Code Critiquer Usability Test Report

Andrea Lee
Team 6: Code Critters

Introduction

The Code Critiquer is a program designed to detect antipatterns, patterns of behavior which lead to errors, within programs provided by users. Users submit their own code using either the text entry field or the file upload function. Currently, only the Java programming language is supported by the application.

It is intended to be used by programming students as well as independent learners who are trying to program by themselves. Checking their code using this application will allow these two groups to learn better practices by ending bad habits they may develop with their programming skills. Thus, the application will not only allow users to find and fix their errors, but end patterns of behavior that would lead to future errors.

The central feature of the code critiquer's user interface is a large text entry field, allowing users to enter their own code as if they were using a text editor or an integrated development environment. Each line is numbered, which is used by the critiquer to notify users where an antipattern has been identified.

When the user has finished writing their code, they can click the "check code" button above the entry field. After the button is pressed any antipatterns identified by the application will be presented within the code itself in red boxes. These red boxes contain the line number the antipattern was detected as well as the type of antipattern identified.

If the user has already written the code they want to check for antipatterns, they can use the "upload file" button next to the "check code" button. This function would allow users to submit a file from their own computer to be uploaded and critiqued. At the time of usability testing, this function was not available.

Usability testing for the Code Critiquer was focused on two scenarios, one for each method of input. The first scenario involved allowing participants to write their own code into the text entry function. To maximize the potential detection of different antipatterns, the only guidelines that participants were given was to make the program at least 10 lines long. Once the program was complete, the participants would begin to check their code and edit it to the best of their ability based on the critiques provided.

The second scenario was originally meant to test the file upload function. However, as the file upload function was not available at the time of testing, scenario to was instead used to test how different participants would correct the errors found in the same file. A questionnaire was given before and after the test to determine users initial experience and interest before testing as well as their thoughts after testing.

Test Plans
Test Scenario 1: Coding Within Application

Test Goals

1. Test text input functionality of app interface
2. Test antipattern detection function
3. Feedback from critiquer should allow for errors in the code to be corrected

Required Software/Equipment

- Zoom video conference software (tester and test giver)
- desktop/laptop computer (tester and test giver)
- Keyboard and mouse
- Microphone
- webcam
- Internet access
- Code critiquer app
- Timer (only for test giver)

Description

The usability tester will be prompted to write their own program within the user interface. As testers are sampled from computer science classes, they will be expected to be capable of programming with java. Testers will be encouraged to make the program at least 10 lines long, excluding any blank lines. Allowing testers to independently generate code to be tested will allow for a wider variety of antipatterns to be detected within the same scenario. However, testers may still ask the attending consultant for possible code to test.

Once the program is complete the tester will be prompted to check the program for antipatterns. If any antipatterns are detected the tester must attempt to correct these errors and check the code again. This process will repeat until no errors are present.

Scenario Text

"This program uses two methods of input, file uploads and the text box in the center of the screen. We will be testing the text entry first. Imagine that you are just writing a program for yourself as practice. While the program can be simple, it should be at least 10 lines long. If you need help thinking of a program I can give some suggestions.Use the 'check code' function to find any errors in your code."

Measurement List

- Time to complete tasks – Slow response times from testers may indicate problems with app navigation.

- Error detection – the code critiquer should automatically detect any errors within the code
- System understandability – users must understand the functions of each button within the app interface
- Error correction – users are capable of correcting their code when errors are detected

Potential Observations

- There is a chance that the code could be completed without errors on their first attempt, this may require prompting the tester to knowingly add an error to their code.
- When an antipattern is detected within the code, the tester's next actions will be informed by their interpretation of the error message. A misinterpretation will lead to a lack of error correction.
- An error may also be undetected by the critiquer itself, this will require a bug report.

Bug report form

See Bug report form

Post Test Questionnaire

See "post test questions" on test questionnaire form

Post Test Interview

1. Were there any elements in the interface you did not understand?
2. Did the format of the critique help you understand where and how an error occurred?
3. Were any of the antipattern messages you encountered unclear?
4. Were there any messages that you understood but did not know how to correct?

Test Setup

The tester will begin a zoom call with the consultant and any attending programmers. They will be given access to the Code Critiquer application as well as permission to share their screen. The tester's microphone will need to be active to maintain communication with the consultant and programmers.

