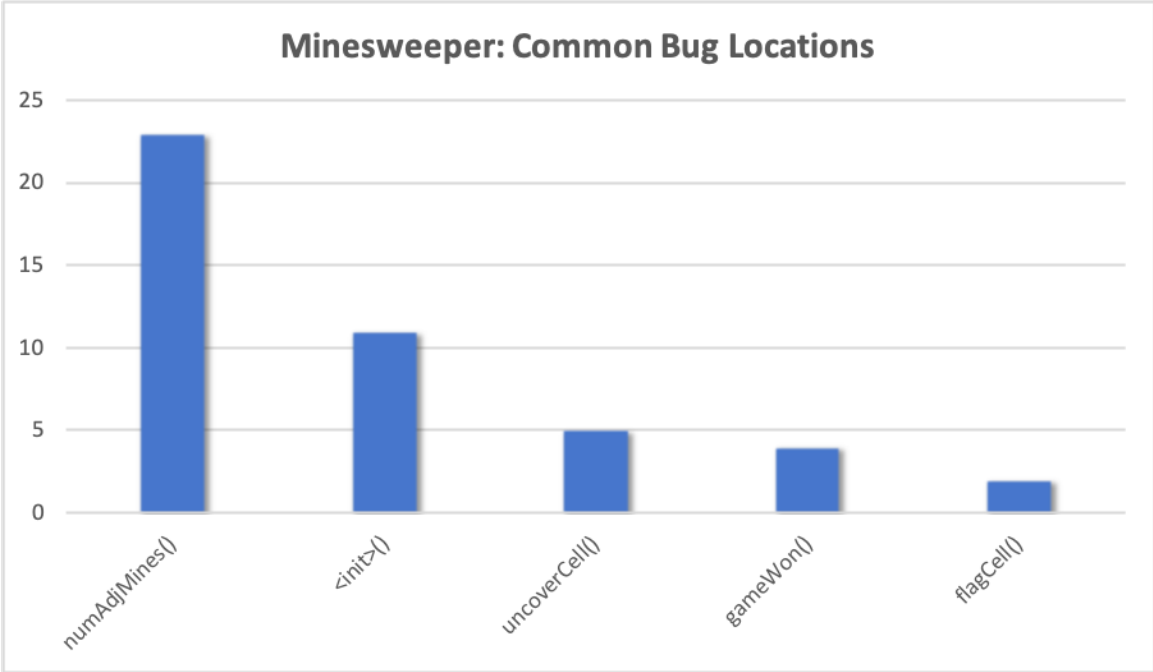


Figure 3.3: Methods in the minesweeper assignment most likely to have defects.

3.2.3 GZoltar Rankings

For each program containing a bug–that is, failing an instructor-provided reference test–the GZoltar results were collected and turned into method-level suspicion scores. Methods scored by GZoltar were then ranked by their suspicion score, and then the top-ranked method containing a bug was identified based on the manual debugging results.

Figure 3.5 summarizes GZoltar's performance. 73.3% of the time, GZoltar's top-ranked

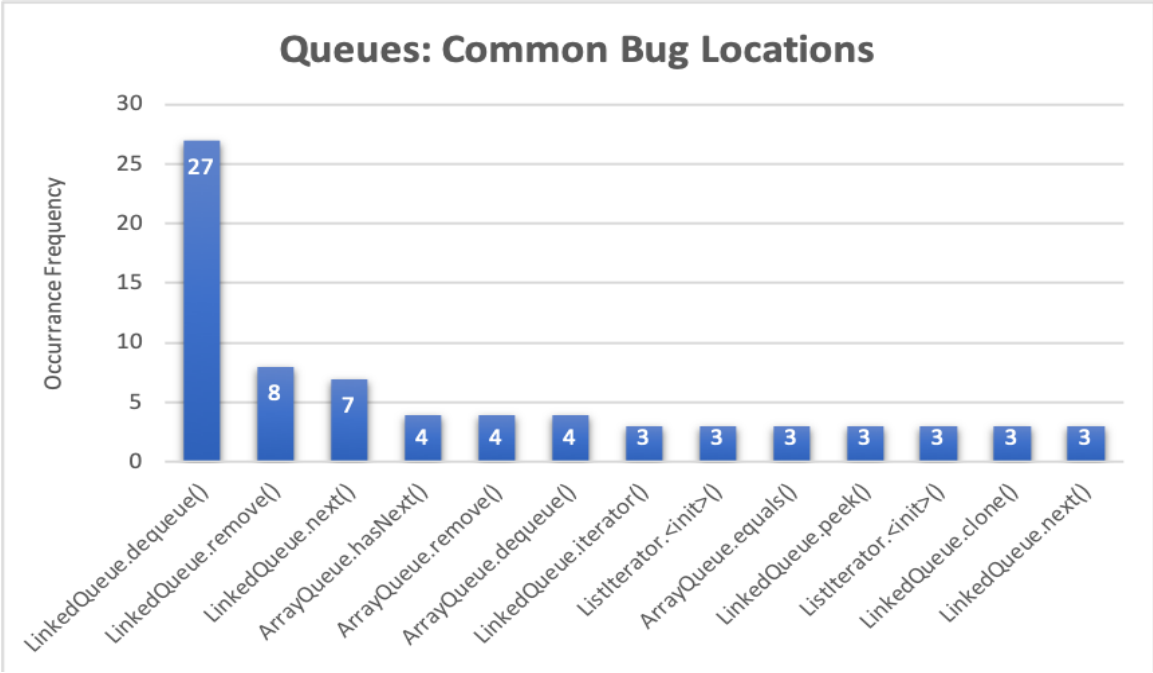


Figure 3.4: Methods in the queue assignment most likely to have defects.

method contained a bug. When considering the two highest-ranked methods, GZOLTAR pointed to a bug 84.4% of the time, while the three highest-ranked methods included a bug 90% of the time. Only 7% of the time did GZOLTAR fail to identify a bug within the top four method scores.

Figure 3.6 presents a scatter plot of the actual suspicion scores given to these bugs of different ranks. From this plot, it appears that GZOLTAR is most accurate when it is most confident –when suspicion scores are above 0.8.

One question that arose during this investigation is: what kinds of methods would not contain a bug, but end up with a higher suspicion score? Since suspicion scores are calculated directly from the spectra of failing tests, code has to be executed in predominantly failing test cases to receive high suspicion scores. Scores are lower when code is included in spectra for both passing and failing tests.

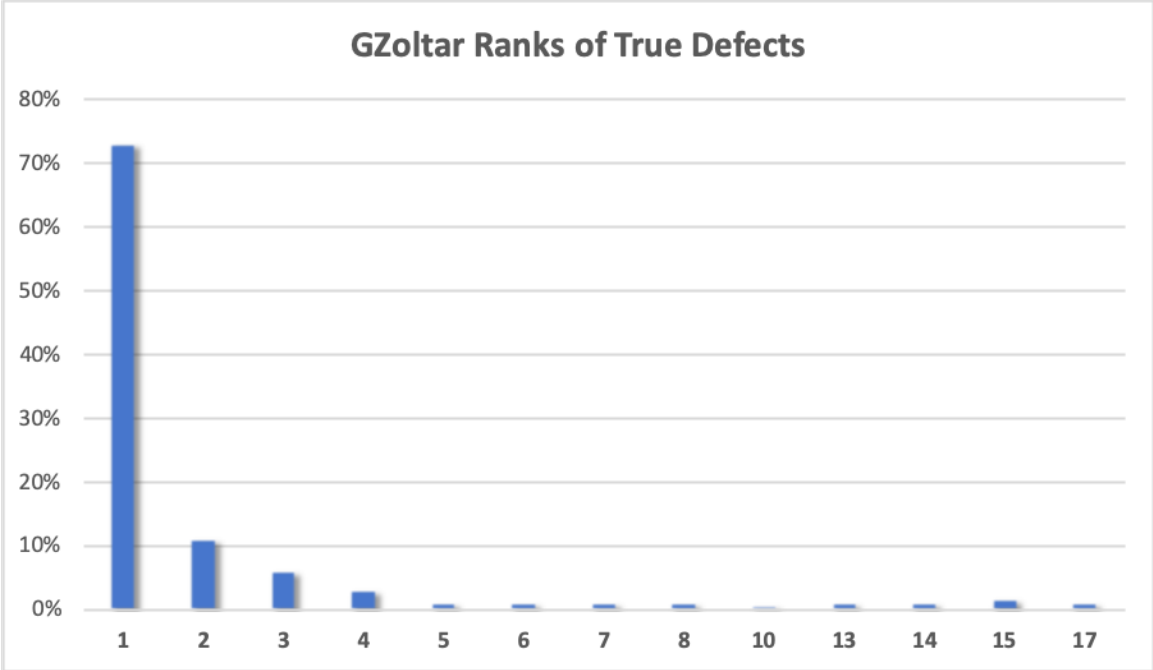


Figure 3.5: Where true defects appeared in GZoltar's suspicion ranking.

To address this question, we hand-examined a selection of cases where methods that did not contain bugs appeared higher in GZoltar's ranking than the defect(s) revealed by manual debugging. In the majority of these cases, we found that the more highly ranked method called the method containing the bug, and served as the primary route by which reference tests invoked the buggy method. For example, consider the queue project, where students wrote their own implementation of a linked queue. Students also created their own internal node class to represent segments of the queue's linked chain. Sometimes, a student may have a bug in how their internal node was set up in its constructor, or a bug in a helper method they placed on this internal class. However, such a bug might only be revealed when a reference test invoked the queue's `enqueue()` or `dequeue()` method to add to or remove from the data structure.

In that situation, if every call to a top-level method used by the reference tests resulted in a test case failure, then that top-level method would receive a very high suspicion score. This

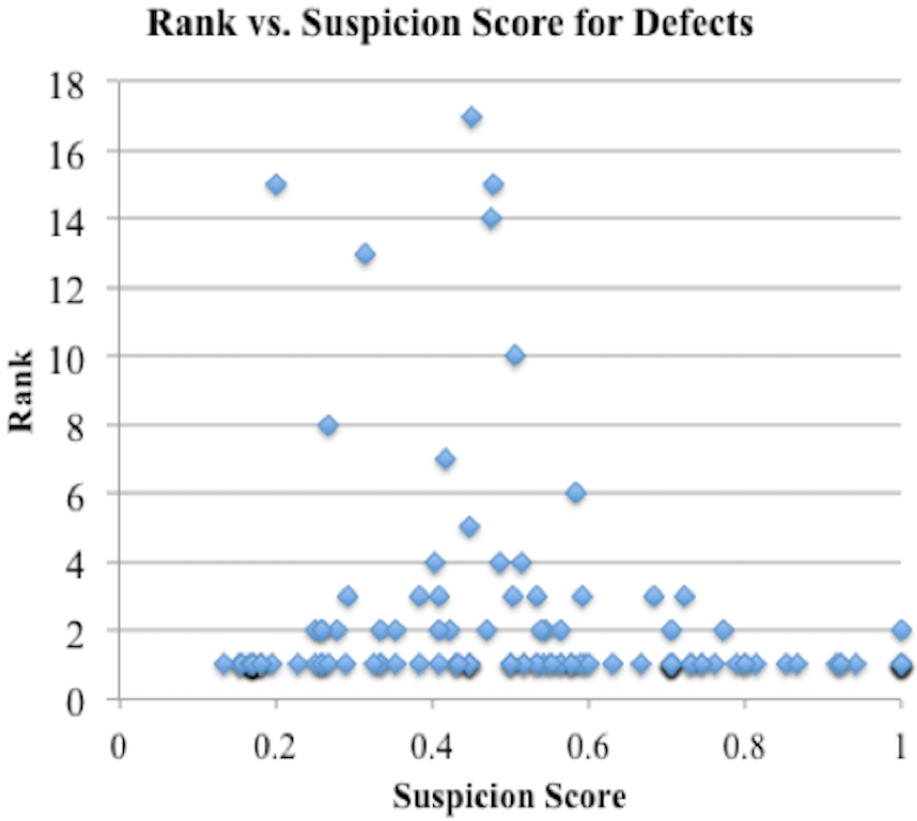


Figure 3.6: Variation in suspicion scores given to bugs by GZOLTAR, compared to their ranks.

makes sense, since the defect was revealed by calling that method, even though that method itself does not contain the error. However, this does make directing the student's focus to the defective method more challenging.

3.3 Providing GZOLTAR Results as Feedback To Students

To turn GZOLTAR's method rankings into student suggestions, the most direct route is to devise a strategy for selecting one or more methods from GZOLTAR's suspicion-ranked list, and present them to the student as candidate locations to inspect. For example, we might envision a system where a student receives a debugging suggestion like this:

> Based on its behavior, your solution still contains bugs. By analyzing where it fails, there may be a bug in one of these methods:
>
> - `numAdjacentMines()`
> - `uncoverCell()`
>
> It may help to spend more time examining how you think these methods could possibly fail, and then testing them more thoroughly yourself.
>
> Remember, if there's not a bug in one of these methods, it may be in something called by them.

Ideally, we want this list to be accurate–that is, to point the student in the right direction. This means that it is important for the list to contain a buggy method, and also important that it not contain non-buggy methods, if possible.

One way to assess this balancing act is to calculate the precision and recall [28] of the method selection strategy. For example, if we simply show the student the top-ranked method, that will be a bug-containing method approximately 73% of the time, for this pool of programs. That provides a precision of 73%. While the true number of latent bugs is unknown, preventing us from calculating true recall, we can use a more conservative notion of recall by considering whether the first known bug is presented. If only the top-ranked method is chosen, it will include this bug 73% of the time, leading to a 73% recall rate.

If we increase the number of top-ranked methods we present to the student, precision will fall, because more non-buggy methods will be included. At the same time, recall will go up, because there is a greater likelihood we will include a buggy method in the longer list. Unfortunately, recall goes up relatively slowly, as indicated by the very steep drop off

in Figure 3.5–each additional method we show increases the chance of including a buggy method, but the amount of that increase falls toward zero very rapidly. Precision, on the other hand, drops rapidly as we report more methods, since each additional method is dramatically less likely to represent a bug.

Still, because the non-buggy methods ranked high by GZOLTAR are usually related to the location of the bug by the code's call chain, presenting additional methods that are thematically related to the bug may be less of a distraction. As a result, for the purposes of recommending directions to students, some loss of precision (i.e., including additional non-buggy methods) may be acceptable, if it results in greater recall (i.e., a higher chance to include an actual bug in the list of methods presented). Based on this, showing the student the top three methods would provide 90% recall, while only requiring the student to focus on three methods.

The idea of deciding which methods to include in a debugging suggestion using a fixed threshold, such as the top three, is simple and easy to implement, but may not provide the best results. In particular, by including information about the call chain relationships between methods, and also by looking at the magnitude of differences in the suspicion scores of methods, it may be possible to come up with a more intelligent decision procedure that selects different numbers of methods in different situations. To be effective, such an approach would need to maintain a high recall rate, including buggy methods in the list at a high frequency, while also increasing precision by leaving out non-buggy methods that would otherwise be included by simpler rules. To this end, we look now to a method of visualizing all of the defects identified, within the context of the student's solution.

Chapter 4

Visualization of Detected Defects

4.1 Background

The work described in this section was presented at the Nineteenth Australasian Computing Education Conference (ACE '17) in the article "Using Spectrum-Based Fault Location and Heatmaps to Express Debugging Suggestions to Student Programmers"[37].

4.2 Source Code Heat Maps

An aspect of feedback that students often find frustrating is the lack of assistance in finding bugs. Students often (wrongly) presume an automated assessment system "knows" where the problems are, but just isn't telling. The truth is that, while assessment systems can be quite effective showing that bugs are present, simply knowing a problem exists is not the same as knowing the cause. Some instructors use different techniques to create reference test suites intended to provide assistance in the task of locating bugs, but this work is time consuming and not everyone has the necessary experience. Other instructors resort to giving the test data to students, so students can examine the tests and run them themselves, but this relieves students of the obligation of self-checking their own work by writing their own tests, and still does not address the problem of finding bugs once they are revealed. As we have discussed earlier, spectrum-based fault location provides a mechanism to identify the

likely location of the defects. The question then becomes: How do we present the debugging suggestions to the student in a way that provides guidance without giving them "too much" help? We believe a visual feedback tool based on heat maps addresses this issue.

Research indicates that at least 77% of CS faculty hold opinions aligning with the "Geek Gene" hypothesis [70]–that some students are born with the skills necessary for learning computing, while others are not. Thus, it is unsurprising that classroom assessment approaches in computing focus on purely objective, points-driven scoring, with feedback that is framed in the negative about what features "do not work", what points were lost because of "mistakes", or what software tests "do not pass". The problem is further exacerbated by automated grading tools, which are typically coldly objective in assessing the pure "correctness" of student programs, without any recognition of the effort or hard work that went into constructing them. The feedback students receive tends to reinforce performance-oriented goals, and fails to meaningfully support learning goals, offering no explicit value or reinforcement of growth mindset beliefs. Students often report their initial experiences with automatic grading in computing courses as cold, impersonal, depressing, and discouraging–facing a zero score accompanied only by complaints about what doesn't work in a program is a sobering experience that "turns off" many students. While the evaluation may reflect "the truth" in regards to the end product, it is too easy for students to interpret such an assessment as a value judgment on their own abilities, and on the work and effort they have invested to get to that point in their solution process. In short, it is too easy for students to interpret such feedback as a discouraging slap in the face that says "you don't have it", and that encourages thoughts of giving up.

We want to encourage students not to give up. Learning is influenced both by the internal motivations of the learner as well as the external situation wherein the learning occurs. By adopting a feedback mechanism that guides the student's discovery of their errors rather than

just a cold list of defects, we hope to provide students the guidance they need to identify their errors as well as encourage their own discovery.

There are many advantages to providing this form of visual feedback. First, it provides holistic feedback about the entire assignment, rather than textual feedback that is much more limited in scope. Second, it provides location information within the student's solution, not just a feature-oriented description. In effect, it is direct, location-based defect feedback instead of indirect information about what test case(s) failed. Third, by providing information about the entire assignment, students can choose which areas they wish to investigate. Fourth, by aggregating the statement-level data used to generate the visualization, it is also possible to rank entire methods in order of probability of containing bug(s). Fifth, it is fully automatic and requires no additional work on the part of instructors to write hints or feedback messages for students, or to consider how students might interpret (or misinterpret) the names of test cases.

4.3 Generating Heat Maps

Heat maps [96] are most commonly used in showing the concentration of occurrences at the intersection of two data points. Previous work has [43][97] has used heat maps to visualize the results of software testing over time as it relates to the changes in the source code.

In our case, we created heat maps based on the GZoltar suspiciousness scores. For each student submission, we created an HTML file that contains the source code for the student's submission. We then added styling to the file to visualize all of the lines of code that GZoltar identified as suspicious. We also highlighted the three methods that have the highest score reported by GZoltar (See Figure 4.1). This method of visualization shows the suggestions within the context of the source code for the submission. We used the HSV colorspace [59]

to highlight lines that were suspicious, because it made it simple to convert the GZOLTAR scores into hue angles on the colorwheel. The color space we chose was between 0 (red) and 70 (yellow) degrees, such that for a GZOLTAR suspiciousness score:

$$colorScore = 70 - (GzoltarScore * 70)$$

Thus, the higher the suspiciousness, the closer to red the source code line is displayed. By using heat maps, the student can quickly see the lines, and thus the methods with the highest suspicion. This is similar to the approach applied by Jones et al. [62] in the Tarantula system, except that in that system code lines that are successfully executed are displayed with a green highlighting. Our focus is on guiding students toward their defects, and thus we believe highlighting successes would create too much visual noise. Additionally, non-executable lines are not encoded. Simply highlighting the lines where GZOLTAR indicated some level of suspiciousness is not terribly helpful. In a submission with many defects, it is possible that the entirety of the student's submission could be considered suspicious at some level. In order to provide students with a place to start their review, we also highlight the three methods that contained the highest suspiciousness scores. We discuss the rationale for why we chose three as the number of methods to highlight in Section 3.3. Each of the methods is surrounded by a colored border (see Figure 4.1). We purposefully do not designate which method is the "most" suspicious as the goal is to provide suggestions that encourage the student to investigate their code and find the defect, rather than tell them exactly where the defect occurred.

The result is a set of marked-up source code that shows all of the suspicious code that has been identified by GZOLTAR. All potential issues are shown at a glance to the student. This has the benefit of addressing the 10% of cases where the actual root-cause defect was not

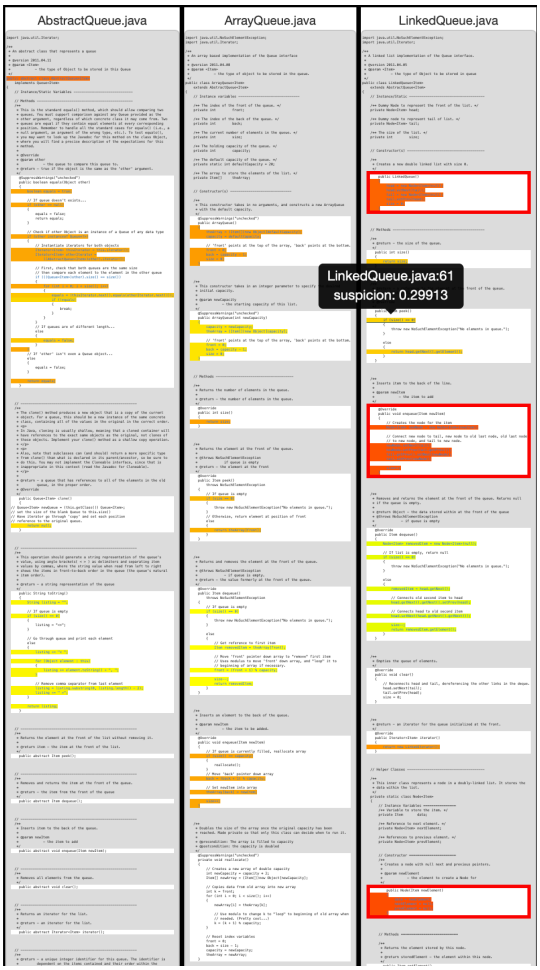


Figure 4.1: Example heat map for a student submission of the Queue assignment, with the top three suspicious methods highlighted, as well as the other lines that GZOLTAR considers suspicious. Additionally, when the student mouses-over a suspicious line, the line number and actual suspiciousness score are displayed.

one of the top three highest scored locations reported by GZOLTAR. For example, in Figure 4.1, the top three scored locations are all in the `LinkedQueue.java` source file. Since we mark all of the lines in the file that were scored with a non-zero suspicion score, the student will be directed to first look at the methods marked, but then can see all of the other lines of code that might be potentially problematic.

We believe this has several advantages. First, it provides context to defects, not only the top methods identified as being defective, but to the structure of the student's code. For

example, in Figure 4.1, the top three methods (identified by the red boxes, are all located in `LinkedQueue.java`. This class inherits from the base class `Queue`. The heat map shows there that `Queue.java` does not contain any suspicious lines. If there were, the student could first devote their attention to correcting those defects first, since it is the base class. Since there are no errors in Queue, the student can focus on the subclass LinkedQueue for detecting and correcting the defects.

Secondly, the top methods identified as defective are shown in the context of all of the defects in the code. Thus, if a student is reviewing one of the most suspicious methods, they can look at the method code and also see quickly if that method calls any other methods that also have suspicious code. This focuses the student's attention. We can provide meaningful direction without telling the student exactly where the error is located.

Third, this method of displaying the results of the fault location allows us to show the student the location of ALL defects, rather than just those with the highest score. This addresses the issue of having "false positive" methods being highlighted in the list as defective when they are not. By showing the student all potentially defective lines, as well as the suspiciousness of each line, the student can see every location that might be problematic. Additionally, this addresses the issue of defects cause by a lack of code. If a defect is caused by something that has not been implemented, such as a missing else condition on an if statement, the method will still be marked as defective.

Finally, this method should enhance the accessibility of feedback to differently-abled learners. The language that is used for feedback can impact students in different ways [72][74]. For those who are not native speakers of the language in which the feedback is provided, a graphical feedback solution provides a language-agnostic feedback mechanism. The explanation of how to use the feedback can be done outside the context of the feedback, possibly in the student's native language. Also, because this feedback uses native HTML to mark up

the student's code, assistive technologies such as screen readers can be used to interpret the feedback for the student. This is of particular interest because of recent research into the difficulty people with visual deficits have with navigating programming tools [16].

4.4 Possible Inaccuracy in Heat Map Visualization

Heat maps show the probable (not certain) locations of bugs. However, this raises the possibility of whether a heat map might mislead a student by being inaccurate, resulting in confusion and frustration. Addressing this possible misdirection is an important issue.

While this is a valid concern, it is important to note that current feedback approaches (such as text-based "hints" about program failures) can also be misleading, and actually have the potential to be much more misleading, since such feedback is written by people with specific goals in mind. Software failures often occur irrespective of the testing goals of the test writers, and our past experience with Web-CAT is that there is a learning curve for instructors to avoid trying to "diagnose bugs" in their tests, since such feedback is much more likely to be misleading.

In comparison, the heat maps are always accurate, in that they show which lines of the software are most heavily used by failing test cases. They are immune from the biases of any individual charged with writing feedback messages for students. While it is true that this does not "pinpoint" the bug, it is not possible for a detected bug to be "outside" the color highlighted areas of the heat map–that is, it is not possible to have a true/false/negative situation, where part of the code looks "clean" in the heat map, but actually harbors a bug detected by the reference tests. Even in the case of bugs of omission, such as the missing `else` condition, the code associated with the `if` clause will still be highlighted as it is associated with the missing branch.

The results of the validation study, discussed in Chapter 3, demonstrate the level of accuracy of the heat maps. In our study, it was found that when considering the top three most deeply highlighted areas of student code, the location of the bug was included in those deeply highlighted areas 90% of the time. Further, it was included in the single most deeply highlighted area 73% of the time.

To ameliorate concerns regarding the 10% of the time the bug is in less deeply "hot" areas, the heat map approach has a further advantage over textual feedback: it shows a holistic view of the entire work, rather than only containing a textual reference to one feature, method, or location. As a result, students see the full spectrum of information available about all locations in the code simultaneously. This provides greater recovery support in those cases where the bug is more difficult to find and providing heat maps alongside traditional textual feedback allows them to be mutually supportive and minimizes their individual weaknesses regarding misleading students.

Now that we developed a method for visualizing the defects identified by the GZOLTAR SFL analysis, it was time to explore the application of this method in an actual CS course. We decided that we would focus on CS 1114 to start. At this level, the students are focused on test-driven development and implementing their solutions in an object-oriented fashion. This results in software with many small methods, where method-oriented debugging suggestions are likely to be effective.

Chapter 5

Experience in Classroom and Student Perceptions

In this chapter, we describe the results of using this context feedback approach to help guide student attention with heat map visualizations over two semesters of CS1 involving over 700 students. We analyze the utility of the heat maps, describe student perceptions of their helpfulness, and describe the unexpected challenges arising from student attempts to understand and apply this style of feedback.

5.1 Background

The work described in this section was accepted for publication at the 2019 Technical Symposium for Computer Science Education (SIGCSE '19) in the article "Experiences Using Heat Maps to Help Students Find Their Bugs: Problems and Solutions" [36].

5.2 Using Heat Maps to Help Students

A significant amount of work, going back over 30 years, has been conducted looking at how novice programmers work and how they debug their code[34][81][86][10]. Additionally, much

research [48][49][90], has been conducted that studied the differences in debugging abilities and techniques employed by novice and expert programmers. Additionally, we know that situating the feedback within the context of the assignment is important to the learning process [93]. Finally, the feedback cycle of edit→submit→receive→revise can be critical for student performance [77]. This feedback cycle can create a problem for students when they encounter a bug in their code and the feedback they receive from the automated grader doesn't provide them the guidance they need to resolve the defect, or is provided out of context.

Students often believe, incorrectly, that an automated assessment system can figure out where the defects in their programs are, but that the instructor has configured the system not to reveal that information. In point of fact, while assessment systems can be quite effective in showing that bugs are present, simply knowing a problem exists is not the same as knowing the cause defect. Some instructors use different techniques to create reference test suites intended to aid in the task of locating bugs, but this work is time-consuming and the complexity of assignments can make this task challenging. Other instructors resort to giving the test data to students so students can examine the tests and run them themselves, but this relieves students of the obligation of self-checking their own work by writing their own tests, and still does not address the problem of finding bugs once they are revealed. Also, there is a balance that has to be struck between prompting the student as to where they might look to solve an issue versus telling them specifically "you're using an assignment instead of an equality operator in line 12".

The heat map visualization we have developed leverages (see Chapter 4) automation to provide contextualized feedback to the students while encouraging them to debug on their own. There are many advantages to providing this form of visual feedback. First, it provides holistic feedback about the entire assignment, rather than textual feedback that is much more

limited in scope. Rather than the instructor having to write feedback for every possible outcome, the heat map will automatically display all possible outcomes. Second, it provides location information within the student's solution, not a feature-oriented description. In effect, it is direct, location-based defect feedback instead of indirect information about what test case(s) failed. Novices don't have the domain knowledge that experts have [49], which makes a feature-based approached less appropriate for novice programmers. Third, by providing information about the entire assignment, students can choose which areas they wish to investigate. Fourth, by aggregating the statement-level data used to generate the visualization, it is also possible to rank entire methods in order of probability of containing bug(s). Fifth, it is fully automatic and requires no additional work on the part of instructors to write hints or feedback messages for students, or to consider how students might interpret (or misinterpret) the names of test cases.

5.3 Experience in the Classroom

To test the efficacy of heat maps as a feedback strategy, we deployed the plug-in for Web-CAT, that we developed (see Chapter 3). When students submitted their assignments, the **GZoltar** library evaluated their code against the instructor's references tests. A heat map was then created and displayed back to the student within the context of the other feedback Web-CAT provided. We used this system during the 2017/18 academic year in the *CS 1114: Introduction to Software Design* course. We evaluated five assignments in the Fall semester, and four in the Spring. In total, 893 students were enrolled, of which 262 consented to participate in this study.

Table 5.1: Heat map creation summary

Program	Students	Attempts	Displayed	Avg
Maze	259	1770	497	28.08%
Segregator	256	2287	983	42.98%
Battleship	254	1427	165	11.56%
AntsVsBees	256	2554	1386	54.27%
WxStation	174	1476	750	50.81%
	Total	**9514**	**3781**	**39.74%**

Heat maps were generated for each student submission, but not necessarily presented each time. We reserved heat map presentation until two conditions were satisfied:

- The student had at least 85% code coverage with their own test cases, **AND**
- The submission had to pass at least one reference test case.

The restrictions on presenting the heat maps were meant to encourage students to continue to write their own tests, and not use the heat maps as a crutch or as an artificial view into the reference tests. The heat map is meant as a tool to provide a scaffold for them to approach debugging, not a substitute to writing their own tests, and imposing these restrictions before presenting the heat maps encouraged that behavior [23].

5.4 Problems Identified in Use

At the end of each semester, we asked the students participating in the study to complete a survey (See Appendix B for details of the survey) about their experience with the heat map tool, and to provide their feedback. Based on observations on the behavior of the tool, and from reviewing the students' feedback, we have identified a number of issues associated with this approach.

One major shortcoming is the lack of visibility into the tests that failed. We are using the results of the instructor's reference tests when creating the heat maps, so the students don't have any access to the tests, which means they can't use the information encapsulated in the test case to localize the defect. Secondly, the nature of SFL means that the score for each line is based on the execution of the set of tests. We don't have a way of saying "The failure to test 1 contributed 23% and the failure of test 2 contributed 77% to the suspiciousness score of line 93".

