Intelligent Goal-Oriented Feedback for Java Programming Assignments

Nischel Kandru

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

Master of Science
in
Computer Science and Applications

Stephen H Edwards, Chair

Donald Scott McCrickard

Francisco Javier Servant Cortes

23 May 2018

Blacksburg, Virginia 24061

Keywords: Goal Formulation, Web-CAT, Prioritized Errors, Summary

Intelligent Goal-Oriented Feedback for Java Programming Assignments

Nischel Kandru

(ABSTRACT)

Within computer science education, goal-oriented feedback motivates beginners to be engaged in learning programming. As the number of students increases, it is challenging for teaching assistants to cater to all the doubts of students and provide goals. This problem is addressed by intelligent visual feedback which guides beginners formulate effective goals to resolve all the errors they would incur while solving a programming assignment.

Most current automated feedback mechanisms provide feedback without categorization, prioritization, or goal formulation in mind. Students may overlook important issues, and high priority issues might be hidden among other issues. Also, beginners are not well equipped in formulating goals to resolve the issues provided in the feedback.

In this research, we address the problem of providing an effective, intelligent goal-oriented feedback to student's code to resolve all the issues in their code while ensuring that the code is well tested. The goal-oriented feedback would eventually implicitly navigate the students to write a logically correct solution. The code feedback is summarized into four categories in the descending order of priority: Coding, Student's Testing, Behavior, and Style. Each category is further classified into subcategories, and a simple visual summary of the student's code is also provided.

Each of the above-mentioned categories has detailed feedback on each error in that category to provide a better understanding of the errors. We also offer enhanced error messages and diagnosis of errors to make the feedback very useful.

This intelligent feedback has been integrated into Web-CAT, an open-source automated grading tool developed at Virginia Tech that is widely used by many universities. A user survey was collected after the students have utilized this

feedback for a couple of programming assignments and we obtained promising results to claim that our intelligent feedback is effective.

Intelligent Goal-Oriented Feedback for Java Programming Assignments

Nischel Kandru

(GENERAL AUDIENCE ABSTRACT)

Within computer science education, goal-oriented feedback motivates beginners to be engaged in learning programming. As the number of students increases, it is challenging for teaching assistants to cater to all the doubts of students and provide goals. This problem is addressed by intelligent visual feedback which guides beginners formulate effective goals to resolve all the issues they would incur while programming.

Most current automated feedback mechanisms provide feedback without categorization, prioritization, or goal formulation in mind. Students may overlook important issues, and high priority issues might be hidden among other issues. Also, beginners are not well equipped in formulating goals to resolve the issues provided in the feedback.

In this research, we address the problem of providing an effective, intelligent goal-oriented feedback to student's code to resolve all the issues in their code. The goal-oriented feedback would eventually implicitly navigate the students to write a logically correct solution. The code feedback is modularized smartly to guide students to understand the issues easily.

A simple visual summary of the student's code is also provided to help students obtain an overview of the issues in their code. We also offer detailed feedback on each error along with enhanced error messages and diagnosis of errors to make the feedback very effective.

Acknowledgments

I would like to acknowledge a lot of people for their immense contributions to my Master's life. It would not have been so hassle-free without all of them.

First and foremost, I would like to thank Dr. Stephen Edwards for being a great advisor/mentor throughout my Master's journey. He is one of the smartest and nicest professors that I have ever met. He is so humble and easy going that I have never felt that I had any strict deadlines to meet. He was always there for me, whenever I hit any technical roadblock or stuck at something non-technical as well. Thanks a lot, Dr. Edwards for all the financial support and valuable time during these two years and for playing a major part in me leading a successful career.

Dr. Servant and Dr. McCrickard, thanks a lot for all your inputs, advise and suggestions during my research. Also, appreciate your valuable time spent on reviewing my work.

I would like to dedicate all my success to my Mom, Dad and Sister, you guys are just amazing. Also, I would like to thank my relatives, Balaji Kandru for making sure that I always lead a smooth life during my Master's.

Mukund, cannot thank you enough for your continuous support, and guidance. Also, thanks a lot for contributing to the User Interface part of my thesis.

I would like to thank my roommates in Blacksburg, Radhakrishnan, Shreyas, and Surabhi for always being there for me. Deepthi, Pooja, Nageen, Ky, Sowrabha, Sachin and all other friends, thanks a lot for all the encouragement and guidance.

I would like to thank Virginia Tech, CS department for providing this incredible opportunity to pursue my Master's at VT. Also, I apologize if I have missed out on any names, please understand that it was not intentional.

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 author and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter covers the motivation behind a new improved intelligent feedback. In the later sections of the chapter, we cover the problem statement pertaining to this research work and the outline of this dissertation.

1.1 Motivation

Computer programming is a process of converting a computation problem to an executable computer program. Programming is essentially an art of developing new software tools using intellectual skills. It involves multiple subtasks like analysis of the problem, forming an algorithm to solve the problem, evaluating the correctness of the algorithm, implementing that algorithm in a programming language and testing for the correctness of the program. Programming requires creativity and a clear understanding of the programming language to convert ideas into working software. It is widely considered as a challenging task because the underlying intricacies of the programming languages are quite challenging for a beginner to understand and it is usually complicated to transform ideas into a completely correct working program.

"As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs." – M. Wilkes [5]

Programming is indeed a tough art to master because of all the complex subtasks involved and more often than not most programmers incur a lot of issues in initial attempts at building a solution.

With such complexities involved, it is quite common and expected that beginners would find programming very challenging and demotivating. Most of them would even lose interest in programming as they incur a lot of issues while coding and are not sure of the efficient order in which they should resolve these issues. Therefore, it is essential that beginners are provided with appropriate feedback that guides them in resolving the issues by providing clear goals that would help them understand the order in which the issues should be resolved and also the priority of different issues.

1.1.1 Goal-Oriented Feedback is Important

Feedback is one of the most crucial aspects of the learning process as it helps students to improve their knowledge. A clear and effective feedback can help many students understand most of the issues that were not clear to them. Feedback is productive when the learner utilizes the provided feedback and builds upon that to overcome the roadblocks.

"Without goals, and plans to reach them, you are like a ship that has set sail with no destination." - Fitzhugh Dodson [8]

When the feedback lists out the goals to the students, they can easily know what to work on next even when there are a lot of issues in their code. This is especially helpful for beginners as they see a lot of errors in their code during the initial attempts to solve a programming assignment. Beginners find it difficult to understand the priority of different issues and how to resolve them in an efficient order. So, effective goal-oriented feedback can provide intelligent guidance and improve their learning abilities and keep them motivated and glued to the subject.

Programming being a challenging task requires effective goal-oriented feedback being provided to the students whenever they face any issues. With high-quality feedback, we can not only keep students interested in programming but continue to attract new students as well. It would effectively reduce the number of dropouts and would increase the contributions to the research of computer science field.

With the growing number of students interested in programming, it is not practical for the instructor to provide effective and personal goals to each student. It is plausible to give out in-class generic feedback of the most common errors that the students would face in an assignment or a concept. But, this was not effective and

did not address the difficulties of most students. This lead to universities employing several teaching assistants so that individual attention is given to each student to provide guidance in resolving different issues.

While teaching assistants did serve the purpose for a long time, it was slowly turning out to be unscalable as the number of beginner programmers increased. It was not feasible even for the teaching assistants to address the concerns of the students. This lead to students being frustrated and unhappy with the support they have been receiving to solve a programming assignment.

Thus, there is a need for research and development in the field of automated feedback to provide students with instant goals to resolve the issues in their code. The feedback should be able to provide all kinds of support in resolving all the issues students would incur while solving their programming assignments. Able automated feedback would successfully help students reach a completely correct solution for their programming assignments by providing smart goals.

1.2 Problem Statement

This research aims to address specific issues in automated feedback on programming assignments to help beginners formulate effective goals to resolve the errors they would incur while solving a programming assignment. It would be beneficial for beginners to know what to do next along with a simple summary and understanding of the issues and their priorities. We describe the problems with the current feedback mechanisms:

- Students are provided with a lot of uncategorized feedback that leaves beginners with the difficult task of interpreting the issues and their severity correctly.
- Students fail to identify high priority issues that might be hidden with other issues when there is a lot of feedback.
- Just like how web-users skim through a web-page to obtain the gist of it, students also skim through the feedback visually and may oversee important details.
- It is very cumbersome for beginners to formulate goals or understand “what to do next” just based on the feedback or by viewing the errors.

- Students cannot understand the uncategorized errors easily and formulate goals to work on similar issues at a time.
- Lack of emphasis on the intelligent ordering of errors to catalyze faster progress towards completion.
- Novice students find most compiler error messages to be unclear and convoluted.

We tackle the above problems utilizing HCI principles and smart UI design instead of a lot of text. Significantly visual-based feedback is easy to understand and requires less effort. It keeps students engaged in the feedback and attracts them to read and interpret the feedback easily. The key intelligent and improved design aspects that address those problems are:

- **Modularized Summary:** Code feedback is categorized into multiple modules that aid students understanding of their code. We provide a smart visual summary of each module instead of forcing the students to analyze entire feedback to understand the issues. This modularization helps us to easily indicate the goal category that the student needs to work on.
- **Goal Formulation:** Priority items are highlighted appropriately in the feedback to guide students in formulating goals. Goal formulation helps students to clearly emphasize on the most important issues to be worked on.
- **Prominent Error Code Snippets:** This feedback locates and describes the error appropriately, along with the actual error message and a detailed explanation of the error. This provides a better illustration of the error that enables students to understand their goals effectively rather than students traversing through every source code file looking for errors.
- **Priority Order of Errors:** All the errors are listed out in priority order so that students can resolve the errors in that order and reach their goal faster. We also ensure that students get feedback on all kinds of errors incurred in their code by enabling smart limits on the feedback.
- **Enhanced Error Messages:** The compiler error messages are enhanced to provide a better understanding of the error and to provide a hint to resolve the error. Better diagnosis of “StackOverflow” exceptions is developed so that students can clearly understand the cycles occurred in the stack. We provide enhanced error messages to many of the subcategories.

1.3 Outline

In the further chapters, we describe the Literature Review (Chapter 2) associated with this work. Then, we move to the core of this thesis. We describe the design and implementation of the intelligent feedback in Chapter 3. Then, we provide a walkthrough of the intelligent feedback user interface in Chapter 4. It is followed by survey details and conclusions in Chapter 5 and 6 respectively.

Chapter 2

Literature Review

In this chapter, we describe the past research work pursued in the field of automated feedback for programming assignments and then explain the significant improvements about automated feedback carried out in this research. We describe Web-CAT, an automated grading tool setup to utilize our intelligent feedback.

2.1 Related Work

There have been many studies in the field of automated feedback generation for programming assignments. We will describe the ones that are most relevant to our work; they include the ones pertaining to automated feedback generation utilizing instructor provided data, automated program repair by providing next-step hints, visual feedback, and enhanced error messages.

2.1.1 Next-Step Hint Generation

We describe the studies pertaining to automated feedback wherein data from instructors is utilized as a major source to provide hints to the student. Most of the current research pertaining to automated feedback is based upon Next-Step hint generation.

Pardha et al. [9] devised an automated feedback strategy wherein they obtain specific software metrics like Cyclomatic Complexity (CC), Lines of Code (LOC)… for each assignment through empirical studies. These metrics are computed for both the Reference Code and Student Code, and the metrics that incurred higher deviations are specified to the instructor who can provide appropriate feedback to the students.

Victor et al. [10] presented a semantic-based technique where each code snippet pattern is associated with specific and relevant feedback provided by the instructor. The student's submission is transformed into a program dependence graph, and a subgraph pattern matching is performed on this, and the instructor provided patterns to obtain personalized feedback associated with that pattern as indicated by the instructor.

Rishabh Singh et al. [11] along with Sumit Gulwani from Microsoft Research have formulated a new method for automatically generating feedback for programming assignments. In this method, instructors are required to submit the reference implementation of the assignment and an error model that consists of corrections to issues that students encounter in their code. These errors are described in an error model language EML, wherein it consists of a set of correction rules that are used for providing feedback to the student. They used a Sketch synthesizer to synthesize a correct program utilizing EML.

Very little research has been carried out in the aspect of feedback generation for performance problems. Sumit Gulwani et al. [12] addressed this issue by designing a light-weight programming language extension that allows the instructor to specify certain key values that should occur at different steps of the execution of the code. By utilizing a dynamic analysis based approach, student's code is tested to check if it matches the instructor's specification, which in turn helps in providing the feedback.

One of the most recent work in automated feedback is done by Georgiana Haldeman et al. [13] at Rutgers University. Their methodology involves instructor building a knowledge map of all the concepts and skills important to the assignment. Reference tests designed by instructors are used to assess the submission and group the submissions into buckets based on the class of errors. A hint for each bucket would be pre-written, and the autograding system provides those hints for submissions that fail on one or more tests.

Luciana Benotti et al. [14] have built a web-based coding tool that provides formative feedback and assessment. To automatically provide feedback to the students, this tool requires the instructor to provide a suite of unit tests to generate correctness feedback and to provide quality feedback the instructor can define certain patterns (called Expectations) and if they should be present or not present in the student's solution. Each Expectation is evaluated against the student's code to

generate appropriate feedback. For instance, an Expectation could be: “direct recursion is not allowed in this assignment.”

Lukas Ifflander et al. [15] have implemented a tool (called PABS) for automated feedback generation for programming assignments on the Java virtual machine. PABS is a web-application that provides students an opportunity to submit code and receive automated feedback. PABS employs separate processes called agents to generate the feedback. The agent utilizes the build and test execution information to create the feedback.

Automated Program Repair(APR) is a recent technology that automatically fixes software bugs, and Jooyong Yi et al. [16] have explored the idea of combining APR with Intelligent Tutoring System for Programming (ITSP). APR tools are used for generating hints so that students can learn better rather than providing a complete repair of the program. Partial repairs are generated; which is like a popular approach in education field known as “next-step hint” generation. So, this research has discussed different strategies to tailor the repair policy so that effective hints can be provided using APR.

Kelly Rivers [17] designed a data-driven tutor authoring tool that uses student data to identify a goal state for any submission. Using this, next-step hints are generated that helps students to make necessary code changes to proceed closer to a perfect submission.

Hannah Blau et al. [18] built an Eclipse plug-in called FrenchPress that offers Advanced Beginner students feedback on their Java Programming Style. The system identifies a limited set of issues wherein object-oriented paradigm is not followed. The plug-in provides feedback to such issues and is up to the student to incorporate those changes. FrenchPress does not check for spacing, comments and other Java style conventions that are covered by static analysis tools like Checkstyle. From the user survey conducted by Hannah et al., it was evident that most of the students were satisfied with the feedback provided by FrenchPress.

2.1.2 Visual Feedback

There have been few studies pertaining to the visual feedback that helped us understand the importance and effects of visual feedback.

Petri Ihantola [19] worked on improving the automated assessment of programming assignments. They have developed visual feedback describing the behavior and functionalities of student's code. Most of the visualizations work carried out by Petri (Object graph-based visual feedback) has been for the static data structures (test data and expected output), and from the evaluation study, they have observed that visual feedback was not worse than good textual feedback. Teachers would invest very less time to produce visual feedback (as automated tools generate them) than good textual feedback. So, this is an encouraging sign and a major motivation for further research to be carried out in visual feedback.

Minjie et al. [20] have developed an interesting approach to improve the skills and understanding of novice programmers. Their approach utilizes a visual programming language and the concepts of "goal" and "plan" wherein, the goals and plans are displayed in a visual form. Their experiments have indicated a significant improvement in the learning abilities of beginner programmers.

Teemu et al. [21] have investigated the use of program visualizations within an interactive Ebook of an introductory programming course. Their analysis shows that most students spend a lot of time on these visualizations and this improved the students' engagement to the course. They also articulated that students are more inclined to this kind of visual-based learning.

2.1.3 Complex Error Messages

Marceau et al. [22] have inferred from their studies that the error messages do not convey the information to the students and they have suggested few enhancements to the error messages in the form of color-coded highlights of the code, consistent vocabulary, and non-biased error messages. This research encouraged us to provide an enhancement to many error messages generated by the tools.

2.1.4 Investigating Static Analysis Errors

Edwards et al. [23] have investigated the Static Analysis Errors in Java programs and examined very interesting research questions regarding the most frequent static analysis errors, most frequent category of static analysis errors, error rates, and the duration to fix different errors. We have utilized this work to obtain the subcategories of Style category.

2.2 Key Improvements in this Work

As described in the previous section, most of the automated feedback research has been carried out in providing next-step hints and automated program repair. This kind of feedback requires a lot of data from the instructor about reference tests, different kind of solutions to obtain all possible paths to goal state and different possible error patterns to provide hints to students. While this is useful to reach the desired goal state regarding the logical correctness of the program, it is not particularly effective for beginners who run into various other issues related to their code and testing.

We have observed from Minjie's [20] and Teemu's [21] studies and HCI principles that visual feedback has far more penetration and engagement levels as compared to text-based feedback. So, we have modeled our feedback to be more of a visual-based one with the minimal amount of text.

Our feedback's design as mentioned in the problem statement section of the previous chapter addresses the issue of providing effective goal-oriented feedback to resolve all kinds of issues related to the coding and testing aspects of the program. We utilize different error logs, static analysis tool results and test results to provide such intelligent feedback. Also, we can see that our design does not require any additional data from the instructors to provide appropriate feedback.