Test Scenario 2: File Upload

Test Goals

1. Test file upload function of code critiquer
2. Test sign in function of code critiquer
3. Correct errors of an existing program through the code critiquer interface
4. Test the “view previous critiques” function of the code critiquer

Required Software/Equipment

- Zoom video conference software (tester and test giver)
- desktop/laptop computer (tester and test giver)
- Keyboard and mouse
- Microphone
- webcam
- Internet access
- Code critiquer app
- Timer (only for test giver)
- Sample program file

Description

The usability tester will be required to download a sample program file provided by the usability consultant. Once the file has been acquired, the tester will need to upload it into the application and check it for antipatterns. As this code was created by the researchers, it will have the same errors each time. Like in the previous scenario, the tester will need to correct and recheck the program until no errors are present.

Once no errors are present, the tester will be instructed to view the past critiques saved. The tester must use this function to view the first critique of the provided program file.

Scenario Text

“In addition to typing directly into the app, the code critiquer allows for files to be uploaded for critique as well. By logging in, you will also gain access to previous versions of your code. For this scenario, imagine that the file I gave to you is a homework assignment that you want to check for errors before turning in. Log into the Code Critiquer and correct any errors it finds. Once the code is free of errors, go through the past critiques to find the original version of the file to compare it to the final version.”

Measurement List

- Time to complete tasks – Slow response times from testers may indicate problems with app navigation.

- Error detection – the code critiquer should automatically detect any errors within the code
- System understandability – users must understand the functions of each button within the app interface
- Error correction – users are capable of correcting their code when errors are detected

Potential Observations

- Time spent searching for functions will need to be tracked by the consultant. As this scenario is more structured than scenario one, response times can be compared between participants.
- Misinterpretation of error messages will lead to their errors remaining uncorrected.
- Errors may also be uncorrected due to a lack of user knowledge.
- Repeated searches through past critiques would indicate that the naming scheme used by the program is unclear

Bug report form

See bug report form

Post Test Questionnaire

See “post test questions” on test questionnaire form

Post Test Interview

1. Were any functions difficult to locate?
2. Were the names of the previous versions of code easy to understand?
3. Did the error messages help you understand the flaws in the program?
4. What part of this scenario did you consider to be the most difficult?

Test Setup

Using the same zoom call from scenario one, the consultant will provide the tester with a .java file containing a sample program to test. As this scenario requires viewing past critiques, login info would need to be provided to the tester as well.

Results

Pre Test

Before testing began, each participant was asked how many years of experience they had with programming, whether they have done any programming outside of class, and their initial interest in the project. These questions help illustrate the variance of experience of each participant as well as how interest in the project may change before and after use.