Additionally, there is a misperception held by students that the auto-grader is "pointing to the bug", when, in fact, it is just evaluating the spectra of defects in the code, which may or may not include the actual location of the defect. Students believe that the heat maps should include the location of the defect, but that is not always true. If the student implemented the code, and the code is defective, the defect will be present. However, if the error is a defect of omission, there is nothing to point to. The spectra will suggest an area to investigate, and the control code associated with the omitted code will be highlighted, but it can't point to the location because the defect manifests as code that doesn't exist.

5.4.1 Block-level Rather than Statement-level Localization

As discussed earlier, the nature of SFL means that suspiciousness scores for statements in a single block of code have the same score, because they are executed together. Unless the block has a single statement, the scores for a block will always be the same. In Figure 5.1, taken from a student submission, we see a constructor method has each line identified as suspicious. In fact, they are all considered equally suspicious.

Additionally, the way that SFL libraries typically behave is to create a *cumulative* suspiciousness score across all of the tests applied during the evaluation. As we showed in Chapter

3, a manual evaluation of GZOLTAR results demonstrated that because of this cumulative behavior, the suspiciousness was closer to the block-level than the statement-level.

From a student perspective, this can be confusing, because the heat map is presenting a block as being suspicious but isn't differentiating within the block where to look. That isn't terribly useful, and in fact could be less useful than looking at the error log, since the heat map will also reflect suspiciousness for locations along the call tree towards the defect. In the case of Figure 5.1, the error may be in `setActorChoices()`, and thus the call to the offending method is marked as suspicious.

```
public Colony()
{
    super();
    QueensChamber queen = QueensChamber();
    add(queen, 0, 3);
    Hive bees = new Hive();
    add(bees, 9, 3);

    setActorChoices(
        HarvesterAnt.class,
        ThrowerAnt.class,
        WallAnt.class,
        FireAnt.class,
        HungryAnt.class);

    food = 10;
}
```

Figure 5.1: A method showing multiple statements with the same suspiciousness.

5.4.2 Faults Due to Missing Code

In some cases, the defects are due to code that is missing, for example, unimplemented features, or a missing `else` conditional. In these cases, the heat map can't highlight the problem (since the root cause isn't present in the code), but it can highlight the code around the problem. Additionally, the heat map may highlight correct code that is executed at the wrong time because of the missing code. This *mis*-highlighting can cause students to mistrust heat maps or find them counter-productive, since they directly conflict with the model of the heat map helping point to the fault location.

Figure 5.2 demonstrates this behavior. In this case, the student has not implemented the `else` clause on the `if..else` statement, and the test case has attempted to test the scenario of `size == 0`. Notice that the SFL analysis has assigned a non-zero suspiciousness to the closing brace on the `if` clause. Since the `else` clause does not exist, this is the only place it makes sense for the suspiciousness score to be assigned for the missing `else` clause.

5.4.3 Multiple Simultaneous Faults

In some cases, the heat map that is generated can have so many lines highlighted that it doesn't provide any meaningful insight to the student. Additionally, it can cause information overload for the student, making it difficult to know where to start. In Figure 5.3, a heat map of a student submission is shown where every non-comment line is highlighted. This isn't useful to the student, and in fact is frustrating, as the survey feedback showed.

Over-highlighting can happen when there aren't many simultaneous faults. It can happen when a key piece of code is defective, or one that is called may times throughout the execution of the application. In either scenario, the spectra generated for the execution will incorporate that code and all of the code that depends on it, resulting in spectra that indicate near

```
public String toString()
{
    String st = "";
    st += "<";
    if (size > 0)
    {
        Node temp = new Node(null);
        temp = front.getNext();
        while (temp != rear)
        {
            if (temp == rear.getPrevious())
            {
                st += temp.getElement() + ">";
                return st;
            }
            st += temp.getElement() + ", ";
            temp = temp.getNext();
        }
    }

    return st;
}
```

Figure 5.2: Heat map of method missing an `Else` condition.

universal code suspiciousness.

5.5 Evaluation of this Approach

We evaluated the heat maps in two ways. First, we surveyed students at the end of the semester (see Appendix B for the survey text) and asked the students to indicate their agreement to a series of questions relating to the usefulness of the heat maps. The student responses (n=43) indicated that most found the heat maps hard to use, confusing, and that they did not improve the student's class performance. This was very discouraging, as we believed that this approach would be helpful to students, because it provides feedback for

Figure 5.3: Heat map with all lines highlighted. Heat maps like this are frustrating for students because there is no actionable insight to be gained from this feedback.

debugging that is within the context of their own source code, not just as a list of error statements in a separate display.

We also manually reviewed the heat maps that had been generated, created a summary analysis of the heat maps themselves, and after completing this review, we understood one reason why the heat maps were confusing. Figure 5.4 shows a graph of the percentage of non-comment lines of code (NCLOC) in the submissions that were highlighted in the heat maps. In our initial deployment, we showed the results of *all failed reference* tests (indicated by the blue column).

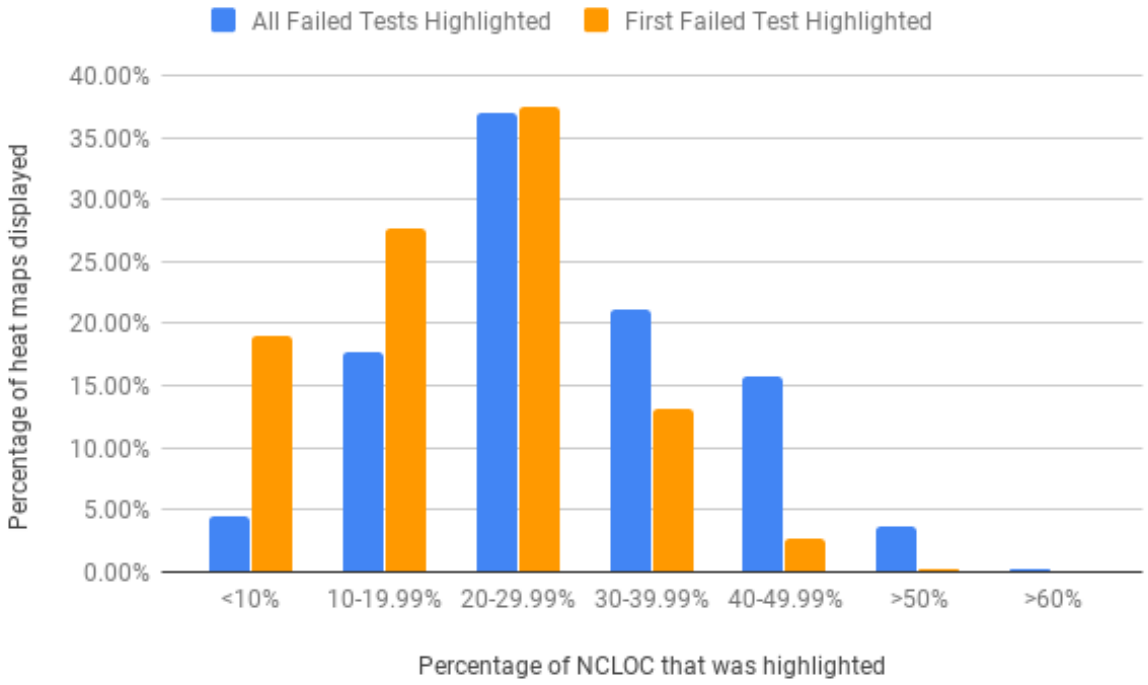


Figure 5.4: Percentage of heat map highlighted code as a percentage of NCLOC in submissions, all tests (in blue), versus first failed test (in orange).

Additionally, Figure 5.5 indicates the heat maps also have a broad highlighting of methods as well. Thus, the highlights are not concentrated in a small set of larger methods. They are spread out across the classes, touching code in many methods. This means that most students will be looking at a heat map with a significant portion of the source code identified as suspect in the heat map.

In our attempt to address this issue, we reassessed the student submissions with a modified version of the heat map generator. Instead of showing the results for all of the failed tests, we generated heat maps based on the first failed test that was encountered by GZOLTAR. In order to insure that the suspiciousness for the entire set of spectra was consistent, we didn't stop testing after the first failure, we merely noted the first failed test and displayed it by itself. The orange column in Figure 5.4 shows the results of the reassessment and suggests

we were correct, but not to the extent that we expected. The farther the data shifted left, as indicated by the orange columns, the more focused the heat maps would be, based on the reduced percentage of highlighted code. We would have hoped that the results would have shifted more to the left for the "first failed", indicating that fewer lines of code were being highlighted in more heat maps.

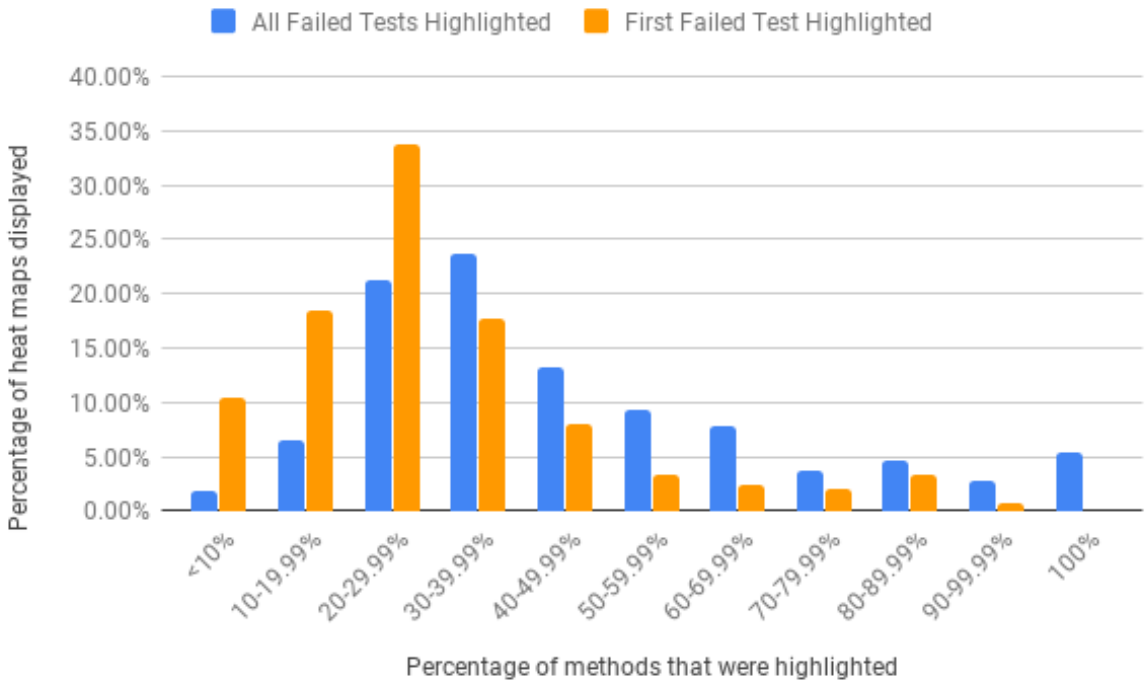


Figure 5.5: Percentage of heat map highlighted code as a percentage of methods in submissions, all tests (in blue), versus first failed test (in orange).

Rather than focusing the student's efforts, the heat maps in their current configuration actually diffuse the focus. However, notice that like the lines of code, changing to a "first failed" reporting option does increase the percentage of the heat maps that have a lower percentage of methods highlighted in them, further reducing the noise to which students are subjected. Coupling this with the survey responses, it is clear that the current approach provides too much information in a way that is not directed enough for students, especially

novice students, to gain much benefit.

5.6 Lessons Learned

Based on our use of heat maps for two semesters, we have come away with some lessons learned about our experience. First, it is not useful to generate heat maps when too many tests fail, or, more precisely, when too much of the program is highlighted by the spectra of failing tests. When the whole design is highlighted due to multiple simultaneous failures, the heat map is no longer helpful for localization. Above a certain point, the highlights become noise.

Second, instead of mapping all of the failed tests, choose a model such as the `first failing test` to "mask" the heat map. This effectively focuses on displaying the slice of the heat map corresponding to one feature (and hopefully, one failure mode or fault). There is still the possibility that even in this model, a heat map could be drawn where all of the lines could be highlighted, such as a case where all test cases are failing regardless. This could be mitigated by adding additional checks in the auto-grader to limit the heat map display if the percentage of NCLOC from the project represented in the map is above a certain threshold.

As a step in this direction, we have developed a version of the heat map generator that allows the instructor to set a "failed test" depth, such that the map will only show the number of failed tests in the map as specified, regardless of the actual number of tests the code fails. Using the same submissions from the students in Fall 2017/Spring 2018 CS 1 courses, we re-created the heat maps, and then evaluated whether less code was being highlighted, as a percentage of NCLOC and methods in the student submissions. As indicated by the orange columns, figure 5.4 shows that the number of heat maps with a 20% or more of highlighted lines has decreased when we limit the heat map to reporting the results of the first failed

test.

Third, label the heat map with the goal/feature being used as the filter. This can address the fact that students don't have access to the reference tests and don't know what is being tested. This helps balance the need to keep reference tests hidden, with the need of programmers to know what fault is being highlighted in the heat map. This will require additional effort on the part of the instructor, as this is effectively combining the functionality of a hinting system with the heat map. The hints or prompts will need to be meaningful, but not give too much away to the student. Some of this effort can be mitigated by appropriately scoping the reference tests when they are written, and using meaningful names for the tests that can be displayed to the students as part of the feedback display.

Fourth, provide intentionally chosen labeling for heat maps to help students understand the correct interpretation and to preempt misconceptions about "the computer points out the bug". The heat maps as we have currently implemented them are displayed as simply the source code displayed in-line with the rest of the feedback provided by the auto-grader. We don't provide any prompts as to the interpretation of the heat map. Providing specific wording could help clarify the use of the heat map as well as address some of the issues associated with the limitations of the underlying SFL analysis. One possible description that could be offered:

> Your submission has been analyzed using statistical fault location. In this heat map, colored lines indicate code executed during a failing test case, with more red lines indicating code that is less commonly (or never) executed in other passing test cases. A colored line is not necessarily defective itself, but it was definitely involved in the execution of a reference test case that failed. Note that a test may have failed because of missing code, as well as defective code.

Heat maps provide a way to visually display the results reference test outcomes within the context of student code assignments. However, their use has to be carefully planned and considered. As we have shown, it is easy to overload the student with more information than they are able to use or process, leading to frustration.

We have provided several alternative approaches for addressing these issues. Our plan is to first implement the *first failed test* option in a class setting, along with the additional interface enhancements discussed. We have shown here that this approach will lower the "noise" the students encounter. However, even without these changes, another question occurred to us: Is is possible that the heat maps provided some advantage to the students, even if they didn't necessarily directly perceive the benefit of having this type of feedback available? We decided to dig into the data that Web-CAT collects to find out.

Chapter 6

Quantitative Evaluation of Heat Map Effects on Student Performance

In this chapter, we describe the results of an analysis of the performance of two cohorts of students, one that was provided the heat map feedback, and one that was not. We see in this analysis that students that were presented the heat map had more frequent incremental improvements to their scores on the instructor-provided reference tests for the programming assignments. Also, they were able to achieve their highest correctness score in less time than students that did not have access to the heat maps.

6.1 Questions Addressed in this Study

The object of the heat map feedback is to improve the student's focus for identifying the root cause in their code of the failure of the instructor's reference tests. By improving their focus, and helping to guide the review of their code, we believe that students will be better able to understand how to identify and correct defects in the future. We looked at two primary axes to determine whether this intervention was having an impact. First, we looked at the proportion of submissions that improved the student's score towards the highest number of points they achieved. Secondly, we looked at whether students who had access to the heat map feedback needed less time to reach their highest score for the correctness portion of the

assignment, which corresponded to passing all of the reference tests. This led to two specific research questions:

- **RQ1:** Does the heat map feedback make it easier for students to improve their code?
- **RQ2:** Does the heat map feedback allow students to complete their projects faster than students who do not have access to it?

We will provide a background for our CS1 course to provide a context for these questions.

6.1.1 Course Context

The *CS 1114: Introduction to Software Design* is typically one of the first formal programming courses taken by computer science majors at Virginia Tech. It uses a test-driven development [19] approach. Students are taught to create their test cases first, and then develop the implementations necessary to correctly address the tests. The course offerings being compared were taught during the Fall semesters of 2015 and 2017, and the Spring semester of 2019.

The course is taught with students meeting for a joint class meeting twice weekly, with the third weekly meeting being in a smaller lab section. The course is taught using Java, and includes several projects (five in Fall 2015, six in Fall 2017 and Spring 2019) to assess student progress. In addition to projects, CS 1114 uses weekly labs, homework assignments, and examinations to assess students progress through the course. The overall structure and course delivery has not changed significantly over the period of the study. The only significant delivery change was transitioning to a new enterprise learning management system in the Fall of 2017[1].

[1]Virginia Tech started a transition to the Canvas LMS[54] platform in Fall 2015. This transition was

The projects are multi-week assignments that require the students to use the concepts discussed in class and the lab sections in a meaningful way. We use Web-CAT [39] to provide automated assessment and feedback on the students work. Students use Web-CAT for submitting all of their project solution attempts, and the heat map feedback is provided in Web-CAT as another feedback method. Web-CAT applies instructor-supplied reference tests to the student's submission, and then reports to the student the results of the assessment. There were three parts to the student's grade:

- Automated correctness assessment: **40 points**
- Automated assessment of coding style and structure: **20 points**
- Manual review of the code by the instructional staff for items such as design, readability, and any assignment-specific requirements: **40 points**

The heat map is displayed by default when the student reviews the results of their submission's assessment.

Previous work on mining the data of automated graders [14] informed our work on how we could use the data that Web-CAT collects to assess submission behavior. As part of the setup for assessing student submissions, the instructor provides reference test cases that are applied to the student submissions. Web-CAT allows an instructor to apply a point-value to the code successfully addressing the scenarios included in the instructor reference tests. Web-CAT also captures a wide variety of data points associated with each student submission, including timestamps for each submission, scoring for the various assessment components for the assignment, and metrics about the student code that is submitted. These metrics

completed in Fall 2017. The Department of Computer Science at Virginia Tech had its own Moodle[33] server for course delivery until Fall 2016, at which time course delivery transitioned to using the Virginia Tech Canvas instance.

include the number of non-comment lines of code, the number of methods and the number of conditionals in the submitted code.

6.1.2 Comparing Assignments Across Terms

One of the difficulties with comparing student performance on programming assignments across terms is the difference in the assignments themselves across course offerings. While the assignments may seek to assess student knowledge on the same course topics, the assignment scenario itself can add or remove complexity depending on how it is constructed. In order for comparisons between projects to be meaningful, we needed a way to determine if the projects were of a similar complexity.

The approach we took was to look at students submissions across semesters for specific projects to see if the solutions submitted had a similar complexity. We reviewed each semester and grouped the projects based on the concepts the projects were designed to assess. This resulted in four projects being grouped across Fall 2015, Fall 2017 and Spring 2019. Our task was simplified to some extent because in the case of two of the projects, the same project specification was used in each semester. The other two projects had students perform similar tasks and focus on similar concepts, but used different programming scenarios. However, for consistency, we performed a complexity analysis on all four project groups.

Two of the pieces of metadata that Web-CAT records about student code are the number of methods and conditionals in the student's submission. With these two values, we can calculate a complexity value that is similar to cyclomatic complexity [75]. To calculate the complexity of a project, we added the number of methods and conditionals for each student's submission where they achieved their highest correctness score, and then took an average of the complexity score for all students across a project. This gave us an average complexity

Table 6.1: Project Complexity Comparison, Fall 2015 vs Fall 2017 and Spring 2019

Project Group	Concepts	$\overline{Project\ Complexity}$				
		Fall 2015	Fall 2017	p	Spring 2019	p
1	Fields and Parameters	45.8	44.1	0.19	47.1	0.31
2	Grouping Objects	66.9	64.5	0.21	63.4	0.20
3	Variable Scoping	135.2	138.8	0.20	113.9	0.16
4	Arrays and Maps	78.5	76.0	0.10	76.4	0.10

($*$ indicates significant at the $p<0.05$ level)

for the project. We then compared projects that cover similar concepts and conducted an ANOVA analysis to test if the means for the complexity were significantly different across terms. We determined that the complexity across the project groups was not significantly different. The results of this analysis are presented in Table 6.1 and the data is summarized in Figure 6.1.

To further validate the similarity between these projects, as well as the student cohorts that made up the classes in these terms, we reviewed some of the metadata associated with the way that students completed their assignments generally. We wanted to further insure that any effects we saw on student performance could be attributed to the impact of the heat map feedback. There was a significant time lapse between these three cohorts, and many factors could have impacted who these students were when they arrived at Virginia Tech to take CS 1114. Similar overall submission counts, as well as similar overall times to complete the projects would suggest that one group was not "more prepared" or "more talented" than the other group, indicating they all started the course, as a group, from a similar position.

First, we assessed whether there was a significant difference in the number of submissions made by students across terms. We compared the average number of total submissions for all students for each project group. We found that there was not a significant difference in the average number of submissions made by students in the Fall 2015 semester compared to

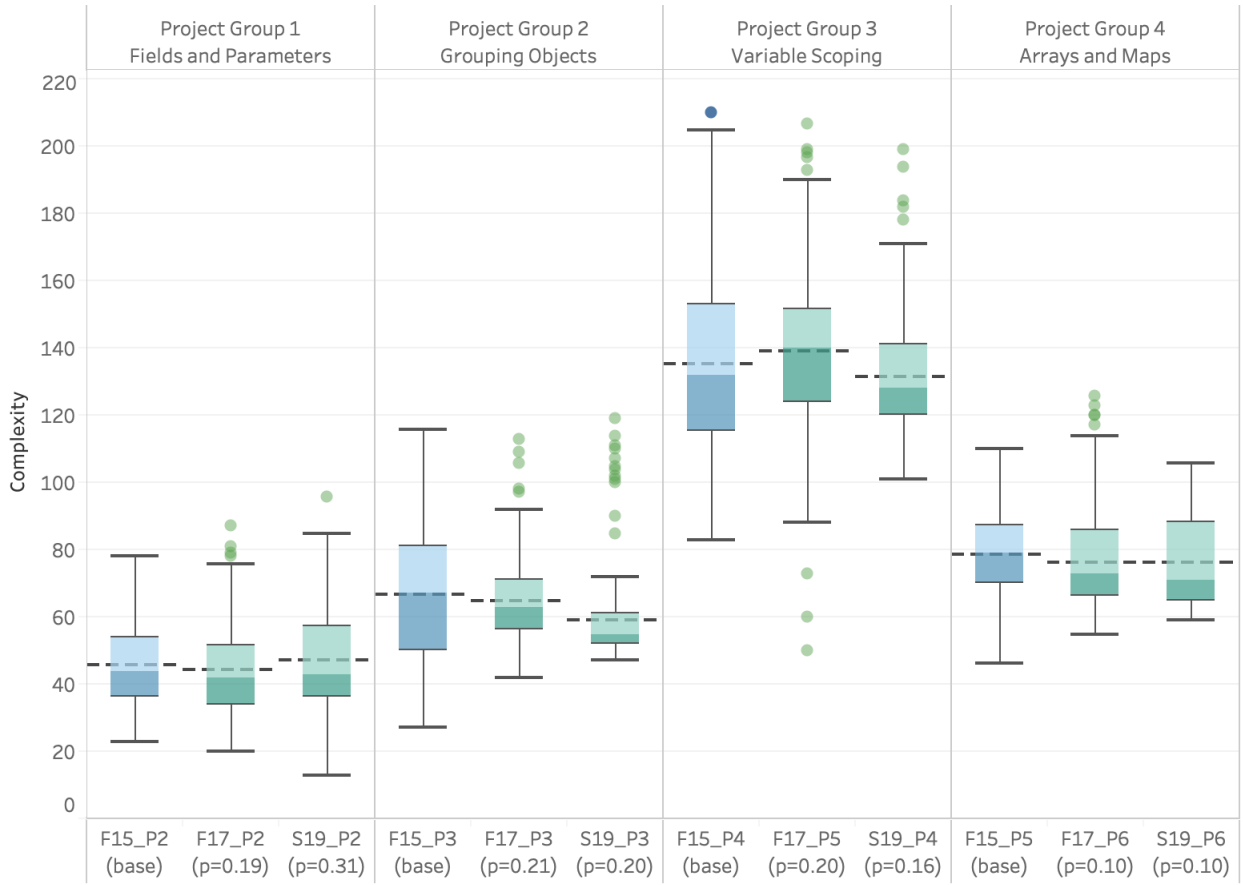


Figure 6.1: Box plots for the data analyzed to determine if there was a difference in the average complexity of the submissions for the Fall 2017 and Spring 2019 projects, relative to the Fall 2015 projects. ($*$ indicates significant at the $p<0.05$ level)

the students in the Fall 2017 semester, nor was there a difference between the submissions by the Fall 2015 students and their Spring 2019 counterparts. The results of this analysis when comparing Fall 2015 to Fall 2017 and Spring 2019 is presented in Table 6.2, and the data summarized in Figure 6.2.

Secondly, we looked at the total elapsed time students took from the point of their initial submission to the time of their final submission. This is not a measure of the total time the students take on the project, as it does not account for any work time done before the first submission. As in the case for the number of submissions, we found no significant difference

Table 6.2: Project Submission Count Comparison, Fall 2015 vs Fall 2017 and Spring 2019

	$\overline{Submission\ Count}$				
Project Group	Fall 2015	Fall 2017	p	Spring 2019	p
1	7.27	7.79	0.32	7.03	0.61
2	8.68	9.50	0.12	7.86	0.08
3	11.23	10.66	0.43	10.22	0.12
4	9.09	8.59	0.37	8.47	0.24

($*$ indicates significant at the $p<0.05$ level)

Table 6.3: Project Submission Elapsed Time Comparison, Fall 2015 vs Fall 2017 and Spring 2019

	$\overline{Total\ Elapsed\ Time}$				
Project Group	Fall 2015	Fall 2017	p	Spring 2019	p
1	25.27	27.68	0.48	21.40	0.23
2	35.51	29.11	0.11	25.76	0.08
3	25.63	28.32	0.37	19.61	0.10
4	27.42	23.47	0.26	20.84	0.07

($*$ indicates significant at the $p<0.05$ level)

in the amount of time students spent on submitting their assignments to Web-CAT in Fall 2015 than they did in Fall 2017 or Spring 2019. The results of this analysis is presented in Table 6.3. The data is summarized in Figure 6.3.

This analysis supports the argument that not only were the projects being compared across semesters similar, but the student groups were also similar. If one group or the other were more prepared, we would expect that group to take significantly less time and to require fewer submissions. This was not the case. There was no significant difference in the average time students took to complete their projects, from first to last submission to Web-CAT, nor was there a significant difference in the number of times students submitted their projects to Web-CAT. Finally, the average complexity of the solutions that students developed was not

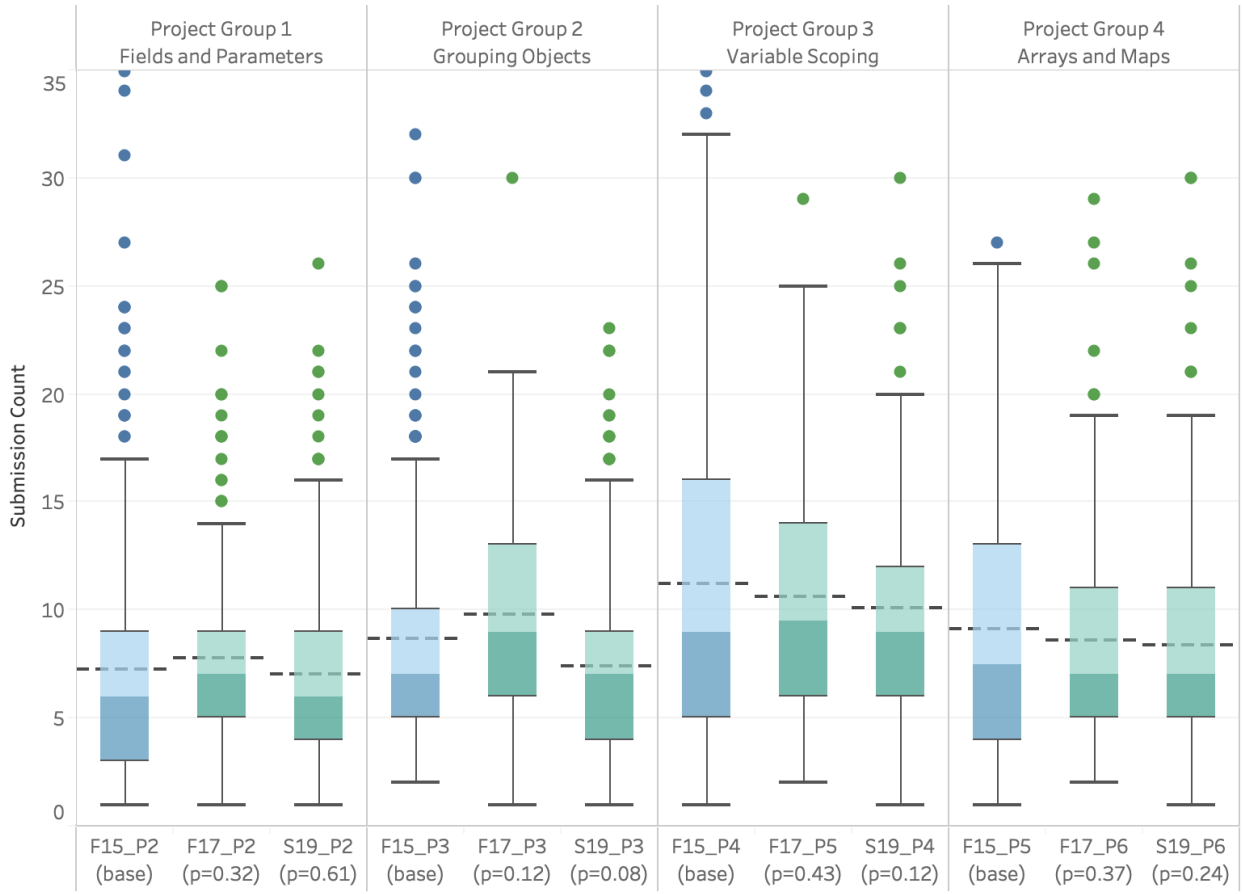


Figure 6.2: Box plots for the data analyzed to determine if there was a difference in the average number of submissions for the Fall 2017 and Spring 2019 projects, relative to the Fall 2015 projects. ($*$ indicates significant at the $p<0.05$ level)

significantly different between the cohorts compared. Once we established that the groups were similar, we then moved on to see what impact the heat map feedback had on student performance.