2.3 What is Web-CAT?

"Web-CAT is a plug-in-based web application that supports electronic submission and automated grading of programming assignments. It supports fully customizable,

scriptable grading actions and feedback generation for any assignment" [29]. It is widely used by many universities across the world. The different plug-ins of Web-CAT support different programming languages. In addition to the support for different programming languages, Web-CAT plugins also enable the use of static code analysis tools like Checkstyle and PMD. Web-CAT also provides lots of support for grading students based on how well they test their code. JaCoCo [38] is integrated into Web-CAT to obtain the code coverage features.

When a student code is submitted to Web-CAT, it can provide a score on three measures: 1) correctness of the code, 2) how well students test their code and 3) how well code conforms to coding standards. The correctness of the code measures how many of the instructor provided reference tests the student's code pass. Students' testing of their code is measured how much of their code is covered by their tests and also how many of their tests pass. Static analysis tools like CheckStyle and PMD enables Web-CAT to provide a score for conforming to coding standards by checking for formatting/indentation, JavaDocs, Naming standards and following best coding practices. Web-CAT is very flexible to the instructors regarding selecting their grading scheme [45]. They can choose to enable or disable any of the three measures of grading and also choose to have a portion of the grade awarded by Web-CAT and the rest done manually.

We implemented our intelligent feedback in one of the plugins that support Java programming assignments in Web-CAT and then evaluated the impact it had on the students.

Chapter 3

Intelligent Feedback: Design and Implementation

As described in Chapter 1, this work focuses on structuring feedback to help students formulate effective goals to resolve the issues they encounter while solving a programming task. We describe below the design goals of the intelligent feedback that aids us to accomplish our objective:

- **Modularized Summary:** Code feedback is categorized into four categories in the descending order of priority: Coding, Student's Testing, Behavior, and Style. This modularization helps students to interpret the feedback effortlessly. By exploiting this modularization, we provide a visual summary of each category. Modularized Summary assists us in visually indicating the goal category to the student.
- **Goal Formulation:** As mentioned in the previous design goal, priority module is visually highlighted in the feedback to guide students in formulating goals. This helps students to understand the highest priority category that needs attention and the issues that need to be worked on.
- **Prominent Error Code Snippets:** Intelligent feedback locates and describes all the error code snippets along with error messages and detailed explanation of the error in the main feedback page. This enhances the understanding of errors that supports students to accomplish the formulated goals efficiently.
- **Priority Order of Errors:** By utilizing the priority ordering of different categories and subcategories and custom prioritization strategies for each subcategory we list the errors in priority order so that students can easily resolve many errors effectively. This guides students to accomplish their goals faster. We also ensure that students get feedback on all kinds of errors incurred in their code by enabling smart limits on the feedback.
- **Enhanced Error Messages:** Intelligent feedback aims to provide a better perception of the errors by providing a detailed explanation of the error. We specifically enhance the compiler error messages as they are usually the most

difficult ones for beginners to understand. We also provide a diagnosis to "StackOverflow" exceptions as it is complicated for beginners to locate the method calls that caused infinite recursion.

In this chapter, we express our initial approaches towards solving the problem and then later describe the design formulation of the intelligent feedback based on the design goals listed out in the problem statement. We also provide the implementation details of this intelligent feedback in Web-CAT.

3.1 Initial Approaches

To tackle the problem of intelligent feedback and summarization of student's program, we looked at several NLP based techniques like goal generators, and text summarizers. But we subsequently realized that this problem could be solved by providing visual feedback with appropriate modularization, implicit goals, progress indicators, extracting essential information from student's code analysis tools and test results. We learned that an intelligent visual-based layout of all this data would provide effective feedback to the student's code.

3.2 Modularized Summary

We describe below the different modules of our intelligent feedback and the design details about the smart summary of errors pertaining to each module. We also detail the scheme for progress charts that are utilized to convey the progress made in each category.

3.2.1 Modularized Feedback

Many students might find it easier to interpret feedback when it is broken into multiple categories, as it makes it easier for the students to understand how different parts of the feedback are related to each other. Modularized feedback assists students in understanding which part of the feedback to look at for specific issues of interest. This is helpful for the beginners to interpret the feedback effectively when a lot of it

is generated. Modularization also assists us to formulate goals for the students; we can clearly convey the goal module which needs to be worked on.

We detail below the categories pertaining to our intelligent feedback and the subcategories of each category. We also describe the priority order of the categories and subcategories that would be exploited in providing smart guidance for students to reach completion faster. The prioritization also helps us in conveying most important issues appropriately so that students do not miss these important issues when a lot of feedback is generated.

3.2.1.1 Different Modules in the Feedback

From the empirical study of all the student's submissions, it has been observed that most issues students experience are related to their code and testing, with style issues also playing a major part but not highly critical. The Testing aspects can be divided into two categories: Student testing and Instructor Reference tests (Behavioral Testing). Therefore, we understood that the best way to present feedback to the students is by modularizing the feedback into these four categories: Coding, Testing, Behavior, and Style as this would provide a simple, better and clear understanding of the feedback to the students. This covers all possible aspects of their code. Each of these four categories is further classified into subcategories. We will discuss them below.

A. Coding

The Coding category describes the errors related to the code. This contains errors pertaining to both the semantic and static analysis of the code and also contains severe issues generated by Static Analysis tools. The Coding category is classified into the following subcategories:

- **Compiler Errors:** These are the errors that occur when the compiler fails to compile the code due to errors in the code [31].
- **Compiler Warnings:** Compiler warnings are issues that the compiler marks as things that might cause problems [32].

- **Signature Errors:** These are the errors that occur while executing the instructor tests on student's code. These are issues like incorrect class names, incorrect constructor parameters and so on. These are essentially the issues pertaining to the student not following the naming, parameter types and other specifications as described in the instructions for the program; based on which the instructor tests are built. A Reflection Error is thrown signifying the same.
- **Potential Coding Bugs:** These are the subset of all the errors/checks from the Static Analysis tools that are very likely to represent bugs in the code and are high priorities to fix [23]. These are also called "Coding Flaws."

Static Analysis Tool	Error
Checkstyle	CovariantEquals
Checkstyle	EmptyStatement
Checkstyle	FallThrough
Checkstyle	HiddenField
Checkstyle	InnerAssignment
Checkstyle	StringLiteralEquality
PMD	AvoidBranchingStatementAsLastInLoop
PMD	AvoidMultipleUnaryOperators
PMD	BrokenNullCheck
PMD	ClassCastExceptionWithToArray
PMD	IdempotentOperations
PMD	JumbledIncrementer
PMD	MisplacedNullCheck
PMD	MissingStaticMethodInNonInstantiatableClass
PMD	NonCaseLabelInSwitchStatement
PMD	ProhibitedGreenfootImport
PMD	SwitchStmtsShouldHaveDefault
PMD	UnusedLocalVariable
PMD	UnusedPrivateField
PMD	UnusedPrivateMethod
PMD	UselessOperationOnImmutable

Table 3.1: Classification of Potential Coding Bugs

- **Unit Test Coding Problems:** These are different problems/checks that have been reported by the Static Analysis tools pertaining to JUnit tests [23].

Static Analysis Tool	Error
PMD	JUnit3ConstantAssertion
PMD	JUnit3TestsHaveAssertions
PMD	JUnit4ConstantAssertion
PMD	JUnit4TestsHaveAssertions
PMD	JUnitSpelling
PMD	JUnitTestClassNeedsTestCase
PMD	UseAssertNullInsteadOfAssertTrue
PMD	UseAssertSameInsteadOfAssertTrue

Table 3.2: Classification of Unit Test Coding Problems

B. Student's Testing

The Testing category comprises of issues that occur by executing student's unit tests and the coverage of their tests. This category is classified into the following subcategories:

- **Unit Test Errors:** This consists of all the issues where JUnit runs the test, and an exception is thrown out that is not caught [35].
- **Unit Test Failures:** JUnit provides a class called Assertions to write unit tests. These are used to verify different behavioral aspects of the source code, and JUnit reports a test failure if the assertion fails. All these issues are bucketed into the "Unit Test Failures" subcategory [35].
- **Unexecuted Methods:** This consists of all the methods that are unexecuted by the tests.
- **Unexecuted Statements:** This consists of all the statements that are unexecuted by the tests.
- **Unexecuted Conditions:** This consists of all the conditions that are not fully covered by the tests; if the tests do not cover all the branches then its reported under this subcategory.

Note that, JaCoCo provides the Code Coverage data.

C. Behavior

The Behavior category comprises of issues that would occur by executing instructor's tests on the student's code. This category is classified into the following subcategories:

- **Unexpected Exceptions:** This consists of all the issues where JUnit runs the test, and an exception is thrown but is not caught [35]. This can be any error apart from "StackOverflowError", "TestTimedOutException", and "OutOfMemoryError".
- **Infinite Recursion Problems:** This consists of all the issues where JUnit runs the test, and "StackOverflowError" is thrown; that is due to the stack being overflown because the code recurses too deeply [36].
- **Infinite Looping Problems:** This consists of all the issues where JUnit runs the test, and a "TestTimedOutException" is thrown; because the test failed on a timeout as the code ran into an infinite loop [37].
- **Behavior Issues:** These are failed Assertions in the instructor's unit tests. These Behavior Issues signify that the student's code does not behave as expected and suffers from correctness issues. All these issues are bucketed into the "Behavior Issues" subcategory [35].
- **Out of Memory Errors:** This consists of all the issues where JUnit runs the test, and "OutOfMemoryError" is thrown; because the Java Virtual Machine is unable to allocate an object as it is out of memory [39].

D. Style

The Style category consists of all the issues pertaining to the styling of student's code and unit tests. These issues are mostly harmless to the execution/result of the code but provide better design, readability, and understandability. This category is classified into the following subcategories:

- **JavaDoc:** This consists of all the issues pertaining to missing or incorrect JavaDoc elements [23].

Static Analysis Tool	Error
Checkstyle	JavadocMethod
Checkstyle	JavadocType
Checkstyle	JavadocVariable
Checkstyle	RegexpSingleline
Checkstyle	TodoComment
PMD	UncommentedEmptyConstructor
PMD	UncommentedEmptyMethod
PMD	UncommentedEmptyMethodBody

Table 3.3: Classification of JavaDoc Issues

- **Formatting/ Whitespace:** This consists of all the formatting issues like the improper indentation of the code or missing whitespace. This reduces readability [23].

Static Analysis Tool	Error
Checkstyle	ArrayTypeStyle
Checkstyle	FileTabCharacter
Checkstyle	GenericWhitespace
Checkstyle	Indentation
Checkstyle	LineLength
Checkstyle	MethodParamPad
Checkstyle	MultipleVariableDeclarations
Checkstyle	NoWhitespaceAfter
Checkstyle	NoWhitespaceBefore
Checkstyle	OneStatementPerLine
Checkstyle	RightCurly
Checkstyle	WhitespaceAfter
Checkstyle	WhitespaceAround

Table 3.4: Classification of Formatting/Whitespace Issues

- **Naming:** This consists of all the names that violate naming conventions like capitalization conventions, too short and are not meaningful [23].

Static Analysis Tool	Error
Checkstyle	ClassTypeParameterName
Checkstyle	ConstantName

Checkstyle	LocalFinalVariableName
Checkstyle	LocalVariableName
Checkstyle	MemberName
Checkstyle	MethodName
Checkstyle	ParameterName
Checkstyle	StaticVariableName
Checkstyle	TypeName
PMD	AvoidFieldNameMatchingTypeName
PMD	FormalParametersNeedMeaningfulNames

Table 3.5: Classification of Naming Issues

- **Readability:** This consists of issues other than the formatting issues that reduce the readability of the code. We also add issues pertaining to Braces such as omitting optional Braces [23].

Static Analysis Tool	Error
Readability	
Checkstyle	NeedBraces
Checkstyle	UpperEll
Checkstyle	DefaultComesLast
Checkstyle	EmptyBlock
PMD	AvoidUsingOctalValues
PMD	MethodWithSameNameAsEnclosingClass
PMD	EmptyCatchBlock
PMD	EmptyFinallyBlock
PMD	EmptyIfStmt
PMD	EmptyInitializer
PMD	EmptyStatementBlock
PMD	EmptyTryBlock
PMD	EmptyWhileStmt
Braces	
PMD	ForLoopsMustUseBraces
PMD	IfElseStmtsMustUseBraces
PMD	IfStmtsMustUseBraces
PMD	WhileLoopsMustUseBraces

Table 3.6: Classification of Readability Issues

- **Coding Style:** This consists of all issues pertaining to "Style"; such as any code that can be simplified, or that does not follow common idioms. This also contains "Excessive Coding" issues like size issues where methods, classes or parameter lists exceed limits and may pose readability problems [23].

Static Analysis Tool	Error
Style	
Checkstyle	RedundantThrows
Checkstyle	RegexpMultiline
Checkstyle	SimplifyBooleanReturn
PMD	AvoidRethrowingException
PMD	AvoidThrowingNewInstanceOfSameException
PMD	AvoidThrowingNullPointerException
PMD	BooleanInstantiation
PMD	CollapsibleIfStatements
PMD	DontImportJavaLang
PMD	DuplicateImports
PMD	EqualsNull
PMD	ExtendsObject
PMD	ForLoopShouldBeWhileLoop
PMD	ImportFromSamePackage
PMD	InstantiationToGetClass
PMD	LogicInversion
PMD	ReturnFromFinallyBlock
PMD	SimplifyBooleanAssertion
PMD	SimplifyBooleanExpressions
PMD	SimplifyBooleanReturns
PMD	SimplifyConditional
PMD	SingularField
PMD	StringInstantiation
PMD	TooFewBranchesForASwitchStatement
PMD	UnconditionalIfStatement
PMD	UnnecessaryCaseChange
PMD	UnnecessaryConversionTemporary
PMD	UnnecessaryFinalModifier

PMD	UnnecessaryReturn
PMD	UnusedImports
Excessive Coding	
PMD	ExcessiveClassLength
PMD	ExcessiveMethodLength
PMD	ExcessiveParameterList
PMD	TooManyFields

Table 3.7: Classification of Coding Style Issues

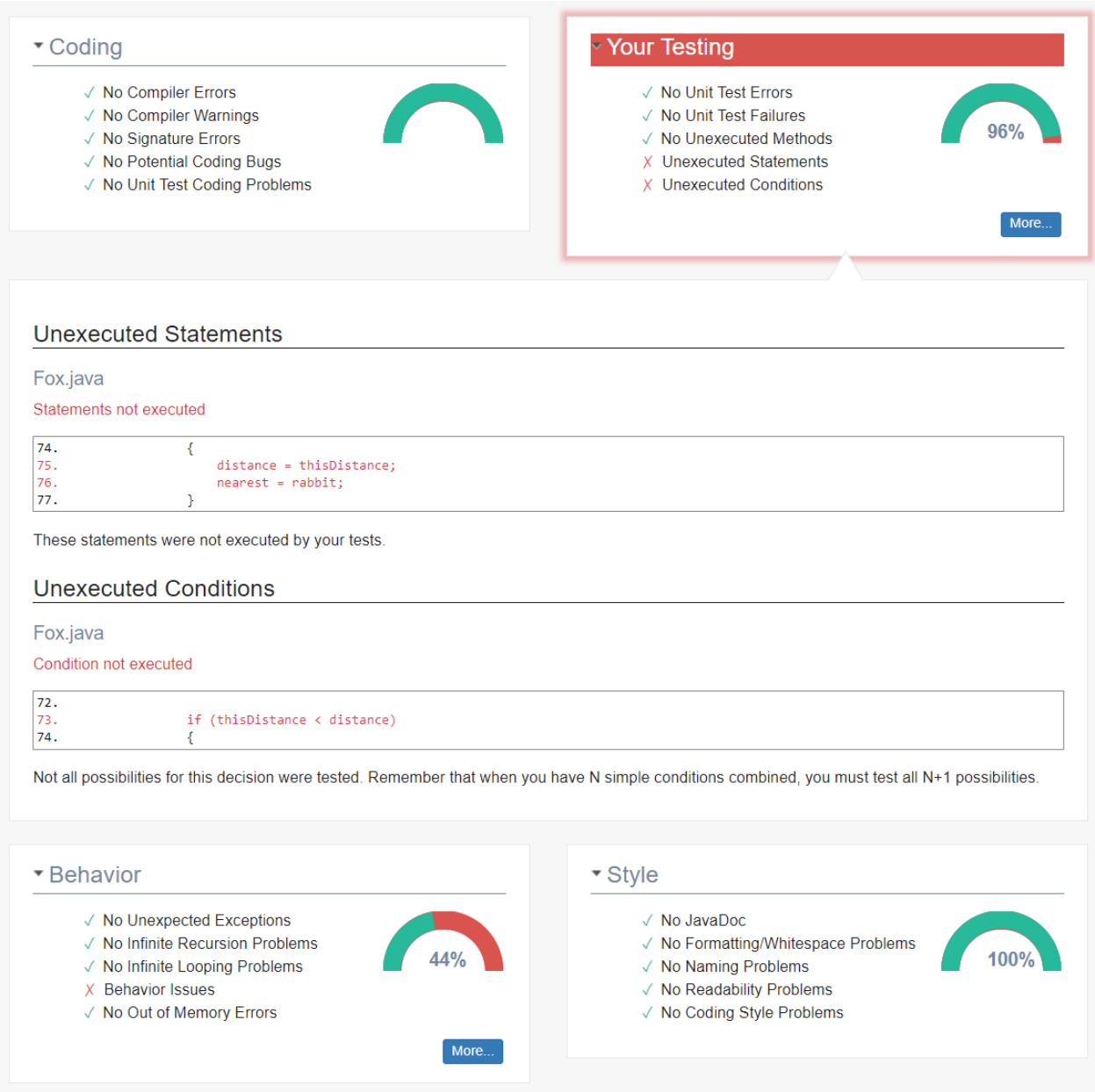


Figure 3.1: Intelligent Feedback on a Submission with Issues

3.2.1.2 Priority Order of Categories

It is of prime importance that high priority categories are present at the initial parts of the feedback so that students resolve them first and consider them to be important. This theory helps us to easily formulate goals; as described in the Goal-Oriented feedback section. We describe below the principles behind the priority order of categories.