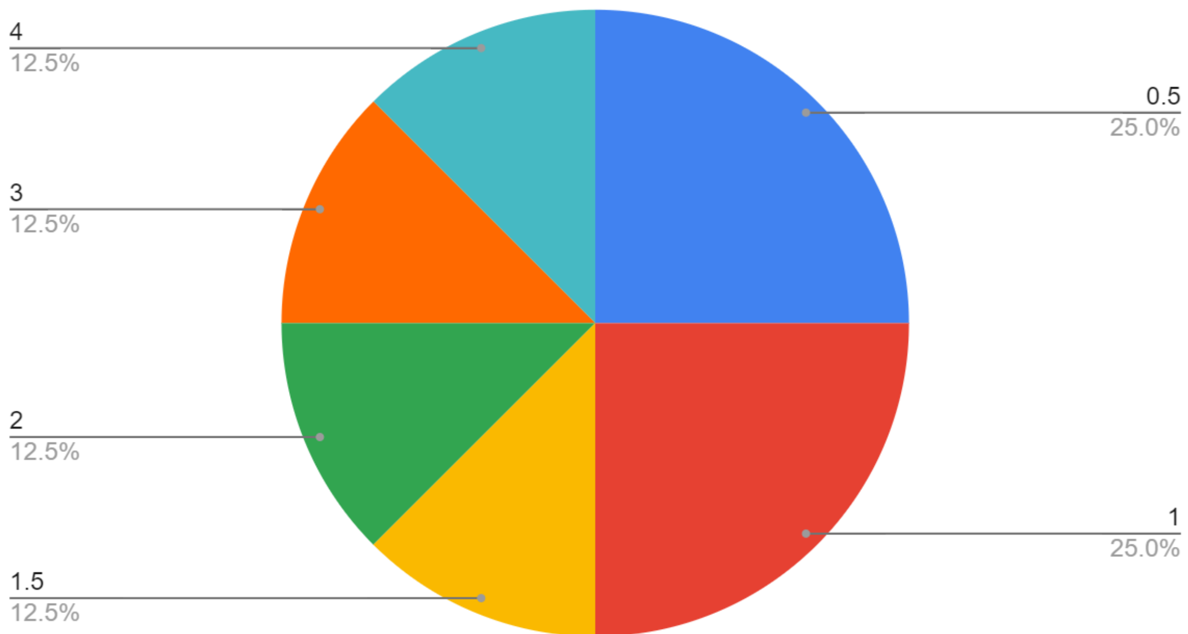


Figure 1: The years of experience in java possessed by each participant

The participants possessed an average of 1.7 years of experience with java. Half of the participants had one year of experience or less with the language. More experienced programmers who have written programs in java for three to four years participated in usability testing as well. While these experienced programmers may not be the primary intended users of the Code Critiquer, comparing their behavior with less experienced programmers can still be useful.

Figure 2: Percentages of participants programming outside of class

An equal number of participants programmed out of class or only programmed for school assignments. This allowed for equal time devoted to both groups. As only students participated in usability testing, the students who program outside of class are a close approximation to the independent learners the application is also intended for. In fact, the participant who programmed out of class could be considered both students and independent learners.

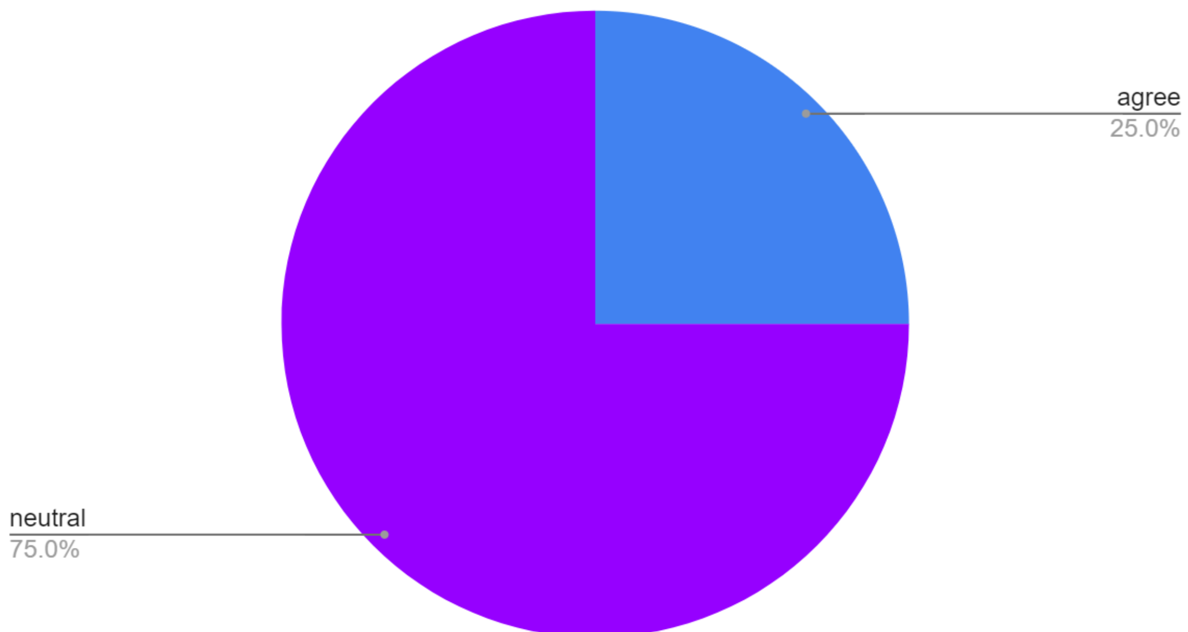


Figure 3:Percentages of participants interested in the project before beginning testing

The final question asked before beginning usability testing was the level of interest each participant had in the project. The majority of participants stated that while they were interested in the project, they were also at least partially motivated by course credit as well.