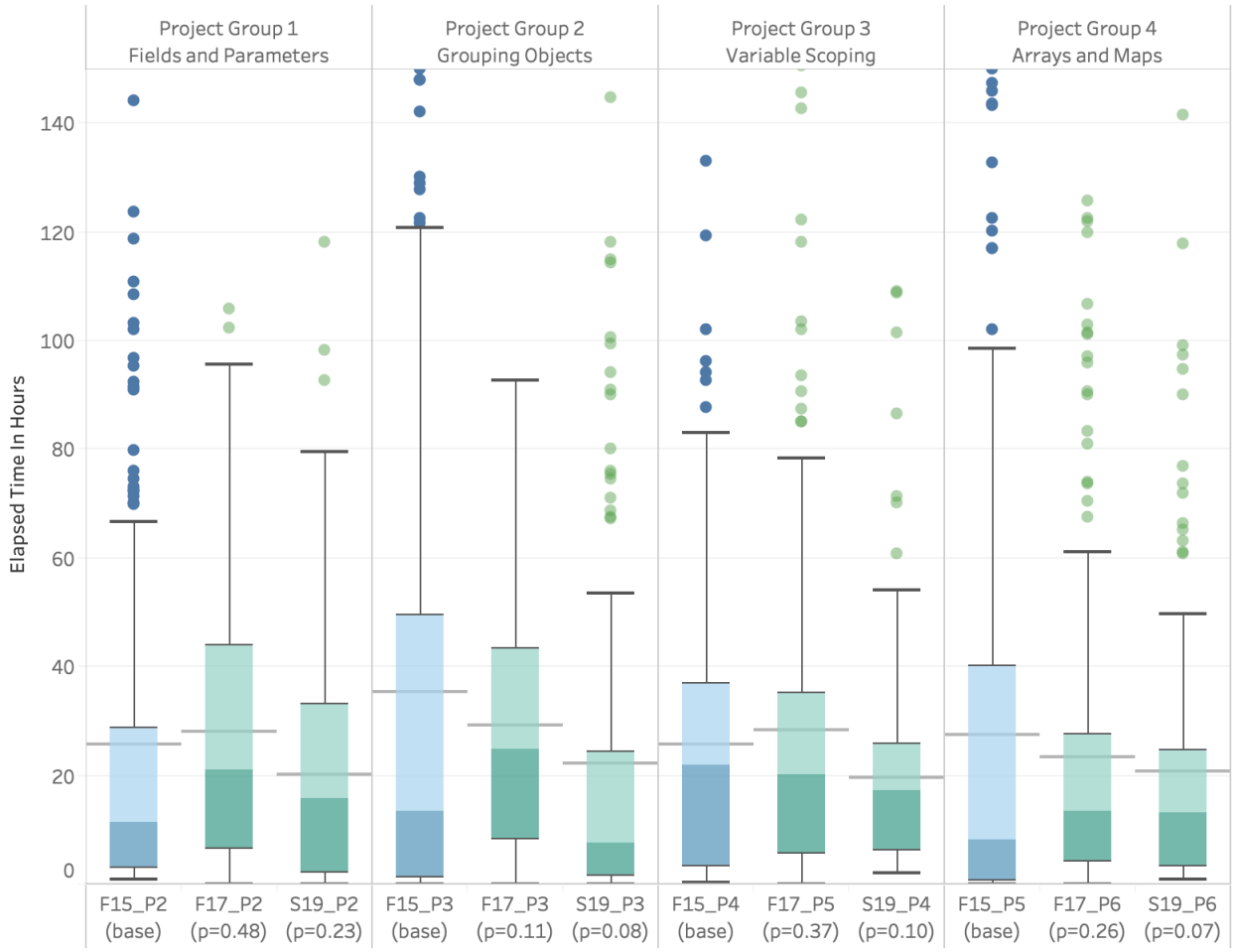


Figure 6.3: Box plots for the data analyzed to determine if there was a difference in the average total elapsed time take from the first submission to the final submission, for the Fall 2017 and Spring 2019 projects, relative to the Fall 2015 projects. ($*$ indicates significant at the $p<0.05$ level)

6.2 Results

6.2.1 Question 1: Heat map access makes it easier for students to improve their code

One of the purposes of the heat map feedback is to allow students to more easliy find the defects in their code, by providing a context for the feedback within their code. We want

Table 6.4: Relative Frequency of Improvement, Fall 2015 vs Fall 2017 and Spring 2019

	Relative Improvement Frequency				
Project Group	Fall 2015	Fall 2017	p	Spring 2019	p
1	0.31	0.42	**<0.001***	0.46	**<0.001***
2	0.34	0.53	**<0.001***	0.51	**<0.001***
3	0.43	0.62	**<0.001***	0.56	**<0.001***
4	0.47	0.56	**<0.001***	0.53	**0.002***

($*$ indicates significant at the $p<0.05$ level)

to provide students a means for focusing their efforts on defective code, without actually showing them exactly where the defects are located. This should encourage students to submit changes that are more correct with each attempt. For our purposes, "ease" was defined as making more frequent improvements to the correctness score portion of the project. To address whether it is easier for students to improve their code, we looked at the frequency of submissions that improved the correctness score, as a percentage of the total number of submissions. Frequent, incremental improvement is fundamental to test-driven development[19]. If students more frequently improve their code, this supports the TDD methodology that we encourage of "*test-early, test-often*". If a larger number of submissions improved the student's correctness score, it was "easier" for the student to improve their code. We counted the number of student submissions where the student improved their correctness score over their previous high score. Figure 6.4 shows a hypothetical student and which submissions would be considered "increasing" submissions.

Every student has their own individual way of approaching programming assignments. Some students submit more frequently than others. To account for the absolute differences in the number of submissions, we determined the percentage of increasing submissions for each student. We divided the number of increasing submissions by the total number of project submissions for the student to calculate this percentage. We then conducted an ANOVA

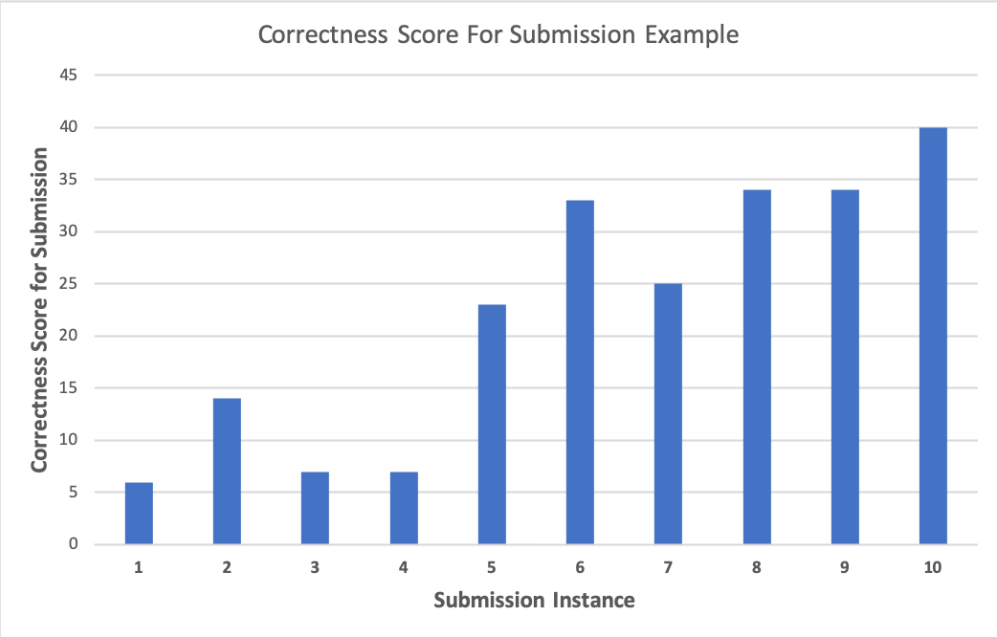


Figure 6.4: An example of how the number of increasing correctness score submissions was determined. In this example, submissions 1, 2, 5, 6, 8, and 10 show a score increase over the previous highest scoring submission. Thus, in total, six of the 10 submissions showed increases.

analysis on these values to determine if there was a significant difference between projects in each project group. Table 6.4 summarizes the results of this analysis.

As shown in Figure 6.5 and Table 6.4, in each project group there was a significant increase in the proportion of submissions for each of the Fall 2017 and Spring 2019 projects where a student improved their correctness score when compared to the Fall 2015 projects.

Additionally, we conducted an analysis to determine actual increase in the number of submissions that improved the students' scores. We used Cohen's d_s measurement[29] since we had samples of unequal size. The results of these analyses are listed Tables 6.5 and 6.6. Cohen defines effect sizes as 'high' ($d_s > 0.8$), medium ($d_s > 0.5$) and low ($d_s > 0.2$). In the case of Fall 2017, two of the four projects had a large effect size, one had a medium effect, and one had a small effect. The largest effect, for project 3, had $d_s = 1.05$. This corresponds

Table 6.5: Effect Size of Improvement Frequency, Fall 2015 vs Fall 2017

	Fall 2015		Fall 2017			
Project Group	$\overline{\#Sub}$	$\overline{\#Improve}$	$\overline{\#Sub}$	$\overline{\#Improve}$	Cohen's d_s	ΔFrequency
1	7.68	1.86	7.82	2.95	*0.60*	0.34
2	8.42	2.40	9.68	4.87	**0.99**	0.97
3	15.39	5.74	10.63	6.23	**1.05**	1.25
4	11.59	4.36	9.42	5.23	<u>0.44</u>	0.45

(Values <u>underlined</u> are low effect, *italic* are medium effect, **bold** are high effect.)

Table 6.6: Effect Size of Improvement Frequency, Fall 2015 vs Spring 2019

	Fall 2015		Spring 2019			
Project Group	$\overline{\#Sub}$	$\overline{\#Improve}$	$\overline{\#Sub}$	$\overline{\#Improve}$	Cohen's d_s	ΔFrequency
1	7.68	1.86	7.12	2.81	*0.77*	0.48
2	8.42	2.40	7.46	3.58	**0.86**	0.65
3	15.39	5.74	9.42	5.23	*0.66*	0.80
4	11.59	4.36	7.83	3.94	<u>0.28</u>	0.24

(Values <u>underlined</u> are low effect, *italic* are medium effect, **bold** are high effect.)

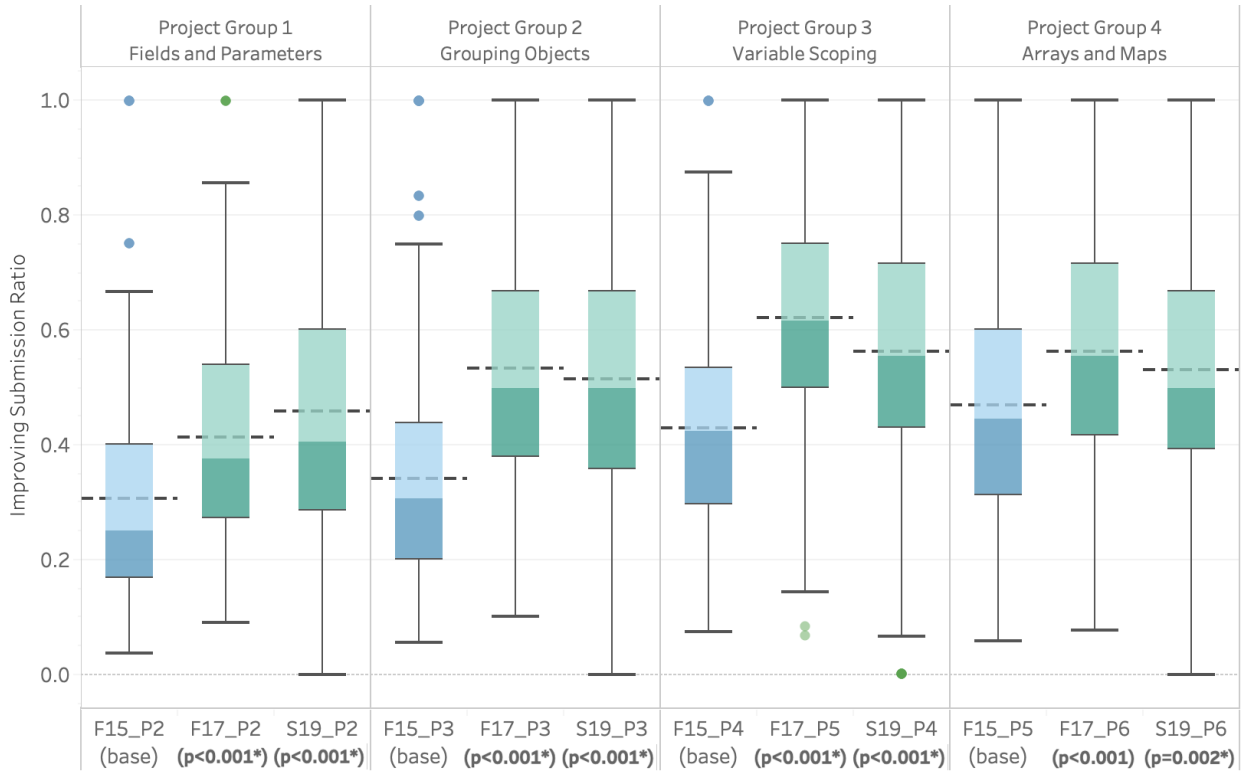


Figure 6.5: Box plots for the data analyzed to determine the ratio of improved submissions to total submissions for the Fall 2017 and Spring 2019 projects, relative to the Fall 2015 projects. ($*$ indicates significant at the $p<0.05$ level)

to an additional 1.25 submissions that improve the students score, on average, for the Fall 2017 students. The smallest effect, for project 4, where $d_s = 0.44$. On average, this would result in an additional 0.45 submissions that improve the student's score. When looking at the Spring 2019, one project, project 2, has a large effect, with $d_s = 0.86$. This corresponds to an increase of 0.65 submissions that increase the student's score, on average. The Spring 2019 project with the smallest effect was also project 4, where $d_s = 0.28$. This resulted in an additional 0.24 submissions, on average, that increased the student's score.

6.2.2 Question 2: Heat map access allows students to achieve their highest score faster

Each student works at their own pace. Some students will start early, working on the project for a while, then setting it aside for some period of time. Others will work steadily, spending consistent blocks of time as they work to complete the project. Still others will wait until the last minute, and then complete the project in a single marathon sitting. To determine whether students achieve their highest score faster, we needed to determine a metric that would account for the times between when students worked. This would allow us to track the time students actually spent working on their submissions, rather than the elapsed time that might include long breaks for class attendance, sleeping or eating, and the like.

In [55], an analysis was conducted using similar data to the data we had access to, where the researchers only had the timestamps for the submissions to determine when students were workings. When these event occurred in groups, they defined them as "work sessions". As defined in [55], a work session was considered to be all of the submissions that occur without an elapsed time of more than two hours between submissions. Figure 6.6 shows a hypothetical example of a student working on a project across four work sessions. We adopted this metric to describe how students worked.

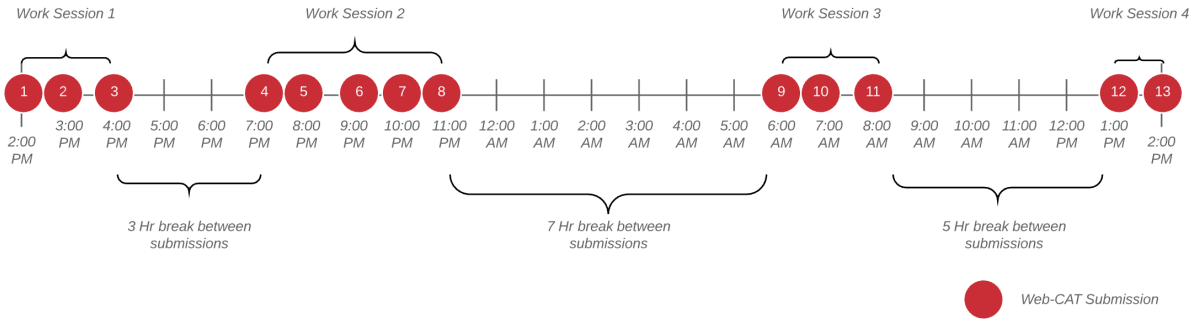


Figure 6.6: A hypothetical student working on their project across four work sessions.

Table 6.7: Time to Highest Score, Fall 2015 vs Fall 2017 and Spring 2019

	$\overline{Time\ To\ Highest\ Score}_{\text{hours}}$				
Project Group	Fall 2015	Fall 2017	p	Spring 2019	p
1	1.70	1.40	**0.03***	1.39	**0.03***
2	1.65	1.89	0.10	1.86	0.09
3	3.07	2.53	**0.04***	2.50	**0.02***
4	2.37	1.96	**0.04***	1.88	**0.01***

($*$ indicates significant at the $p<0.05$ level)

Once we had a way to address the "non-working" time for a project, we were able to calculate the time needed for the student to achieve their highest correctness score for each project. We determined the distinct work session for each submission, and then calculated the elapsed time for each submission from the previous submission in the work session. To find the total working time for the submission that achieved the highest correctness score, we added the times for all of the submissions together, including the submission where the highest score was achieved. If the highest score was achieved in multiple submissions, we used the submission where the score was achieved the first time. Once we had these times for each student for each project, we then conducted ANOVA comparisons of the means for each project group to determine if there was a significant difference between semesters in the time each class took to reach the highest correctness score. Figure 6.7 summarized the results of this analysis.

There was a significant reduction in the average time need to attain the highest score between project groups 1, 3 and 4 (see Table 6.7) for both Fall 2017 and Spring 2019 students. Thus, students in both classes took less time on projects covering *Fields and Parameters*, *Variable Scoping*, and *Array and Maps*, then students in Fall 2015 working on the corresponding projects. We also calculated the effect size of these reductions, again using the Cohen's d_s metric. Overall, while significant, there was only a small effect, as defined by Cohen, on project groups three and four in Fall 2017 and on projects one, three, and four in Spring

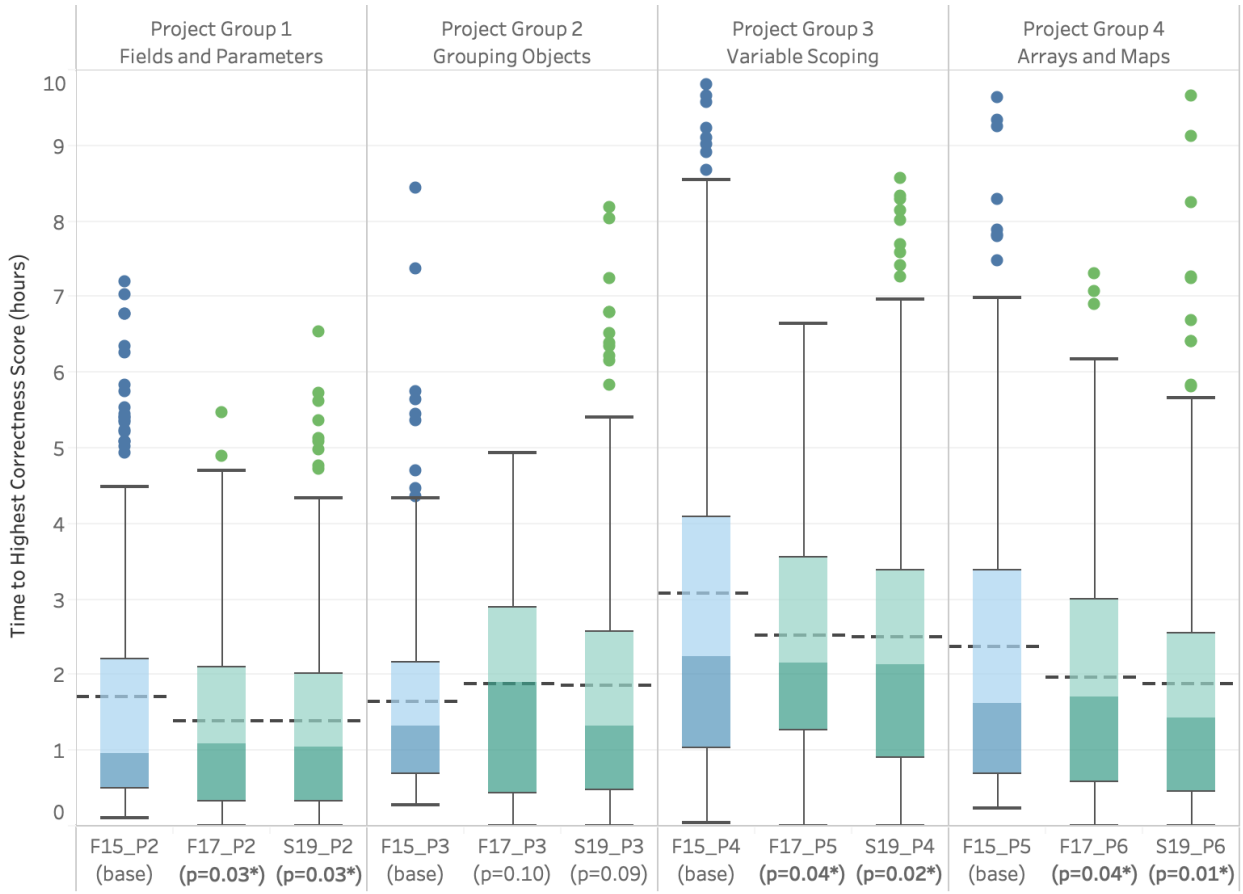


Figure 6.7: Box plots for the data analyzed to determine the difference in average times to achieve the highest correctness score for the Fall 2017 and Spring 2019 projects, relative to the Fall 2015 projects.($*$ indicates significant at the p<0.05 level)

2019. The reduction in time to achieve the highest correctness scores, on average, ranged from 16.2 minutes for project 1 in Fall 2017 to 28.2 minutes in project 3 in Spring 2019. These findings are summarized in Table 6.8

As noted, a concern was that one group of students may have been more prepared that the other. For example, if the 2017 cohort had been made up of more students with more programming experience, that experience might account for them making more frequent improvements, and achieving their highest correctness scores faster. As the analysis reported in Tables 6.2 and 6.3 show, that was not the case. The three cohorts used, on average,

Table 6.8: Effect Size of Time to Highest Score Change, Fall 2015 vs Fall 2017

	Fall 2015	Fall 2017			Spring 2019		
Project Group	$\overline{T}_{\text{hours}}$	$\overline{T}_{\text{hours}}$	Cohen's d_{s}	$\Delta\text{T}_{\text{hours}}$	$\overline{T}_{\text{hours}}$	Cohen's d_{s}	$\Delta\text{T}_{\text{hours}}$
1	1.70	1.40	-0.19	-0.27	1.39	<u>-0.20</u>	-0.28
2	1.65	1.89	0.18	0.25	1.86	0.13	0.25
3	3.07	2.53	<u>-0.22</u>	-0.40	2.50	<u>-0.23</u>	-0.47
4	2.38	1.96	<u>-0.21</u>	-0.33	1.88	<u>-0.24</u>	-0.46

(Values <u>underlined</u> are low effect, *italic* are medium effect, **bold** are high effect.)

the same number of overall submissions, and the same amount of time once they started submitting to Web-CAT. This supports the claim that the heat map feedback provided the students with a means for making more frequently improvements to their code, as well as completing the reference test portions of the projects faster.

Chapter 7

Discussion

7.1 Where We Have Been So Far...

We started this effort with this overarching question: **how can computer science education use automated fault localization tools to guide and assist students' debugging efforts?**, and set out on a multi-step approach to determine the answer. Let us review each one of those steps in turn.

1) Identify a Technique

We started by working to **identify an automated defect localization technique and validate its ability to reliably identify defects in student code submissions** (See Chapter 3). The GZoltar library was found to be a good fit for our needs. It is open-source, and based on Java, which is our language of choice for our core computer science courses. After applying GZoltar against a set of student submissions, and then manually inspecting the results, we determined that GZoltar identified the location of the defect as being in the top three most suspicious methods 90% of the time.

2) Create Contextualized Feedback of the SFL Analysis

After validating that the GZoltar analysis was a viable method for conducting an SFL

analysis of student code, we needed to **create a feedback method that provides contextualized feedback to students based on the results of the defect localization process** (See Chapter 4). We used a "heat map" visualization model to create an overlay of the suspiciousness scored for each code line on top of the students' submission. The result is a set of marked-up source code that shows all of the suspicious code that has been identified by GZoltar. All potential issues are shown at a glance to the student. Since we mark all of the lines in the file that were scored with a non-zero suspicion score, the student will be directed to first look at the methods marked, but then can see all of the other lines of code that might be potentially problematic. The coloring of the lines indicates the level of suspicion.

3) Apply the SFL Technique at Scale

Next, we worked to **design and develop a tool to apply this defect localization technique to student submissions at scale, within the context of an automated grader** (See Chapter 5). We developed an Apache Ant-based plugin for our auto-grader Web-CAT, which allows the SFL analysis and visualization to be applied to student submissions at-scale, along side the existing feedback methods already built into Web-CAT. This allows us to offer this new feedback while continuing to support the use of existing feedback incentives, such as achieving a minimum code coverage percentage from the students own tests before feedback on reference cases is provided. This solution was deployed in Web-CAT in the Fall semester of 2017, and has been used in CS 1114 since that time. There have been 2,267 students enrolled in CS 1114 since Fall 2017[1]. For the courses involved in this study alone, the GZoltar tool processed 47,477 student submissions. We showed our research tool can function in a production environment.

[1]Enrollment figures captured 11 Sept 2019

4) ASSESS STUDENT PERCEPTIONS OF THE FEEDBACK

After providing the students with the heat map feedback for a semester, we surveyed the students to **assess the feedback provided based on students' perceptions of its utility in helping them understand the defects in their code and in correcting those defects** (See Chapter 5). The survey results suggested that students had misconceptions about what the heat maps could and could not do. The results also suggested several changes to the heat map display that could make them easier to use.

5) ASSESS THE IMPACT OF THE FEEDBACK ON STUDENT PERFORMANCE

Given the lukewarm responses to the heat maps we received from students, we decided to **assess the feedback provided based on students' performance on programming assignments where the feedback is provided, versus when the feedback was not provided** (See Chapter 6). We compared the performance of students across three semesters, one where students were not provided the heat maps (Fall 2015) and two where they were (Fall 2017 and Spring 2019). We discovered that despite the lukewarm perception of the heat maps, students improved their submissions more frequently and achieved their highest correctness scores faster when they had access to the heat map feedback.

7.2 What Have We Discovered?

We have discovered a number a of interesting things while answering our overarching question, some of them quite unexpected.

7.2.1 SFL is a Practical Analysis Solution

When we first started looking at using SFL as an approach for analyzing student code, we did not completely understand how SFL would operate in an educational setting. We expected that we would get results that would provide different suspiciousness scores for different lines of codes. This was not the behavior we observed. While GZoltar did not always produce uniform scores throughout a method, it quickly became apparent that all of the lines that were consistently executed together were going to participate in the same spectra, and thus receive the same suspicion score. Because of this, it is unreasonable to expect SFL to pinpoint defects down to the line level. However, GZoltar was much more accurate at identifying the method that contained a bug. As a result, we switched our attention from using SFL to identify lines containing bugs to using SFL to identify methods containing bugs. This was actually a better outcome, as our overall goal was always to provide a scaffold to guide students in their debugging efforts, rather than pointing them directly to the lines where the failures occurred. This also addressed the issue of bugs occurring because of omitted code (such as a missing `else` condition) where there is, by definition, no line of code to point the student toward.

Additionally, we manually inspected the student programs to verify that the method identified by GZoltar as containing a bug–that is, failing an instructor-provided reference test–was the actual location of the defect that GZoltar was turning into a suspiciousness score. Methods scored by GZoltar were then ranked by their suspicion score, and then the top-ranked

method containing a bug was identified based on the manual debugging results. As we reported earlier, (see Figure 3.5), GZOLTAR identified the true location of the bug in the top three ranked methods 90% of the time. This is a good performance, but as suggested earlier, it leaves a very long tail of possible misidentified locations where the true defect could exist. One possible way of presenting the results from GZOLTAR to students could have been just presenting a long list of method names, but that is not much different from the types of feedback that are provided now. Also, a list would not provide any more context than students are currently provided. We needed to find a way to represent all of the locations that GZOLTAR had identified as suspicious, without just presenting a long list of method names.

7.2.2 Heat Maps Provide More Contextualized Feedback

The heat map approach to providing the results from the GZOLTAR analysis addresses these context issues, as well as the issues associated with the "long tail" of methods (see Figure 3.5) where the defect might be located. This model provides a variety of advantages over traditional, text-based feedback approaches. First, it provides context, not just to the defects, but to the structure of the student's code. Second, the top methods identified as defective are shown in the context of all of the defects in the code. Thus, if a student is reviewing one of the most suspicious methods, they can look at the method code and quickly see if that method calls any other methods that also have suspicious code. This focuses the student's attention. We can provide meaningful direction without telling the student exactly where the error is located.

Third, this method of displaying the results of the fault location allows us to show the student the location of *all* defects, rather than just those with the highest score. This addresses the issue of having "false positive" methods being highlighted in the list as defective when they

are not. By showing the student all potentially defective lines, as well as the suspiciousness of each line, the student can see every location that might be problematic. Additionally, this addresses the issue of defects cause by a lack of code. If a defect is caused by something that has not been implemented, such as a missing `else` condition on an `if` statement, the method will still be marked as defective. Finally, this feedback approach has the added benefit of enhancing the accessibility of feedback to differently-abled learners, as well as learners who are not native-speakers of the language the textual feedback is provided. By augmenting the markup used to implement the heat map, we can enable screen readers to more accurately report to the student location information about the defects. Also, non-native speakers can look at the code and get information from the visual display that may not be immediately obvious from the the textual feedback that could be provided in a language other than their first.

7.2.3 More is Not Necessarily Better: How the Heat Maps Are Presented Matters

We studied the use of the heat maps across two semesters, and learned some lessons from that experience. First, more isn't not necessarily better. It is not useful to students to generate heat maps when too many tests fail, or, more precisely, when too much of the program is highlighted by the spectra of failing tests. When the whole design is highlighted due to multiple simultaneous failures, the heat map is no longer helpful for localization; the highlights just become noise. This leads to student confusion and frustration. They don't know where to start looking, which is the exact opposite of our goal.

Second, instead of mapping all of the failed tests, our experience in the classroom suggests we should choose a model such, as "first failing test" to "mask" the heat map. In this model,

we would conduct the full SFL analysis, but we would only show the SFL results of the first failed test we encounter. This effectively focuses on displaying the slice of the heat map corresponding to one feature (and hopefully, one failure mode or fault). There is still the possibility that even in this model, a heat map could be drawn where all of the lines could be highlighted, such as a case where all test cases are failing regardless. This could be mitigated by adding additional checks in the auto-grader to limit the heat map display if the percentage of NCLOC from the project represented in the map is above a certain threshold. To this end, we have created an updated version of the heat map plug-in for Web-CAT that allows instructors to select the "test depth" they would like the heat map to report.

Third, labeling the heat map with the goal/feature being used as the filter would be a benefit. This can address the fact that students don't have access to the reference tests and don't know what is being tested. This helps balance the need to keep reference tests hidden, with the need of programmers to know what fault is being highlighted in the heat map. This will require additional effort on the part of the instructor, as this is effectively combining the functionality of a hinting system with the heat map. The hints or prompts will need to be meaningful, but not give too much away to the student. An alteration we have made to the heat map plug-in for Web-CAT reports the name of the test case that is being reported in the heat map. If the instructor uses an appropriate level of granularity, as well as meaningful test names, this feature can achieve both outcomes without any additional effort on the instructor's part.

Fourth, providing intentionally chosen labeling for heat maps will help students understand the correct interpretation of the feedback and help preempt misconceptions about "the computer points out the bug". The heat maps as we have currently implemented them are simply displayed as the source code displayed in-line with the rest of the feedback provided by the auto-grader. We don't provide any prompts as to the interpretation of the heat map. We

believe providing specific wording will help clarify the use of the heat map as well as address some of the issues associated with the limitations of the underlying SFL analysis (see the text in subsection 5.6). Additionally, we have created a web page that links from the heat map feed back with a more detailed explanation of how the SFL analysis works, what it does and does not do, and examples for students who are interested in learning more about SFL.

7.2.4 Heat Maps Allow Students to Improve Their Projects More Quickly

After our somewhat disappointing experience in the classroom, we decided to take another look at the heat maps, this time from a more quantitative perspective. As we discussed in Chapter 6, we wanted to see if students still improved their performance when they had access to the heat maps, even though our surveys suggested that the students perceptions of the utility of the heat map were mixed at best. We found that students with the heat maps had more frequent incremental improvements, as well as achieving their highest correctness score faster, than students with out the heat maps. This raised an important question:

Why does having access to the heat maps promote more frequent incremental improvement?