The modules are listed in the priority order as follows: "Coding," "Student's Testing," "Behavior," and "Style." We observed from studies that "Coding" is the most important category amongst all of them; as a compiler error-free (and all other subcategories from "Coding" category) code can be improved further by appropriate testing only after everything with regards to the "Coding" category is resolved. Then, we observe that the student's testing ("Testing" category) helps to figure out hidden errors that would have gone unnoticed while coding. Well-written tests would provide full coverage of the code that makes it easier to figure out all possible hidden errors.

Once the code is well tested by the student to make sure that all possible errors (as far as possible) have been resolved, the instructor's tests (Behavior category) would be the most important one. It helps to improve the correctness of the code so that the goal state (regarding problem coverage) is reached. Also, instructor's tests would help to resolve hidden errors that the student would have never expected.

After we have a logically correct code with zero runtime errors; appropriate styling of the code would improve the readability and understandability. This is why we prompt students to maintain appropriate styling by marking failed subcategories under Style appropriately.

3.2.1.3 Priority Order of Subcategories

By prioritizing the subcategories, we can list out the errors pertaining to each category in the priority order of the subcategories and subsequently prioritize the errors in each further subcategory as described in the "Priority Order of Errors" section. This prioritization helps us to describe the high priority goals at the initial parts of the feedback. We describe below the principles behind the priority order of subcategories.

The subcategories of the Coding category are listed in the priority order as follows: "Compiler Errors," "Compiler Warnings," "Signature Errors," "Potential Coding Bugs," and "Unit Test Coding Problems." We can see that "Compiler Errors" is the highest priority subcategory followed by "Compiler Warnings." "Signature Errors" would be the next most important one as it will cause all the instructors' tests to fail. "Potential Coding Bugs" would take higher precedence compared to "Unit Test Coding problems" as fixing the source code is of higher priority than test code.

The subcategories of the "Testing" category are listed in the priority order as follows: "Unit Test Errors," "Unit Test Failures," "Unexecuted Methods," "Unexecuted Statements," and "Unexecuted Conditions." We can easily understand the fact that "Unit Test Errors" have more impact on the code and are difficult to resolve; we have "Unit Test Errors" as the highest priority category followed by "Unit Test Failures." A "100% code coverage" from tests would improve the testing quality, and if an uncovered method is covered, it will improve the coverage drastically. So, "Unexecuted Methods" would be the next one in the priority order followed by "Unexecuted Statements," and "Unexecuted Conditions."

The subcategories of the Behavior category are listed in the priority order as follows: "Unexpected Exceptions," "Infinite Recursion Problems," "Infinite Looping Problems," "Behavior Issues," and "Out of Memory Errors." From our studies, we have inferred that "Unexpected Exceptions" is the most important subcategory under "Behavior" followed by "Infinite Recursion Problems" – as these are usually complicated for beginners and require immediate attention to resolve the dangerous recursions. "Infinite Looping Problems" is important to be resolved soon; as it implies that the code can be written efficiently and resolving this might also resolve "Out of Memory Errors" indirectly. So, "Behavior Issues" would be the next one followed by "Out of Memory Errors."

The priority order of the subcategories of the Style category is: "JavaDoc," "Formatting/Whitespace," "Naming," "Readability" and "Coding Style." "JavaDoc" is the most occurring error, and "Formatting" is the most occurring category from our study [23]. These are followed by "Naming" and "Readability" from our research [23] and usually cause more points deduction than "Coding Style." So, this leaves us with "Coding Style" as the last subcategory.

3.2.2 Summary of Errors

We have observed that most students skip a lot of text in the feedback and usually skim through the feedback thereby missing a lot of feedback pertaining to important issues. So, we have devised our feedback to be non-textual, and as part of this strategy, we have designed a visual-based summary that students would be engaged to. Also, we are specifically interested in providing a summary that can be interpreted effortlessly even when students skim through the feedback and ignore most of the feedback.

This visual summary illustrates the high priority category (and subsequently subcategories) which students might fail to identify when there is a lot of feedback. As we can see in Figure 3.2, this summary enables us to convey the goal category visually which can be easily observed by the student. We can understand the goal formulation design details from Section 3.3.

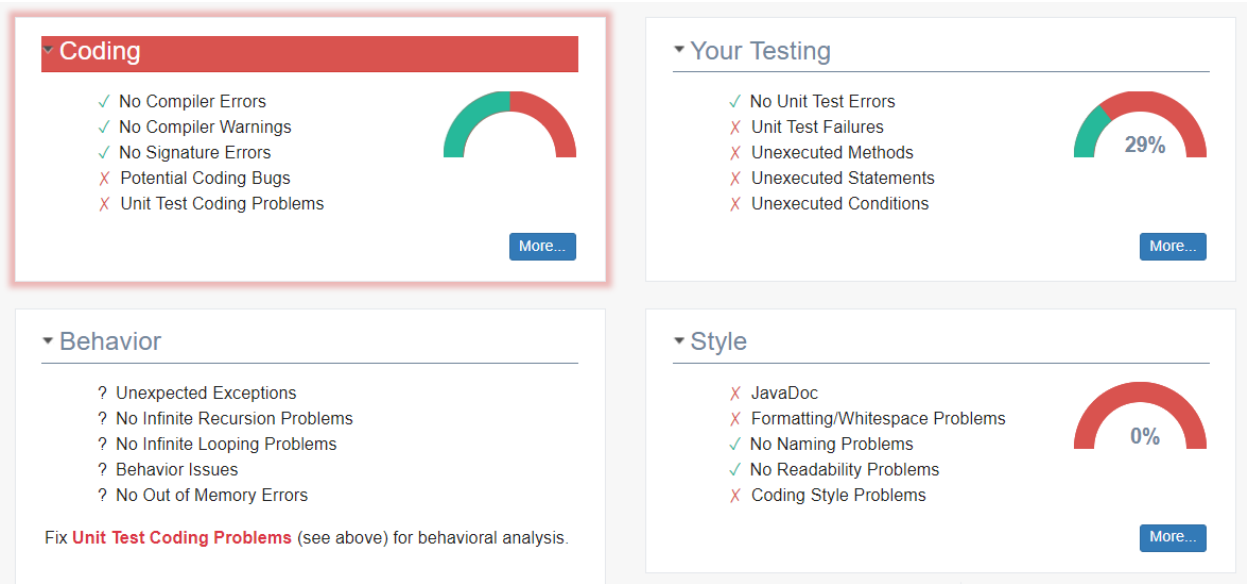


Figure 3.2: Summary of a Submission with Issues

Also, it is imperative for any automated feedback to provide a clear summary of the entire code as it would help students to obtain a quick understanding of the shortcomings of their code without having to look around in multiple feedback pages or logs. We have designed this summary utilizing the modules described above so that the summary is smart and simple.

As described in the above figure, we provide a visual summary of each category by marking each subcategory with a visual clue to signify if there are any issues concerning that category in the code.

A “check mark” is used to signify that there are no issues pertaining to that subcategory in the code whereas a “cross mark” is used otherwise. We also append a “question mark” to a sub-category to signify that it has not been evaluated.

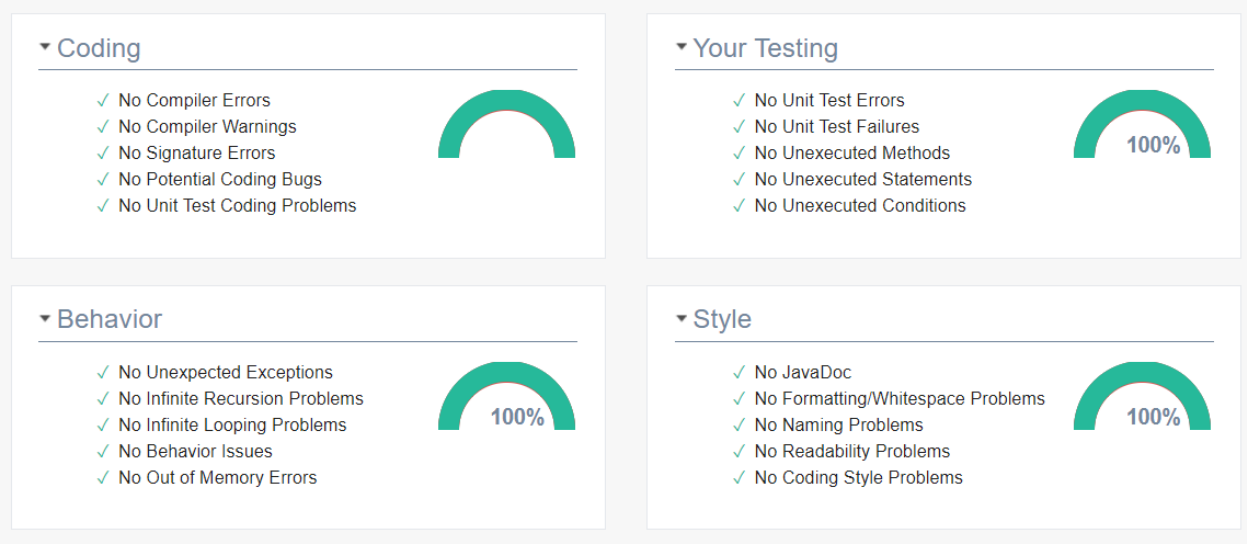


Figure 3.3: Summary of a Completely Correct Submission

We can see from the above figure that at a single glance, we can obtain a lot of information pertaining to each category and its subcategories.

3.2.2.1 Implementation

We have implemented the summary in Web-CAT by parsing different logs. With regards to the “Coding” category; “Ant log” is used to mark the “Compiler Errors” and “Compiler Warnings” subcategories with appropriate visual clues. “Signature Errors” is marked by parsing the instructor test results for “ReflectionError.” “Potential Coding bugs” is marked utilizing “Checkstyle” and “Pmd” logs and “Unit Test Coding Problems” is marked utilizing the “Pmd log.”

With regards to the Testing category, “Unit Test Errors” and “Unit Test Failures” are marked utilizing the Student’s test results, and the remaining categories are marked utilizing the coverage report from “Jacoco.”

The Behavior category is marked by parsing the instructor test results for appropriate errors and behavior issues (if any).

The Style category is marked by parsing the “Checkstyle” and “Pmd” reports.

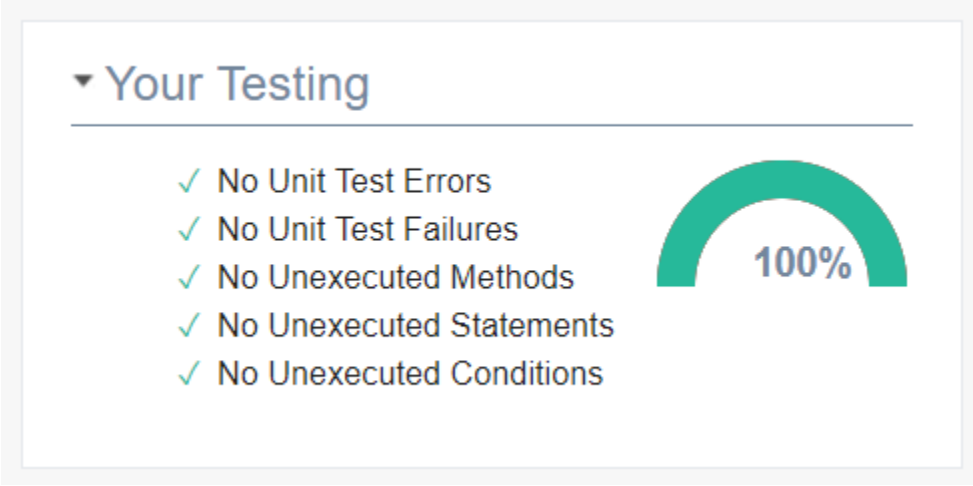


Figure 3.4: Progress Chart of Completely Correct Testing Category

3.2.3 Progress Charts

As observed in Figure 3.4, we provide Progress Charts (as part of the Summary) for each category so that students can obtain a clear and simple understanding of the progress made in each category. This will give them clear evidence of the progress made in each category and based on their improvement across multiple submissions; students can estimate the time it would take to reach the goal state (a 100% progress bar) [40]. The fact that the chart is visually based and also contains the numeric value- “percentage of progress” makes it highly valuable.

3.2.3.1 Implementation

For the “Coding” category, if the code does not contain any of “Compiler Errors,” “Compiler Warnings” and “Signature Errors” then it contributes to 50% of the radial bar (half of the radial bar) being positive (green). Otherwise, 50% of the radial bar would turn red. Similarly, if “Potential Coding Bugs” and “Unit Test Coding Problems” have been marked with a “check” visual clue then it contributes to the

other 50% of the radial bar (another half of the radial bar) being positive (green). Otherwise, 50% of the radial bar would turn red.

For the Testing category, the student's unit test case pass percentage contributes to 50% of the radial bar and the student's code coverage from tests contributes to the other half of the radial bar.

For the Behavior category, the Problem Coverage percentage (number of instructor test cases that have been passed); is directly used as an estimate to signify progress.

For the Style category, the number of points lost due to Styling issues is used to estimate the negative part of the radial bar. So, essentially the percentage of progress is 100% – the percentage of points lost.

3.3 Goal Formulation

Goal-Oriented feedback would help the students clearly understand the issues that they should work on for the next submission (attempt). It is very helpful especially for beginners to clearly know their goals rather than them extrapolating the goals from the error logs. This kind of feedback that provides goals to the students would save a lot of time and efforts of the students while keeping them motivated.

Goal-Oriented feedback would keep the students highly motivated as they clearly know "what to work on next," and with an effective illustration of the errors (Prominent Error Code Snippets), they can easily accomplish all the goals.

By the objective of having visual-based feedback, we convey the goal visually without explicitly describing the goal in textual form. This keeps the students engaged and visually guides them effectively even if they glance through the feedback.

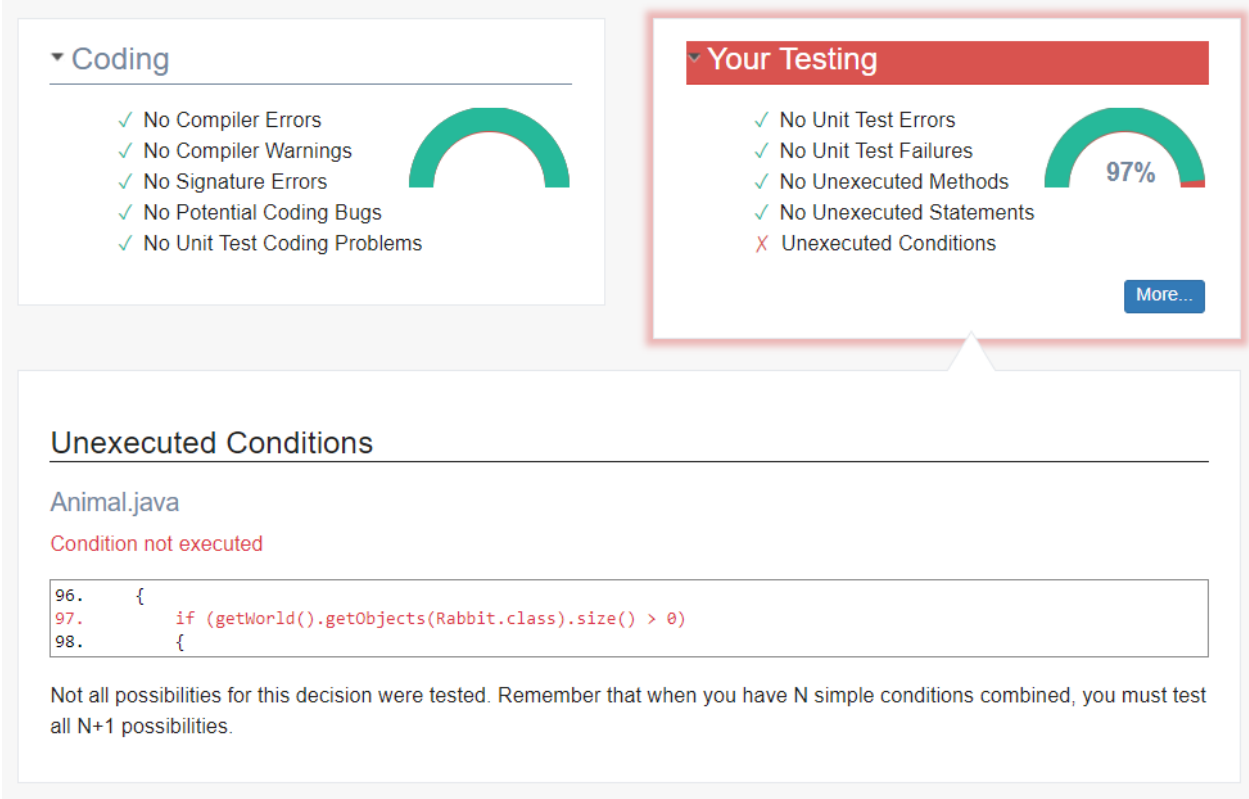


Figure 3.5: An Example of Testing Category being the Goal and is Expanded by Default to signify the same

3.3.1 Implementation

As seen in the above figure, we compute the goal category (goal section) as the first one that has an error in the categories listed in the order: Coding, Testing, Behavior, and Style. The goal category is easily identifiable due to the smart priority ordering of the categories. We highlight (shaded with red color) the goal category to convey that it requires some work to be done.