Incremental improvement is a cornerstone of test-driven development[19], and of CS course designs that implement a test-first paradigm[13]. Previous research has shown that contextual feedback is important to a continuous improvement process[52][23]. The standard behavior of Web-CAT is to provide a textual hint outside the context of the student's code. The hint is tied to the reference test that failed, and by extension to the feature being tested, but not to any specific code segments. The student is not provided any type of direction to suggest where to look, other than knowing which test failed. The heat map feedback approach bridges this contextual gap, providing the feedback directly within the context of

the student's code.

As Figure 4.1 demonstrates, the heat map feedback approach allows students to focus only on the code that is known to be involved in the failed test cases. While it does not tell the student exactly where the defect is, it provides a focus for their ongoing efforts, without having code that "works" be a distraction. The heat map also provides an additional point of focus by providing information about the likelihood the code was involved in a defect test through the color gradient provided. The more red the line background, the more likely the line is involved in the defective test case. This provides a focus for the student of not just the feature being tested, but the specific lines involved. Additionally, as they make progress on passing the reference tests, the heat map will update after each submission, providing more and more focus for the remaining issues to be addressed. This results is more frequent improvements to the reference tests.

Given this focus, the students seem to spend less time debugging code that is known to work. The heat map shows them the portions of their code that are most defective. Fixes to the most defective code will result in an overall faster progress towards completion. Additionally, the heat map may be preventing regression defects, since it is focusing the student toward defective code and away from code that is not involved in a defect. Depending on the location of the defect, it is possible that multiple reference test defects can be addressed by changes in one area of suspicious code, also increasing the proportion of improvements, and reducing the number of regression defects.

7.2.5 Heat Maps Allow Students to Achieve Their Highest Correctness Scores Faster

Our review of the quantitative data for Web-CAT submissions for students who had access to the heat maps raises another interesting question:

Why does having access to the heat maps allow students to complete the reference tests faster in their projects?

The reasons for more frequent progress to completion also support decreasing overall time to completion. The heat maps provide increased focus for student efforts. The students make more progress toward successfully meeting the reference test requirements with each submission. Students making more consistent progress to addressing the reference test requirements will take less time overall, assuming they spend the same time on other aspects of the project, such as design and code style. Students will continue to take different amounts of absolute time, but the overall time will be reduced.

7.2.6 Threats to Validity

There are several threats to both internal and external validity. First, while we have reason to believe that the heat maps do help students improve their work more easily, and complete their work more quickly, we cannot say there is a direct connection between the heat maps and these outcomes. There are no direct measures that we have to insure that the heat map feedback is the only factor that has influenced the improvements analyzed in this study. To fully address this issue, we would have to provide the feedback to only half the students in CS 1114 in a given term. This is not practical, as it may provide one set of students an advantage over another set of students.

Another threat is the mapping of the projects from one semester to another. While this analysis suggests the projects were equal in complexity from the perspective of the students' submissions, and the number of submissions made and total time elapsed were not significantly different, two of the four project groups in each semester comparison were not identical. Also, there could be aspects of the assignments that influenced a set of students that were not present in another semester. The fact that the instructor was the same for both semesters would mitigate these factors, but not completely remove it.

The preparation of the students themselves may play a role in the differences. We have focused on presumably "novice" developers in our CS 1114 course. However, students bring a wide range of background skills with them to college. It is entirely possible that the students in the Fall 2017 cohort were better prepared coming into college than their 2015 counterparts. However, the data analysis of student submission behavior and total submission time indicated that there is no significant difference between the 2015 and 2017 student cohorts. Indeed, the replicative analysis between Fall 2015 and Spring 2019 produced similar results. Students, on average, submitted their assignments for each project group to Web-CAT the same number of times each semester, and took the same amount of time once they began submitting. If one group or the other were more prepared, we would expect that group to take less time and to require few submissions. This was not the case.

Although we collected submission data from all of the students in our CS 1114 course, the structure of the course itself may influence student performance unevenly. All students attend a lecture led by a single instructor, but they also attend a smaller-enrollment lab session that is led by a teaching assistant. The group of teaching assistants assigned to each term was different, and we do not have any metrics about those interactions. Also, we do not have a way to control for the interactions between students. We provide a CS undergraduate meeting center to encourage interactions between students, although each

student was required to provide their own submission. This peer interaction could have had an influence on performance, as well, but we do not have data to address this possibility.

Another area of possible concern is the complexity of the projects. This work focused on CS 1114. Later courses in our curriculum use more sophisticated projects, and while the SFL analysis module in Web-CAT will work with those more advanced projects (and in principle any Java project that can be assessed in Web-CAT), the GZoltar results have not been manually validated against exemplar projects from our *CS 2114: Software Design and Data Structures*, or *CS 3114: Data Structures and Algorithms* courses. Finally, we don't know how many of these students did not have previous experience with automated grading systems, or with programming in general. Learning the tool chain can impact the time taken for some students to submit, while others could have relied on previous experience to reduce their time.

Chapter 8

Future Work and Conclusions

8.1 Future Work

There are a number of avenues for further development of this work. We are looking at implementing the changes to address the lessons learned described in section 5.6. We have developed an updated version of the GZoltar evaluation engine that will "mask" the test results based on the "test depth" value specified by the instructor. Also, a more detailed description of the heat map has been added to the feedback output. Another enhancement to the heat map we have implemented is a way for the student to be able to mouseover and get a zoomed-in view of their marked-up source code. These changes are slated for deployment into Web-CAT. We have added a link to a page which explains the workings of the heat map. To this point, we have been focused on the instructor-provided reference tests. We want to also investigate integrating student tests into the heat map, using the heat map as a visualization of code coverage. Figure 8.1 shows a example of the updated interface.

Another area of improvement in the current tool would be to allow students to select the "mask" of the test case they wish to review if a number of tests were included in the analysis. Our original work tested all of the reference tests, but the feedback we received from students suggested that this often resulted in too much information in the heat map. The current iteration allows the instructor set a "test depth", where up to the n^{th} failed test information is shown in the heat map. With the redesign of the underlying heat map generator, it is

possible that separate heat map masks could be generated for each failed test, as well as a mask that is the sum of all failed tests. The student could be given a control to all them to select which test to review, which would update the heat map being displayed. This would not change the suspiciousness scores, since they are cumulative across all tests run in a single analysis session, but it would still allow the student know which lines were involved in specific failed tests.

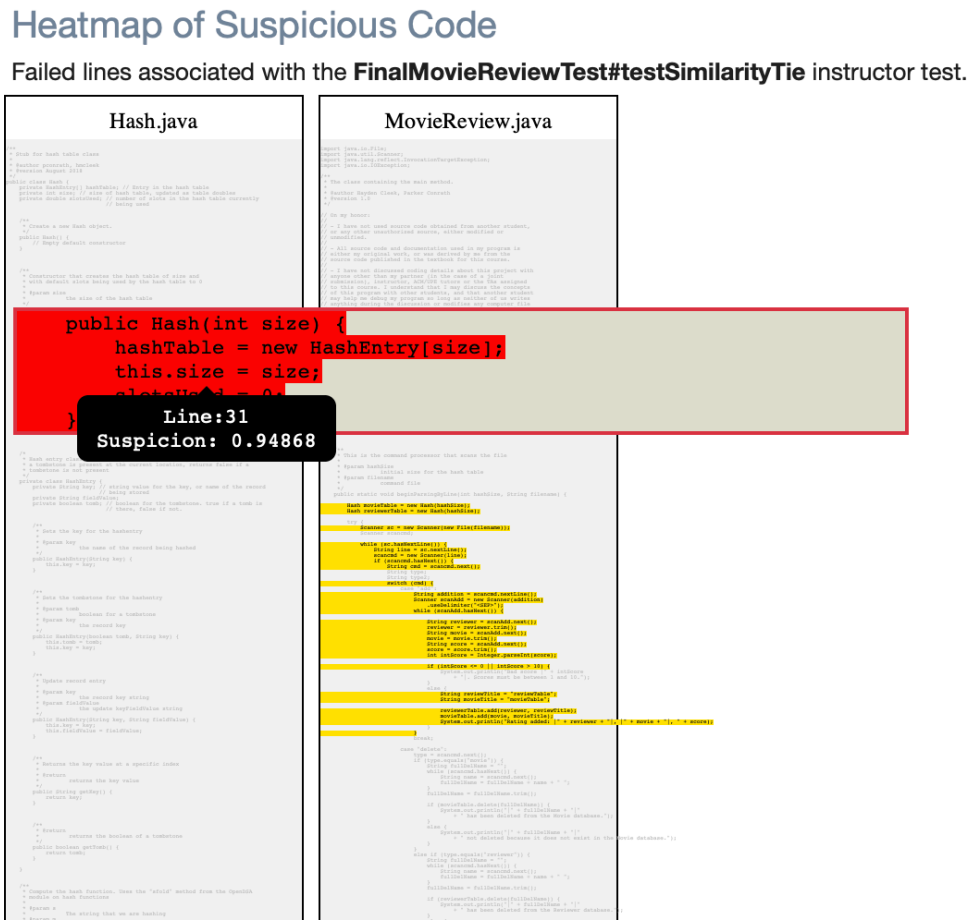


Figure 8.1: Example of the updated heat map interface, featuring a mouseover zoom-in feature, more distinct highlighting of the most suspicious methods, the name of the failed testcase, as well as line and suspiciousness score identification.

An additional area of future work is to provide more guidance to the students about what the heat map can and, more importantly, cannot do. The underlying SFL analysis has a

difficult time handling the "errors of omission" scenario. We believe there are some situations where we can create a visual in the heat map to guide the student when the SFL analysis has identified defects that have a "fingerprint":

- Defects of Omission, such as a missing `else` condition.
- "Off by one" errors in `while` and `for` loops.
- `switch`/`select` statements where multiple cases have similar suspiciousness.

This visual would provide the student with guidance appropriate to the situation. For example, a missing `else` condition might have a prompt as displayed in Figure 8.2. The location would use a different color that contrasted with the yellow-to-red spectrum of the suspiciousness scores. In this example, blue is used to identify the closing brace of the `if` condition with the missing `else`. In addition to providing the suspiciousness score for the line, we could add an additional prompt appropriate to the control block. It this case, we suggest the student review the `if..else` to make sure it is complete. We would need to be careful to make sure not to provide too much detail, such as "You are missing an `else` condition in your code", but that something that can be easily managed.

One of the first steps to enable work on defects of omission would be a study similar to that which started our work: Analyze a set of student submissions with GZOLTAR and then manually inspect the submissions, to determine how GZOLTAR handles applying suspiciousness scores to missing code. This analysis would have to answer similar questions, such as:

- How frequently does GZOLTAR identify the missing code as the most suspicious in a method?
- Does the missing code's suspiciousness always attach to the control syntax where it should appear (such as the closing branch of the `if` in the case of a missing `else`?

- What is the set of missing code that GZoltar will identify?

Once these questions are understood, it would be possible to craft changes to the heat map to account for these scenarios.

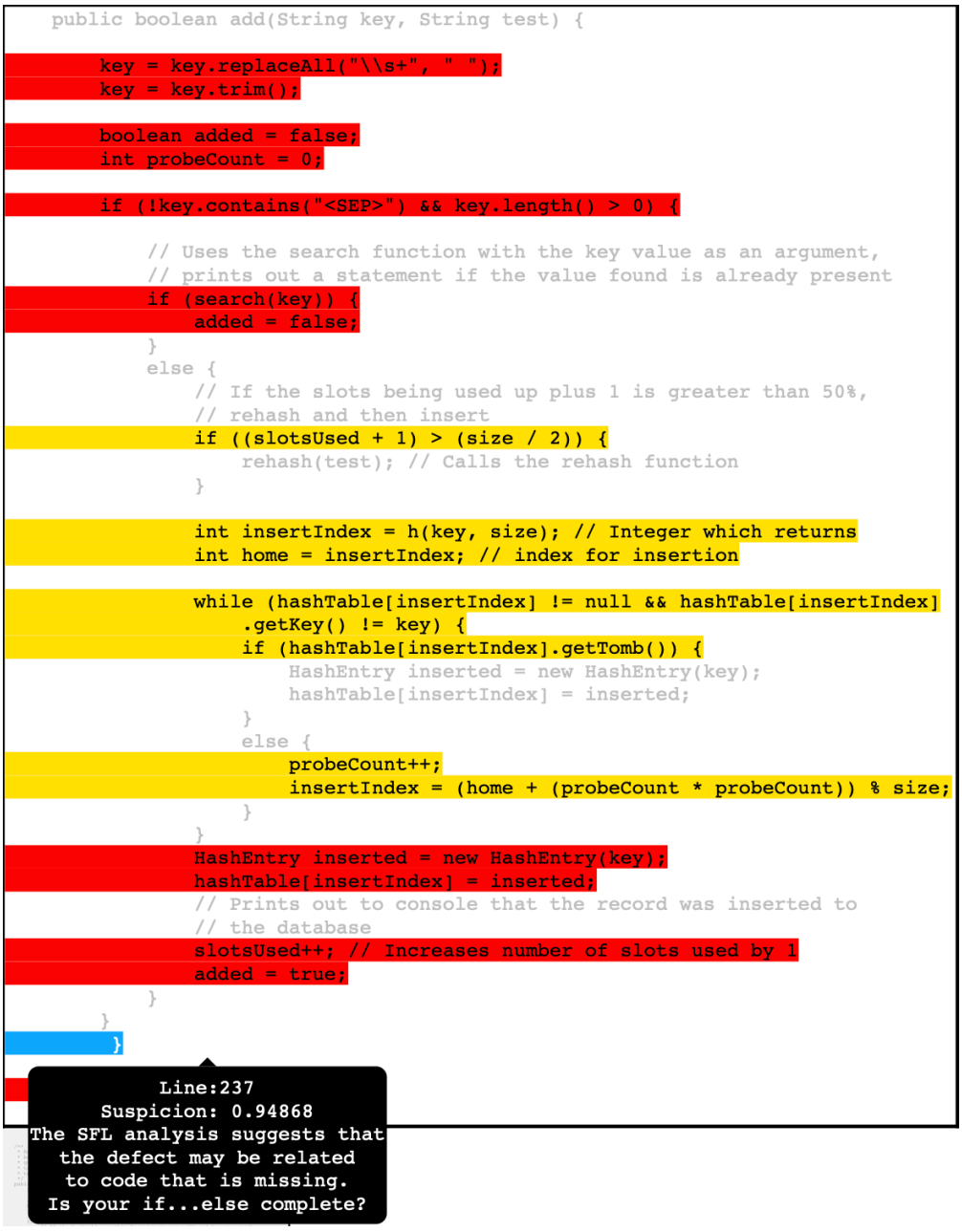


Figure 8.2: A mockup of one potential option for displaying prompt information for defects related to missing code.

There are additional user interface approaches that may also provide benefits to students. One of these approaches is similar to the revised interface we have developed but with additional functionality and detail. Furnas [42] developed the idea of the FISHEYE view, where items at the focal point of the students attention are shown in great detail and high "magnification", while items that are farther away are show in less detail at lower resolution. In both cases, the complete file is shown at all times. Since Furans' original proposal, significant work has been done to apply the FISHEYE approach to text editors used by programmers [57][58]. This approach could also be used for the heat maps, using scalable techniques [53] to determine the points of interest in the files, which in our case would be the suspicious code, and building the fisheye effect around those suspicious points of interest.

Another approach that might prove beneficial is one called "code bubbles" proposed by [22]. Each bubble represents a code fragment, such as a function or method. Additionally, bubbles that represent code that is associated via call sequences are visually connected in the interface. Their research into the efficacy of this interface model suggests that this is a faster means of navigating and completing tasks than traditional, file-based interface designs [21]. This model could be adapted as another way of showing the results of the SFL analysis to students. One possible way is shown in Figure 8.3.

In this case, the student is presented a list of methods from their code, with suspicious methods indicated by colors derived from the original heat map display. The student then selects a suspicious method, in this case `main()`, and the code for main, as well as the code for methods that have suspicious methods that are called by main are displayed, here `beginParsingByLine()`. Information about "fingerprint" defects such as errors of omission can also be called out as needed. A similar study, as outlined above, for determining how missing code is treated by GZOLTAR, would be needed for this work as well.

The code bubble model would also have a potential negative outcome. It would trade the

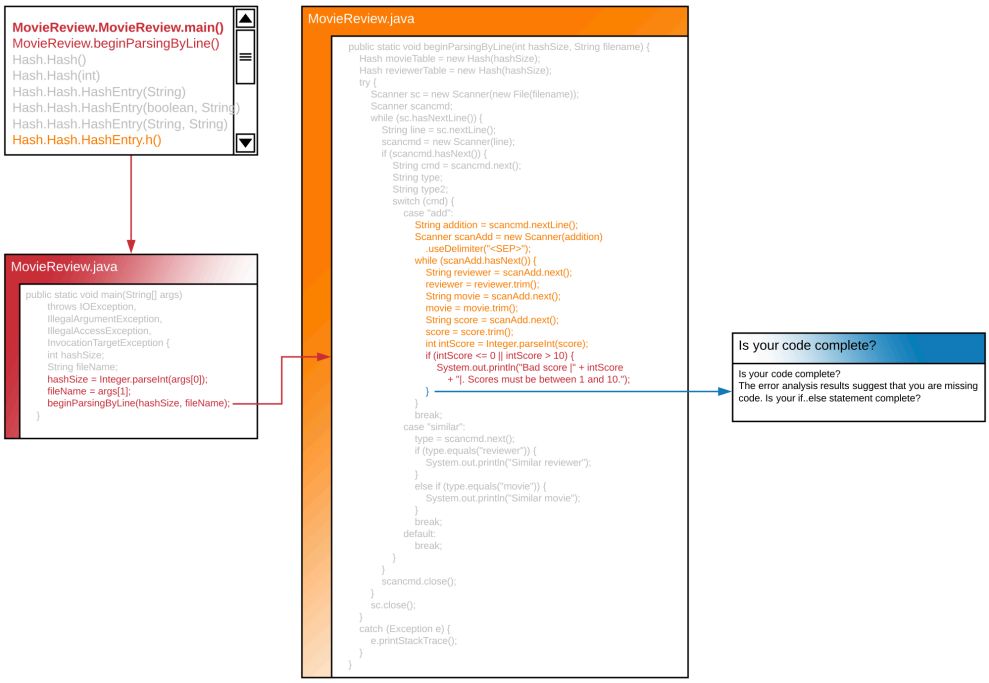


Figure 8.3: A prototype for melding the heat map feedback to the Code Bubble concept for reporting the SFL results to students.

focus on all of the defects in one place that the current heat map model provides with the more narrowed focus and additional required navigation that would be required for the code bubbles. The additionally navigation requirements could possibly result in students not seeing all of the feedback because they would have to click through the interface to find the feedback that as being provided. However, the code bubble model might be appropriate for more complex assignments for more advanced students, where the number of modules is such that presenting everything to the student at once overloads them with information.

Finally, we plan to use the heat map in our CS 3114 data structures and algorithms courses. We have questions about whether CS 1114/CS 2114 students may not be the best audience for this type of feedback system. Is it possible that we are assuming a certain level of skill to use this tool that CS 1114/CS 2114 students don't yet possess when we present this feedback style to them? By offering the heat maps to more advanced students, we want to see if they

are better equipped to benefit from them, or if they have similar issues as the CS 1114 students.

8.2 Conclusion

When we started looking at this area, the goal was to determine if we could design a system using automated fault localization tools to guide and assist students in their debugging efforts, pointing them in the direction they should look to resolve their defects without going so far as to specify the exact location of the defect. We believe the answer to this is a definitive "Yes".

This work has shown that spectrum-based fault location is a feasible approach for generating automated feedback to students from their their own source code, based on the execution of instructor-provided reference tests. When using the GZOLTAR library to implement SFL for Java-based student projects, we are able to identify the actual location of the defect in the top three methods identified by this approach 90% of the time. In addition, we are able to completely automate this process, generating this analysis and feedback without requiring any additional instructor time or planning. Further, the analysis is driven by the student's own code structure and does not suffer from some of the "error guessing" problems that instructor-written hints may experience.

Also, we have developed a plugin for Web-CAT that automatically apply the GZOLTAR analysis to student submissions. As part of this plugin application, we created visualizations of the results of the SFL output, and overlaid those results using a heat map motif on the student's source code submissions. Finally we provided this feedback technique to a set of CS 1114 students. This results in software with many small methods, where method-oriented debugging suggestions are likely to be effective. Students where provided the heat

map feedback on their own assignments, and then asked to evaluate the technique through a survey.

Based on the responses from the students, there were several lessons learned. A heat map with too many lines highlighted is not helpful to students, and actually may hinder them. A masking for the heat map, such as the "first-failed" approach, will show the suspiciousness scores calculated for the full evaluation of all of the test cases, but only highlight the lines for the "first-failed" test. As part of this, we also believe we should show the goal or feature that is being tested which is used for the filter. Finally, we learned that the heat map itself is not as obvious in its usage as we expected. We need to provide additional directional language to the students for them to grasp the meaning of the heat map. We have made modifications to the Web-CAT plugin to implement this feedback from students.

Additionally, we analyzed student performance when the heat maps were available to students in two semesters, versus when they were not. We found that students who had access to the heat maps were able to make more frequent incremental improvements on their reference test completion. Also, we found that students who had access to the heat maps were able to reach their highest correctness score faster on three out of the four projects studied, when compared to students that did not have access to the heat maps. This suggests that although there was mixed feelings from students about the utility of the heat maps, students do benefit from the contextual feedback the heat maps provide. Because of concerns about the significance of the initial study results, we repeated the analysis with an additional semester of student submissions. This produced results that were similar to the first.

Taken together, we believe that this approach to providing contextualized feedback to students is both useful and promising. With the growth in demand for computer science courses showing no signs of diminishing, automated tools can be used to "take the pressure off" instructors to some extent. However automating context is hard, and this tool shows that it

is possible to provide automated contextual feedback to show students where to look when they are debugging rather than just telling them exactly where the error occurred. There are additional avenues of exploration for this work, but we believe this is solid beginning.

Bibliography

[1] JUnit, accessed 2018-12-19. URL https://junit.org/.

[2] Apache Ant Build System, accessed 2019-01-04. URL https://ant.apache.org/.

[3] Blackboard Learning Management System, accessed 2019-01-04. URL https://www.blackboard.com/index.html.

[4] IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains, accessed 2019-03-11. URL https://www.jetbrains.com/idea/.

[5] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. Mutation Analysis vs. Code Coverage in Automated Assessment of Students' Testing Skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 153–160, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1. doi: 10.1145/1869542.1869567. URL http://doi.acm.org/10.1145/1869542.1869567.

[6] ABET. ABET Computing Accreditation Committee 2018-2019 Criteria Version 2.0. Technical report, ABET, Baltimore, MD, October 2017. URL https://www.abet.org/wp-content/uploads/2018/02/C001-18-19-CAC-Criteria-Version-2.0-updated-02-12-18.pdf.

[7] Rui Abreu, Peter Zoeteweij, and Arjan. J. C. van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46, December 2006. doi: 10.1109/PRDC.2006.18.

[8] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82 (11):1780–1792, November 2009. doi: 10.1016/j.jss.2009.06.035.

[9] ACM Computing Curricula Task Force, editor. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, Inc, January 2013. ISBN 978-1-4503-2309-3. doi: 10.1145/2534860. URL http://dl.acm.org/citation.cfm?id=2534860.

[10] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. An Analysis of Patterns of Debugging Among Novice Computer Science Students. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 84–88, New York, NY, USA, 2005. ACM. ISBN 978-1-59593-024-8. doi: 10.1145/1067445.1067472. URL http://doi.acm.org/10.1145/1067445.1067472.

[11] Vincent Aleven, Ido Roll, Bruce M. McLaren, and Kenneth R. Koedinger. Help Helps, But Only So Much: Research on Help Seeking with Intelligent Tutoring Systems. *International Journal of Artificial Intelligence in Education*, 26(1):205–223, March 2016. ISSN 1560-4306. doi: 10.1007/s40593-015-0089-1. URL https://doi.org/10.1007/s40593-015-0089-1.

[12] Richard C. Anderson, Raymond W. Kulhavy, and Thomas Andre. Feedback procedures in programmed instruction. *Journal of Educational Psychology*, 62(2):148–156, April 1971. doi: 10.1037/h0030766.

[13] Anonimyzed. Anonimyzed title. *J. Educ. Resour. Comput.*, 3(3), September 2003. ISSN 1531-4278.

[14] Anonimyzed. Anonimyzed title. *Educational Data Mining 2008*, 2008.

[15] Dave Astels. *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003. ISBN 978-0-13-101649-1.

[16] Catherine M. Baker, Cynthia L. Bennett, and Richard E. Ladner. Educational Experiences of Blind Programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 759–765, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3287410. URL http://doi.acm.org/10.1145/3287324.3287410. event-place: Minneapolis, MN, USA.

[17] David J. Barnes and Michael Kolling. *Objects First With Java: A Practical Introduction Using BlueJ (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 978-0-13-197629-0.

[18] Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, October 1999. ISSN 0018-9162. doi: 10.1109/2.796139.

[19] Kent Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003. ISBN 978-0-321-14653-3. Google-Books-ID: CUlsAQAAQBAJ.

[20] Barry W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1), 1984. doi: DOI:10.1109/MS.1984.233702.

[21] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806866. URL

http://doi.acm.org/10.1145/1806799.1806866. event-place: Cape Town, South Africa.

[22] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2503–2512, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-929-9. doi: 10.1145/1753326.1753706. URL http://doi.acm.org/10.1145/1753326.1753706. event-place: Atlanta, Georgia, USA.

[23] Kevin Buffardi and Stephen H. Edwards. A Formative Study of Influences on Student Testing Behaviors. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 597–602, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2605-6. doi: 10.1145/2538862.2538982. URL http://doi.acm.org/10.1145/2538862.2538982. event-place: Atlanta, Georgia, USA.

[24] Andrew C. Butler, Jeffrey D. Karpicke, and Henry L. Roediger III. Correcting a metacognitive error: Feedback increases retention of low-confidence correct responses. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 34(4):918–928, 2008. ISSN 1939-1285(Electronic),0278-7393(Print). doi: 10.1037/0278-7393.34.4.918.

[25] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. GZoltar: An Eclipse Plug-in for Testing and Debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 378–381, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1204-2. doi: 10.1145/2351676.2351752. URL http://doi.acm.org/10.1145/2351676.2351752.

[26] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: problem determination in large, dynamic Internet services. In *International Conference on Dependable Systems and Networks*, page 595, 2002. URL https://www.computer.org/csdl/proceedings/dsn/2002/1597/00/15970595.pdf.

[27] Mohammad. Y. Chuttur. Overview of the technology acceptance model: Origins, developments and future directions. *Sprouts Working Papers on Information Systems*, 9(37), 2009.

[28] Cyril W. Cleverdon. On the Inverse Relationship of Recall and Precision. *Journal of Documentation*, 28(3):195–201, 1972. doi: http://dx.doi.org/10.1108/eb026538.

[29] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[30] Simon P. Davies. Models and theories of programming strategy. *International Journal of Man-Machine Studies*, 39(2):237–267, August 1993. ISSN 0020-7373. doi: 10.1006/imms.1993.1061. URL http://www.sciencedirect.com/science/article/pii/S0020737383710618.

[31] Fred. D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):319–340, September 1989.

[32] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical Slicing for Software Fault Localization. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 121–134, New York, NY, USA, 1996. ACM. ISBN 978-0-89791-787-2. doi: 10.1145/229000.226310. URL http://doi.acm.org/10.1145/229000.226310. event-place: San Diego, California, USA.

[33] Martin Dougiamas and Peter Taylor. Moodle: Using Learning Communities to Create

an Open Source Course Management System. *Proceedings, Edmedia 2003*, page 14, 2003.

[34] Benedict du Boulay. Some Difficulties of Learning to Program. *Studying the Novice Programmer*, pages 286–300, 1988. doi: 10.4324/9781315808321-5.

[35] Bob Edmison and Stephen H. Edwards. Applying spectrum-based fault localization to generate debugging suggestions for student programmers. In *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–99. ©2015 IEEE, reprinted by permission., November 2015. doi: 10.1109/ISSREW.2015.7392052.

[36] Bob Edmison and Stephen H. Edwards. Experiences Using Heat Maps to Help Students Find Their Bugs: Problems and Solutions. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 260–266, Minneapolis, MN, February 2019. ACM. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3287474. URL http://dl.acm.org/citation.cfm?id=3287324.3287474.

[37] Bob Edmison, Stephen H. Edwards, and Manuel A. Pérez-Quiñones. Using Spectrum-Based Fault Location and Heatmaps to Express Debugging Suggestions to Student Programmers. In *Proceedings of the Nineteenth Australasian Computing Education Conference*, ACE '17, pages 48–54, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4823-2. doi: 10.1145/3013499.3013509. URL http://doi.acm.org/10.1145/3013499.3013509.

[38] Stephen H. Edwards. Using Software Testing to Move Students from Trial-and-error to Reflection-in-action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 26–30, New York, NY, USA, 2004.

ACM. ISBN 978-1-58113-798-9. doi: 10.1145/971300.971312. URL http://doi.acm.org/10.1145/971300.971312.

[39] Stephen H. Edwards and Manuel A. Pérez-Quiñones. Web-CAT: Automatically Grading Programming Assignments. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, pages 328–328, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-078-4. doi: 10.1145/1384271.1384371. URL http://doi.acm.org/10.1145/1384271.1384371.

[40] Stephen H. Edwards and Zalia Shams. Do Student Programmers All Tend to Write the Same Software Tests? In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, pages 171–176, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2833-3. doi: 10.1145/2591708.2591757. URL http://doi.acm.org/10.1145/2591708.2591757.

[41] Eclipse Foundation. The Eclipse Integrated Development Environment, January 2019. URL https://www.eclipse.org/ide/.

[42] George W Furnas. The FISHEYE view: a new look at structured files. In Stuart K. Card, J. Mackinlay, and Ben Shneiderman, editors, *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann Publishers, 1981.

[43] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. An Interactive Functional Programming Tutor. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 250–255, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1246-2. doi: 10.1145/2325296.2325356. URL http://doi.acm.org/10.1145/2325296.2325356.

[44] Fischer Gerhard and Scharff Eric. Learning Technologies in Support of Self-Directed Learning. *Journal of Interactive Media in Education*, 98:98–94, 1998.

[45] David A. Gilman. Comparison of several feedback methods for correcting errors by computer-assisted instruction. *Journal of Educational Psychology*, 60:503–508, 1968. URL https://psycnet.apa.org/fulltext/1970-04255-001.pdf.

[46] David J. Gilmore. Models of debugging. *Acta Psychologica*, 78(1):151–172, December 1991. ISSN 0001-6918. doi: 10.1016/0001-6918(91)90009-O. URL http://www.sciencedirect.com/science/article/pii/000169189190009O.

[47] Carlos Gouveia, José Campos, and Rui Abreu. Using HTML5 visualizations in software fault localization. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–10, September 2013. doi: 10.1109/VISSOFT.2013.6650539.

[48] Thomas RG Green and Marian Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, June 1996.

[49] Leo Gugerty and Gary Olson. Debugging by Skilled and Novice Programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '86, pages 171–174, New York, NY, USA, 1986. ACM. ISBN 978-0-89791-180-1. doi: 10.1145/22627.22367. URL http://doi.acm.org/10.1145/22627.22367.

[50] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An Efficient Relevant Slicing Method for Debugging. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, Lecture Notes in Computer Science, pages 303–321. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-48166-9.