Then, we expand that high priority category to display the expanded feedback (read as “Prominent Error Code Snippets”) by default so that student’s attention is guided to this category in all possible ways and improvement is highly desired with regards to this category.

3.4 Prominent Error Code Snippets

The expanded category as seen in Figure 3.1 is an extension of a category; it is utilized to provide more detailed feedback for each category. Every category that has some issues would have a detailed category attached to it (which would expand upon clicking the "More…" button).

Fox.java

Condition not executed

```
79.                 double thisDistance = distanceTo(rabbit);
80.                 if (thisDistance < distance)
81.                 {
```

Not all possibilities for this decision were tested. Remember that when you have N simple conditions combined, you must test all N+1 possibilities.

Figure 3.6: Detailed Feedback on an Issue

We designed our detailed feedback such that students obtain all the essential guidance and knowledge from the feedback that would make it very easy for them to resolve the errors. We also drafted our feedback to be in a convenient way wherein the error messages are displayed along with the code snippet. The code snippet has a clear highlighting of the line of code that has that error. We also provide a line above and below the error line (usually) so that it is easier to locate the error line.

When this code snippet is bound to the error message, it makes it very easy to understand the error and act upon it rather than students scanning through every source file looking for errors. Traversing through every source file might be very laborious and difficult when there are a lot of source files. So, this better illustration of the error benefits students to interpret their formulated goals effectively and work on them effortlessly.

As described in the "Enhanced Error Messages" section below, we also enhance most of the error messages to make them less complicated and easily understandable. This provides a more detailed explanation of the error so that it is easier to understand the error.

3.5 Priority Order of Errors

One of the most important design goals of our feedback is to provide fruitful feedback so that students can accomplish the formulated goals sooner (which would in a way increase the score they would obtain in the assignment). This can be achieved by listing out the errors in priority order so that students can resolve the errors faster. By listing the errors in priority order, we ensure that students do not miss the high priority issues which might be hidden when there is a lot of feedback.

Also, it is desirable that by resolving the errors in our listed priority order students would make tremendous progress – by making it easier for students to resolve many errors effectively based on the priority order.

Ant.java

The field 'wallHealth' should be declared private. Only constants (static final fields) are allowed to be public. If this field is intended to represent a constant value, declare it to be static and final.

```
26.     public int fireHealth = 1;
27.     public int wallHealth = 4;
28.     public int hungryHealth = 1;
```

Ant.java

The field 'throwerHealth' should be declared private. Only constants (static final fields) are allowed to be public. If this field is intended to represent a constant value, declare it to be static and final.

```
24.     public int harvesterHealth = 1;
25.     public int throwerHealth = 1;
26.     public int fireHealth = 1;
```

Likewise, there are 8 issue(s) in Ant.java, 1 issue(s) in Insect.java.

Figure 3.7: Limiting the Errors of a Subcategory

We limit (global value to limit errors) the number of errors that would be displayed in each subcategory as part of the feedback. This would prohibit the feedback from growing tremendously large that would make it complicated, uninteresting and demotivating to students. We smartly describe the count of remaining errors (after the limit) in each file so that students can obtain a fair understanding of such errors in each file.

We also limit the number of errors in each further subcategory of a subcategory. For example, we would limit the number of errors that we would exhibit in each subcategory of "Potential Coding Bugs."

We have discussed the efficient priority ordering of all the categories and their subcategories in Section 3.2.1; which is a major contribution to this design goal. We will discuss the priority ordering of errors in each subcategory of all the categories.

3.5.1 Coding

- **Compiler Errors**

  As studied by Brett A. Becker et al. [41], fixing the first compilation error and ignoring the rest is the most efficient way to deal with multiple compiler errors. So, we have employed this technique to the compiler errors wherein we only present the first compilation error in a source code file in the feedback.

  We order those errors based on the count (decreasing order) of compiler errors in each source code file so that the file with the most number of errors is resolved first that would help the student to fix all the errors (implicitly) sooner.
- **Compiler Warnings**

  We employ a technique like that of Compiler Errors (mentioned above) for presenting Compiler Warnings in priority order.
- **Signature Errors**

  We group all the "Signature Errors" by source code file so that all such errors in a specific file can be resolved easily as they might have occurred due to the same issue. It is also easier to resolve all similar issues in a file in one go. We sort these groups based on their count (descending order) and then limit them (if necessary) as mentioned above.

  The "Signature Errors" would only contain the hint message, as code snippet from instructor test is not appropriate.

  This technique of priority order is used in multiple subcategories as mentioned below. All these subcategories do not have further subcategories of errors, so we group them by source code file instead.

- **Potential Coding Bugs**
  We group each of the issues that fall under "Potential Coding Bugs" (mentioned in 3.2.1.1) by its subcategory of "Potential Coding Bugs" so that all such similar errors can be resolved easily together.
  We sort these groups based on their count (descending order) and then limit them (if necessary) as mentioned above. We limit each subcategory of "Potential Coding Bugs" by (number of subcategories/limit value). This ensures that all kinds of errors are present in the feedback with high priority ones being at the top.

  This technique of priority order is used in multiple subcategories as mentioned below.
- **Unit Test Coding Problems**
  We employ same technique as that of "Potential Coding Bugs" (mentioned above) to obtain the priority order of errors for "Unit Test Coding Problems." We utilize the count of errors in subcategories of "Unit Test Coding Problems" (mentioned in 3.2.1.1) to compute their priority order.

3.5.2 Student's Testing

- **Unit Test Errors**
  We employ same technique as that of "Potential Coding Bugs" (mentioned above) to obtain the priority order of errors for "Unit Test Errors." We utilize the count of errors in subcategories of "Unit Test Errors" – different exceptions; to compute their priority order.
- **Unit Test Failures**
  We employ same technique as that of "Signature Errors" (mentioned above) to obtain the priority order of errors for "Unit Test Failures." We group all the "Unit Test Failures" by source code file.

  We also add few lines of code (five lines) above the assertion failed line so that students can easily recognize the computation being performed in that assertion failed method.

Unit Test Failures

AnimalTest.java

testTurn: expected:<1> but was:<2>

```
66.         animal.setGridY(2);
67.
68.         animal.turn();
69.         animal.move(1);
70.
71.         assertEquals(1, animal.getGridX());
72.         assertEquals(1, animal.getGridY());
```

Figure 3.8: Feedback for Unit Test Failures

- **Unexecuted Methods**
  All the methods that have been uncovered are ranked based on the number of uncovered instructions (size of the method). This technique will begin the feedback with the largest uncovered method which upon being covered will improve the coverage drastically. This is followed by the second largest and so on.

  The first statement that contains the method definition is the error line of code with regards to the feedback type mentioned in Section 3.6.

Insect.java

Method not executed

```
39.      */
40.     public int getHealth()
41.     {
```

This method or constructor was not executed by any of your software tests. Add more tests to check its behavior.

Figure 3.9: Feedback for Unexecuted Methods

- **Unexecuted Statements**
  We obtain blocks of code (sequence of statements) that have not been covered and compute the size of the block. We rank the blocks based on their size and utilize this as the priority order in our feedback.

Fox.java

Statements not executed

```
81.                 {
82.                     distance = thisDistance;
83.                     nearest = rabbit;
84.                 }
```

These statements were not executed by your tests.

Figure 3.10: Feedback for Unexecuted Statements

- **Unexecuted Conditions**
  For each uncovered condition (branch) we count the number of missed branches and use this to rank the uncovered conditions. So, the priority order of feedback would begin with the uncovered condition with the highest number of missed branches followed by the second highest and so on.

Hive.java

Condition not executed

```
61.         Colony colony = (Colony)this.getWorld();
62.         if (count == 0 && this.getBees() >= 0)
63.         {
```

Not all possibilities for this decision were tested. Remember that when you have N simple conditions combined, you must test all N+1 possibilities.

Figure 3.11: Feedback for Unexecuted Conditions

3.5.3 Behavior

- **Unexpected Exceptions**
  We employ same technique as that of "Potential Coding Bugs" (mentioned above) to obtain the priority order of errors for "Unexpected Exceptions." We utilize the count of errors of different exceptions under "Unexpected Exceptions" to compute their priority order.

Fox.java

IndexOutOfBoundsException: Index: 0, Size: 0

```
75.         List<Rabbit> rabbits = this.getWorld().getObjects(Rabbit.class);
76.         Rabbit nearest = rabbits.get(0);
77.         double distance = distanceTo(nearest);
```

Figure 3.12: Feedback for Unexpected Exceptions

- **Infinite Recursion Problems**
  We employ same technique as that of "Signature Errors" (mentioned above) to obtain the priority order of errors for "Infinite Recursion Problems." We group all the "Infinite Recursion Problems" by source code file.
- **Infinite Looping Problems**
  We employ same technique as that of "Signature Errors" (mentioned above) to obtain the priority order of errors for "Infinite Looping Problems." We group all the "Infinite Looping Problems" by source code file.
- **Behavior Issues**
  We employ same technique as that of "Signature Errors" (mentioned above) to obtain the priority order of errors for "Behavior Issues." We group all the "Behavior Issues" by source code file and this would contain the hint message as code snippet from instructor test is not appropriate.
- **Out of Memory Errors**
  We employ same technique as that of "Signature Errors" (mentioned above) to obtain the priority order of errors for "Out of Memory Errors." We group all the "Out Of Memory Errors" by source code file.

3.5.4 Style

- **JavaDoc**
  We employ same technique as that of "Potential Coding Bugs" (mentioned above) to obtain the priority order of errors for "JavaDoc." We utilize the count of errors in subcategories of "JavaDoc" (mentioned in Section 3.2.1.1) to compute their priority order.
- **Formatting/ Whitespace**
  We employ same technique as that of "Potential Coding Bugs" (mentioned above) to obtain the priority order of errors for "Formatting/ Whitespace." We

utilize the count of errors in subcategories of "Formatting/ Whitespace" (mentioned in Section 3.2.1.1) to compute their priority order.

- **Naming**
  We employ same technique as that of "Potential Coding Bugs" (mentioned above) to obtain the priority order of errors for "Naming." We utilize the count of errors in subcategories of "Naming" (mentioned in Section 3.2.1.1) to compute their priority order.
- **Readability**
  We employ same technique as that of "Potential Coding Bugs" (mentioned above) to obtain the priority order of errors for "Readability." We utilize the count of errors in subcategories of "Readability" (mentioned in Section 3.2.1.1) to compute their priority order.
- **Coding Style**
  We employ same technique as that of "Potential Coding Bugs" (mentioned above) to obtain the priority order of errors for "Coding Style." We utilize the count of errors in subcategories of "Coding Style" (mentioned in Section 3.2.1.1) to compute their priority order.

3.6 Enhanced Error Messages

Students find most of the error messages very difficult to understand as they are slightly vague and do not convey the issues clearly. So, as mentioned in Section 3.4, we provide a detailed explanation of many error messages in the detailed feedback. This is placed right below the code snippet in the feedback and enables students to obtain a better understanding of the error which subsequently helps in accomplishing the formulated goals faster.

While most errors are convoluted for beginners, we specifically aim to enhance the compiler error messages as most of the beginners find it very difficult to interpret them. We also provide a smart diagnosis for the "Infinite Recursion" problems that many students tend to struggle.

3.6.1 Enhanced Compiler Error Messages

As studied by Brett A. Becker et al. [25], compiler errors are usually the most difficult ones for beginners to resolve. It is necessary for beginners to clearly understand the compiler errors so that they can fix them and make further progress with their code. They are considered to be convoluted mostly because of the way the default compiler error messages are implemented. These messages are inadequate, time-taking to understand and often require assistance from experts to understand the errors.

This lead to the idea of providing better and detailed compiler error messages to students. We have utilized Brett A. Becker et al. [25] work on Enhanced compiler error messages and implemented them in Web-CAT.

3.6.1.1 Implementation

We learned about all the enhanced compiler messages from Devon's [26] work and built a regex matcher that maps the existing compiler error messages to their corresponding enhanced messages. The enhanced messages contain the entity names from the user's code instead of being a generic message.

3.6.2 Infinite Recursion Diagnosis

StackOverflowErrors/Infinite Recursion problems are very complicated for seasoned programmers, let alone beginners. It is very difficult to locate the method calls that are involved in the unterminated recursion from the stack trace as these are sandwiched between multiple other calls. There are usually three kinds of issues that cause StackOverflowErrors: Unintended Recursions, Unintended Recursions with Cyclic Relationships and Intended Recursion with inappropriate Termination Conditions [38].

Fox.java

StackOverflowError

```
104.        {
105.            nearestRabbit();
106.        }
```

There is a recursion with cyclic relationships:
Fox.java:105. nearestRabbit(); → Animal.java:80. animalsNearBy() → Fox.java:103. rabbitsNearBy(); → Fox.java:105. nearestRabbit();

Figure 3.13: Feedback for Infinite Recursion Errors

3.6.2.1 Implementation

We compute the cycles in the stack and sort them based on the number of such cycles in the stack. We display every cycle with the start of the cycle as the code snippet and the detailed message consisting of each call in the cycle represented by their file name, line number and code snippet.

The fact that the code snippet is tied along with the error message and the detailed error message provides students with a better understanding and impression of the error. They would be able to precisely locate the error and interpret it in detail with the help of different elements of feedback that is provided to them for each error.

Chapter 4

Intelligent Feedback Walkthrough

In this chapter, we navigate through an example of a live intelligent feedback page of a student submission in Web-CAT. We will demo how the student would utilize this feedback. Then, we describe the benefits of intelligent feedback in different situations in comparison with the old feedback in Web-CAT.

4.1 Usability of Intelligent Feedback

We describe few of the captures of intelligent feedback that a student (let us call the student as "Carla") might view to understand his goals and feedback:

1. First glance at the feedback (Figure 4.1) will help Carla understand the Summary and realize that his "Testing" category is of high priority and needs to be worked upon first because its expanded by default, highlighted with a red color and progress bar is not 100% green.

   Carla can also figure out that "Behavior" category is not complete because few of the subcategories are marked with a cross mark, and the progress bar is not 100% green, but "Coding" and "Style" categories are good.

   As Carla inferred that "Testing" category is the "Goal" and could also see the detailed feedback of that category, Carla would go ahead and fix the Testing issues: "Unexecuted Statements" and "Unexecuted Conditions."