[51] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Journal of Software Testing, Verification and Reliability*, 10(3):171–194, September 2000. doi: 10.1002/1099-1689(200009)10:3<171::AID-STVR209>3.0.CO;2-J.

[52] John Hattie and Helen Timperley. The Power of Feedback. *Review of Educational Research*, 77(1):81–112, March 2007. doi: 10.3102/003465430298487.

[53] Jeffrey Heer and Stuart K. Card. Efficient User Interest Estimation in Fisheye Views. In *Conference on Human Factors in Computing Systems: CHI'03 extended abstracts on Human factors in computing systems*, volume 5, pages 836–837, 2003.

[54] Instructure. Learning Management System | LMS | Canvas by Instructure, December 2018. URL https://www.canvaslms.com/.

[55] Michael S. Irwin and Stephen H. Edwards. Can Mobile Gaming Psychology Be Used to Improve Time Management on Programming Assignments? In *Proceedings of the ACM Conference on Global Computing Education*, CompEd '19, pages 208–214, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6259-7. doi: 10.1145/3300115.3309517. URL http://doi.acm.org/10.1145/3300115.3309517. event-place: Chengdu,Sichuan, China.

[56] David Jackson and Michelle Usher. Grading Student Programs Using ASSYST. In *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '97, pages 335–339, New York, NY, USA, 1997. ACM. ISBN 978-0-89791-889-3. doi: 10.1145/268084.268210. URL http://doi.acm.org/10.1145/268084.268210. event-place: San Jose, California, USA.

[57] Mikkel R. Jakobsen and Kasper Hornbæk. Evaluating a Fisheye View of Source Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 377–386, New York, NY, USA, 2006. ACM. ISBN 978-1-59593-372-0. doi: 10.1145/1124772.1124830. URL http://doi.acm.org/10.1145/1124772.1124830. event-place: Montréal, Québec, Canada.

[58] Mikkel Rønne Jakobsen and Kasper Hornbæk. Fisheyes in the Field: Using Method Triangulation to Study the Adoption and Use of a Source Code Visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1579–1588, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-246-7. doi: 10.1145/1518701.1518943. URL http://doi.acm.org/10.1145/1518701.1518943. event-place: Boston, MA, USA.

[59] George H. Joblove and Donald Greenberg. Color Spaces for Computer Graphics. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78, pages 20–25, New York, NY, USA, 1978. ACM. doi: 10.1145/800248.807362. URL http://doi.acm.org/10.1145/800248.807362.

[60] James A. Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM. ISBN 978-1-58113-993-8. doi: 10.1145/1101908.1101949. URL http://doi.acm.org/10.1145/1101908.1101949.

[61] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization for Fault Localization. In *Proceedings of ICSE 2001 Workshop on Software Visualization*, pages 71–75, Toronto, CA, November 2001.

[62] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477, May 2002. doi: 10.1145/581396.581397.

[63] Andrew J. Ko and Brad A. Myers. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference*

on Human Factors in Computing Systems, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM. ISBN 978-1-58113-702-6. doi: 10.1145/985692.985712. URL http://doi.acm.org/10.1145/985692.985712. event-place: Vienna, Austria.

[64] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988. ISSN 0020-0190. doi: 10.1016/0020-0190(88)90054-3. URL http://www.sciencedirect.com/science/article/pii/0020019088900543.

[65] Raymond W. Kulhavy. Feedback in Written Instruction. *Review of Educational Research*, 47(2):211–232, June 1977. ISSN 0034-6543. doi: 10.3102/00346543047002211. URL https://doi.org/10.3102/00346543047002211.

[66] Raymond W. Kulhavy and Richard C. Anderson. Delay-retention effect with multiple-choice tests. *Journal of Educational Psychology*, 63(5):505–512, 1972. ISSN 1939-2176(Electronic),0022-0663(Print). doi: 10.1037/h0033243. URL https://psycnet.apa.org/fulltext/1973-03232-001.pdf.

[67] Jean Lave and Etienne Wenger. *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press, September 1991. ISBN 978-0-521-42374-8. Google-Books-ID: CAVIOrW3vYAC.

[68] Timotej Lazar and Ivan Bratko. Data-Driven Program Synthesis for Hint Generation in Programming Tutors. In Stefan Trausan-Matu, Kristy Elizabeth Boyer, Martha Crosby, and Kitty Panourgia, editors, *Intelligent Tutoring Systems*, Lecture Notes in Computer Science, pages 306–311, Honolulu HI, June 2014. Springer International Publishing. ISBN 978-3-319-07221-0. doi: https://doi.org/10.1007/978-3-319-07221-0_38.

[69] Nguyen-Thinh Le and Wolfgang Menzel. Using Weighted Constraints to Diagnose Errors in Logic Programming–The Case of an Ill-defined Domain. *International Journal of Artificial …*, 2009.

[70] Clayton Lewis. Attitudes and beliefs about computer science among students and faculty. *ACM SIGCSE Bulletin*, 39(2):37–41, June 2007. doi: 10.1145/1272848.1272880.

[71] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM. ISBN 978-1-59593-056-9. doi: 10.1145/1065010.1065014. URL http://doi.acm.org/10.1145/1065010.1065014. event-place: Chicago, IL, USA.

[72] Michael H. Long. Linguistic and Conversational Adjustments to Non-Native Speakers. *Studies in second language acquisition*, 5(2):177–193, April 1983. doi: 10.1017/S0272263100004848.

[73] Jonas Löwgren and Erik Stolterman. *Thoughtful Interaction Design: A Design Perspective on Information Technology*. MIT Press, 2004. ISBN 978-0-262-12271-9. Google-Books-ID: g573kw36QMUC.

[74] Alison Mackey. Feedback, noticing and instructed second language learning. *Applied linguistics*, 27(3):405–430, 2006. doi: 10.1093/applin/ami051.

[75] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.1976.233837.

[76] Renée McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: a review of the literature from an

educational perspective. *Computer Science Education*, 18(2):67–92, June 2008. doi: 10.1080/08993400802114581.

[77] Sudhir Mehta and Nem W. Schlecht. Computerized assessment technique for large classes. *Journal of Engineering Education*, 87(2):167–172, April 1998.

[78] Susan Meyer. Report on the initial test of a junior high school vocabulary program. *Teaching machines and programmed learning*, pages 229–246, 1960.

[79] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing, 3rd Ed.* John Wiley & Sons, September 2011. ISBN 978-1-118-13315-6. Google-Books-ID: GjyEFPkMCwcC.

[80] Chris Parnin and Alessandro Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001445. URL http://doi.acm.org/10.1145/2001420.2001445.

[81] Roy D. Pea. Language-Independent Conceptual "Bugs" in Novice Programming:. *Journal of Educational Computing Research*, 2(1):25–36, 1986. doi: 10.2190/689T-1R2A-X4W4-29J2.

[82] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A Survey of Literature on the Teaching of Introductory Programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '07, pages 204–223, New York, NY, USA, 2007. ACM. doi: 10.1145/1345443.1345441. URL http://doi.acm.org/10.1145/1345443.1345441.

[83] Zalia Shams and Stephen H. Edwards. Toward Practical Mutation Analysis for Evaluating the Quality of Student-written Software Tests. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 53–58, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2243-0. doi: 10.1145/2493394.2493402. URL http://doi.acm.org/10.1145/2493394.2493402.

[84] B. F. Skinner. *Science and human behavior*. Science and human behavior. Macmillan, Oxford, England, 1953.

[85] Troy A. Smith. Learning from feedback: Spacing and the delay–retention effect. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 36(1), January 2010. doi: 10.1037/a0017407. URL /fulltext/2009-24668-022.html.

[86] James G. Spohrer and Elliot Soloway. Analyzing the High Frequency Bugs in Novice Programs. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, pages 230–251, Norwood, NJ, USA, 1986. Ablex Publishing Corp. ISBN 978-0-89391-388-5. URL http://dl.acm.org/citation.cfm?id=21842.28897.

[87] Strategic Planning and Economic Analysis Group. The economic impacts of inadequate infrastructure for software testing. Final Report Planning Report 02-3, National Institute of Standards and Technology., Gaithersburg, MD, May 2002. URL https://www.nist.gov/document/report02-3pdf.

[88] Lucy Suchman. Making work visible. *Communications of the ACM*, 38(9):56–64, September 1995. doi: 10.1145/223248.223263.

[89] Lucy Suchman. *Human-machine reconfigurations: Plans and situated actions*. Cambridge University Press, New York, NY, 2nd edition, 2007.

[90] Shinji Uchida, Akito Monden, Hajimu Iida, Ken-ichi Matsumoto, and Hideo Kudo. A multiple-view analysis model of debugging processes. In *Proceedings International Symposium on Empirical Software Engineering*, pages 139–147, October 2002. doi: 10.1109/ISESE.2002.1166933.

[91] Anneliese von Mayrhauser and A. Marie Vans. Program understanding behavior during debugging of large scale software. In *Papers presented at the seventh workshop on Empirical studies of programmers - ESP '97*, pages 157–179, Alexandria, Virginia, United States, 1997. ACM Press. ISBN 978-0-89791-992-0. doi: 10.1145/266399.266414. URL http://portal.acm.org/citation.cfm?doid=266399.266414.

[92] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 1–11, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771797. URL http://doi.acm.org/10.1145/2771783.2771797. event-place: Baltimore, MD, USA.

[93] Shu-Ling Wang and Pei-Yi Wu. The role of feedback and self-efficacy on web-based learning: The social cognitive perspective. *Computers & Education*, 51(4):1589–1598, December 2008. doi: 10.1016/j.compedu.2008.03.004.

[94] Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 978-0-89791-146-7. URL http://dl.acm.org/citation.cfm?id=800078.802557. event-place: San Diego, California, USA.

[95] Mark Weiser. Programmers Use Slices when Debugging. *Commun. ACM*, 25(7):446–

452, July 1982. ISSN 0001-0782. doi: 10.1145/358557.358577. URL http://doi.acm.org/10.1145/358557.358577.

[96] Leland Wilkinson and Michael Friendly. The History of the Cluster Heat Map. *The American Statistician*, 63(2):179–184, May 2009. ISSN 0003-1305. doi: 10.1198/tas.2009.0033. URL https://doi.org/10.1198/tas.2009.0033.

[97] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. Effective Fault Localization using Code Coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 449–456, July 2007. doi: 10.1109/COMPSAC.2007.109.

[98] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, August 2016. ISSN 0098-5589. doi: 10.1109/TSE.2016.2521368.

[99] Andreas Zeller. Yesterday, My Program Worked. Today, It Does Not. Why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 253–267, London, UK, UK, 1999. Springer-Verlag. ISBN 978-3-540-66538-0. URL http://dl.acm.org/citation.cfm?id=318773.318946. event-place: Toulouse, France.

[100] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 319–329, May 2003. doi: 10.1109/ICSE.2003.1201211.

[101] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning Dynamic Slices with Confidence. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 169–180, New York,

NY, USA, 2006. ACM. ISBN 978-1-59593-320-1. doi: 10.1145/1133981.1134002. URL http://doi.acm.org/10.1145/1133981.1134002. event-place: Ottawa, Ontario, Canada.

Appendices

Appendix A

Project Specifications

A.1 Project Group 1—Fields and Parameters

Students were asked in this project to create a character, a "jeroo" that was able to navigate a maze, collecting flowers as it travelled, and disabling any nets it encountered. The goal of the project was to encourage students to think about program structure and about the fields and parameters they would need to pass between method invocations to meet the goals outlined in the project specification.

The same project specification was used in all three semesters studied, Fall 2015, Fall 2017 and Spring 2019. This representation is from the Fall 2017 semester when we had transitioned to the Canvas LMS.

A.1.1 All Three Studied Semesters—Walking a Maze

Program 2

Walking a Maze

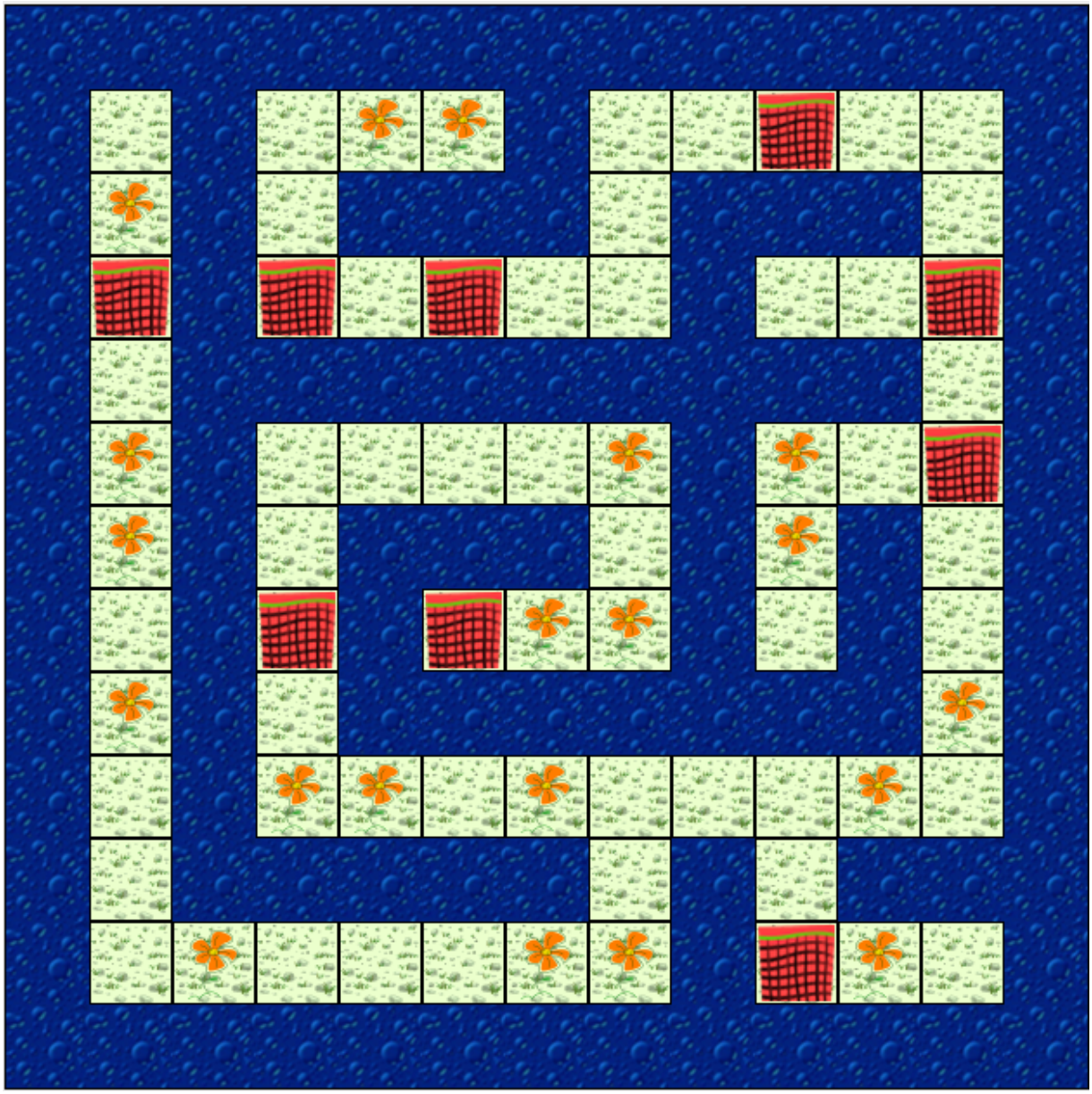
- It is an Honor Code violation to resubmit prior work from an earlier semester in this course.
- It is an Honor Code violation to submit work authored by someone else as your own.

For this (and all future) assignments, you **must** include the following Virginia Tech Honor Code pledge at the very top of each Java file you write (separate from the class Javadoc comment). Include your name and PID as shown, to serve as your signature:

```
// Virginia Tech Honor Code Pledge:
//
// As a Hokie, I will conduct myself with honor and integrity at all times.
// I will not lie, cheat, or steal, nor will I accept the actions of those
// who do.
// -- Your Name (pid)
```

Goal

In your second program assignment, your task is to create a jeroo smart enough to navigate a simple maze, picking all of the flowers and disabling all of the nets:



The **starting position** for your jeroo will always be the southeast corner of the island at (*width* - 2, *height* - 2).

After running the maze, the jeroo should always end at the **ending position**, which is at location (1, 1).

The catch is that Maze Island changes every time. Sometimes it is wider, sometimes narrower--the only constant is that it is rectangular and surrounded by water. Flowers can grow anywhere and everywhere on the island. Also, there may be **up to fifteen nets** located on the island, although there will never be a net at the starting position.

Further, you will have to **write software tests to demonstrate your jeroo can handle this task**.

How to Walk a Maze

While there are many strategies for navigating through a maze, one easy strategy is to simply hug the righthand wall as you walk through the maze--you'll eventually make it all the way through the maze, back to your starting position (hugging the lefthand wall also works the same way, as long as you

always stick to one side). If you traverse the maze completely, ending back at the starting location you are then ensured that all the flowers and nets have been cleared. Then your Jeroo simply needs to hop to the northwest corner. You can use this strategy to create a simple maze walker (or runner) with just a few lines of code. Other strategies are possible, of course, so use the one you find easiest to understand.

Structuring Your Solution

Download the scenario for this assignment, which contains the `MazeIsland` class: **program2.zip**.

In the `program2` scenario, **you do not need to create any island subclasses**. Instead, create your own subclass of `Jeroo` called `MazeRunner`, and structure all of your solution code in it. Give your `MazeRunner` a `myProgram()` method that will completely clear the island of all flowers and nets, and then position the jeroo at the ending position. You will need to right-click on your MazeRunner class (in the Actor classes pane) and select new MazeRunner() and then place it at the southeast corner in the Maze. Although there is no island subclass for program 2, your MazeRunner `myProgram()` method will be automatically executed when you press the **>Run** button.

Your `MazeRunner` class should provide a constructor that takes no parameters and that creates the jeroo holding 15 flowers (a parameterless constructor is called a *default constructor*). You can choose to provide additional constructors if you wish. Constructor methods are **not** inherited by subclasses. However subclass constructors can invoke super class constructors using the `super()` keyword. So your `MazeRunner` default, i.e., parameterless, constructor can invoke the Jeroo super constructor `Jeroo(int flowers)`.

Remember guidelines for breaking problems up into methods. Including methods that are too long, or methods with poorly chosen names, will lower your score. You will be graded in part on the design and readability of your solution using the posted grading criteria, so consider these factors when devising and naming your methods. The **Program Grading Rubric (http://moodle.cs.vt.edu/mod/page/view.php?id=35574)** (same as on Program 1) describes the grading criteria.

Testing Your Solution

You are expected to write software tests to confirm your solution works the way you intend for this assignment. Write a test class for your jeroo called `MazeRunnerTest`. Ensure that your test class checks all of the methods you have written.

We **strongly** recommend that you write your tests as you write each method in your solution, since it will help you find problems faster. Saving testing until the end usually results in extra time spent trying to find bugs long after they were written.

Keep Your Solution Simple

When approaching a problem like this, many students start out writing a basic piece of code they hope will run, and then try to "improve" it piece by piece--adding if statements and actions here and there in order to handle the situations they notice as they run their solution on a few islands. However, be aware that **this approach may lead to a poor solution**. Often, such students end up with code that involves a large number of nested if statements with complicated conditions intended to handle specific situations that the student saw, or with multiple loops that are intended for specialized behaviors on maps with certain kinds of pathways. Please **do not do this**. You'll end up with a complicated (and probably buggy) mess.

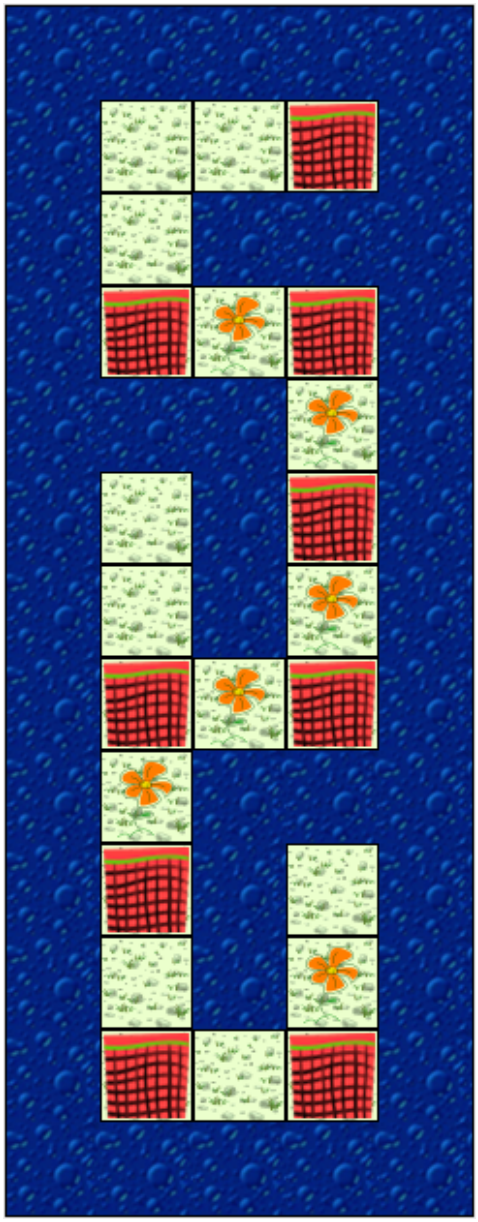
In truth, the minimal solution to this assignment is not large. You do not need more than one if statement with a simple condition to handle flowers, or more than one if statement with a simple condition to handle nets. Navigation around the island can be done with as few as two if statements, and no boolean AND or OR operators are needed. You can also create a solution with one (or perhaps two) loops.

If, instead, you find yourself with very long methods (more than a handful of lines) involving many if statements (or many else-if branches) that use complicated conditions with lots of &&'s and ||'s scattered around, then that is a warning sign that your solution is difficult to read, difficult to understand, and poorly planned.

If you find yourself in that situation, it is better to step back and try to think more systematically about all the possible situations the Jeroo needs to appear in (for navigation, for example, there are only 3 distinct situations you need to consider). Try to plan to handle the distinct situations, instead of just "adding logic" to a solution that sometimes fails.

Expectations for Testing

For easier testing, the `MazeIsland` class provides a second constructor that takes a "code number" (called a `long` number) that identifies a specific island. For example, the island numbered **75** looks like this:



For writing your tests, you can pick your own number (or numbers) and use it to create exactly the same island layout over and over. This will give you a fixed, predictable situation for testing testing individual pieces of your solution.

Expectations for your test class are as follows:

1. For each method in your solution that has *no control statements* (i.e., no if statements, loops, etc.), you should write one test method to confirm that method works as intended.
2. For each loop contained in any method, you should write two separate test methods: one that exercises the method in a situation where the loop is completely skipped, and one that exercises it in a situation where the loop repeats more than once.
3. For each if statement in any method, you should write a separate test method for *each branch* (for each if, each else-if, and each else) that exercises the method in a situation where that branch is executed.

4. If your logic or your control statements are particularly complicated, you might need to ask for help about what else you should test.

You can easily write software tests that follow this pattern:

- Create an island using the code number you have selected (so it is the same every time)
- Add your jeroo at a specific location of your choosing on that island, in order to set up the situation needed for the method/condition you are trying to test
- Call the method you intend to test
- Confirm that the desired behavior was achieved

Note that a portion of your grade will be based on your approach to testing your solution.

Submitting Your Solution

All program assignments are submitted to Web-CAT. Use the Controls->Submit... menu command to submit your work. Don't forget to include the Honor Code pledge in your source file(s).

Web-CAT Submission Energy

The submission energy bar is a new feature you will use on Web-CAT for this assignment. This section helps outline some of the questions you may have, as well as some of the background with the energy bar. If you have additional questions, feel free to post them on Piazza.

How does it work?

For each assignment, you have an *energy bar* that holds up to 3 units of submission energy. Each time you submit to Web-CAT, one unit of energy is used. Your bar will then regenerate energy at a rate of 1 unit per hour until it is full again. If your energy runs out, you cannot submit to Web-CAT again until another unit of energy is regenerated.

Because of the energy regeneration limit, you should not plan to have an unlimited number of Web-CAT submissions in one sitting. If you're working on the assignment for one evening, in 3 hours you'll be able to have around 6 submissions, so don't spend a submission trying to fix one formatting error or changing one line of code--work smarter, and fix all the formatting errors you can see in your previous Web-CAT submission before submitting again, or try to reproduce problems using your own software tests so you'll be able to figure out whether your code is fixed on your own, without submitting to Web-CAT every time.

On programs of this size, most students use only 8-10 Web-CAT submissions, which you can easily get in two short work sessions of 3 hours each, so plan out your time so you will be able to work without pressure, instead of saving all your Web-CAT work until the last night.

Help! The deadline is approaching and I'm out of energy!

Lack of energy won't prevent you from submitting your final work before the deadline. If you run out of energy and your next unit won't regenerate until after the deadline, don't worry--Web-CAT will allow you to submit your work without energy and process it immediately, until the deadline is reached. Simply put, if you submit within an hour of the deadline, Web-CAT won't "lock you out" of submitting, even if you have no energy left.

What happens if I submit, but have no energy?

If your next unit of energy won't regenerate until after the due deadline, your submission will be processed immediately (see above).

Otherwise, if you are submitting through the web interface, you cannot submit if you have no energy.

If you are submitting from inside your IDE, the results page you receive will indicate you're out of energy and ask you to wait until your energy regenerates.

In either case, simply wait until the next unit of submission energy regenerates to resubmit.

Where did the idea come from?

The submission energy bar was inspired by many mobile games (like Clash of Clans) in which the user is given a limit on the number of actions they can perform, but the limit replenishes over time. Research has shown that such systems have encouraged gamers to change their behavior by revisiting tasks periodically over a longer period of time instead of binge-playing in just one or two sittings.

It turns out that on *programming assignments*, research has shown that starting earlier and working over more work sessions leads individuals to earn higher scores than they would if they binge-programmed in one or two marathon sessions just before the deadline. Binge programming at the last minute results in lower scores for most students. Thus, the submission energy bar was born.

Why was the energy bar developed?

There are several reasons the energy bar helps students...

1. **Help students get better grades.** If you start earlier, you will have the opportunity to make more submissions and get more feedback over a longer period of time. Earlier feedback gives you more time to ask questions in class, visit office hours, or participate in discussions in class forums. If you get stuck, you have more time to get help becoming unstuck. And finally, it allows you to recognize opportunities to apply what you hear in class to your assignments. If, on the other hand, you wait until the last minute to get started, often there isn't time to seek help.
2. **Encourage students to think about their changes.** When you are developing your solution and come across a problem, some students fall back on a "guess and check" strategy where they make small changes and resubmit their work, sometimes leading to a very large number of Web-CAT submissions in order to resolve the issues. Unfortunately, research indicates that this pattern isn't helpful for learning, because the student may not understand the source of the problem, or may not

understand the effects of the changes they are trying out. The recharge rate on submission energy encourages students to think through the meaning of any change, to double-check changes locally first, and to resolve multiple errors at a time, rather than trying out guesses with submission after submission. When you think harder about the effects of your changes, and confirm for yourself they do what you want before you submit, you learn more and learn faster.

3. **Reduces Web-CAT congestion.** Submission energy keeps other students from overloading Web-CAT when you need to get feedback on your own submissions, too.

What strategies can I apply to get the most out of the energy bar?

- **Start working earlier.** By working earlier, you will get the opportunity to make more submissions. As you work earlier, you can recognize how labs, lecture, and other assignments provide clues on how to complete the assignments. It's hard to recognize these if you wait until the last minute.
- **Plan out how you spend your time.** Think explicitly about when you want to schedule your programming sessions into your weekly schedule, so you are more deliberate about spreading your effort out instead of focusing on a small number of marathon programming sessions or an all-nighter.
- **Make the most of each submission.** If Web-CAT indicated you have several errors or issues, try to fix as many as possible before making a new submission. Try to fix or add test cases. Basically, make smart submissions and think through the consequences of the changes you make--because that's how you exercise your skills and learn how to improve.

A.2 Project Group 2 - Grouping Objects

In this project class, students were tasked with navigating around a two-dimensional game world, and having their game agents take actions based on the situations that they encounter. In this project, the state of the two-dimensional grid was managed for them. In a later project, they would have to manage it themselves. In Fall 2015, students were asked to implement a game where tomato-loving aliens called "Greeps" landed on Earth and wanted to acquire as many tomatoes as possible. In Fall 2017 and Spring 2019, students were asked to consider a more serious topic, and implement a model of Schelling's Model of Segregation.

A.2.1 Fall 2015 Project 3 - Invasion of the Greeps

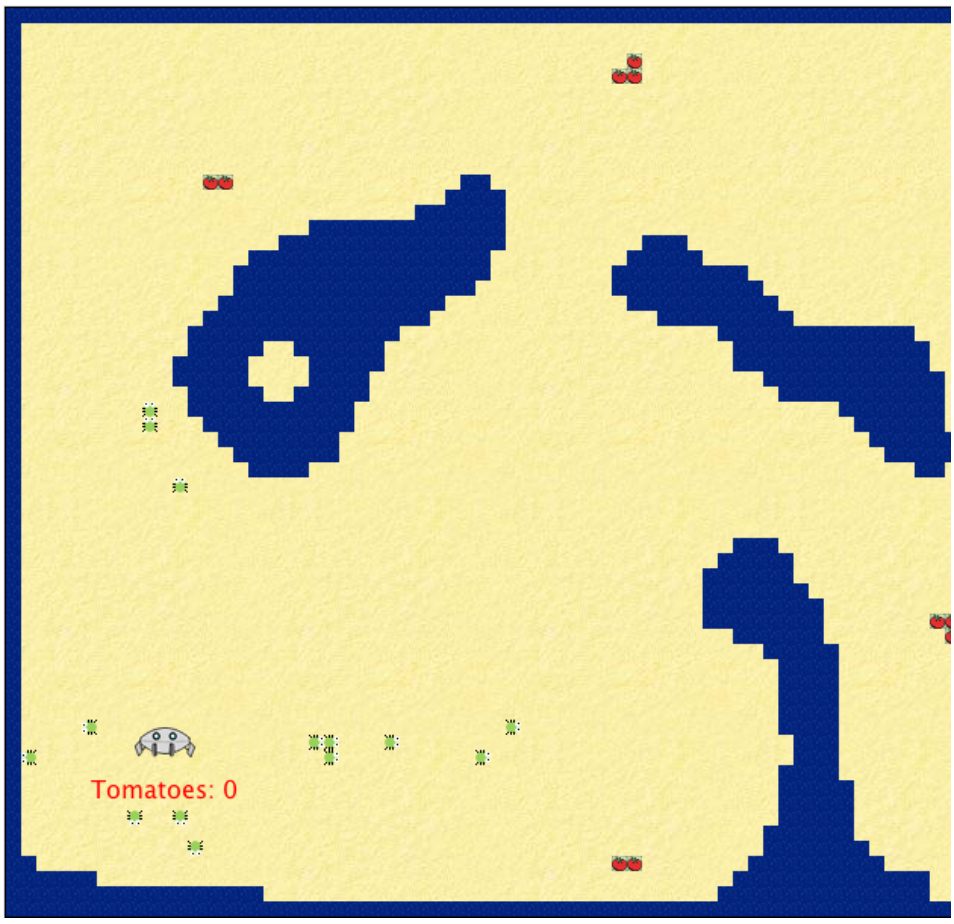
Program 3

Program 3: Invasion of the Greeps

Goal

Alien creatures have landed on Earth--they are **the Greeps**. Greeps like tomatoes. By some incredible stroke of luck, they have landed in an area where tomato patches are found randomly spread out over the otherwise barren land. Help your Greeps collect as many tomatoes as possible and bring them back to their ship.

You will accomplish this task by implementing your own Greep class (the other classes are already provided as part of a library). At a minimum, your solution must be able to bring at least one tomato back to its ship.



Starting Materials

Download the scenario for this assignment, which contains all the classes you need to start.: program3.zip.

The starting scenario has the following classes already defined (none have methods you can use in your solution):

`Earth`
: The Earth represents the map (or world) where Greeps operate. The `Earth` class provides **two constructors** for your convenience as you work on this project:

  - The `Earth()` constructor takes no parameters and uses three separate maps (with randomly placed tomato patches) to show a simulated ship landing on each map. An entire ship's crew of Greeps will be dispatched and allowed to harvest for a limited time. After all three maps have been used, your "score" (the number of tomatoes retrieved) will be shown. You can use this for running an interactive mission test of your Greep class.
  - The `Earth(`*`int`*`)` constructor takes a number from 1-3 as parameter, and shows only the specified map, with tomato patches at fixed locations. You can use this to create a fixed map in test class and test you Greeps behaviour for that specific map. You can also drop a greep (and a ship) anywhere you choose to test out individual methods.

`Water`
: As with jeroo scenarios, water represents an obstacle that cannot be crossed. If a Greep enters the water, it drowns and is removed from the simulation. Water completely surrounds the ground on which the Greeps have landed.

`Tomato`
: Tomato objects grow on tomato patches and represent the food that Greeps love. However, Greeps are pretty small, and cannot simply pick up a tomato by themselves. In order for a Greep to carry a tomato, two greeps must be together--then one of them can load the tomato onto the other.

`TomatoPatch`
: Represents a fertile area of the ground where tomatoes grow. If a tomato is loaded by one Greep onto another, a new tomato will replace it after 10 turns.

`PaintTrail`
: Greeps cannot communicate directly with each other, but they can leave a trail of paint on the ground so that other Greeps can find it. This paint disappears over time.

`Ship`
: The space ship the Greeps landed in serves as their home base. Greeps can always sense roughly in which direction their ship lies. Tomatoes must be returned to the ship in order to be counted as "harvested". Ship class has a `tomatoCount()` method which returns the number of tomatoes the ship currently has. You can use this method in you test class to check if your greeps are bringing back tomatoes successfully to their ship.

`Alien`
: The base class for `Greep`. This class defines the methods that all Greeps share, and which you can use to construct your solution. This class is documented in the next section.

Structuring Your Solution

Your task is to write the `Greep` class, which must be a subclass of `Alien`. However, unlike jeroos, the `Alien` class is a bit more general and **does not use a `myProgram()` method**.

Working with the `Alien` class in this assignment requires us to understand a more general model of how micro-worlds operate. Micro-worlds are a kind of *discrete event simulation*--think of a "clock" that ticks, and for each clock tick, all of the actors in the world get a

chance to do something. At every clock tick, every actor gets a "turn" to act. When we were working with Jeroos, we provided a fully scripted sequence of actions in a `myProgram()` method, and on each "turn" the Jeroo took "the next step" in that plan, methodically plodding along. In this simulation, however, there will be a large number of Greeps all working at the same time. For each turn, every one of them will have a chance to act. Their actions will change the state of the world, so they may act differently on the next turn--they will *react* to their surroundings instead of carrying out a predetermined sequence of actions.

To make the simulation work, your `Greep` class will provide an `act()` method that determines what that object will do for its current turn. Running the simulation a single step by pressing the "Act" button amounts to calling `act()` on every single actor--giving each one its single turn. Running the simulation by pressing the "Run" button amounts to running turn after turn after turn, where each turn consists of calling `act()` on every single actor. The machinery for calling `act()` and managing all the actors is all built into the micro-world, so all you have to do is say what a given actor does on its turn--you have to write the `act()` method for the `Greep` class. For this assignment, you have to write the `act()` method and all other methods inside the Greep.java file.

Because `Greep` is a subclass of `Alien`, let's see what that class provides.

The Alien Class

The `Alien` class, by a stroke of luck, includes a number of methods that are similar to Jeroo methods, plus a few more. You can use any of these inside your `Greep` class. Also, Greeps understand the same directions that jeroos do: NORTH, SOUTH, EAST, and WEST for compass directions, and LEFT, RIGHT, AHEAD, and HERE for relative directions.

Method	Purpose
`hop()`	Hop one space ahead. If the Greep jumps into the water, it drowns. Note that Greeps *can* occupy the same location (be next to each other), so water is the only hazard.
`turn(`*`relativeDirection`*`)`	Turn in the indicated direction [`LEFT` or `RIGHT`, just like jeroos; `turn(AHEAD)` and `turn(HERE)` are meaningless]
`hasTomato()`	Is this Greep carrying a tomato?
`isClear(`*`relativeDirection`*`)`	Is there a clear space in the indicated direction? A clear space contains no other objects. [`isClear(HERE)` is meaningless]
`isFacing(`*`compassDirection`*`)`	Is this Greep facing in the indicated direction?
`sees(`*`objectType, relativeDirection`*`)`	Is there an object of the specified type in the indicated direction? For example, `this.sees(Greep.class, LEFT)` or `this.sees(Tomato.class, AHEAD)`.
`loadTomato(`*`relativeDirection`*`)`	Load a tomato onto a neighboring Greep in the indicated direction. Nothing happens if there is no tomato here, or if there is no Greep in the indicated direction, or if the Greep is already holding a tomato of its own. Note that, since Greeps can be in the same cell at the same time, it is possible to `loadTomato(HERE)` if there is another Greep in the same spot--but you cannot load a tomato on yourself.
`unloadTomato()`	If the Greep is currently carrying a tomato and is at the ship, the Greep can unload the cargo onto the ship, which counts as a "score". If the Greep is not at the ship, it will just drop the tomato and, splat, it's gone. Thankfully, Greeps don't need any help dropping off a tomato and can do this by themselves.

Method	Purpose
`paint()`	Drops a blob of paint at the Greep's current location, so the Greep can leave a trail. Paint gradually disappears over 50 turns.
`getDirection()`	Returns the current `CompassDirection` in which the Greep is facing.
`getShipDirection()`	Returns the `CompassDirection` in which the Greep's ship lies. Note that the ship is not necessarily in a straight line this way--it may be off in a diagonal direction instead, but the closest cardinal compass direction is returned.
`turnTowardShip()`	Turns the Greep so that it is facing in the direction (NORTH, SOUTH, EAST, or WEST) closest to where the ship lies. Note that the ship is not necessarily in a straight line this way--it may be off in a diagonal direction instead, but the closest cardinal compass direction is where the Greep will turn.

You can use any of these methods in constructing your solution. At the same time, however, there are several restrictions you must follow:

1. **Only one move per turn.** Your Greep can only `hop()` once per turn (which is why there is no `hop(n)` method). "Once per turn" means only once per call to `act()`.
2. **Greeps cannot communicate directly to each other.** That is: no field accesses or method calls to other Greep objects are allowed. (Greeps can communicate indirectly via the paint spots on the ground.)
3. **No long vision.** You are allowed to look at the world only using the Greep's `sees()` method, and may not look farther. That is: no use of methods similar to `getObjects...()` or `getOneObject...()` are permitted.
4. **No creation of objects.** You are not allowed to create any scenario objects (instances of any Actor classes, such as Greep or Tomato). Greeps have no magic powers--they cannot create things out of nothing.
5. **Only boolean fields.** You can add boolean fields to your Greep class, but cannot add other kinds of fields. In particular, you cannot "remember" coordinate positions for key features, like the ship or tomato patches. Only yes-or-no values can be kept in fields (other than test classes, of course). Make sure that all the fields are private.
6. **No teleporting.** Methods from Actor that cheat normal movement (such as `setGridLocation()`) may not be used.

Generating Random Numbers

Although it is not necessary in this assignment, sometimes you might find it helps to make random choices so your Greeps are a little more independent and varied. For example, sometimes you might want a Greep to wander around randomly looking for tomatoes. Or maybe when it finds water, sometimes you want it to turn right and other times turn left. Java provides a built-in class called `Random` for generating random numbers, and the Sofia library includes a special version of this class that is helpful for beginners. To use this class, add the following import statement at the top of your Greep.java file:

```
1 import sofia.util.Random;
```

The `Random` class provides a method called `generator()` to get an object that represents a random number generator. Here, we only need to deal with generating random integers, and the generator provides a method that is very useful for this purpose. You can use it like this:

```
    int value = Random.generator().nextInt(4);  // generate a random number from
```

The generator provides a method called `nextInt()` that generates a random integer. It takes a single parameter, which is an upper limit. When you provide this upper limit, the `nextInt()` method will generate a number from 0 (inclusive) up to (but not including) the upper limit.

So, for example, if you want to generate a number from 0–99, you would call `nextInt(100)`. Suppose that you would like to perform some action 15% of the time. You could do this:

```
if (Random.generator().nextInt(100) < 15)
{
    // ...
}
```

Here, the call to `nextInt()` will produce a number from 0–99 (that is 100 possible values), and the if statement will execute its true branch if the generated number is in the range 0–14 (which is 15 possible values, or 15% of the time). You should also check the Sofia Random API for more information on `Random` class.

Testing Random Behaviors

Random behaviors are great for chance-based events. But random behaviors also make software testing hard. When you add random behavior to your Greep and then want to test it, what will your test case do? Suppose you want your Greep to turn left at the water's edge half the time, and right the other half. If you write a test case where the Greep sees the water ahead, it might turn left ... or it might not. How can you write tests for that???

The answer is simple: the `Random` class helps you. Consider the following code sequence, which generates three random numbers less than 100:

```
int x = Random.generator().nextInt(100);
int y = Random.generator().nextInt(100);
int z = Random.generator().nextInt(100);
```

It would be difficult to write a test case that used this code, since you have no way of controlling what values end up in *x*, *y*, and *z*. For test cases, however, the `Random` class provides a special method called `setNextInts()` that *lets you control what numbers are generated* for testing purposes. You use it like this:

```
// In your test case, do this:
Random.setNextInts(40, 50, 60);

// In the code you are testing, this happens:
int x = Random.generator().nextInt(100);
int y = Random.generator().nextInt(100);
int z = Random.generator().nextInt(100);

// You know x will get the value 40, while y is 50, and z is 60
```

So, when you are testing behaviors that are random, you can *force* the actions to be predictable just by saying in your test cases what sequence of values you want the random number generator to produce. Outside of test cases, the generator will produce a truly (pseudo-)random sequence, but inside your test cases, the numbers will be completely determined by you.

Comments on Design and Testing

As in other assignments, you will be graded in part on the design and readability of your solution using the posted grading criteria, so consider these factors when devising and naming your methods. The Program Grading Rubric (same as on Programs 1 and 2) describes the grading criteria. Note that a portion of your grade will be based on your approach to testing your solution. Make sure to write test methods to test each of the methods in your Greep class. Also remember to use `setUp()` method in your test class to set up the initial conditions. In the test class, you are free to create any scenario objects

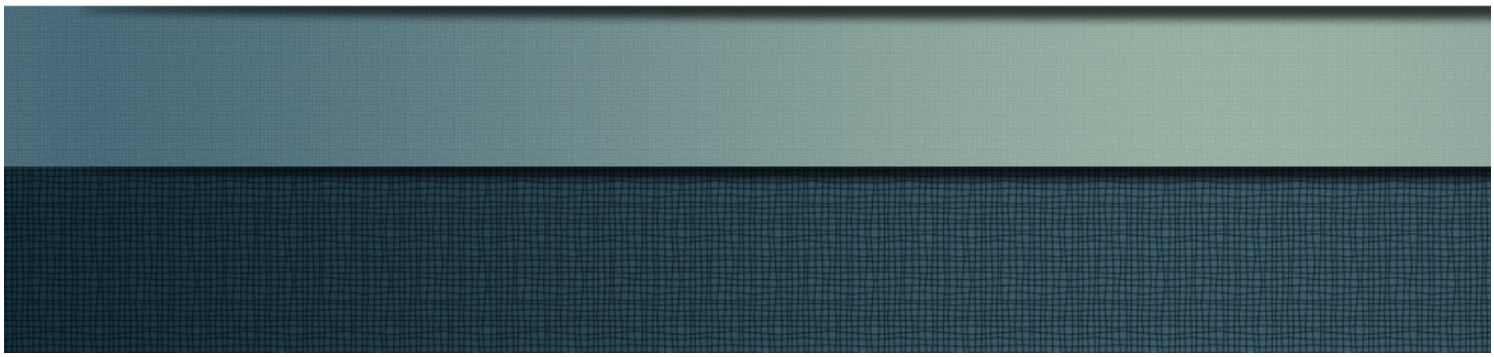
(instances of any Actor classes, such as Greep or Tomato). In the test class, you can also teleport your Actor to any position you want in order to test the behaviour of the Actor (using `setGridLocation(x, y)` method).

A Contest - ignore during summer

At a minimum, your solution must be able to retrieve one tomato within the turn limit provided by the `Earth` class. However, your goal is to retrieve as many tomatoes as possible before time runs out.

So, for for this assignment, all submissions that are **completed on time** will be entered in a **tournament**. Each submission will be played off on a set of new maps that you haven't seen, and scored based on the total number of tomatoes harvested across all of these maps. Awards for the tournament winners will be given in class. Good luck!

All program assignments are submitted to Web-CAT. Use the Controls->Submit... menu command to submit your work.



A.2.2 Fall 2017 & Spring 2019 Project 3—Schelling's Model of Segregation

Program 3

Program 3: Schelling's Model of Segregation

- It is an Honor Code violation to resubmit prior work from an earlier semester in this course.
- It is an Honor Code violation to submit work authored by someone else as your own.

You **must** include the following Virginia Tech Honor Code pledge at the very top of each Java file you write (separate from the class Javadoc comment). Include your name and PID as shown, to serve as your signature:

```
// Virginia Tech Honor Code Pledge:
//
// As a Hokie, I will conduct myself with honor and integrity at all times.
// I will not lie, cheat, or steal, nor will I accept the actions of those
// who do.
// -- Your Name (pid)
```

Goal

Racial segregation has always been a pernicious social problem in the United States. Although much effort has been extended to desegregate our schools, churches, and neighborhoods, the **US continues to remain segregated (https://tcf.org/content/commentary/racial-segregation-is-still-a-problem/)** by race and economic lines. The New York Times has an **interesting interactive map (http://projects.nytimes.com/census/2010/explorer?ref=censusbureau)** showing the distribution of racial and ethnic groups in cities and states around the country as recorded during the 2010 census. Why is segregation such a difficult problem to eradicate?

The forces that lead people to select where they live are complex and it is unlikely that any simple model can truly answer the questions of how such segregation arises and why it persists. However, it is still possible to ask simple questions that may shed some light on the situation.

In 1971, the American economist **Thomas Schelling (http://en.wikipedia.org/wiki/Thomas_Schelling)** created an agent-based model that might help explain why segregation is so difficult to combat. He was trying to answer a simple question: is it possible that even if a group of individuals would prefer to live in

integrated neighborhoods, might they still end up in segregated neighborhoods as the result of their collective choices?

His simple model of segregation showed that even when individuals (or "agents") didn't mind being surrounded or living by agents of a different race, they would still choose to segregate themselves from other agents over time! Although the model is quite simple, it gives a fascinating look at how individuals might self-segregate, even when they have no explicit desire to do so. In this assignment, students will create a simulation of Schelling's model, and see how it operates.

How the Model Works

We will work with a slight simplification of Schelling's model. To explain the model, suppose there are two types of agents: X and O. For this assignment, these two types of agents will represent: those who prefer their iced tea sweetened, and those who prefer their iced tea unsweetened. We'll use separate visual representations for these individuals so it is easier to see how agents move. Green elephants are sweet tea drinkers, while orange monkeys prefer unsweetened tea instead.

Two populations of the two agent types are initially placed into random locations of a neighborhood represented by a grid. After placing all the agents in the grid, each cell is either occupied by an agent or is empty as shown below.

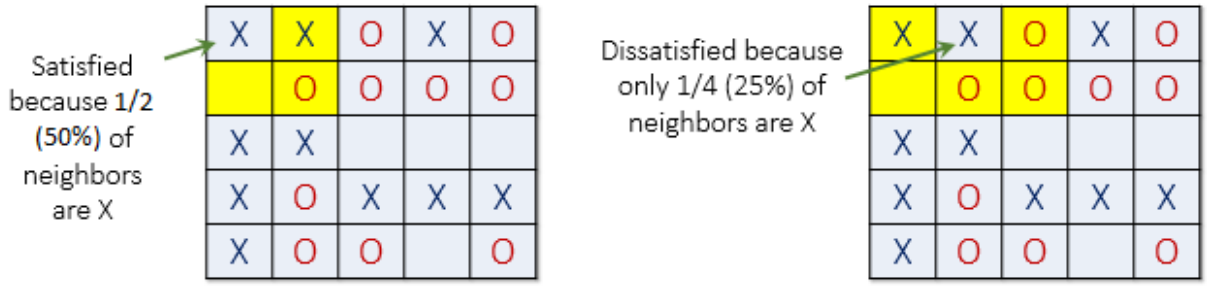
Agents placed randomly in grid

X	X	O	X	O
	O	O	O	O
X	X			
X	O	X	X	X
X	O	O		O

Now we must determine if each agent is *satisfied* with its current location. A **satisfied** agent is one that is surrounded by at least *t* percent of agents that are like itself. That means each agent is perfectly happy to live in an integrated neighborhood where 1 - *t* percent of its neighbors have the opposite tea preference. We'll call *t* the agent's *satisfaction threshold*. Note that the higher the threshold, the higher the likelihood the agents will not be satisfied with their current location.

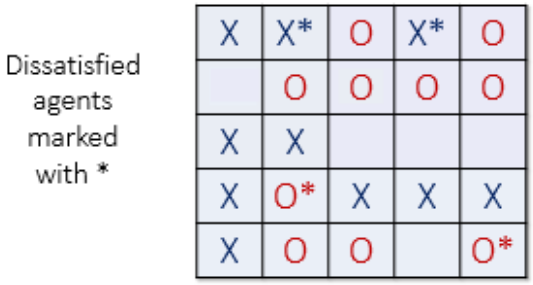
For example, if *t* = 30%, agent X is satisfied if at least 30% of its neighbors are also X, meaning up to 70% of its neighbors can be O's. If fewer than 30% are X, then the agent is not satisfied and it will want to change its location in the grid. For the remainder of this explanation, let's assume a threshold *t* of 30%. This means every agent is *fine with being in the minority* as long as there are at least 30% of similar agents in adjacent cells.

The picture below (left) shows a satisfied agent because 50% of X's neighbors are also X (50% > *t*). The next X (right) is not satisfied because only 25% of its neighbors are X (25% < *t*). Notice that **empty cells are not counted** when calculating satisfaction.



When an agent is not satisfied, it can move to any vacant location in the grid where it would be satisfied.

In the image below, all dissatisfied agents have an asterisk next to them. These are the agents that would choose to move when it is there turn, although they will only move to a different cell if that cell is empty, and if they would be satisfied after the move. Note that such a move may cause some agents that were previously satisfied to become dissatisfied!



Design Exercise 1

In our implementation, on every call to `act()` an `Agent` will either move or stay put. You'll have to write the logic to determine whether it will move or not, and to where. Recall that the `World` provides a `getOneObjectAt()` method that takes an (x, y) location and returns the object, if any, that is located there (if multiple objects are located there, it just returns one of them at random).

Before you read any further in this assignment, take a piece of paper and outline which method(s) you would create for your `Agent` class to implement this choice. What logic would you use to decide whether you are satisfied? What logic would you use to decide where to move?

Outline your logic now, and the methods you would use to implement it. Then proceed to read the rest of the assignment.

Starting Materials

Download the scenario for this assignment, which contains all the classes you need to start.: **program3.zip**.

The starting scenario does not contain any starting classes--you'll have to create all the classes yourself from scratch. The starting scenario only contains the necessary images for the green elephants and the orange monkeys.

Classes You Create

You will create two classes: one to represent the world, and one to represent the "agents" (that is, residents) who live in and move around in the world. For this simulation, we will divide our population into two groups of agents: those who prefer their iced tea sweetened, and those who prefer their iced tea unsweetened. We'll use separate visual representations for these individuals so it is easier to see how agents move. Green elephants are sweet tea drinkers, while orange monkeys prefer unsweetened tea instead.

A City Class

Create a subclass of `World` called `City` that represents a single rectangular region where your agents will live. You can use any image background you wish for your city.

Your `City` class must provide a constructor that takes its width and height as parameters. Grid cells in your city should be 24 pixels square. Each `City` should be initially empty (that is, no agents).

Your `City` class must also define a `populate()` method that takes three double parameters between 0.0 - 1.0. The first parameter represents the percentage of elephants in the city, and the second represents the percentage of monkeys. The third parameter represents the satisfaction threshold for the agents. For example, consider this call:

```
city.populate(0.3, 0.4, 0.3);
```

This call indicates that approximately 30% of the cells in the city should be filled with elephants, and approximately 40% of the cells should be filled with monkeys, which will leave approximately 30% of the cells unoccupied or empty. All of the agents will use 0.3 (30%) as their satisfaction threshold, according to the third parameter.

You can implement populate using a pair of loops:

- For each possible X coordinate ...
- And for each possible Y coordinate ...
- Generate **one** random number between 0.0 - 1.0 for the cell at (x, y).
- If that random number is less than or equal to the "elephant" parameter, then place an elephant at (x, y), or
- If that random number is less than or equal to the sum of the "elephant" and "monkey" parameters, then put a monkey at (x, y), or
- Otherwise, leave that cell empty.

Note that you **must** limit yourself to just a single random number per cell. For example, for a 10x10 city, you should make *exactly* 100 calls to generate random numbers, no more, no less. Use the *same* random value for all tests you perform to determine the contents of a single cell location in the city.

Finally, your `City` class must also provide a default constructor that initializes the city to a size of your choice (something that looks good on your screen), and then calls `populate()` with population ratios and a threshold that you pick. You should implement your default constructor by using `this` to call the `City` constructor that takes the width and height as parameters instead of using `super()`.

An Agent Class

Create a subclass of `Actor` called `Agent` that represents a single agent.

Your `Agent` should provide a constructor that takes two parameters: a `String` indicating the kind of animal, and a `double` value between 0.0 - 1.0 indicating its satisfaction threshold.

Note that while we are describing this scenario in terms of elephants and monkeys (i.e., the kind of animal might be "elephant" or "monkey"), **you may not assume these are the only kinds of agents**. Your code should work for *any kind specified*, without assuming only these two values might be provided.

Your constructor should set the image used for the `Agent` using the animal name. For example, the elephants use the "elephant.png" image, while the monkeys use the "monkey.png" image (and rabbits use the "rabbit.png" image, and kangaroos use the "kangaroo.png" image, etc.).

Your `Agent` must also provide the following methods:

`getKind()`

A getter method that returns the kind of agent, as a string (such as "elephant", for example).

`getThreshold()`

A getter method that returns the agent's satisfaction threshold, as a double value.

`isSameKindAs(agent)`

A boolean method that returns true if this agent is the same kind of agent as the one provided.

`isSatisfiedAt(x, y)`

A boolean method that returns true if this agent would be satisfied at the specified location, or not. Remember to watch out for out of bounds errors when an agent is on an edge or corner of the city.

`isSatisfied()`

A boolean method that returns true if this agent is satisfied at its current location. Note: you **may not** simply repeat the code from isSatisfiedAt(). Your implementation here should be a single line (no more). Be careful with how you compare this agent with its neighbors to count up the number of neighbors of the same kind.

`relocate()`

A method that moves the `Agent` to a new location in the grid where it will be satisfied, if there is one. You may use a pair of nested loops to examine each (x, y) location to find the first empty location where this agent will be satisfied, then move there. Remember that if there are no empty spaces where this agent will be satisfied, it should stay put. You can use the agent's `setGridLocation()` method (which is a combination of `setGridX()` and `setGridY())` to move it to a new location.

`act()`

Executes one "turn" for this agent, which means determining if the agent is satisfied, and relocating if it is not.

Design Exercise 2

Go back to the outline you made of your program logic and your methods earlier (from Design Exercise 1). Compare your method breakdown with the breakdown given above.

Does the method breakdown given above *help simplify the problem*? Does it make it easier to outline the logic of the `act()` method?

Identify the key differences in your own outline and the method breakdown shown above for what is required. Try to determine why those differences are present, and look for the reasons behind the methods specified above.

Comments on Design

As in other assignments, you will be graded in part on the design and readability of your solution using the posted grading criteria, so consider these factors when devising and naming your methods. The **Program Grading Rubric** (same as on Programs 1 and 2) describes the grading criteria. Note that a portion of your grade will be based on your approach to testing your solution.

For testing, remember that you can create very small, completely empty cities (even as small as 1x2, 2x2, or 3x3), and place exactly the agents you want at exactly the positions you want in order to test behaviors. All of the methods for cities can be tested without calling act(). Constructing simple, tiny situations to test out your behaviors is much cleaner and easier than trying to "run" a big simulation with a large population of elephants and monkeys spread around.

Submitting Your Solution

All program assignments are submitted to Web-CAT. Use the Controls->Submit... menu command to submit your work.

A.3 Project Group 3 - Variable Scoping

In this project, the focus was on inheritance and scoping. The students were tasked with designing a tower defense where they protected the queen of an ant colony from invading bees. Unlike earlier projects, the students were responsible for their own class design in this project. In Spring 2019, students also focused on the inheritance and scoping, but in the context of a falling sand game.

The same project specification was used in both Fall 2015 and Fall 2017 semesters. This representation is from the Fall 2017 semester when we had transitioned to the Canvas LMS.

A.3.1 Fall 2015 Project 4 & Fall 2017 Project 5—Ants and Some-Bees

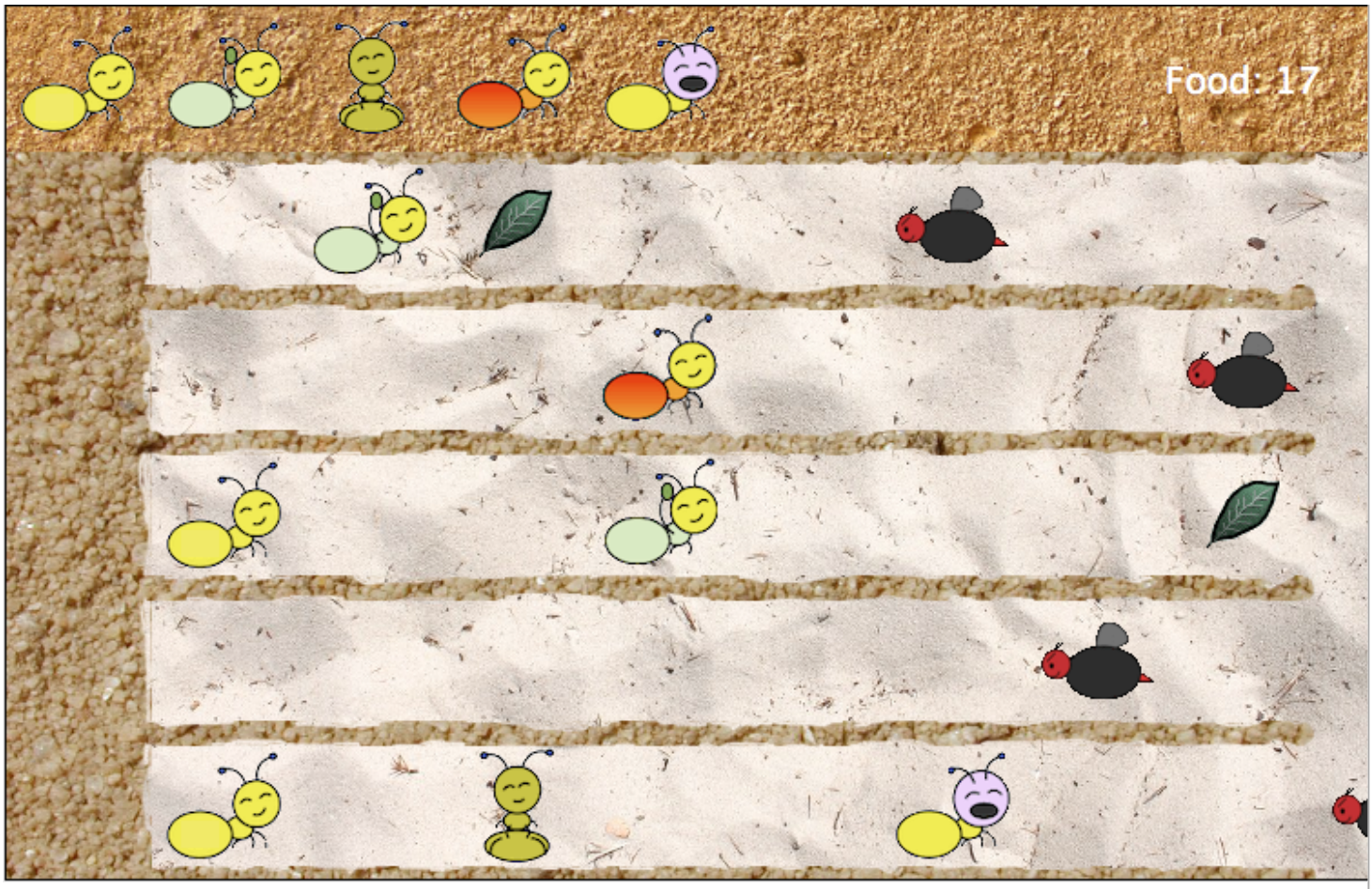
Program 5

- It is an Honor Code violation to resubmit prior work from an earlier semester in this course.
- It is an Honor Code violation to submit work authored by someone else as your own.

For all assignments, you **must** include the following Virginia Tech Honor Code pledge at the very top of each Java file you write (separate from the class Javadoc comment). Include your name and PID as shown, to serve as your signature:

```
// Virginia Tech Honor Code Pledge:
//
// As a Hokie, I will conduct myself with honor and integrity at all times.
// I will not lie, cheat, or steal, nor will I accept the actions of those
// who do.
// -- Your Name (pid)
```

Ants vs. SomeBees



Goal

In this project, you will create a **tower defense** (https://secure.wikimedia.org/wikipedia/en/wiki/Tower_defense) game called **Ants Vs. SomeBees**. As the ant queen, you populate your colony with the bravest ants you can muster. Your ants must protect the colony from the evil bees that invade your territory. Irritate the bees enough by throwing leaves at them, and they will be vanquished. Fail to pester the airborne intruders adequately, and your queen will succumb to the bees' wrath. This game is inspired by PopCap Games' **Plants Vs. Zombies** (http://www.popcap.com/games/pvz/web) ®.

(If you've never played Plants vs. Zombies, you can **play it for free right in your web browser** (http://www.popcap.com/games/plants-vs-zombies/online) .)

This project requires you to implement a larger number of classes. It is **very important** to think about how these classes might relate to each other, and **how you can use inheritance** to make the code easier to write and easier to understand, while eliminating duplicate code. Consider how you will use inheritance very carefully--*designing* the classes yourself is an important aspect of this assignment.

Starting Materials

Download the scenario for this assignment, which contains the classes and images you need to start: **program5.zip**. **Do not forget to unpack the zip file before you use it!**

The starting scenario provides two classes for you to use in your solution:

`Colony` is a skeleton you will complete yourself. It forms the "world" for this game. The source code provided only contains a constructor, and a stub for a method called `onTouchDown()`. Read the comments in the source file for details on these methods. See below for **required features you have to add** to this class yourself.