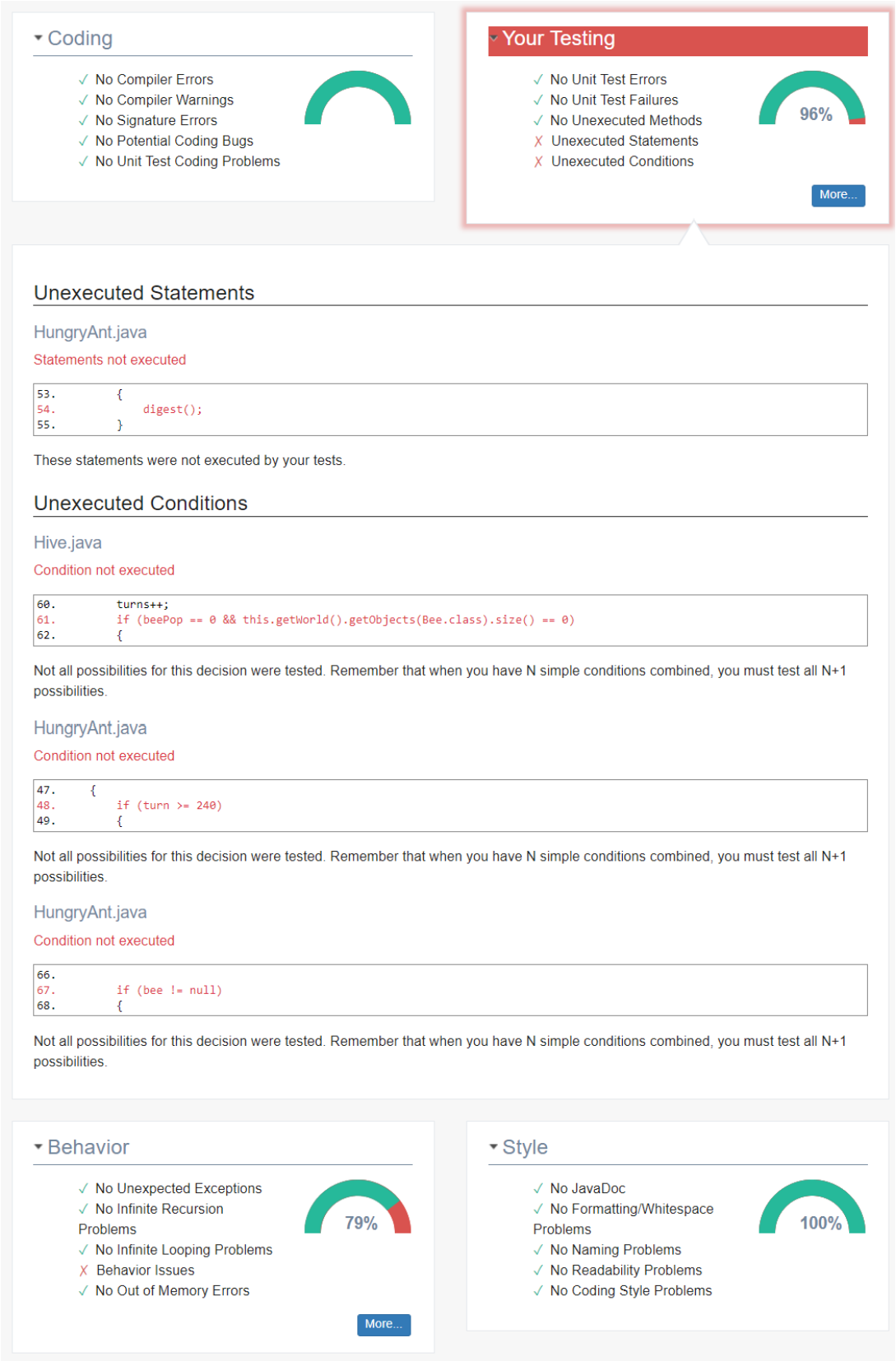


Figure 4.1: An Initial glance at the Feedback

2. Carla will expand the “Behavior” category and resolve the “Behavior Issues” exploiting the feedback (Figure 4.2).

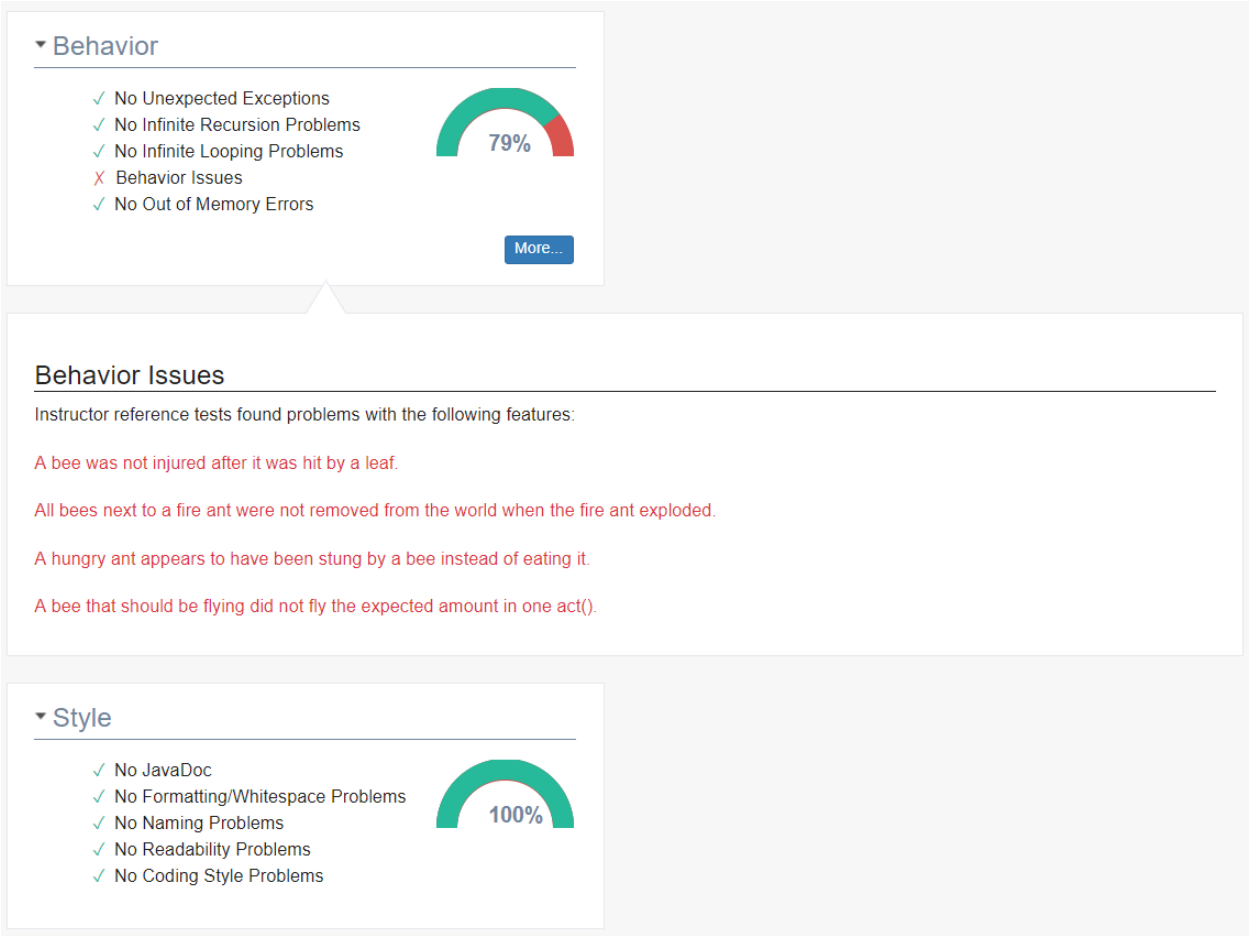


Figure 4.2: A Second glance at the Feedback

3. Carla will understand that the “Style” category is completely resolved; so, all the issues are fixed and would resubmit his code.

4.2 Benefits of Intelligent Feedback over Old Feedback

We describe the benefits of the intelligent feedback for different problems/situations listed in the Problem Statement by comparing the feedback provided by the old and new feedback mechanisms.

We compare both the feedbacks for few scenarios:

1. The code has a lot of Style issues, and one such kind of errors among them is the “Potential Coding Bugs” – which are the most important and severe errors among all the Style errors.

File Details

File	Staff Cmts	Staff Pts	AutoGrade Cmts	AutoGrade Pts	Methods and Conditions Executed
Animal.java	0	0.0	1	-9.7	16.7%
AnimalTest.java	0	0.0	5	-14.1	0.0%
Field.java	0	0.0	4	-4.0	100.0%
FieldTest.java	0	0.0	0	0.0	
Fox.java	0	0.0	1	-19.4	7.7%
Rabbit.java	0	0.0	0	0.0	100.0%
README.TXT	0	0.0	0	0.0	

Figure 4.3: Point Deductions and Code Coverage of each File

AnimalTest.java

Select from previous comments ▾ | Error ▾ | To Everyone ▾ | B I U | Show all comments ▾

```
1   import sofia.micro.*;
2   import sofia.util.Random;
```

Error [PMD]: -1
The package or class imported by this statement is never used. If you are not going to use the package or class you should remove the import statement.

```
3   import java.util.List;
```

Error [PMD]: -1
The package or class imported by this statement is never used. If you are not going to use the package or class you should remove the import statement.

```
4
5   // Virginia Tech Honor Code Pledge:
6   //
7   // As a Hokie, I will conduct myself with honor and integrity at all times.
8   // I will not lie, cheat, or steal, nor will I accept the actions of those
9   // who do.
10  XXX
11
12  //-----------------------------------------------------------------------
13  /**
14   *  Tests all Animal
15   *  class methods.
16   *
17   *  @author XXX
18   *  @version 2018.04.24
19   */
20  public class AnimalTest extends TestCase
21  {
22      //~ Fields ..............................................................
23      private Field field;
```

Error [PMD]: -2
The private field 'field' is never used, so it appears to be unnecessary.

Error [PMD]: -2
This field is only used in one method (or constructor) in this class. Make it a local variable within the method instead.

```
24
25      //~ Constructor .........................................................
26
27      // -------------------------------------------------------
28      /**
```

Figure 4.4: Old Feedback for Style Issues in a File

We can see from Figure 4.3 and Figure 4.4 that there are many Style issues and Code Coverage issues, so there is a lot of feedback for the student to be bothered. But, the most important ones are the “Potential Coding Bugs” that are not highlighted appropriately to convey their priority. So, students cannot understand the importance of such issues.

The new intelligent feedback as depicted in Figure 4.5 clearly highlights the “Coding” section as the Priority one and lists the “Potential Coding Bugs” to signify that it is the “Goal” and needs to be worked on.

We also clearly display the code snippet along with the error message to convey a better illustration of the error; whereas in the old feedback student should drill down to every source code file to view different errors.

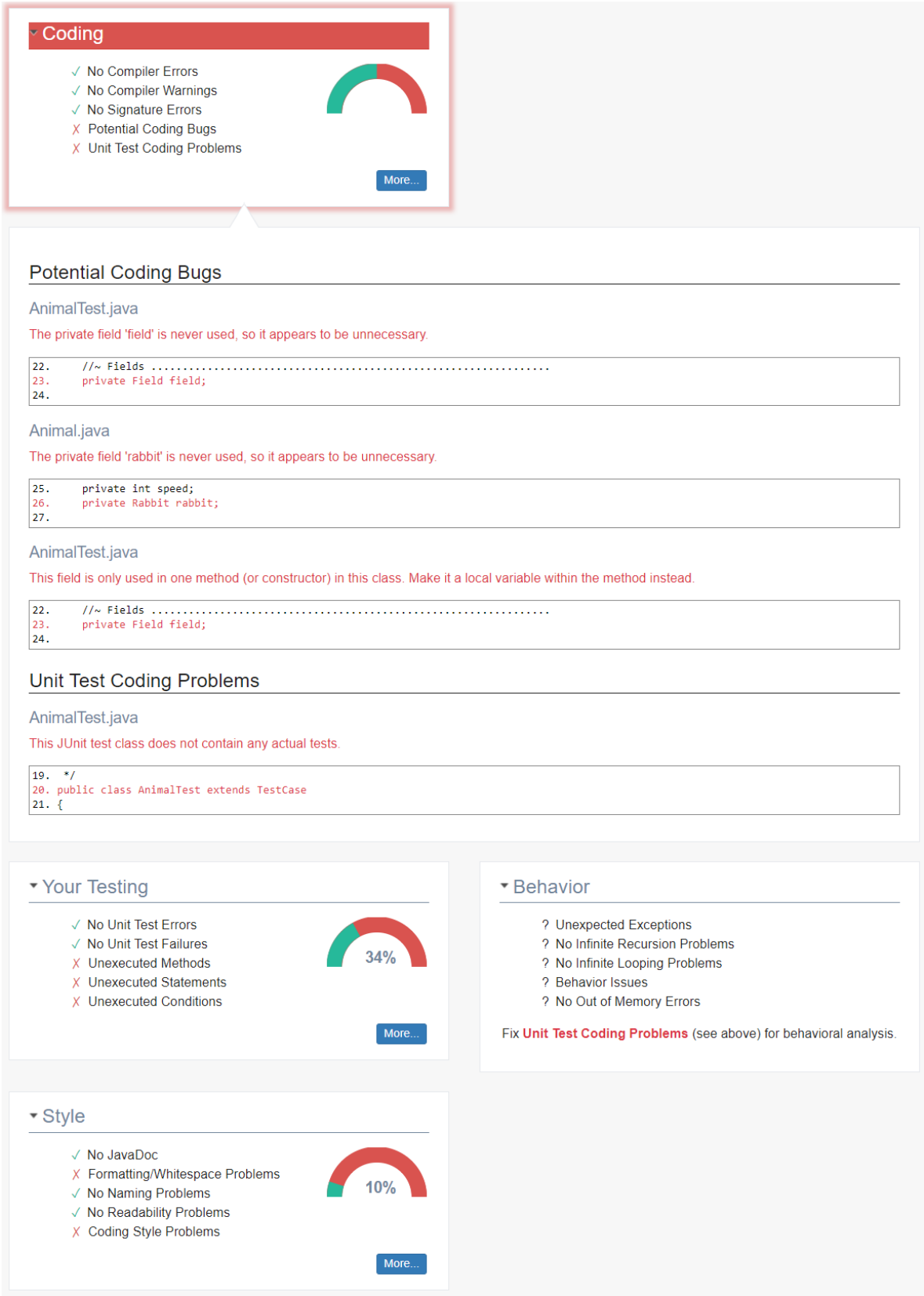


Figure 4.5: Intelligent Feedback on Coding Issues

2. Student incurred Unexpected Exceptions (Unit Test Errors) from their tests and Unexpected Exceptions (Behavior section) from the Instructor's reference tests.

   The old feedback as depicted in Figure 4.6 lists out all the issues from the Student's Tests and the Instructor Reference test results without clearly conveying that resolving the issues pertaining to Student's Tests is of highest priority, as that might resolve issues pertaining to Reference tests as well.

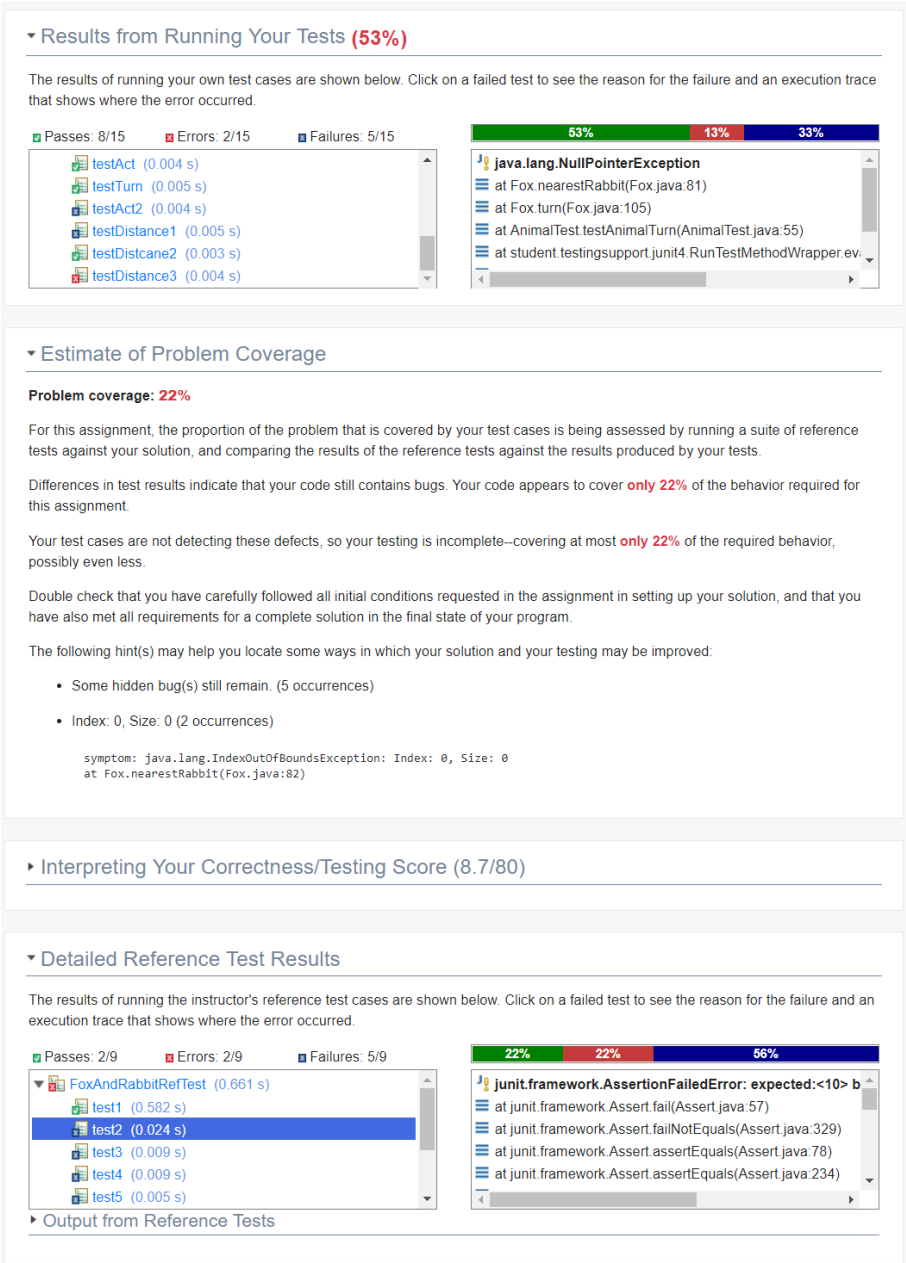


Figure 4.6: Old Feedback for Test Results

Our intelligent feedback as illustrated in Figure 4.7 conveys that Student's Testing category is the Goal and needs to be worked upon before anything else. This feedback also locates the code snippet causing the exceptions and layouts the uncovered elements by student's tests in priority order. This improves the understanding of the exception; as they need not worry about where and how the exception had occurred.

Coding

- ✓ No Compiler Errors
- ✓ No Compiler Warnings
- ✓ No Signature Errors
- ✓ No Potential Coding Bugs
- ✓ No Unit Test Coding Problems

Your Testing

- ✗ Unit Test Errors
- ✗ Unit Test Failures
- ✗ Unexecuted Methods
- ✗ Unexecuted Statements
- ✓ No Unexecuted Conditions

48%

More...

Unit Test Errors

Fox.java

NullPointerException: java.lang.NullPointerException

```
80.     {
81.         List<Rabbit> rabbits = this.getWorld().getObjects(Rabbit.class);
82.         Rabbit nearest = rabbits.get(0);
```

Fox.java

IndexOutOfBoundsException: java.lang.IndexOutOfBoundsException: Index: 0, Size: 0

```
81.         List<Rabbit> rabbits = this.getWorld().getObjects(Rabbit.class);
82.         Rabbit nearest = rabbits.get(0);
83.     }
```

Unit Test Failures

FieldTest.java

testPopulate1: expected:<1> but was:<2>

```
83.         Random.setNextInts(40);
84.         field.populate(1, 0);
85.         assertEquals(1, field.getObjects(Rabbit.class).size());
86.         assertNotNull(field.getOneObjectAt(40, 40, Rabbit.class));
```

FieldTest.java

testPopulate2: junit.framework.AssertionFailedError

```
94.         Random.setNextInts(40);
96.         assertEquals(1, field.getObjects(Rabbit.class).size());
97.         assertNotNull(field.getOneObjectAt(40, 40, Fox.class));
98.     }
```

FoxTest.java

testAct2: expected:<1> but was:<0>

```
82.         field.add(bunny, 6, 6);
85.         assertEquals(1, field.getObjects(Fox.class).size());
86.         assertEquals(1, field.getObjects(Rabbit.class).size());
87.     }
```

FoxTest.java

testDistance1: expected:<Rabbit at (30, 30)> but was:<Rabbit at (4, 4)>

```
94.         field.add(bunny, 4, 4);
97.         field.add(velvet, 30, 30);
98.         assertEquals(sue.nearestRabbit(), bunny);
99.     }
```

Unexecuted Methods

Animal.java

Method not executed

```
62.     */
63.     public int getSpeed()
64.     {
```

This method or constructor was not executed by any of your software tests. Add more tests to check its behavior.

Animal.java

Method not executed

```
75.     */
76.     public void turn()
77.     {
```

This method or constructor was not executed by any of your software tests. Add more tests to check its behavior.

Animal.java

Method not executed

```
86.     */
87.     public void act()
88.     {
```

This method or constructor was not executed by any of your software tests. Add more tests to check its behavior.

Behavior

- ✗ Unexpected Exceptions
- ✓ No Infinite Recursion Problems
- ✓ No Infinite Looping Problems
- ✗ Behavior Issues
- ✓ No Out of Memory Errors

22%

More...

Style

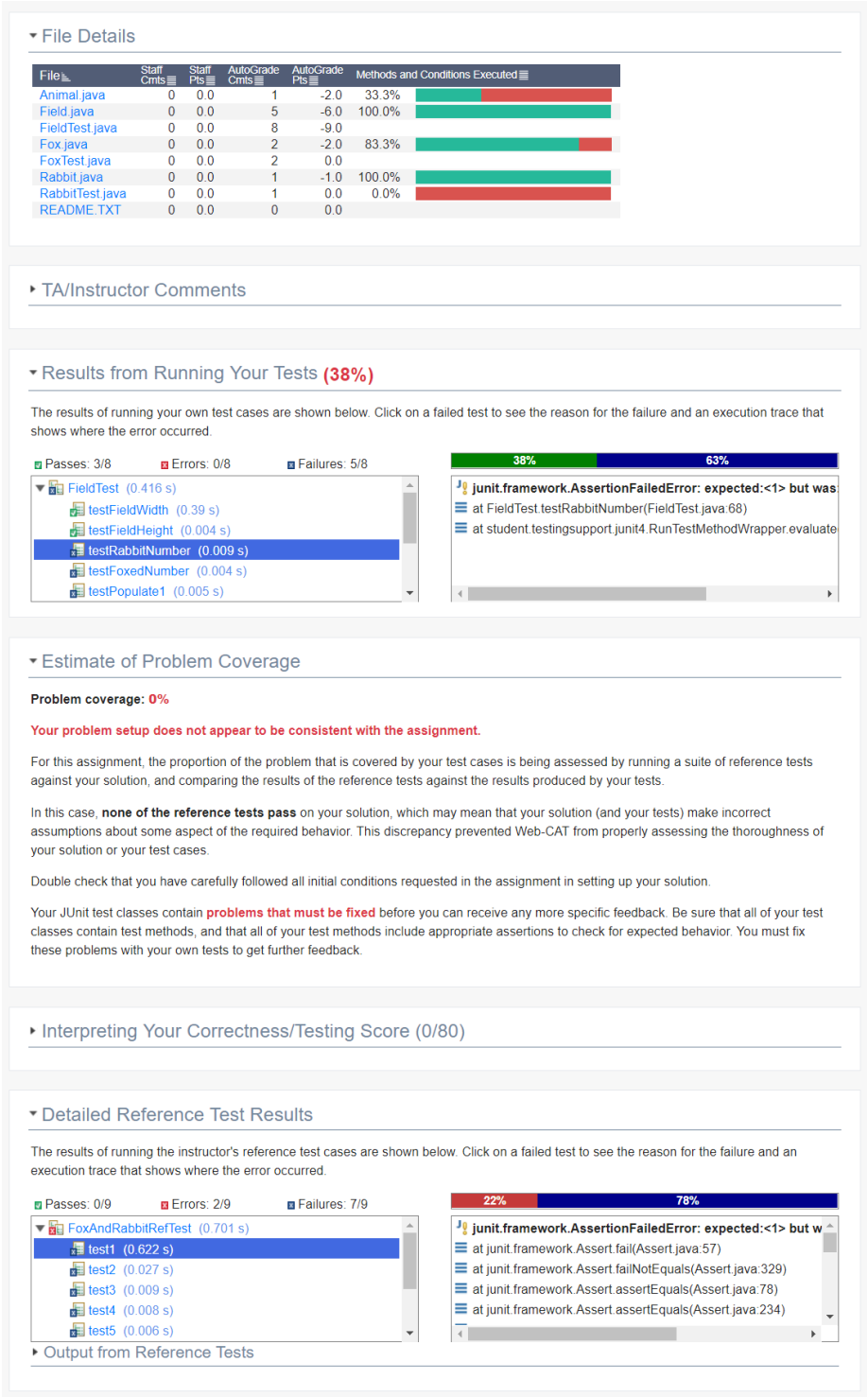
- ✓ No JavaDoc
- ✓ No Formatting/Whitespace Problems
- ✓ No Naming Problems
- ✓ No Readability Problems
- ✗ Coding Style Problems

95%

More...

Figure 4.7: Intelligent Feedback: Detailed Feedback for Student's Unit Tests

3. The code has a lot of issues pertaining to Coding, Testing, and Style categories that lead to a lot of feedback being generated.



File Details

File	Staff Cmts	Staff Pts	AutoGrade Cmts	AutoGrade Pts	Methods and Conditions Executed
Animal.java	0	0.0	1	-2.0	33.3%
Field.java	0	0.0	5	-6.0	100.0%
FieldTest.java	0	0.0	8	-9.0	
Fox.java	0	0.0	2	-2.0	83.3%
FoxTest.java	0	0.0	2	0.0	
Rabbit.java	0	0.0	1	-1.0	100.0%
RabbitTest.java	0	0.0	1	0.0	0.0%
README.TXT	0	0.0	0	0.0	

TA/Instructor Comments

Results from Running Your Tests (38%)

The results of running your own test cases are shown below. Click on a failed test to see the reason for the failure and an execution trace that shows where the error occurred.

Passes: 3/8 Errors: 0/8 Failures: 5/8

38% 63%

FieldTest (0.416 s)
testFieldWidth (0.39 s)
testFieldHeight (0.004 s)
testRabbitNumber (0.009 s)
testFoxedNumber (0.004 s)
testPopulate1 (0.005 s)

junit.framework.AssertionFailedError: expected:<1> but was
at FieldTest.testRabbitNumber(FieldTest.java:68)
at student.testingsupport.junit4.RunTestMethodWrapper.evaluate

Estimate of Problem Coverage

Problem coverage: 0%

Your problem setup does not appear to be consistent with the assignment.

For this assignment, the proportion of the problem that is covered by your test cases is being assessed by running a suite of reference tests against your solution, and comparing the results of the reference tests against the results produced by your tests.

In this case, **none of the reference tests pass** on your solution, which may mean that your solution (and your tests) make incorrect assumptions about some aspect of the required behavior. This discrepancy prevented Web-CAT from properly assessing the thoroughness of your solution or your test cases.

Double check that you have carefully followed all initial conditions requested in the assignment in setting up your solution.

Your JUnit test classes contain **problems that must be fixed** before you can receive any more specific feedback. Be sure that all of your test classes contain test methods, and that all of your test methods include appropriate assertions to check for expected behavior. You must fix these problems with your own tests to get further feedback.

Interpreting Your Correctness/Testing Score (0/80)

Detailed Reference Test Results

The results of running the instructor's reference test cases are shown below. Click on a failed test to see the reason for the failure and an execution trace that shows where the error occurred.

Passes: 0/9 Errors: 2/9 Failures: 7/9

22% 78%

FoxAndRabbitRefTest (0.701 s)
test1 (0.622 s)
test2 (0.027 s)
test3 (0.009 s)
test4 (0.008 s)
test5 (0.006 s)

junit.framework.AssertionFailedError: expected:<1> but w
at junit.framework.Assert.fail(Assert.java:57)
at junit.framework.Assert.failNotEquals(Assert.java:329)
at junit.framework.Assert.assertEquals(Assert.java:78)
at junit.framework.Assert.assertEquals(Assert.java:234)

Output from Reference Tests

Figure 4.8: Old Feedback of Code with a lot of Issues

We can see from Figure 4.8 that Old Style feedback does not contain a simple summary for the student to skim through the page and understand the issues in their code. This contains a lot of feedback that would baffle the students, as they do not know what to look at and how they are related to each other or their progress.

All the Style and Coding issues (as seen in below figure) can only be viewed by drilling down to the specific file that is tedious, and there is no indicator to mark high priority issues appropriately.

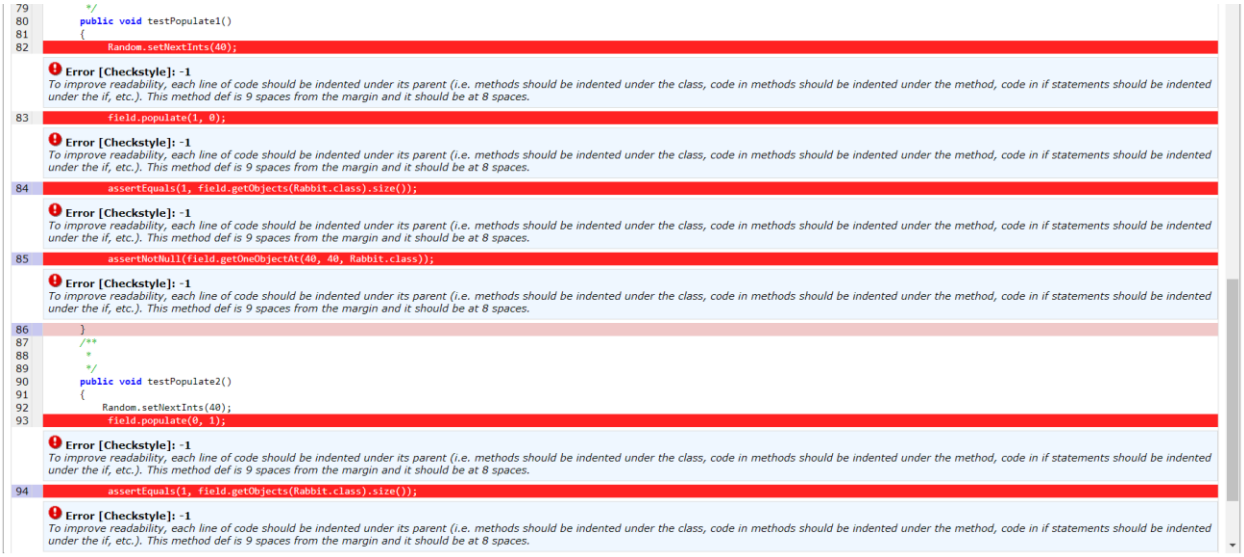


Figure 4.9: Old Feedback Embedded in the File

We can see below in Figure 4.10 that the intelligent feedback provides a simple summary so that the student can have a clear understanding of their code in one view. This feedback also marks high priority issues and describes the issues in context along with the code snippet and error messages to provide easy access to errors.

Smart categorization of errors provides a clear understanding of the errors so that students know the different kinds of issues affecting their code. They would clearly understand “what issue to look at” and “where to look at.”

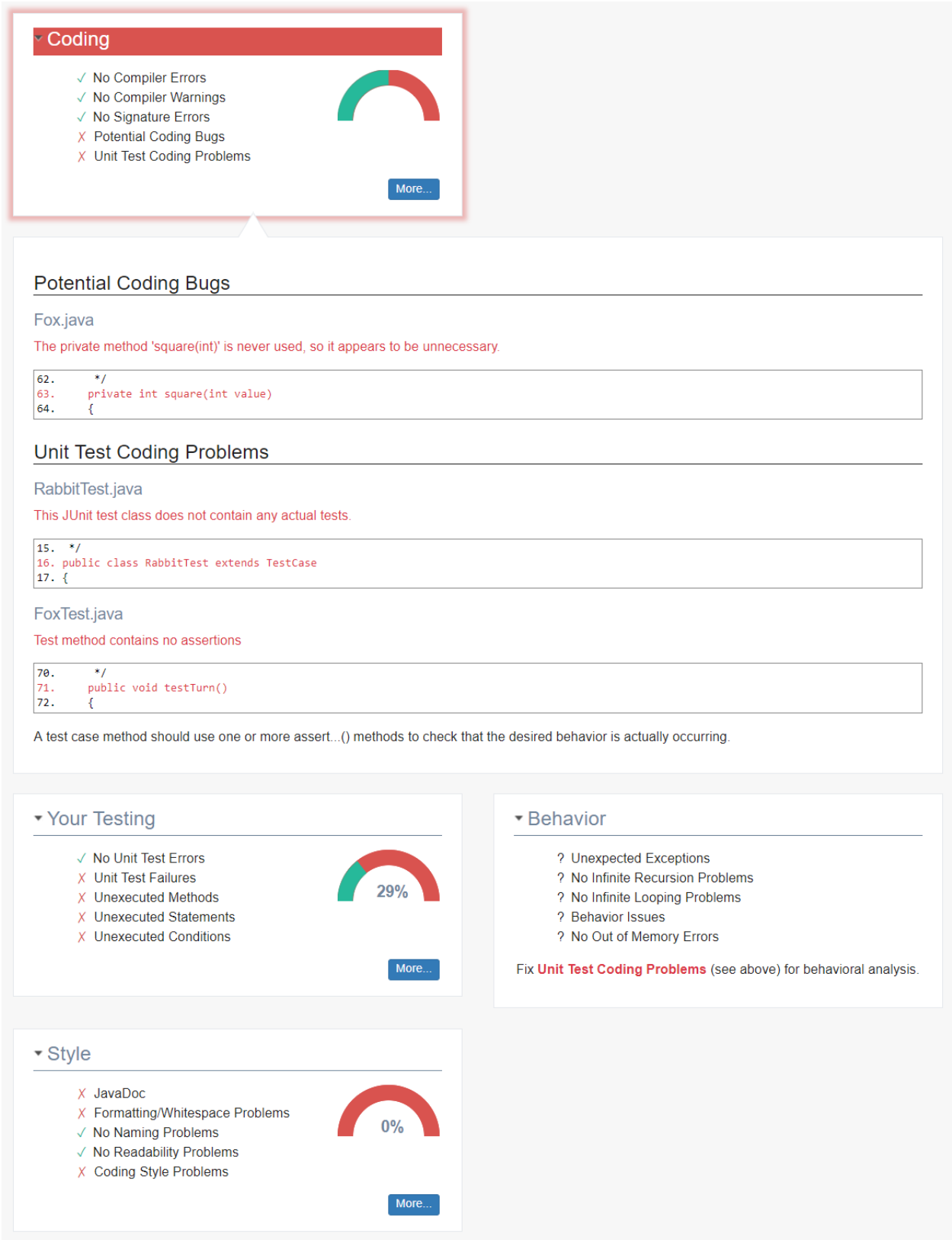


Figure 4.10: Summary generated by Intelligent Feedback

As we can see below (Figure 4.11) in the Detailed Feedback of the Style category; the errors are smartly grouped and listed in priority order so that the feedback acts as an excellent guide to the students. We claim that if the student resolves their errors in the specified order, they would be able to make faster progress.

We limit the amount of feedback that we present and compute remaining such errors in other files and list out the feedback as "Likewise, there are 3 issue(s) in FieldTest.java" so that the student can obtain a fine understanding of all such errors in all the source files.