`HomeBase` is a predefined class that serves as the base class for your `Colony`. This class sets up the game world geometry: the screen is a 10x6 grid, where each grid cell is 80 pixels square. It also provides a number of methods that you can use in implementing your Colony's features:

Method	Purpose
`setActorChoices(class1, class2, ...)`	Your `Colony` class should call this method in its constructor to indicate which actor classes (ant types) you have created. It will cause the world to show a simple row of ant selectors (clickable images) across the top edge of the game screen that you can click on to indicate which kind of ant you wish to place while playing the game.
`newActorOfSelectedType()`	Call this method in your code to create a new ant of the selected type, in order to place it somewhere on the screen. This method will return null if no ant type has been selected by the player yet. Note that this method will return a reference of type `Actor`. In order to use the reference to invoke `Ant` methods you will need to typecast it: `Ant selectedAnt = ((Ant) newActorOfSelectedType())`
`win()`	Your code should call this method when the ants win to pop up a winning message and stop the game.
`lose()`	Your code should call this method when the bees win to pop up a losing message and stop the game.
`act()`	This method is automatically called on each turn of the simulation. The `HomeBase` class uses it to update the food display on-screen,and determine whether the game is over. If you override this method in your `Colony` class, **be sure to call super.act();** so that the food display gets updated properly.

Classes You Create

You are responsible for creating five different types of ants, plus the bees that harrass them. In addition, you'll also create a small number of additional classes that represent objects (not insects) in the game. These are discussed separately below.

Types of Ants

Ants are "created" when the player clicks on one of the available ant types across the top of the screen, and then clicks on an open spot in the playable area of the game to indicate where the new ant should

be placed. Creating ants requires food from the colony, and should be handled in your colony object (discussed below).

You are required to provide five specific kinds of ants that differ in how they behave. All ants have an integer "health" value that indicates how strong they are. Health is decreased when an ant is stung by a bee, and an ant leaves the field (is removed from the world) when its health reaches zero.

Every ant should provide a method called `getHealth()` that returns the ant's current health, as well as a method called `injure(n)` that reduces the ant's health by the provided amount. When an ant's health reaches zero, it should remove itself from the colony. Further, every ant class should provide a default constructor. Finally, every ant should provide a `getFoodCost()` method that indicates how many food units are necessary to add this ant to the colony.

Harvester Ant

A `HarvesterAnt` simply collects food, adding it to the colony's food supply. Every 40 turns, a harvester produces one unit of food to add to the colony. Adding a harvester ant costs 2 food units. A harvester ant begins with a health of 1.

Thrower Ant

A `ThrowerAnt` is an offensive unit that throws a continuous stream of leaves straight ahead. Leaves annoy flying bees, causing them to fly away. Every 120 turns, a thrower throws one leaf. Adding a thrower ant costs 4 food units. A thrower ant begins with a health of 1.

Fire Ant

A `FireAnt` is an offensive unit that explodes when its health reaches zero. When it explodes, it reduces the health of any bee in any neighboring cell by 3 health units. Adding a fire ant costs 4 food units. A fire ant begins with a health of 1.

Wall Ant

A `WallAnt` is a defensive unit that blocks bees. Adding a wall ant costs 4 food units. A wall ant begins with a health of 4.

Hungry Ant

A `HungryAnt` is an offensive unit with a huge appetite that can eat a bee. Any bee that reaches the hungry ant will be eaten--but the catch is that it will take the hungry ant 240 turns to "digest" the bee it just ate. While digesting, the hungry ant is vulnerable to any other bee that comes across it. Adding a hungry ant costs 5 food units. A hungry ant begins with a health of 1.

Bees

A `Bee` flies in a straight line due west, heading for the ant queen's chamber (on the left edge of the screen). Your Bee class, (named `Bee`), should turn to face due west as part of its default constructor.

Like ants, your bee class must provide a method called `getHealth()` that returns the bee's current health, as well as an `injure(n)` method that reduces the bee's health by the specified amount. When the bee's health reaches zero, it should remove itself from the colony. Bees start with a health value of 3.

Since our grid has large cells (80 pixels wide), bee movement would be very jumpy if bee's moved forward by one grid cell each turn--not to mention, they'd be super fast! Instead, the `move()` method provided by the `Actor` class can take floating point numbers, allowing actors to move by fractional portions of a grid cell. Bees move to the left by **0.0125 units** (1/80th of a grid cell) on each turn, as long as they are not blocked by an ant.

As soon as a bee touches an ant, it becomes blocked. The bee will be stuck on top of the ant until it can vanquish the ant. Once the bee reaches an ant, it will wait for 40 turns and then sting (injure) the ant, reducing its health by one. The bee will continue stinging the ant every 40 turns until one of the two is vanquished by the other.

As soon as any bee touches the queen's chamber on the left side of the screen (see below), the bees win the game, (i.e., the ants lose).

Other Actor Classes Representing Objects

`Leaf`: A leaf is thrown by a thrower ant at incoming bees. A leaf simply travels due east at a rate of 0.025 grid units per turn (twice as fast as a bee flies). As soon as a leaf hits a bee, it should injure the bee, reducing its health by 1. The leaf should also be removed from the world. Further, if the leaf reaches the right edge of the screen (x coordinate of 9) without hitting a bee, it should also be removed from the world.

`QueensChamber`: Create a class that represents the bee's goal: the queen's chamber. An image is already provided for you to use. The queen's chamber should be placed at (0, 3). If any bee reaches the queen's chamber, the bees win.

`Hive`: Create a class that represents the bee hive. An image is already provided for you to use. The hive should be placed at (9, 3) on the right edge of the screen. The hive should provide a method called `getBees()` that returns the number of bees in the hive, as well as a corresponding setter method that allows the number of bees in the hive to be set. Initially, the hive's constructor should start it with 30 bees to be released. The hive should do nothing for the first 400 turns. After that, the hive should release one bee at a time, randomly placing the bee in one of the five spots from (9, 1) - (9, 5). The hive should then wait a random delay between 80-400 turns before launching the next bee.

`Colony`: The colony represents the whole world. The queen's chamber and the hive should be created and placed appropriately in the colony's constructor. The colony starts off with 10 food units, and only gains food when it is provided by harvester ants. Your colony must provide a `getFood()` method that returns the number of food units currently in the colony, as well as a `addFood(n)` method that can be used by harvesters to add food to the colony, and a `consumeFood(n)` method that is used to reduce the amount of food in the colony when creating new ants. The ants win the game when the hive has no more bees, and there are no more bees anywhere in the colony. The colony creates a series of *controls* along row 0 at the top of the screen, based on the classes you pass into setActorChoices().Each of these control spots is itself just an actor, (one you don't have to write inherited from `HomeBase`), that remembers a specific ant class, and then calls setSelectedActor() on the colony whenever it is touched.

Testing

To test `onTouchDown()` use the setSelectedActor() method on your `Colony`. You could use it in a test like this:

```
public void testCreatingHarvester()
{
   // Simulate "clicking" the harvester ant control at the top of screen
   colony.setSelectedActor(HarvesterAnt.class);

   // Now test onTouchDown() on the colony somewhere
   colony.onTouchDown(2, 2);

   // The remainder of your test goes here
   ...
}
```

To simulate the passing of time in your tests note that the test class has a method called `run(world, N)` that takes a world (i.e., your colony) and will simulate *N* turns. However, it appears that this method may not work reliably in Greenfoot4Sofia, but it does work reliably on Web-CAT.

Submitting Your Solution

All program assignments are submitted to Web-CAT. Use the Controls->Submit... menu command to submit your work.

A.3.2 Spring 2019 Project 4—Particle Simulator

Program 4

- It is an Honor Code violation to resubmit prior work from an earlier semester in this course.
- It is an Honor Code violation to submit work authored by someone else as your own.

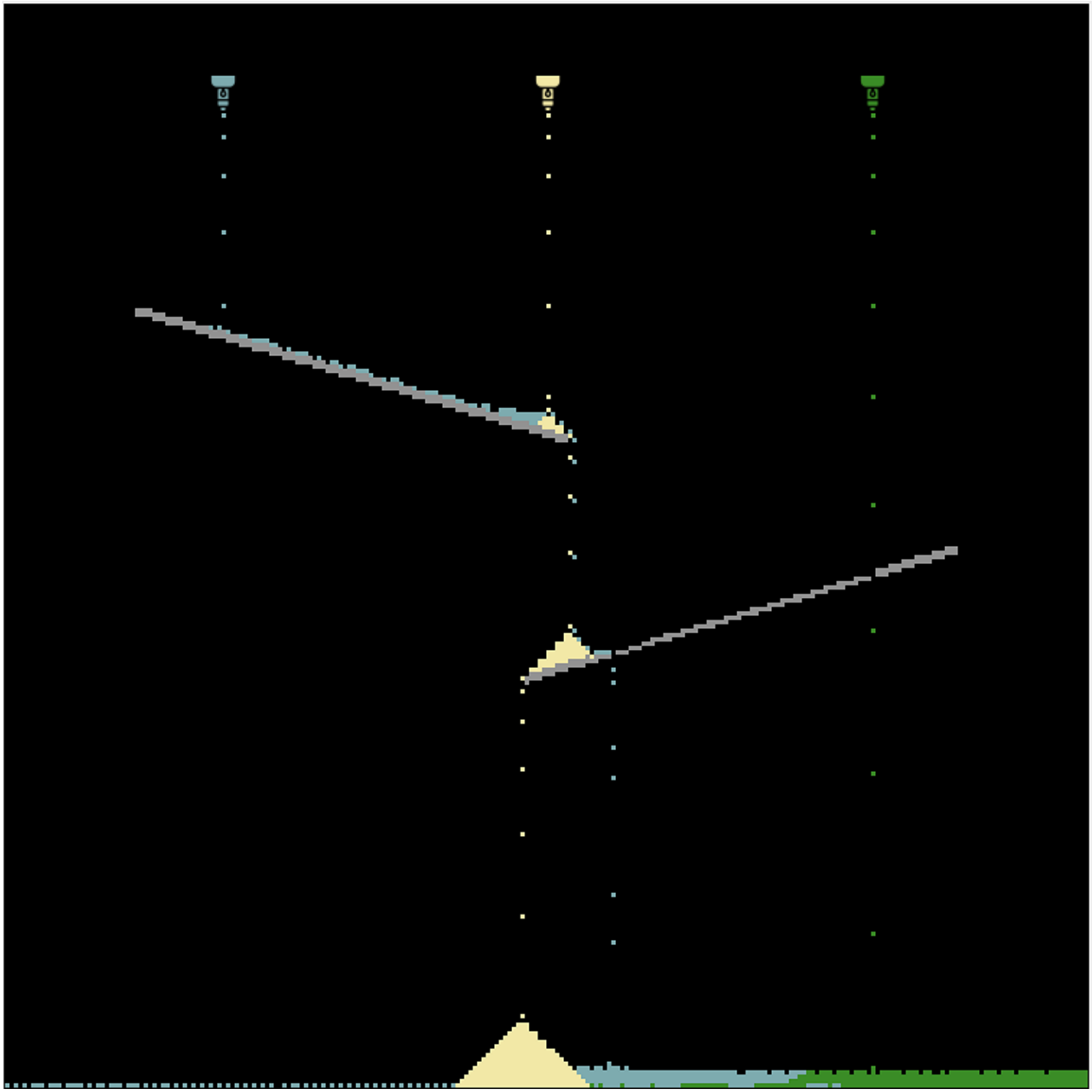
You **must** include the following Virginia Tech Honor Code pledge at the very top of each Java file you write (separate from the class Javadoc comment). Include your name and PID as shown, to serve as your signature:

```
// Virginia Tech Honor Code Pledge:
//
// As a Hokie, I will conduct myself with honor and integrity at all times.
// I will not lie, cheat, or steal, nor will I accept the actions of those
// who do.
// -- Your Name (pid)
```

Goal

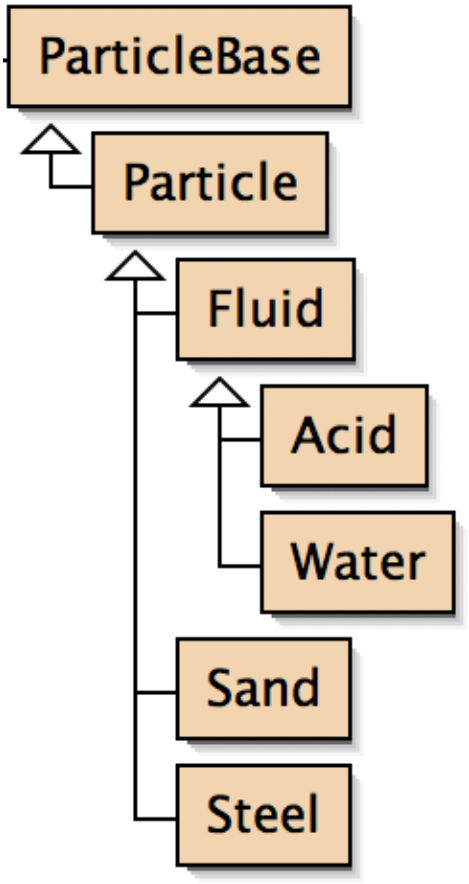
In this assignment, you will be writing multiple classes that **use inheritance and polymorphism**. This assignment will give you practice on implementing these features, as well as continued practice translating written instructions into corresponding code implementations.

A Particle Simulator



"Falling Sand" is a generic name for a number of simple game programs that were created last decade. These programs all shared similar properties. They were driven by a simple model of physics that involved tiny particles, and simulating the basic rules of motion for a single particle. By placing many particles in the same environment, you could then see what would happen. You can **read more about these games on Wikipedia (https://en.wikipedia.org/wiki/Falling-sand_game)** .

In this assignment, you will be creating a version of such a simulation where you implement the behaviors for four different types of particles. Our four types of particles are related using inheritance:



Sand, of course, is the kind of particle that "Falling Sand" is named for, and represents simple grains of sand. You will also model **Steel**, that is used to create fixed ramps, obstacles, or other features for particles to interact with. **Water** models a simple liquid that flows differently than sand, and **Acid** is another form of liquid that can eat through steel.

Starting Materials

Download the scenario for this assignment, which contains all the classes you need to start.: **program4-download.zip**.

The starting scenario contains a `World` subclass called `ParticleWorld` that forms the environment. It also includes a `Dropper` class that drips out particles while the simulation is running. **You will not need to edit the world class or the dropper class in this assignment**. Instead, you will only be creating `Actor` subclasses that represent the different kinds of particles.

The starting scenario also contains a `ParticleBase` subclass of `Actor` that should be the parent of all of your particle classes. It provides minimal features for setting a particle's color in its constructor. All other particle behaviors will have to be implemented by you.

You **must** use an inheritance hierarchy for your particle subclasses, following the structure in the class diagram shown above.

The `ParticleBase` class included in the starting scenario provides the following for your use:

`ParticleBase(color)`

A constructor that takes the particle's color (as a `Color` object). The constructor takes care of setting the particle's image to the appropriate color and size.

`getColor()`

A getter method that returns this particle's color as a `Color` object.

Classes You Create

You will create a total of six classes that model particles. Two of these classes represent parent classes in your inheritance hierarchy that capture common behaviors shared by multiple types of particles. The remaining four classes represent specific kind of particles and their unique attributes.

Note the following:

- Pay attention to the named attributes of each class, and whether there is a corresponding getter method and/or constructor parameter--these are indications of when you should **use a field** to represent a value associated with an object. In this assignment, all methods named like getters should be implemented the "standard" way (as a single return statement that returns the value stored in a field).
- Think strategically about the words used in describing each method's behavior. Key words that appear in method names are used strategically. Don't repeat yourself (known as the "DRY" principle)--instead, call other methods rather than duplicating their contents. If you see a behavior mentioned in multiple places, there should be a path to implementing that behavior just once (and calling it), rather than duplicating the code.
- Avoid adding extra constructors beyond those specified. Try to stick to using the exact constructors described in this assignment, and avoid the temptation to add additional constructor parameters or write separate constructors that take different parameters. For beginners, these would be more likely to introduce possible bugs.

A Particle Class

Create a subclass of `ParticleBase` called `Particle` that will serve as the common parent class for all of your other subclasses. This class represents the common properties and behaviors that all particles share.

Every particle has 6 basic attributes: velocity, acceleration, strength, density, dissolvability, and color. Your `Particle` class must store and manage all of these except the color, which is managed by the `ParticleBase` superclass. Your `Particle` class must provide all of the following:

`Particle(color, willDissolve, density)`

A constructor that takes the particle's color (as a `Color` object), a boolean value indicating if the particle will dissolve in acid, and a floating-point double representing the particle's density (in g/cm^3). Remember to add an import statement for `sofia.graphics.Color` (see Lab 09) in order to be able to

reference the `Color` class. A particle's initial velocity should be 0.0, its initial acceleration should be 1.0, and its strength should be 100.

`getDensity()`

A getter method that returns this particle's density (a floating-point value).

`getVelocity()`

A getter method that returns this particle's downward velocity (a floating-point value).

`getAcceleration()`

A getter method that returns this particle's downward acceleration (a floating-point value).

`getStrength()`

A getter method that returns this particle's strength (an integer value).

`willDissolve()`

A getter method that returns tha boolean value indicating whether this particle can be dissolved (i.e., is it reactive).

`weaken()`

This method reduces the particle's strength by 1. If the strength becomes zero, this method should remove this particle from the world.

`isFalling()`

Returns a boolean value indicating whether this particle is in free-fall or not. A particle is falling if it is above the maximum y value and there is nothing in the space immediately underneath it. Be sure not to access the world out of bounds (you cannot fall outside the world).

`fall()`

This method implements the behavior of falling, using a simple model of Newtonian physics. Because of gravity, objects accelerate as they fall. This method should add the particle's current acceleration to its velocity, in order to update its velocity. The velocity represents the number of cells downward this particle will fall in one turn (as a floating-point number). Once the velocity has been updated, use the integer equivalent of the velocity (use a typecast) as the distance to fall. Note: use a loop to fall one cell at a time until the desired distance is reached, so you can stop falling as soon as the particle lands on something (remember, you already have a boolean method to check for this that you can use in a loop condition). Remember to reset the velocity to zero if the particle stops by landing on something.

`swapPlacesIfPossible(x, y)`

This method changes place with another particle at the specified location, if appropriate, and returns a boolean value indicating whether this action succeeded. If the specified location is empty, then this particle can definitely move there. If the specified location is already occupied by another particle, *and*

that other particle has a lower density than this one, then this method causes the two particles to swap places (both particles move). Be sure to ensure that the specified location is within bounds (you cannot swap places to a location outside the world).

`dodge()`

When this particle is not falling (moving straight down through air), it must be "on top of" some other particle, or the bottom edge of the world. In these situations, even though it isn't falling it may still "slide" or "flow" to another position under some circumstances, and the `dodge()` method implements these alternative motions. Simulating the physics of sliding is beyond the complexity of what we want to do in this assignment. Instead, we will use a really simple model:

1. The particle will "sink" by swapping places with what is immediately below it (x + 0, y + 1) if possible.
2. If it can't sink straight down, the particle will swap places with what is down and to the left one cell (x - 1, y + 1) if possible.
3. Finally, if neither of those options are possible, it will swap places with what is down and to the right one cell (x + 1, y + 1) if possible.

The `dodge()` method returns a boolean value indicating whether or not the particle moved.

`act()`

Executes one "turn" for this particle. Each turn, the particle should fall, if it is falling, or dodge otherwise.

A Fluid Class

Create a subclass of `Particle` called `Fluid` that represents liquid particles. It must provide:

`Fluid(color, willDissolve, density)`

A constructor that takes the particle's color (as a `Color` object), a boolean value indicating if the particle will dissolve in acid, and a floating-point double representing the particle's density (in grams per cubic centimeter). The `Fluid` constructor should simply pass these to its superclass' constructor, and this class *should not declare any fields to store these values*.

`dodge()`

This class should provide an overriding definition of `dodge()`. This definition should first apply the inherited behavior. If the inherited behavior does not result in moving the particle, then this version should:

1. The particle will "flow left" by swapping places with what is immediately to its left (x - 1, y + 0) if possible.

2. If it can't flow left, the particle will "flow right" by swapping places with what is immediately to its right (x + 1, y + 0) if possible.

The `dodge()` method returns a boolean value indicating whether or not the particle moved.

Steel

Create a subclass of `Particle` called `Steel` that represents steel particles "attached" to the background. It must provide:

`Steel()`

A default constructor that initializes the particle by providing appropriate parameter values to the superclass constructor. For `Steel` particles, use `Color.gray` as the particle's color. `Steel` particles will dissolve in acid. The density of `Steel` is 7.87 (g/cm^3).

`isFalling()`

Provide an overriding definition of this method that always returns false (just one line). While other particles "fall", since this type of particle is used to form obstacles that we can consider "attached to the background", steel particles don't fall.

`dodge()`

Provide an overriding definition of this method that always returns false (just one line). While steel particles don't "fall", they also don't slide or flow like other particles either, and overriding this method will prevent that behavior.

Sand

Create a subclass of `Particle` called `Sand` that represents sand particles. This class only needs a constructor:

`Sand()`

A default constructor that initializes the particle by providing appropriate parameter values to the superclass constructor. For `Sand` particles, use `Color.khaki` as the particle's color. `Sand` particles will dissolve in acid. The density of `Sand` is 2.5 (g/cm^3).

The `Sand` class does not need to provide any additional methods, since all of the particle behaviors it uses are inherited from its parent class.

Water

Create a subclass of `Fluid` called `Water` that represents water. It must provide:

`Water()`

A default constructor that initializes the particle by providing appropriate parameter values to the superclass constructor. For `Water` particles, use `Color.cadetBlue` as the particle's color. `Water` particles do not dissolve in acid. The density of `Water` is 1.0 (g/cm^3).

The `Water` class does not need to provide any additional methods, since all of the particle behaviors it uses are inherited from its parent class.

Acid

Create a subclass of `Fluid` called `Acid` that represents liquid acid. It must provide:

`Acid()`

A default constructor that initializes the particle by providing appropriate parameter values to the superclass constructor. For `Acid` particles, use `Color.green` as the particle's color. `Acid` particles do not dissolve in acid. The density of `Acid` is 1.0 (g/cm^3).

`dissolveIfPossible(int x, int y)`

This method dissolves another particle at the specified location, if appropriate. If the specified location is empty, then no action happens. If the specified location is occupied by another particle, *and* that other particle will dissolve in acid, then this method causes the other particle to weaken and also causes the current acid particle to weaken, too. Be sure to ensure that the specified location is within bounds (you cannot swap places to a location outside the world).

`act()`

Provide an overriding definition of this method that first dissolves adjacent particles to the left (x - 1, y + 0), to the right (x + 1, y + 0), or above (x + 0, y - 1), or below (x + 0, y + 1), before performing the other aspects of acting that it inherits from its superclass. Remember that when a particle weakens, it might be removed from the world, and once this acid particle is removed from the world, it should not continue to act on other particles or try to move further.

Comments on Design

As in other assignments, you will be graded in part on the design and readability of your solution using the posted grading criteria, so consider these factors when devising and naming your methods. The **Program Grading Rubric** (same as on Programs 1-3) describes the grading criteria. Note that a portion of your grade will be based on your approach to testing your solution.

For testing, remember that you can create very small, completely empty worlds (even as small as 2x2, or 3x4, etc.), and place exactly the particles you want at exactly the positions you want in order to test behaviors. Some classes are really easy to test, because you only need to test the constructor. In other cases, focus on testing methods one at a time, instead of trying to test everything by calling `act()`. Constructing simple, tiny situations to test out your behaviors is much cleaner and easier than trying to "run" a big simulation with a large number of particles spread around.

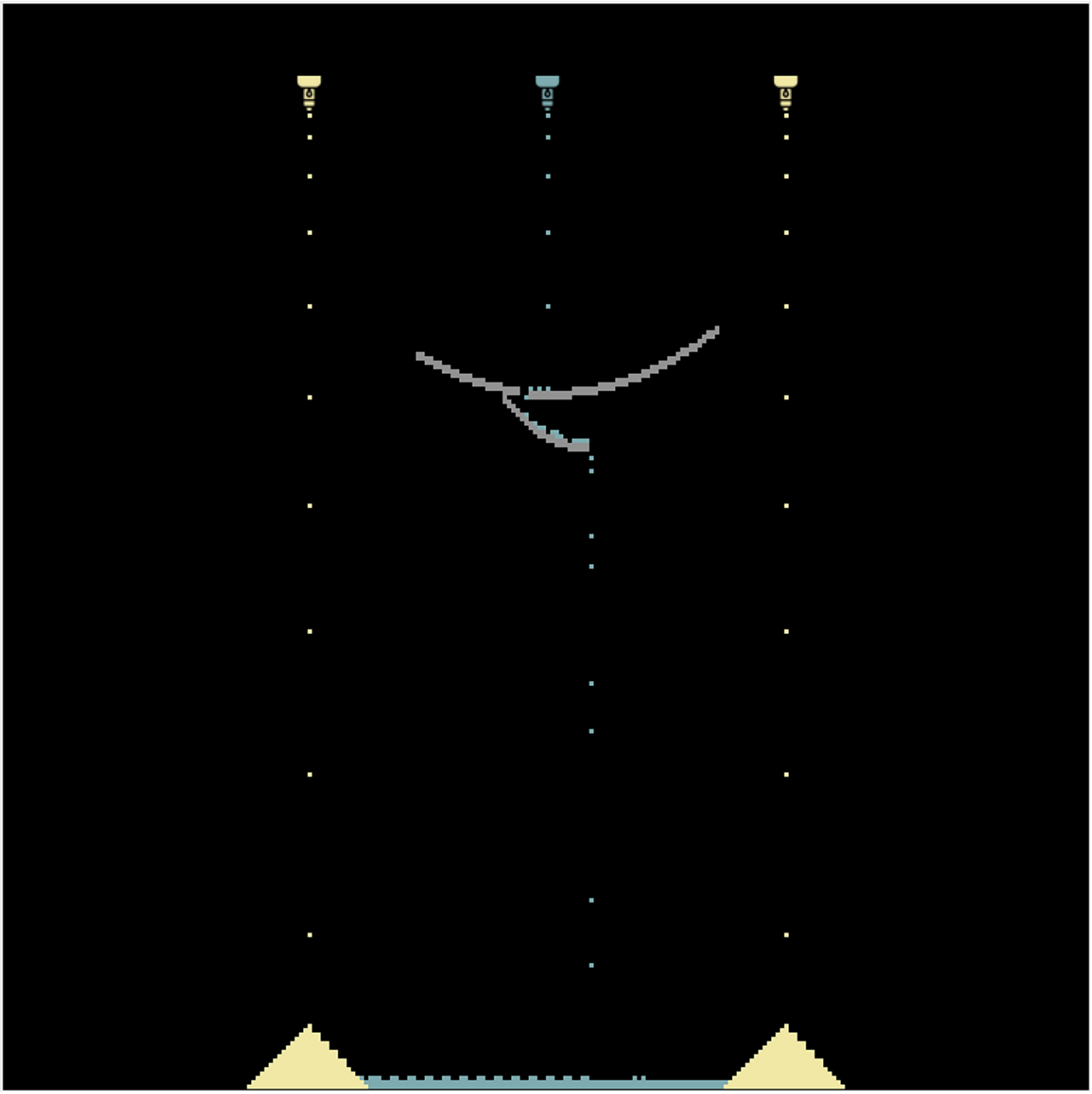
Also, think carefully about writing tests for adverse situations or unexpected situations. What if a particle that is falling "hits" the bottom border? What if a particle that can flow sideways is next to the left or right edge of the world? Will a particle that is falling fast enough "skip over" a steel support some distance below it? Think about "extreme" situations for each method you are testing, and write tests for those situations, rather than only focusing on the simple/easy situations.

Finally, there are plenty of aspects of this simulation that are over-simplified or not physically accurate. Don't worry about those inaccuracies, though--addressing them is possible, but makes the job more challenging and complicated. However, if you see places where more realism might be obtained, feel free to post about it on Piazza and discuss what you've observed and your thoughts about what would be more realistic. You might come up with an even better model!

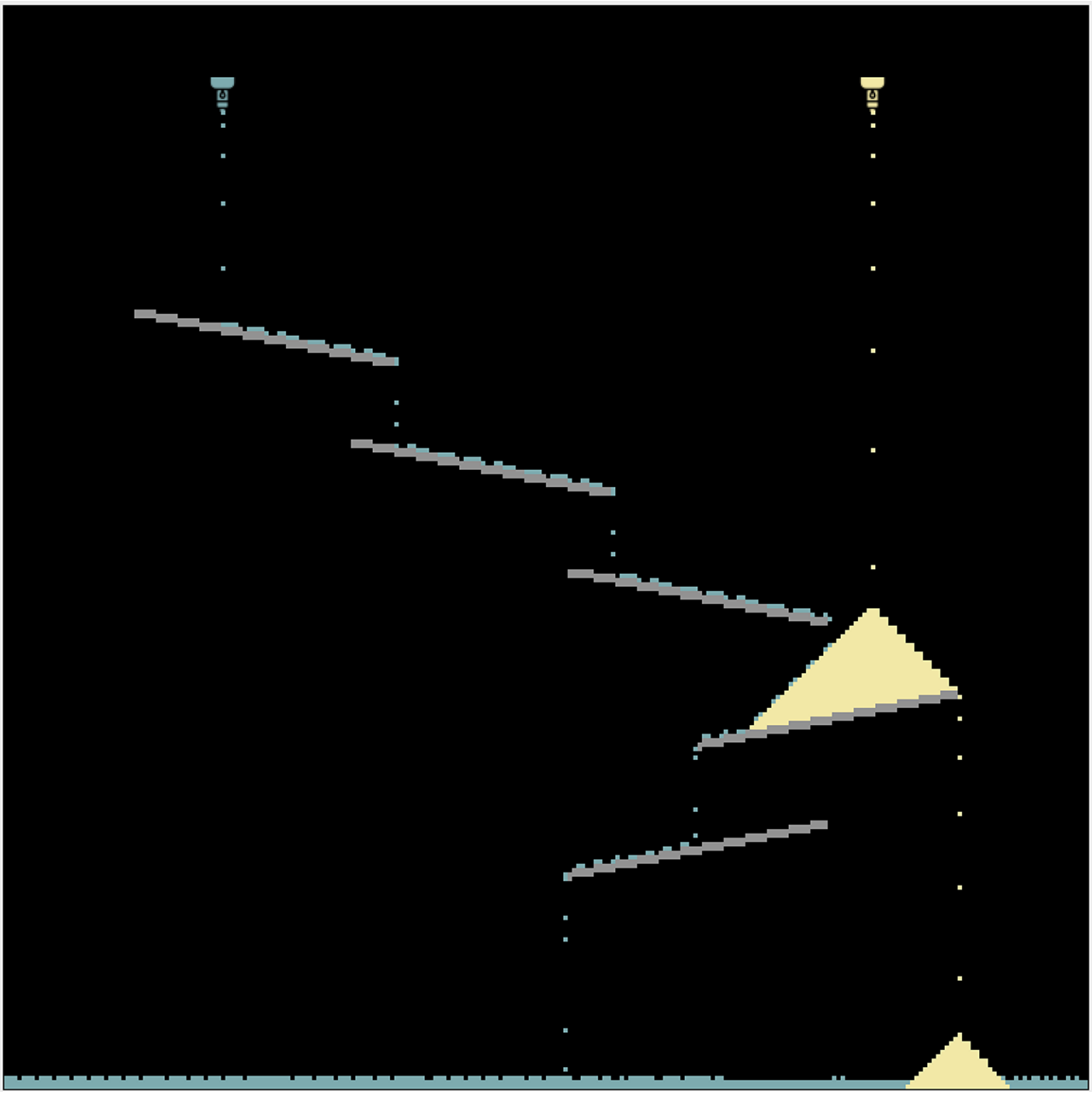
Trying Out Different Layouts

As you develop your solution, you can use the "Run" button to see how your particles behave. By pausing the simulation, you can click and drag to reposition the droppers. You can also right-click on droppers and use their methods to turn them on or off, and can even remove them entirely. You can also move around individual particles the same way (remember they are very small!).