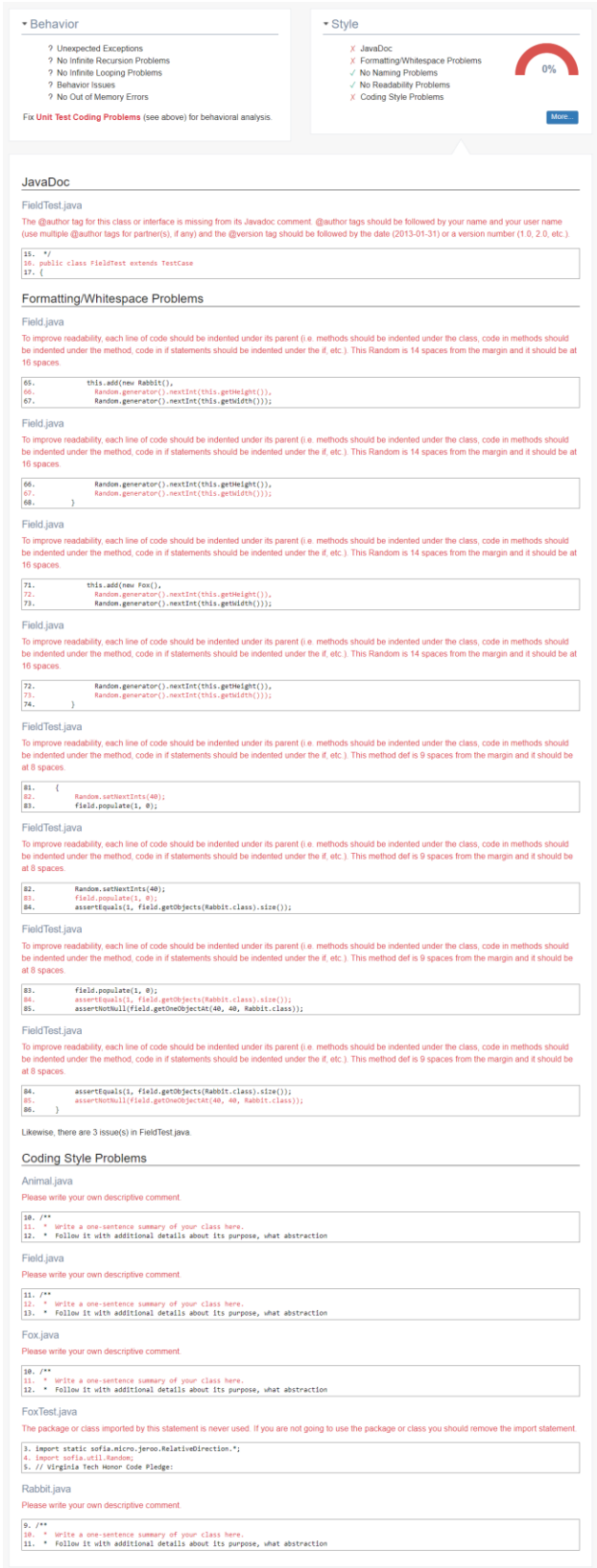


Figure 4.11: Detailed Intelligent Feedback for Style Issues

Chapter 5

Survey Results and Discussion

To assess student perceptions of the new feedback design, a survey was used to measure the impact of this feedback on the students. The primary goal of the intelligent feedback is to convey the errors effectively and reduce the students time and efforts in understanding the errors in their code. Also, another major objective of the intelligent feedback is to provide "Goals" to the students. A user-based survey can best evaluate all these features.

5.1 Survey Design

We have utilized the Qualtrics Survey Software to distribute the survey. The survey was designed to evaluate the uselessness, impact and added learning benefits that the intelligent feedback would provide to the students. The survey questions were constructed to assess our design goals and understand if we have solved all the problems mentioned in the Problem Statement successfully.

The survey consists of 10 questions on a seven-level Likert scale with the available options for each of them being: Strongly agree, Agree, Somewhat agree, Neither agree nor disagree, Somewhat disagree, Disagree and Strongly disagree. Students would mark one of those options to convey their level of agreement with each statement. To obtain consistent and reliable results, we have added negative statements in between the positive ones.

The survey also consists of a free-form question to understand the further improvements that students would want to see in the intelligent feedback.

5.2 Population Used for Study

Our intelligent feedback is integrated into Web-CAT for CS 1114 course at VirginiaTech: "Introduction to Software Design" for a couple of labs and a programming assignment during Spring 2018. This course had 404 students enrolled in a couple of sections. After the usage of this feedback for a week, we circulated the survey that students were requested to complete.

Assignment
Lab 13: Weekly Calendar
Lab 14: Testing and Debugging
Program 5: Ants vs. SomeBees

Table 5.1: Assignment details where Intelligent Feedback was provided

5.3 Results

In this section, we describe the Likert-Scale and free-form question results and the discussion pertaining to them. We used Qualtrics to aggregate the data.

5.3.1 Likert-Scale Question Results

Table 5.2 describes the results of the Likert-Scale Questions; where 60 students responded to these questions. The overall results are very promising and conveyed that students had a very positive impression about the intelligent feedback. We consider these results to be tremendous because this new feedback was used for a smaller duration and students usually take time to get accustomed to a change. So, we expect that longer use of this feedback would deliver incredible results.

	Question	% of strongly/somewhat agreed	Strongly Agreed to Strongly Disagreed
Modularized Summary, Enhanced Error Messages			
1	This new style of feedback was easier to understand than before	**63.33%** agreed	
2	I found the old style of feedback easier to understand	**45.01%** agreed	
3	The new style of feedback made me more discouraged with my work	**15.25%** agreed	
Goal Formulation			
4	It helped me decide on my goal for what to work on next	**66.67%** agreed	
5	It was easier to understand what I needed to work on with the new feedback	**67.24%** agreed	
Prominent Error Code Snippets			
6	The code snippets in the feedback helped me find problems more easily	**83.34%** agreed	
7	The new feedback helped me improve my score faster	**40.68%** agreed	

Table 5.2: Likert-Scale questions and % of agreement (cont.)

Priority Ordering of Errors			
8	I was able to resolve issues with my assignment faster with this form of feedback	**61.67%** agreed	
9	The new feedback is too long and complicated	**37.28%** agreed	
10	I prefer the old style of feedback	**37.28%** agreed	

Table 5.2: Likert-Scale Questions and % of Agreement

We can infer that Q1, Q2, and Q3 are similar, but we obtained slightly ambiguous responses on Q2; which could be attributed to the negative phrasing of Q2. The strong agreement on Q1 implies that students could understand and interpret different parts of the feedback effortlessly. A very low agreement on Q3 indicates that the intelligent feedback kept the students encouraged by guiding them to understand the issues clearly and accomplish the formulated goals effectively.

Q4 and Q5 have obtained a strong agreement that clearly conveys that students understand "what to do next" and the feedback guided them in formulating smart goals.

Q6 achieved a very strong agreement that emphasizes that better illustration of errors by the "Prominent Error Code Snippets" made it easier for students rather than them drilling down to each file to find the errors. Q7 attained slightly lower agreement than expected but we do not expect a significant improvement in students' scores immediately. We believe that with longer use of this feedback students learning abilities would improve significantly and they would understand all the issues in their code easily due to a better illustration of the errors. This will implicitly contribute to improving their score in the long run. But the fact that we got a decent level of agreement for little usage is very promising.

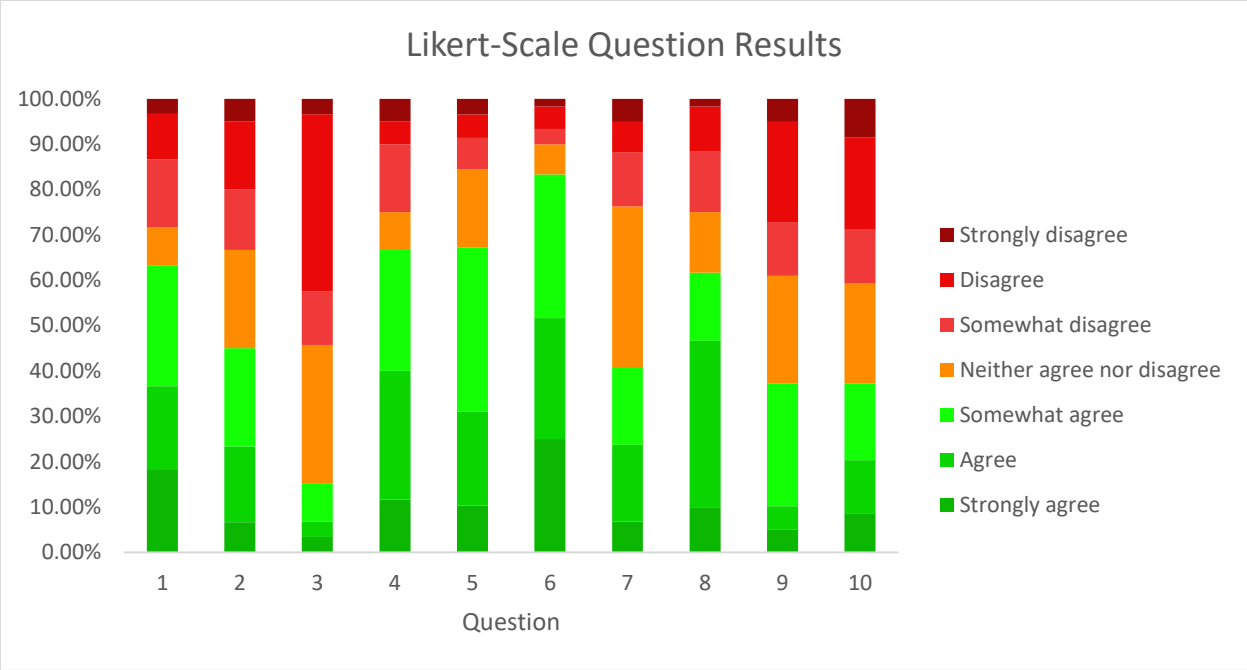


Figure 5.1: Likert-Scale Question Results

High agreement on Q8 indicates that the "Priority Order of Errors" guided students in resolving the issues comfortably and efficiently. We obtained a slightly higher agreement on Q9 due to a high "Neither agree nor disagree" even though we smartly limit the feedback. This might be because of using this feedback for a very low duration, and students should ideally use it longer to appreciate and understand the benefits of this feedback mechanism. To tackle this issue of providing a not so long feedback and to avoid any bias in the survey results, we intend to conduct a user study as mentioned in Section 6.2.4 to learn about the right kind of feedback that must be provided to the students.

We also believe that Q10 obtained slightly higher agreement since the students used this feedback for a limited duration and a few of them might have found it a bit difficult to interpret the feedback effectively (during the initial usage of the feedback). We expect that as students get used to this intelligent feedback, they would prefer this feedback mechanism for all their courses.

We can conclude from the above survey results that few of them have obtained wonderful results pertaining to the design of the intelligent feedback. While all the results are very positive and promising, we yearn for the results to be better. As mentioned in Section 6.2.3 we would like to observe the correlation between

student's performance in the course and their survey results to inspect if their performance caused them to view the feedback differently.

5.3.2 Free-Form Question Results

We had a free-form question: "How can we further improve the feedback generated by Web-CAT?" to understand the further improvements that we should carry out pertaining to the intelligent feedback. We can see from the complete list of responses below that we have got quite a few interesting responses:

- *It was great*
- *Teach how to use feedback in class*
- *It was sometimes difficult to determine what exactly to fix. Still somewhat vague.*
- *Provide feedback about a wider array of problems*
- *Show all issues at once, not just a couple at a time.*
- *Don't display everything that is correct. Only display things that are incorrect to clean up the interface*
- *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.*
- *When the code is wrong, be more specific in how it should function.*
- *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.*
- *Increase relevant feedback, the "heatmap" system does not help diagnose syntax errors, etc.*
- *Don't use it*
- *No submission energy*
- *None*
- *Get rid of the cool down. It didn't change how I submitted at all*
- *N/A*
- *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.*
- *This was my first time using Web-CAT, so I don't have anything to compare it to. That being said, I don't know anybody who has ever said, "Oh wow, using Web-CAT is a real joy." For one assignment, Web-CAT had me losing 18 points because I forgot one this.super();. I recognize that it is difficult to grade so many students, but Web-CAT does not do a very good job.*

- *Increase the amount of errors that aren't hidden.*
- *To view more details from the bars in line rather than grid*
- *dont do the energy*
- *Place more focus on the actual errors in the submitted code. The new categories displaying many potential sources of error (such as infinite recursions or out of memory errors) even though they passed testing was more distracting and confusing than helpful.*
- *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.*
- *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.*

We can see that we have a couple of interesting suggestions about guiding students to a logically correct solution by providing next-step hints; which we have listed out under Future Work. Most of the other recommendations have been incorporated in our feedback, but few students (as seen in the responses) are unable to understand those benefits because they had limited exposure to this feedback. Upon reanalyzing those improvement suggestions, we certainly believe that the students can clearly understand those issues with fairly longer usage.

Chapter 6

Conclusion

In this thesis, we had described the design and implementation of intelligent feedback for Java Programming Assignments. We also outlined the motivation and the related work pertaining to this research. The following features of this feedback were explained in this thesis:

1. Modularized Summary
2. Goal Formulation
3. Prominent Error Code Snippets
4. Priority Order of Errors
5. Enhanced Error Messages

We had also described the implementation details of the above features. We had analyzed the differences and improvements of this new feedback when compared to the old feedback design in Web-CAT. We also expressed our evaluation plan and a user survey plan and the expected results.

6.1 Contributions

The intelligent feedback design as described in this thesis with all the key features mentioned above helps students to interpret the errors appropriately when a lot of feedback is provided. Modularized summary assists students in understanding important issues in their code which they might miss otherwise. Modularized summary and goal formulation helps beginners to identify high priority issues and understand "what to do next." Priority ordering of errors helps students to resolve high priority issues earlier, and this intelligent ordering of errors guides students to accomplish the formulated goals faster. Enhanced error messages provide students with a better understanding of the errors which aids in accomplishing the goals easily.

By utilizing this feedback, we expect beginners to formulate effective goals to resolve all the errors in their code. These errors have been made comprehensible due to different features of intelligent feedback.

6.2 Future Work

In this section, we describe the future work that we would like to pursue regarding the intelligent feedback. We detail below the additional features that we would like to research upon and the student performance data based evaluations.

6.2.1 Additional Features

Even though we have accomplished all the design goals that we had described, we would like to incorporate the following features that would make this feedback much better:

- **Integration to Other Languages**: We would like to utilize this design and build this kind of feedback for other programming languages like Python, C++.
- **Next-Step Hint Generation:** Many researchers have carried out a lot of work pertaining to the Next-Step hint generation of an error-free code. Integrating this feature to our feedback would facilitate in improving the problem coverage of the code easily.
- **Better Diagnosis of All Errors:** We would like to provide hints and diagnosis to resolve all kinds of errors utilizing patterns in the code and Machine Learning techniques. This kind of an automated program repair technique would be of great help to the students.
- **Program Slicing of Error Code Snippets:** Explore different slicing techniques to provide relevant code snippets that affect the error line of code rather than providing a generic context- few lines above and below the error line for all the errors [44]. This would especially be very handy for Unit Test Errors and Failures as smart slicing will assist the students to understand the cause of those issues.

6.2.2 Plan for Additional Evaluation

We would like to study the performance data of students to help us obtain the learning impact of intelligent feedback on students. We describe below the evaluation strategies that we would use for analyzing the performance data of students' submissions:

- **Number of submissions per assignment:** We would like to evaluate the average number of submissions made by a student in an assignment where we provide intelligent feedback and compare that against the previous years' assignment where intelligent feedback was not provided. The expectation is that the average number of submissions in the former case would be lower as students might tend to use fewer submissions because of the better understanding of the errors. This also implies that it saves a lot of students' time in resolving the errors.
- **Difference in error rate between initial and final submission:** We would like to compute the average difference in error rate (number of errors/KLOC) between initial and final submissions. We expect this to be higher (when compared to the old feedback) as students might resolve more errors with the new feedback. This might help us conclude that our feedback made it easier for students to understand the errors and helped them immensely in resolving the errors.
- **Difference in error rate between consecutive submissions:** We would like to compute the average difference in error rate of all consecutive submissions; this gives us the number of errors resolved per submission. We also expect this to be higher (in comparison with the old feedback). This might help us conclude that the errors are resolved faster by utilizing the new feedback.
- **Difference in time between initial and final submission:** We would like to compute the average difference in submission timestamp between initial and final submissions. We expect this to be lower (when compared to the old feedback) as students are expected to accomplish the formulated goals faster with the new feedback.

6.2.3 Correlating Survey Responses with Performance

As discussed in Section 5.3, the survey results were encouraging. However, it would be useful to explore possible relationships between how students perform on assignments and how they react to the feedback changes. As part of this evaluation we would like to answer following questions:

- Whether students who have been only doing well in the course (good scores), did not appreciate the change and so did not find the feedback to be impactful that led them to indicate lower agreement to the survey questions.
- If students who have not been only doing well found the feedback to be very difficult to understand that led them to indicate lower agreement to the survey questions.
- Whether students who have been only doing well in the course (good scores), understood the feedback easily that led them to indicate higher agreement to the survey questions.
- If students who have not been only doing well found the feedback to be very helpful that led them to indicate higher agreement to the survey questions.

By analyzing the above questions, we can learn if students who answered the survey questions in one way performed in a certain way in the course. We can determine the correlation between survey results and students' performance through this evaluation and learn if we should make certain enhancements to the intelligent feedback to cater to the needs of all types of students.

6.2.4 Evaluating Impact with a User Study

We would like to conduct a user study to understand the impact and usefulness of varying lines of code snippets to the students. This would help us obtain the right amount of context-based code snippets that should be provided to the students. Also, we would like to figure out if our intelligent feedback is formulating right kind of goals for the students and whether students understand the error easily.

Bibliography

[1] Stephen H. Edwards and Manuel A. Perez-Quinones. 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. ACM, 2008.

[2] Chris Kite. Codeconquest, http://www.codeconquest.com/programming-help/why-is-programming-so-hard, last accessed 04-04-2018.

[3] Quora Inc. Quora, https://www.quora.com/Why-is-programming-so-hard-1, last accessed 04-04-2018.

[4] Wikimedia Foundation. Wikipedia, https://en.wikipedia.org/wiki/Computer_programming, last accessed 04-04-2018.

[5] Richard E. Pattis. CMU, https://www.cs.cmu.edu/~pattis/quotations.html, last accessed 04-04-2018.

[6] IUPUI, http://www.iupui.edu/~cletcrse/course_mat/06_mark/html/course_files/3_10.html, last accessed 04-04-2018.

[7] University of Reading, https://www.reading.ac.uk/internal/engageinfeedback/Whyisfeedbackimportant/, last accessed 04-04-2018.

[8] Marelisa Fabrega, https://daringtolivefully.com/goal-quotes, last accessed 04-04-2018.

[9] Pardha Koyya, Young Lee, and Jeong Yang. Feedback for Programming Assignments Using Software-Metrics and Reference Code. In *ISRN Software Engineering Volume 2013 (2013)*, Article ID 805963, 8 pages.

[10] Victor J. Marin, Tobin Pereira, and Srinivas Sridharan. Automated Personalized Feedback in Introductory Java Programming MOOCs. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference*, San Diego, CA, USA. 18 May 2017.

[11] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, Pages 15-26. ACM, 2013.

[12] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Feedback generation for performance problems in introductory programming assignments. In

Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, Pages 41-51. ACM, 2014.

[13] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. Providing Meaningful Feedback for Autograding of Programming Assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, Pages 278-283. ACM, 2018.

[14] Luciana Benotti, Federico Aloi, Franco Bulgarelli, and Marcos J. Gomez. The Effect of a Web-based Coding Tool with Automatic Feedback on Students' Performance and Perceptions. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, Pages 2-7. ACM, 2018.

[15] Lukas Ifflander, Alexander Dallmann, Philip-Daniel Beck, and Marianus Ifland. PABS - a Programming Assignment Feedback System. In *CEUR Workshop Proceedings*, Vol-1496, paper 5.

[16] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, Pages 740-751. ACM, 2017.

[17] Kelly Rivers. Designing a Data-Driven Tutor Authoring Tool for CS Educators. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*, ICER '15, Pages 277-278. ACM, 2015.

[18] Hannah Blau, Samantha Kolovson, W. Richards Adrion and Robert Moll. Automated Style Feedback for Advanced Beginner Java Programmers. In *Frontiers in Education Conference (FIE), 2016 IEEE*, Erie, PA, USA. 01 December 2016.

[19] Ihantola Petri. Automated assessment of programming assignments: visual feedback, assignment mobility, and assessment of students' testing skills. In *Aalto University publication series DOCTORAL DISSERTATIONS, 131/2011*. 2011

[20] Minjie Hu, Michael Winikoff, and Stephen Cranefield. Teaching novice programming using goals and plans in a visual notation. In *Proceedings of the Fourteenth Australasian Computing Education Conference*, ACE '12, Volume 123, Pages 43-52. ACM, 2012.

[21] Teemu Sirkia, and Juha Sorva. How Do Students Use Program Visualizations within an Interactive Ebook? In *Proceedings of the eleventh annual International Conference on International Computing Education Research*, ICER '15, Pages 179-188. ACM, 2015.

[22] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Mind your language: on novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, Onward! 2011, Pages 3-18. ACM, 2011.

[23] Stephen H. Edwards, Nischel Kandru, and Mukund B. M. Rajagopal. Investigating Static Analysis Errors in Student Java Programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, Pages 65-73. ACM, 2017.

[24] Stephen H. Edwards, Mukund B. M. Rajagopal, and Nischel Kandru. Pedagogical Agent as a Teaching Assistant for Programming Assignments: (Abstract Only). In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, Pages 1079-1079. ACM, 2018.

[25] Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. Effective Compiler Error Message Enhancement for Novice Programming Students. In *Computer Science Education 26(2-3)*, Pages 148-175. 2016.

[26] Devon Harker. Examining the Effects of Enhanced Compilers on Student Productivity. In *Master of Science Thesis in University of Northern British Columbia*. 2017.

[27] Krishnan Panamalai Murali. CodeWorkout: Design and Implementation of an Online Drill-and-Practice System for Introductory Programming. In *Master of Science Thesis in Virginia Polytechnic Institute and State University*. 2016.

[28] Julio C. Caiza, and Jose M. Del Alamo. Programming Assignments Automatic Grading: Review of Tools and Implementations. In *7th International Technology, Education and Development Conference*, INTED2013, Pages 5691-5700. 2013.

[29] Stephen H. Edwards. Web-CAT, http://web-cat.org/, last accessed 04-04-2018.

[30] Stephen H. Edwards. https://pdfs.semanticscholar.org/presentation/5d09/94da0dffe59c673948ac5f1cb3ed69699105.pdf, last accessed 04-04-2018.

[31] Wikimedia Foundation. Wikipedia, https://en.wikipedia.org/wiki/Compilation_error, last accessed 04-04-2018.
[32] UC3M, http://www.it.uc3m.es/abel/as/DSP/L2/CCErrors_en.html, last accessed 04-04-2018.
[33] Checkstyle, http://checkstyle.sourceforge.net/, last accessed 04-04-2018.
[34] PMD, http://pmd.sourceforge.net/pmd-4.3/, last accessed 04-04-2018.
[35] Jeff Langr, Andy Hunt, and Dave Thomas. Pragmatic Unit Testing in Java 8 with JUnit.
[36] Oracle Corporation. Oracle, https://docs.oracle.com/javase/7/docs/api/java/lang/StackOverflowError.html, last accessed 04-04-2018.
[37] Kent Beck, Erich Gamma, David Saff, Mike Clark (University of Calgary). JUnit, https://junit.org/junit4/javadoc/4.12/org/junit/runners/model/TestTimedOutException.html, last accessed 04-04-2018.
[38] Eclemma. Eclipse Foundation, http://www.jacoco.org/jacoco/, last accessed 05-02-2018.
[39] Oracle Corporation. Oracle, https://docs.oracle.com/javase/7/docs/api/java/lang/OutOfMemoryError.html, last accessed 04-04-2018.
[40] Webre, Elizabeth C. ERIC, https://eric.ed.gov/?id=ED298435, last accessed 04-04-2018.
[41] Brett A. Becker, Cormac Murray, Tianyi Tao, Changheng Song, Robert McCartney, and Kate Sanders. Fix the First, Ignore the Rest: Dealing with Multiple Compiler Error Messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, Pages 634-639. 2018.
[42] Dustin Marx. JavaWorld, https://www.javaworld.com/article/2072881/diagnosing-and-resolving-stackoverflowerror.html, last accessed 05-02-2018.
[43] Wikimedia Foundation. Wikipedia, https://en.wikipedia.org/wiki/Likert_scale, last accessed 05-02-2018.
[44] Josep Silva. A vocabulary of program slicing-based techniques. In *ACM Computing Surveys (CSUR)*, Volume 44 Issue 3, Article No. 12. ACM, 2012.
[45] Stephen H. Edwards. Work-in-progress: program grading and feedback generation with Web-CAT. In *Proceedings of the first ACM conference on Learning @ scale conference*, L@S '14, pages 215-216. ACM, 2014.

Appendix A

Text of Survey

Block 1

Survey on Web-CAT Feedback

This voluntary survey includes questions regarding your opinions on the automated feedback you received on your programming assignments. We will use this information to understand better how you view that feedback, and whether it is helpful to you.

The results from this survey will be used for research purposes and for improving the feedback you receive as you work on assignments. Please complete all items, even if you feel that some are redundant. This may require 5 minutes of your time. Usually it is best to respond with your first impression, without giving a question much thought. Your answers will remain confidential, and will not affect your grade in any way.

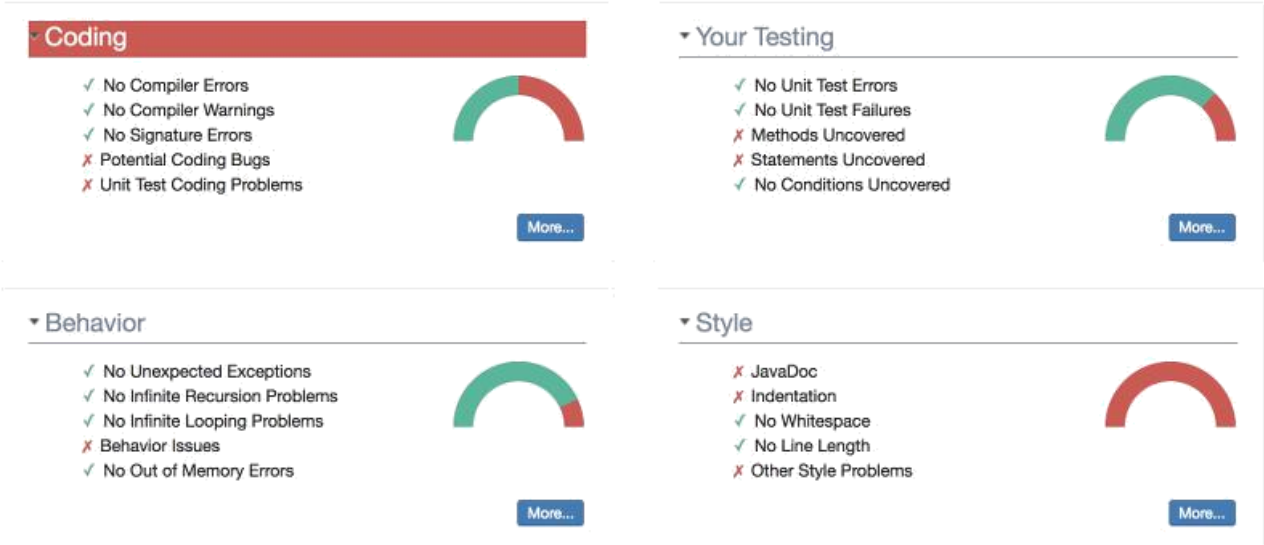
Your participation is voluntary. If you do not wish to participate, simply do not fill out the survey. You must be 18 or older to take part in this research.

Thank you for your participation.

Complete the following items. Select the option that best describes how strongly you agree or disagree.

Block 2

In some of your programming assignments, Web-CAT used a new feedback presentation to present the issues in your code in a different way:



Please indicate your agreement/disagreement with the following statements related perceived ease-of-use or usefulness of this feedback.

	Strongly agree	Agree	Somewhat agree	Neither agree nor disagree	Somewhat disagree	Disagree	Strongly disagree
This new style of feedback was easier to understand than before	○	○	○	○	○	○	○
It helped me decide on my goal for what to work on next	○	○	○	○	○	○	○
The code snippets in the feedback helped me find problems more easily	○	○	○	○	○	○	○
I was able to resolve issues with my assignment faster with this form of feedback	○	○	○	○	○	○	○
I found the old style of feedback easier to understand	○	○	○	○	○	○	○
The new style of feedback made me more discouraged with my work	○	○	○	○	○	○	○
The new feedback helped me improve my score faster	○	○	○	○	○	○	○
The new feedback is too long and complicated	○	○	○	○	○	○	○
It was easier to understand what I needed to work on with the new feedback	○	○	○	○	○	○	○
I prefer the old style of feedback	○	○	○	○	○	○	○

How can we further improve the feedback generated by Web-CAT?

Appendix B

Survey Report

Question	Strongly agree		Agree		Somewhat agree		Neither agree nor disagree		Somewhat disagree		Disagree		Strongly disagree		Total
This new style of feedback was easier to understand than before	18.33%	11	18.33%	11	26.67%	16	8.33%	5	15.00%	9	10.00%	6	3.33%	2	60
It helped me decide on my goal for what to work on next	11.67%	7	28.33%	17	26.67%	16	8.33%	5	15.00%	9	5.00%	3	5.00%	3	60
The code snippets in the feedback helped me find problems more easily	25.00%	15	26.67%	16	31.67%	19	6.67%	4	3.33%	2	5.00%	3	1.67%	1	60
I was able to resolve issues with my assignment faster with this form of feedback	10.00%	6	36.67%	22	15.00%	9	13.33%	8	13.33%	8	10.00%	6	1.67%	1	60
I found the old style of feedback easier to understand	6.67%	4	16.67%	10	21.67%	13	21.67%	13	13.33%	8	15.00%	9	5.00%	3	60
The new style of feedback made me more discouraged with my work	3.39%	2	3.39%	2	8.47%	5	30.51%	18	11.86%	7	38.98%	23	3.39%	2	59

The new feedback helped me improve my score faster	6.78%	4	16.95%	10	16.95%	10	35.59%	21	11.86%	7	6.78%	4	5.08%	3	59
The new feedback is too long and complicated	5.08%	3	5.08%	3	27.12%	16	23.73%	14	11.86%	7	22.03%	13	5.08%	3	59
It was easier to understand what I needed to work on with the new feedback	10.34%	6	20.69%	12	36.21%	21	17.24%	10	6.90%	4	5.17%	3	3.45%	2	58
I prefer the old style of feedback	8.47%	5	11.86%	7	16.95%	10	22.03%	13	11.86%	7	20.34%	12	8.47%	5	59

How can we further improve the feedback generated by Web-CAT?

- *It was great*
- *Teach how to use feedback in class*
- *Get rid of the cool down. It didn't change how I submitted at all*
- *none*
- *No submission energy*
- *Don't use it*
- *Increase relevant feedback, the "heatmap" system does not help diagnose syntax errors, etc.*
- *Provide feedback about a wider array of problems*
- *It was sometimes difficult to determine what exactly to fix. Still somewhat vague.*
- *N/A*
- *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.*
- *This was my first time using Web-CAT, so I don't have anything to compare it to. That being said, I don't know anybody who has ever said, "Oh wow, using Web-CAT is a real joy." For one assignment, Web-CAT had me losing 18 points because I forgot one this.super();. I recognize that it is difficult to grade so many students, but Web-CAT does not do a very good job.*
- *Increase the amount of errors that aren't hidden.*
- *To view more details from the bars in line rather than grid*
- *dont do the enregy*
- *N/A*
- *Show all issues at once, not just a couple at a time.*
- *Don't display everything that is correct. Only display things that are incorrect to clean up the interface*
- *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.*
- *Place more focus on the actual errors in the submitted code. The new categories displaying many potential sources of error (such as infinite recursions or out of memory errors) even though they passed testing was more distracting and confusing than helpful.*
- *When the code is wrong, be more specific in how it should function.*
- *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.*
- *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.*

Appendix C

Virginia Tech IRB Approval

Office of Research Compliance
Institutional Review Board
North End Center, Suite 4120
300 Turner Street NW
Blacksburg, Virginia 24061
540/231-3732 Fax 540/231-0959
email irb@vt.edu
website http://www.irb.vt.edu

MEMORANDUM

DATE: May 4, 2018

TO: Stephen H Edwards, Catherine Amelink, Bob Edmison, Michael Scott Irwin, Zhiyi Li, Nischel Kandru, Mukund Babu Manniam Rajagopal

FROM: Virginia Tech Institutional Review Board (FWA00000572, expires January 29, 2021)

PROTOCOL TITLE: Collaborative Research: Using Automated Feedback to Promote a Growth Mindset in Programming Assignments (NSF 1625425)

IRB NUMBER: **16-014**

Effective April 25, 2018, the Virginia Tech Institution Review Board (IRB) approved the Continuing Review request for the above-mentioned research protocol.

This approval provides permission to begin the human subject activities outlined in the IRB-approved protocol and supporting documents.

Plans to deviate from the approved protocol and/or supporting documents must be submitted to the IRB as an amendment request and approved by the IRB prior to the implementation of any changes, regardless of how minor, except where necessary to eliminate apparent immediate hazards to the subjects. Report within 5 business days to the IRB any injuries or other unanticipated or adverse events involving risks or harms to human research subjects or others.

All investigators (listed above) are required to comply with the researcher requirements outlined at:

http://www.irb.vt.edu/pages/responsibilities.htm

(Please review responsibilities before the commencement of your research.)

PROTOCOL INFORMATION:

Approved As:	**Expedited, under 45 CFR 46.110 category(ies) 5,7**
Protocol Approval Date:	**May 13, 2018**
Protocol Expiration Date:	**May 12, 2019**
Continuing Review Due Date*:	**April 28, 2019**

*Date a Continuing Review application is due to the IRB office if human subject activities covered under this protocol, including data analysis, are to continue beyond the Protocol Expiration Date.

FEDERALLY FUNDED RESEARCH REQUIREMENTS:

Per federal regulations, 45 CFR 46.103(f), the IRB is required to compare all federally funded grant proposals/work statements to the IRB protocol(s) which cover the human research activities included in the proposal / work statement before funds are released. Note that this requirement does not apply to Exempt and Interim IRB protocols, or grants for which VT is not the primary awardee.

The table on the following page indicates whether grant proposals are related to this IRB protocol, and which of the listed proposals, if any, have been compared to this IRB protocol, if required.

Date*	OSP Number	Sponsor	Grant Comparison Conducted?
06/01/2016	P534CP3I	National Science Foundation (Title: Collaborative Research: Using Automated Feedback to Promote a Growth Mindset in Programming Assignments)	Compared on 05/04/2016

* Date this proposal number was compared, assessed as not requiring comparison, or comparison information was revised.

If this IRB protocol is to cover any other grant proposals, please contact the IRB office (irbadmin@vt.edu) immediately.