Because the particles are so small, building a completely new screen arrangement with ramps and other shapes can be tedious. However, by right-clicking on the `ParticleWorld` base class, you can create a starting layout from one of 3 predefined arrangements. Layout 0 is the arrangement shown at the top of this page. In addition, you can use Layout 1:



Or you can use Layout 2:



Submitting Your Solution

All program assignments are submitted to Web-CAT. Use the Controls->Submit... menu command to submit your work.

This tool needs to be loaded in a new browser window

Load Program 4 in a new window

A.4 Project Group 4—Arrays and Maps

For this project topic, the focus was managing a two-dimensional array space. In Fall 2015, this was done in the context of implementing a very minimal Asteroids game. In Fall 2017 and Spring 2019, students were tasked with implementing a tool for drought detection.

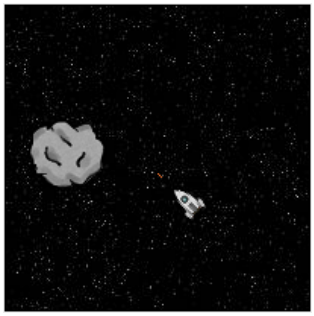
A.4.1 Fall 2015 Project 5—Asteroids

Program 5

Program 5: Asteroids

Goal

In this program assignment, you will implement classes for a *very* minimalist version of the classic video game called *Asteroids*. While the requirements for this assignment are very basic, you also have the freedom to adapt or expand on the basic ideas presented here in many ways if you would like to customize the game.



Note: For simplicity, we will not be implementing the full physics of the original *Asteroids* game. Instead, we will use a simplified physics approach that is much simpler to program. If you're interested in comparing your result here to the original, there are several playable versions of *Asteroids* available on-line, including this one.

In the variant of *Asteroids* you will implement, there are only three kinds of objects: your ship, the bullets your ship fires, and the asteroids. All of these objects move in straight lines, although you can steer your ship and control its speed.

Structuring Your Solution

Your task is to write the four main classes: `Ship`, `Bullet`, `Asteroid`, and `Space` and unit tests for all of the classes. You may write additional classes to structure your solution if you desire. **Hint**: consider consolidating common features of classes into one or more superclasses of your own design. **You will be graded in part on how well you have used this feature in your design**.

As in more recent assignments, here we will use a world with a cell size of *1 pixel by 1 pixel*. In Greenfoot, actors by default have a size determined by their image, so the actors in this program will be much larger than a single cell. Actor locations--that is, cell coordinates--will be in pixel sizes here. Actors are drawn so that their image is centered on their location. Also, the `getIntersectingObjects()` method takes into account the Actor's physical size, not just its location, so it works as you would expect for actors that are larger than a single cell in size.

The Space Class

The `Space` class represents the world for your game. **It must provide two constructors: one is the default constructor with no parameter, and another is a constructor that**

takes two parameters, a width and a height. Both constructors should ensure that the world is created with a cell size of 1 pixel square. At first write the parameterized constructor which takes two parameters, width and height and create a world with given width, height, and a cell size of 1 pixel. Then write the default constructor. For the default constructor, you pick your own default world size--you might start with a size like 500x500 pixels, and then adjust the size larger or smaller so that it looks good on your screen. Inside the default constructor, call the parameterized constructor to pass a default width and height.

For the `Space` image, you can use the `backgrounds/space1.jpg` image provided in Greenfoot's image library, or you can import another image from a file to customize the look of your game (use the "Import from file..." button when setting the image, and provide a PNG, JPG, or GIF image that you have found on the web or created yourself--even a photo will work).

Finally, your `Space` class should provide a method called `populate()` method that creates **one ship** in the center of the world facing north, and creates **five randomly placed asteroids**, each of which has a random speed between 1-5, and a random orientation from 0-359. Update your default constructor so that it calls `populate()` method to set up the world in a ready-to-play configuration. Call the `populate()` method only from the default constructor, not from the parameterized constructor.

The Asteroid Class

The `Asteroid` class represents the "hazard" in your game. It is simply a rock floating through space. **This class must provide a constructor that takes two integer parameters: a speed (in pixels per turn), and a rotation (in degrees)**.

The `Asteroid` class should provide a method called `getSpeed()` that returns the asteroid's speed. Note that the `Actor` class already provides a getter and setter for the object's rotation.

For the `Asteroid` image, you can download and import the image below, or you can import another image from a file to customize the look of your game:

The behavior of an `Asteroid` is simple: it moves ahead on each call to `act()` the distance specified by its first constructor parameter (its speed). Remember that the `Actor` class provides a `move(int)` method that moves a specified distance forward in the direction the actor is currently facing.

There is one extra twist to moving an asteroid--if, after moving the asteroid forward, it is located on any edge of the world, it should "wrap around" to the other side. In other words, if after moving, the asteroid ends up on the northern edge of space, it should be transported to the corresponding spot on the southern edge, as if it flew off the top edge and re-entered the world on the bottom edge. Implement the same processing for the left/right edges of the world as well.

Finally, if the asteroid collides with a ship or bullet, both the asteroid and the ship should be removed from the world (but asteroids harmlessly pass through each other :-)).

The Bullet Class

The `Bullet` class represents the shots your ship fires at the asteroids. It is simply a projectile flying through space. **This class must provide a constructor that takes two integer parameters: a speed (in pixels per turn), and a rotation (in degrees)**.

The `Bullet` class should provide a method called `getSpeed()` that returns the bullet's speed. Note that the `Actor` class already provides a getter and setter for the object's rotation.

For the `Bullet` image, you can download and import the image below, or you can import another image from a file to customize the look of your game:

 <-- It's a really small red streak right there! (The border is being drawn by HTML to make it easier to see. It is not part of the image.)

The behavior of a `Bullet` is simple: it moves ahead on each call to `act()` the distance specified by its first constructor parameter (its speed). When a bullet reaches the edge of the world, it is removed from the world--bullets do not "wrap around" like asteroids do. Also, if the bullet collides with an asteroid, both should be removed from the world.

The Ship Class

The `Ship` class represents your ship in space. Unlike the other objects, it is steerable using an on-screen thumbstick (on an Android device) or the keyboard (on a PC). Like the other objects, it has both a speed and a rotation. It must provide a `getSpeed()` method to get its current speed, but can use the existing getter and setter for rotation that it inherits from the `Actor` class.

The `Ship` class must provide a default constructor (means a constructor with no parameter). A ship always starts with a speed of zero, facing directly north (a rotation of -90 degrees).

Movement of the ship is just like for asteroids, with "wrap around" at the borders of the world.

Unlike the other actors, however, the ship is controlled by your actions. The thumbstick, or *directional pad (D-pad)*, allows you to use your finger to touch on the north, south, east, or west sides of a thumb control on-screen. On your PC's keyboard, you can use the arrow keys, or the W/A/S/D keys, which Greenfoot4Sofia treats the same way as the on-screen D-pad on an Android device.

Implement the following methods in your `Ship` class to respond to D-pad actions:

- `public void dpadNorthIsDown()`
- `public void dpadEastIsDown()`
- `public void dpadSouthIsDown()`
- `public void dpadWestIsDown()`

Implement your ship so that it turns (rotates) 5 degrees left or right if the west or east side of the D-pad is down. Your ship should also speed up by 1 (accelerate) if the north side is pressed, or slow down by 1 (decelerate) if the south side is pressed. Be sure not to allow your ship's speed to become negative.

Finally, to shoot, your ship must create a new `Bullet`. For simplicity, create the bullet at the same location as the ship, with the same rotation as the ship, and a speed of 10. Do this if the user taps/touches the screen (or, on a PC, clicks the mouse). Screen touches cause the following method to be invoked (if it exists):

- `public void onScreenTouchDown()`

For the `Ship` image, you can download and import the image below, or you can import another image from a file to customize the look of your game:

Submitting Your Solution

All program assignments are submitted to Web-CAT. Use the Controls->Submit... menu command to submit your work. The due date for this assignment is June 30, 11.55 pm. Please note that, no late submission will be accepted for this assignment.

For Fun (no bonus points)

After you complete the requirements described above and are happy with your Web-CAT submission, you may wish to add other features to your solution. There are many ways you can do this. Here are some possibilities you might want to think about:

- You may notice that using integer coordinates (and distances and speeds) in your program produces a slightly "jerky" movement pattern for your ship. Greenfoot4Sofia allows you to use floating point coordinates and distances for actors. If you change your ship to use full floating point behaviors everywhere, you will get smoother movement.
- How would you implement the physics model of the original game, where ship orientation and velocity were independent of each other? In the original game, the ship could only accelerate, not decelerate. To slow down, you had to point the ship in the opposite direction and fire your engines to apply thrust opposite to your current direction.
- In the original `Asteroids`, the asteroids came in different sizes. When a bullet hit a larger asteroid, it "broke apart" into smaller pieces that flew in random directions with random speeds. You might try adding three sizes of asteroids. You can even limit larger asteroids to lower speeds, allowing smaller ones to take on higher speeds.
- It is possible to add a new type of object that represents an explosion, and "create" one when a bullet collides with an asteroid (or an asteroid collides with the ship), having it remove itself from the world after a certain number of turns.
- Consider how you would keep score, and how you would indicate the score to the player.
- Video games often do not allow "unlimited" firing speed for the player, and instead impose a "reload" time. Think about how you might add such a restriction to the bullet firing of the ship.
- How would you add a UFO that shoots back at the ship?
- There are many possible visible enhancements as well to make the game more interesting or attractive.
- What other modifications can you think of?

A.4.2 Fall 2017 & Spring 2019 Project 6—Drought Detection

Program 6

A reminder from the syllabus about the Honor Code and your work on this assignment:

> You may freely offer and receive verbal assistance on how to use the programming language, what library classes or methods do, what errors mean, or how to interpret assignment instructions with your classmates. However, you **may not give or receive help from others while working on your program code**. Further, on individual program assignments you *must work alone while typing at the keyboard, or viewing your source code on the screen*. It is OK to work in the Undergraduate Learning Center (McBryde 106), as long as you are not asking for or receiving assistance from others (except course TAs or approved tutors).

That means **it is an Honor Code violation to show your program code to a fellow student at any time**. If your source code is visible on the screen, no other students should be looking at it.

- It is an Honor Code violation to resubmit prior work from an earlier semester in this course.
- It is an Honor Code violation to submit work authored by someone else as your own.

For all assignments, you **must** include the following Virginia Tech Honor Code pledge at the very top of each Java file you write (separate from the class Javadoc comment). Include your name and PID as shown, to serve as your signature:

```
// Virginia Tech Honor Code Pledge:
//
// As a Hokie, I will conduct myself with honor and integrity at all times.
// I will not lie, cheat, or steal, nor will I accept the actions of those
// who do.
// -- Your Name (pid)
```

Drought Detection

Goal

Climate change affects our weather, which in turn impacts many people and industries in different ways. People who are particularly susceptible to weather changes can use technology to help monitor and predict weather situations that can negatively affect their businesses or livelihoods. In this lab, we are going to look at the beginnings of how you can write programs to analyze weather data to compute basic statistics--numerical summaries that are the foundation of predictive models.

Here, we are going to focus on rainfall and drought conditions, which influence many communities in different ways. You may have read about recent drought situations in different areas of this country, or on other continents. But while we all have an idea of what a drought is, it can actually be hard to measure. While droughts are caused by complex and interrelated issues, one of the main underlying forces is the availability of water. Lack of rainfall naturally reduces the amount of water available in an area, which can lead to meteorological drought, the first and most direct kind of drought (see NOAA's **Definition of Drought (https://www.ncdc.noaa.gov/monitoring-references/dyk/drought-definition)**). In this lab, we are going to look at calculating monthly rainfall averages, which is one step in identifying meteorological drought.

One reason to do this is because some communities, particularly farming communities, can be significantly affected by drought, and farmers wish to monitor signs of impending drought conditions. One of the main indexes used to identify (or predict) drought compares accumulated rainfall amounts against historical averages. This information matters in many places in the world.

For example, in East Africa, many countries (and many families) depend heavily on agriculture and might see huge impacts from drought. That region includes many farms that produce coffee and tea--in fact, Kenya is the world's third biggest exporter of tea (behind China and India). However, the tea industry in this region includes a significant number of small, independently operated farms. These small farms operate season to season, and do not have the financial resources to survive the damage that can occur if an entire season's crops are destroyed. For individuals operating these farms, their lives depend on the weather.

Read the following two (brief) articles for more background:

- **East Africa Drought Affects Tea Production (http://worldteanews.com/news/east-africa-drought-affects-tea-production)**
- **Adverse weather conditions hurting tea farmers (https://www.standardmedia.co.ke/article/2001230909/adverse-weather-conditions-hurting-tea-farmers)**

While working on this assignment, think about how a drought where you live would affect you and your family. Also think about how it might affect a poor, family-owned tea farm in Kenya, and what would be similar to or different from the local affects in the area where you live. Finally, think about what would be necessary to beyond the simple data reading/processing in this assignment to create a practical tool for drought identification.

Starting Materials

This assignment does not include a downloadable zip file. You will create all classes from scratch yourself.

Real Weather Data

Your assignment is driven by real-world weather data collected from weather observation stations. It is possible to get this information from various sources online, including NOAA, the National Oceanographic and Atmospheric Administration (see their **Climate Data Online (https://www.ncdc.noaa.gov/cdo-web/)** web site). Here, we will be using some data collected from actual weather stations in Kenya. From NOAA's data sources, we've compiled **daily summary information** from weather station observations into a text format for you to use. The text file looks like this (but a lot longer):

```
KE000063612     3.117   35.617  515     2/10/16 0.04    87      98      73
KE000063820     -4.033  39.617  55      4/25/16 0       88      101     75
...
```

Each line in this text file represents the information from a single day at a single weather station. Each "word" represents a separate piece of information. You'll be reading this input using a `Scanner` so you can use `next()`, `nextInt()`, or `nextDouble()` to read individual data values from the line.

Each line contains 9 separate values, with whitespace in between. These are:

Name	Type	Example	Meaning
Station ID	text	KE000063612	The weather station's identifier (a unique name)
Latitude	floating point	3.117	The latitude of the weather station's location
Longitude	floating point	35.617	The longitude of the weather station's location
Altitude	integer	515	The elevation at the weather station's location (in feet)
Date	text	4/25/16	The date for this daily summary, in m/d/yy format
Precipitation	floating point	0.04	The total rainfall for that day
Avg Temp	integer	87	The average (mean) temperature for that day (in degrees Fahrenheit)
Max Temp	integer	98	The high temperature for that day
Min Temp	integer	73	The low temperature for that day

Note that some records **use -1 as a value** when data is missing in the precipitation or temperature fields. Such -1 values should be ignored in those positions.

Of these, in this assignment we will only use: the station ID, the month (which you can extract from the date using `substring()`), and the precipitation amount. The other values may be relevant in a more

advanced drought identification process (for example, temperature information allows one to account for evaporation rate, which allows for more accurate models than those using rainfall alone).

At the same time, though, you can rely on all 9 values being present on every line. That may simplify the process of reading data from the input source.

Two example weather data files you can look at are:

- **Kenya-short.txt (http://courses.cs.vt.edu/~cs1114/Kenya-short.txt)** : a really small file containing 5 weather summaries from 2 different weather stations in Kenya. Useful for looking at the format, or for simple software tests.
- **Kenya-2014-2016.txt (http://courses.cs.vt.edu/~cs1114/Kenya-2014-2016.txt)** : a full file containing all available weather summaries from all weather stations in Kenya for 2014-2016.

However, when creating software tests for your solution, you can create your own test records directly inside your test cases, without using separate files (see the **Java I/O tutorial** for how to set up a Scanner with input text for testing).

Classes You Create

For this assignment, you will create three classes.

WeatherStation

This class represents the basic statistics collected by one weather observation station. Internally, a weather station should use an array to hold 12 monthly rain totals that represent the sum of precipitation numbers for all the days reported in that month. It should also use a separate array to hold 12 monthly counts that represent the number of daily records that have been processed for that month. **Note that month numbers are passed into methods and returned as values between 1-12, even though your arrays are indexed starting at zero**. Plan accordingly.

This class should provide the following methods:

Method	Purpose
`WeatherStation(identifier)`	This constructor initializes the internal data for the weather station. The identifier that is provided is the Station ID for this weather station (i.e., a unique name). Initially, the weather station does not know about any daily summary reports (i.e., all counts and totals are zero).
`String getId()`	Returns the weather station ID for this weather station.
`recordDailyRain(month, rainfall)`	Record the information from one daily summary line in a data file, which adds the rainfall (a double) to the month (an integer from 1-12 indicating the month of the daily report).

Method	Purpose
`int getCountForMonth(month)`	Returns the number of daily rainfall values that have been recorded for the specified month (a number 1-12). Return zero when no values have been recorded for the specified month.
`double getAvgForMonth(month)`	Returns the average daily rainfall for the specified month (a number 1-12). This is the total rainfall across all reported daily values for that month, divided by the number of daily values that have been recorded for that month. Return -1 if no rainfall amounts have been recorded for the specified month.
`int getLowestMonth()`	Returns the number of the month (a number 1-12) indicating the month that had the lowest average rainfall recorded at this station. If multiple months have the same lowest rainfall average, return the earliest one (the lowest month number). If no rainfall records have been entered for any month, return the earliest month as well (1).

WeatherBureau

This class represents a weather service that keeps track of all the weather stations. Internally, it should use a map to associate weather station IDs with weather station objects. It should provide the following methods:

Method	Purpose
`WeatherBureau()`	This default constructor simply initializes the internal data of the class. A new weather bureau does not yet know about any weather stations.
`recordDailySummary(text)`	Takes a single string representing a single one-line daily weather summary for one day at one weather station (i.e., one line from a weather data input sources). See the "Real Weather Data" section above for information about what is contained on one line. If the rainfall amount in the text is -1, ignore the line (it is missing its rainfall data). Otherwise, record the rainfall from this daily summary in the corresponding weather station object in the bureau's map (create a new weather station object if the station ID doesn't correspond to one you've seen before).
`recordDailySummaries(input)`	Takes a `Scanner` object as a parameter that represents an input data source, such as a file containing a series of daily summary records for one or more weather stations. Record all of the daily summaries from the input source.
`WeatherStation getStation(identifier)`	Return the weather station object for the given weather station ID (or null, if the identifier doesn't match any weather station you've seen so far).
`WeatherStation lowestStation(month)`	Returns the weather station that has the lowest average rainfall for the specified month (or null, if the weather bureau hasn't recorded *any* rainfall daily summaries for *any* station in the specified month).

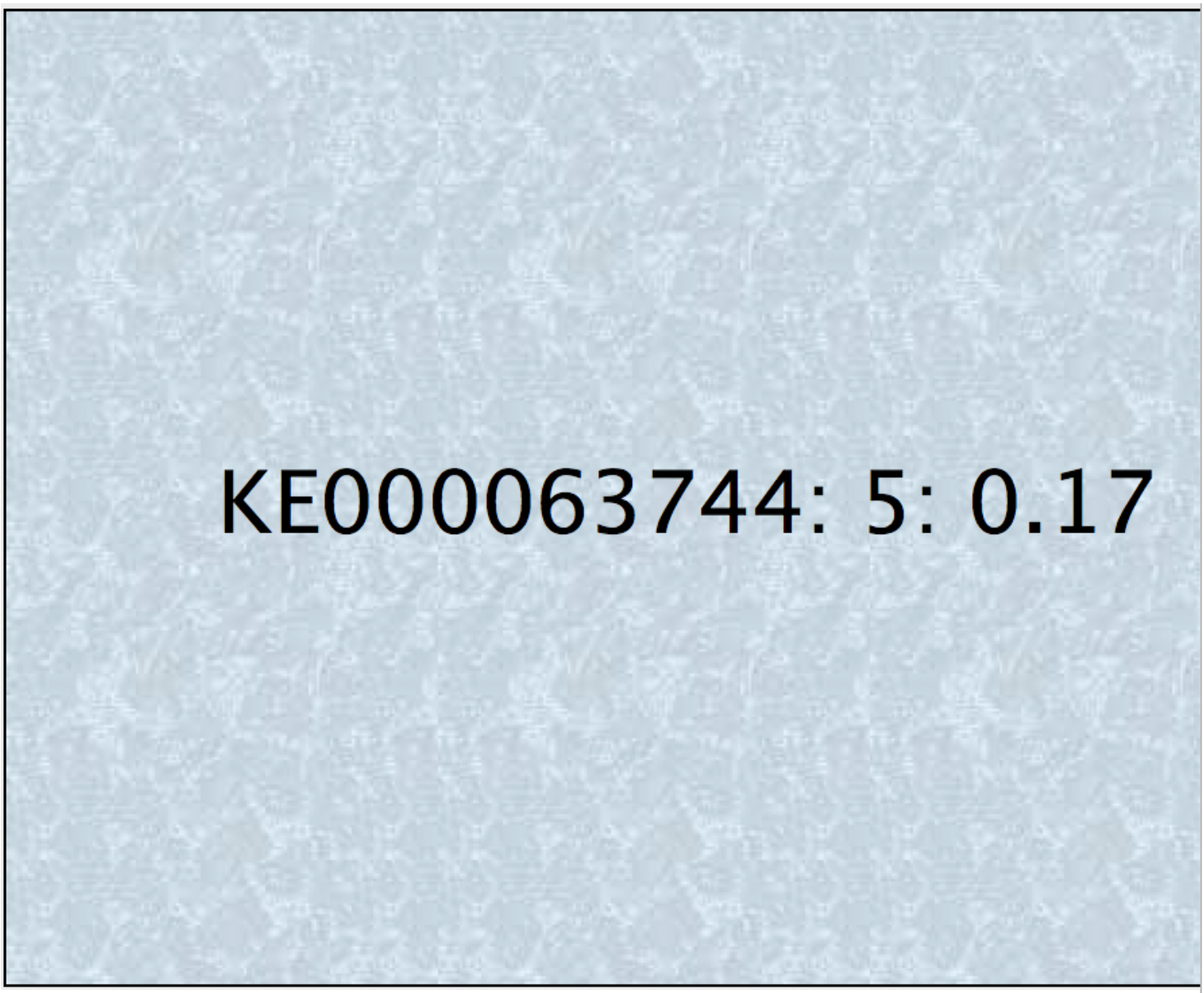
Method	Purpose
`WeatherStation lowestStation()`	Returns the weather station that has the lowest average rainfall recorded for *any* month (1-12) (or null, if the weather bureau hasn't recorded *any* rainfall daily summaries for *any* station for *any* month). If multiple stations have the same lowest average rainfall amount, just return the first one you find (of those that tied for lowest). Remember, you can use the `getLowestMonth()` method on each station to find out which month has its specific lowest average value.

RainfallAnalyzer

The `RainfallAnalyzer` is a subclass of `World`. It is very simple, since it's purpose is to parse some weather data and then display on the screen the weather station with the lowest rainfall using a process like this:

- Create a new weather bureau object.
- Use the weather bureau object to record all the daily summaries in a data source provided as a `Scanner`.
- Use the weather bureau object to find the weather station with the lowest average monthly rainfall.
- Create a `TextShape` and add it (approximately) to the center of the world (see **Chapter 9 (http://sofia.cs.vt.edu/cs1114-ebooklet/chapter9.html)** of the online ebooklet).
- Set the text displayed in the `TextShape` to show the weather station ID of the weather station with the lowest rainfall, the month number (1-12) of the lowest rainfall, and the average amount of that month's rainfall at that station. If no such station exists (because of lack of data), display "no data" instead.
- The rainfall average should be shown with only two significant digits after the decimal (use the `String.format(...)` method; read **Java String format() method explained with examples (https://beginnersbook.com/2017/10/java-string-format-method/)**).
- All these actions should be placed in one constructor (not in the `act()` method).
- You may want to look back at the lab assignment you did involving reading text from a `Scanner` to see how the pair of constructors were handed there.

The resulting display will look something like this (you can pick your own background, font size, and font color; centering does not need to be exact; and the actual text will be different than this example):



The `RainfallAnalyzer` class must provide the following methods:

Method	Purpose
`RainfallAnalyzer(`*input*`)`	This constructor takes a `Scanner` as a parameter. It should initialize the world using 72x72 pixel grid cells arranged in 5 rows by 8 columns, and perform all the actions in the description above for setting up its own weather bureau, recording all the data from the `Scanner` using the appropriate method on the weather bureau, and so on, including the display of the `TextShape` with the appropriate contents.
`RainfallAnalyzer()`	The default constructor should simply create a `Scanner` that reads from **http://courses.cs.vt.edu/~cs1114/Kenya-2014-2016.txt (http://courses.cs.vt.edu/~cs1114/Kenya-2014-2016.txt)**, and call the first constructor to perform the remainder of the initialization (see **Lab 12**).
`WeatherBureau getBureau()`	This getter returns the weather bureau created inside this object to hold all the weather data.

Method	Purpose
`TextShape getText()`	This getter returns the text shape created in the constructor to display the weather station information for the station with the lowest average rainfall.
`WeatherStation getStation()`	This getter returns the weather station with the lowest rainfall that is currently displayed (the one calculated in the constructor).

Notes on Testing

As with prior assignments, you must write software tests for all of your classes. Remember the following when writing your tests (and remember you'll have to customize any code snippets provided below using your own field names and variable names (or declarations) in order for them to work in your own code--you can't just cut-and-paste without thinking about how to adapt to your own situation):

1. Simple getter methods that just return a field value without doing anything else don't need to be tested. You only need to test them if they do anything more than just returning a field.

2. When testing methods that return objects you are defining yourself (such as `getStation()` or similar methods), use `assertEquals()` on *attributes* of the object that are important (such as the station ID), instead of trying to compare the object directly. None of your own classes define an `equals()` method, so using `assertEquals()` on a whole weather station object (or any other class you define yourself in this assignment) isn't appropriate.

3. When checking floating point values, you **must** use the three-argument version of `assertEquals()`:

   ```
   assertEquals(0.27, station.getAvgForMonth(3), 0.001);
   ```

   The third argument specifies the *tolerance*: how close the two values must be to be considered equal. The **tolerance must always be specified when comparing floating point values** using `assertEquals()`, so don't ever leave it out.

4. `TextShape` objects provide a `getText()` method that simply returns their contents. You can use this when testing.

5. When writing test cases that use simple string inputs or outputs, just place the string directly in your code:

   ```
   public void testSomething()
   {
       String line = "KE000063612 3.117 35.617 515 2/10/16 0.04 87 98 73";
       bureau.recordDailySummary(line);
       // place your assertions here
   }
   ```

6. When writing test cases that use `Scanners`, you can use `setIn()` and `in()` (the **Java I/O tutorial** describes these):

```
public void testSomething()
{
    setIn(
        "KE000063612 3.117 35.617 515 2/10/16 0.04 87 98 73\n"
        + "KE000063820  -4.033  39.617  55      4/25/16 0       88      101     75\n"
        + ...
        // put as many lines as you want; use \n as the newline marker in each
        );
    bureau.recordDailySummaries(in());
    // place your assertions here
}
```

The `in()` method refers to a built-in scanner you can use in each test, and you use the `setIn()` method to set its text contents.

Focus on writing very small tests that just use a couple of lines of data input, so you can control what is happening. **Do not** write all your tests using the default constructor that reads the full 3-year dataset from the web--you don't have any way of knowing what the expected values should be for averages, lowest, etc., and will not be able to write tests that way that actually check for bugs. Make up your own rainfall amounts and/or month numbers and use them in just a few data lines so you determine for yourself what average (or lowest, or whatever) values you want. You can make up your own station IDs in your tests also, since they are just unique strings.

7. Make sure you write tests for "edge" cases (that is, situations that are at the very outer limits of what is allowable). Some examples to consider include: -1 values in precipitation entries; station IDs that have not been used, so aren't present in the bureau; calculating the average when no entries have been recorded for a given month; finding the lowest rainfall month (or station) when all of them have the same value; finding the average when only zeroes have been recorded; using months at the limits of acceptable values (such as 1 and 12); and so on. For each method that involves any realistic logic, try to write tests that use any possible missing values you can imagine (or any possible minimum values, if missing is not allowed) to make sure you are handling them in your solution.

Submitting Your Solution

All program assignments are submitted to Web-CAT. Use the Controls->Submit... menu command to submit your work.

Appendix B

Student Survey

This survey was administered to the consenting students in CS 1114 in Fall 2017 and Spring 2018 semesters. The questions were distributed by Qualtrics survey and based on the Davis' Technology Acceptance Model. [31][27]

The students were asked to respond to following questions on a Likert-scale from one to five, with one meaning "strongly agree", two as "agree", three meaning "neutral", four as "disagree" and five meaning "strongly disagree".

These questions assess whether the student perceived the presence of the heat maps.

- P1) I remember seeing heat maps on Web-CAT for some programming assignments.
- P2) I always looked at the heat maps for my assignment when they were available.

These questions assess perceived usefulness.

- U1) Using the heat maps in my assignments enables me to complete my assignments more quickly.
- U2) Using the heat maps improves my class performance.
- U3) Using the heat maps enhances my effectiveness in the class.
- U4) Using the heat maps in my assignments increases my productivity.

- U5) Using the heat maps makes it easier to complete my assignments.
- U6) I find the heat maps useful in my assignments.
- U7) I find the heat maps are frequently incorrect in identifying the locations of possible bugs.
- U8) I find the heat maps to be helpful in finding bugs in my work.

These questions assess perceived ease of use, using the same seven-point scale.

- E1) Learning to use heat maps was easy for me.
- E2) I did not know how to interpret the heat maps.
- E3) The heat maps were confusing.
- E4) It was easy to become skillful at using the heat maps.
- E5) I found heat maps easy to use.
- E6) The information in the heat maps was clear and understandable.

These questions capture student demographic information.

- Your academic level
- Your Major? Minor?
- Anticipated Grade in this course? (optional)
- Gender (optional)
- Describe your level of experience programming before taking this course (This was a free text response).

In order to analyze the level of experience question, the responses were categorized based on the following rubric:

- Novice : I need help when programming
- Intermediate : I am able to independently complete programming tasks, but require help from time to time
- Advanced : I have attained a level of self-efficacy with programming and are recognized as "a person to ask" when difficult questions arise
- Expert : I am recognized as an authority and can provide guidance, troubleshoot, and answer questions regarding programming.

The results of the survey are presented in Figure B.1 .

The students were provided an opportunity to submit free text comments. These are the comments that students entered. Some are comments about Web-CAT in general:

- "Teach how to use feedback in class"
- "I feel like the heat maps often had code covered in red, when there was nothing wrong. Or, more commonly, lots of the code would be in yellow despite only a tiny portion being wrong."
- "It was sometimes difficult to determine what exactly to fix. Still somewhat vague."
- "I know with some of my program submissions, I had unnecessary if-statements, therefore if webcat can somehow show areas where you can condense coding that would be helpful."

- "No hidden tests, just tell us what's wrong. Being clearer with failures would be helpful as well. There were many times I was missing points for problem coverage and the descriptions were super vague. It was frustrating."
- "The biggest problem is that it refuses to tell you what you did wrong in terms of coverage if you don't have complete testing. Ideally, it would tell you everything you did wrong so that you could decide what you wanted to prioritize. This was a huge problem for me when I had a lot of coverage issues but only one testing issue that I couldn't fix. "

Figure B.1: Summary of student responses to survey administered via Qualtrax to CS 1114 students about the usefulness of the heat maps.