Scenario One

As expected, each participant created a different program for the first scenario. None of the participants had any trouble understanding the interface, with only one struggling to find the “check code” button. However, each participant shares some similar issues with the interface.

Every code checked by the code critiquer gave a message reading “Comment Header Block”. While the message likely describes the need for a header for the program with a description of its purpose, none of the participants were able to determine the exact format required by the code critiquer. The application does not currently provide any information on how these antipatterns may be corrected, only whether or not they were found within the code.

Every participant was provided with around 15 minutes to write and edit code. As no participants were able to satisfy the “comment header block message”, none of the participants were able to make their code completely free of antipatterns.

Scenario Two

At the time of usability testing, the file upload function was not available. However, the file upload menu was available to test for demonstration purposes. All participants quickly found the file upload button on the interface with little effort. However, in addition to the lack of function with the “submit file”. The file name field used white text on a white background.

To continue with the usability tests. The text of the text file was simply copied into the text entry field. The formatting of the text did not affect the code critiquer and any differences to text color or indentation were corrected to the preferred formatting of the Code Critiquer as the “Check Code” function was used. This demonstrates that copying text into the text field is a viable method of input in addition to the other methods tested.

While no clear errors avoided detection by the interface, there were certain messages which were confusing to participants. When a “for” loop used the variable “i” as an iterator variable, the message “replacing i” would appear when the “check code” function was used. No other variable names would trigger a similar message.

Similar issues to scenario one, including the “Comment Header Block” were encountered in this scenario as well. As no solution could be found. No participants were able to correct all errors discovered in the test file.

Post Test

At the conclusion of the testing session, participants were asked to report if they found the application easy to use and whether they would want to use the program in the future. This allowed for comparison between interest in the assignment before and after using the application.

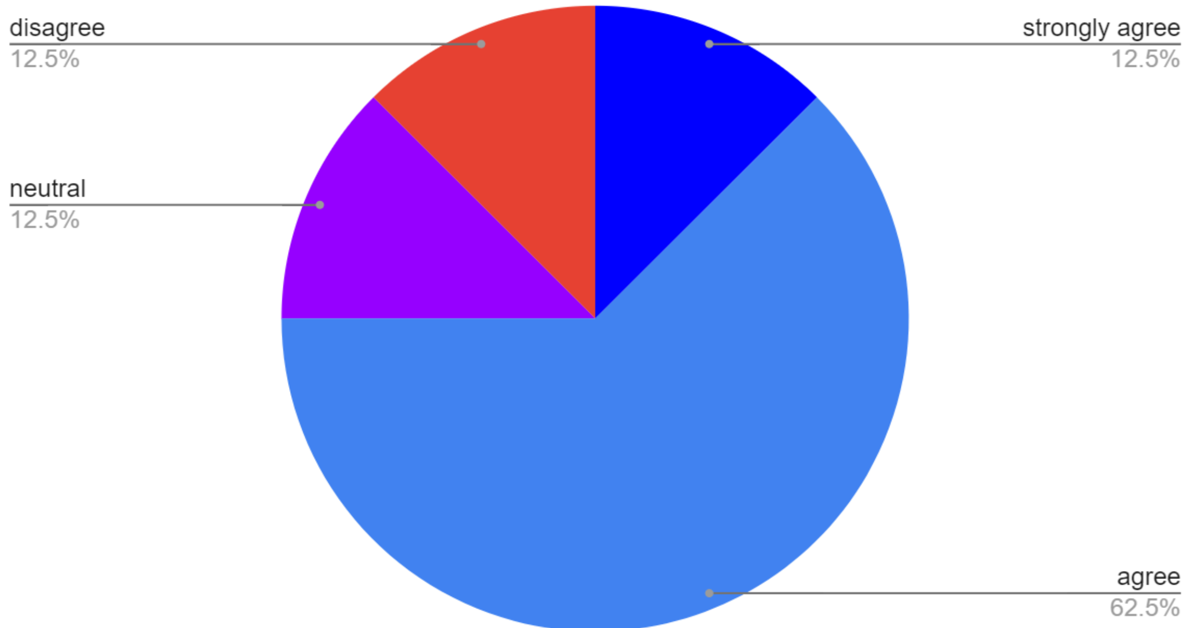
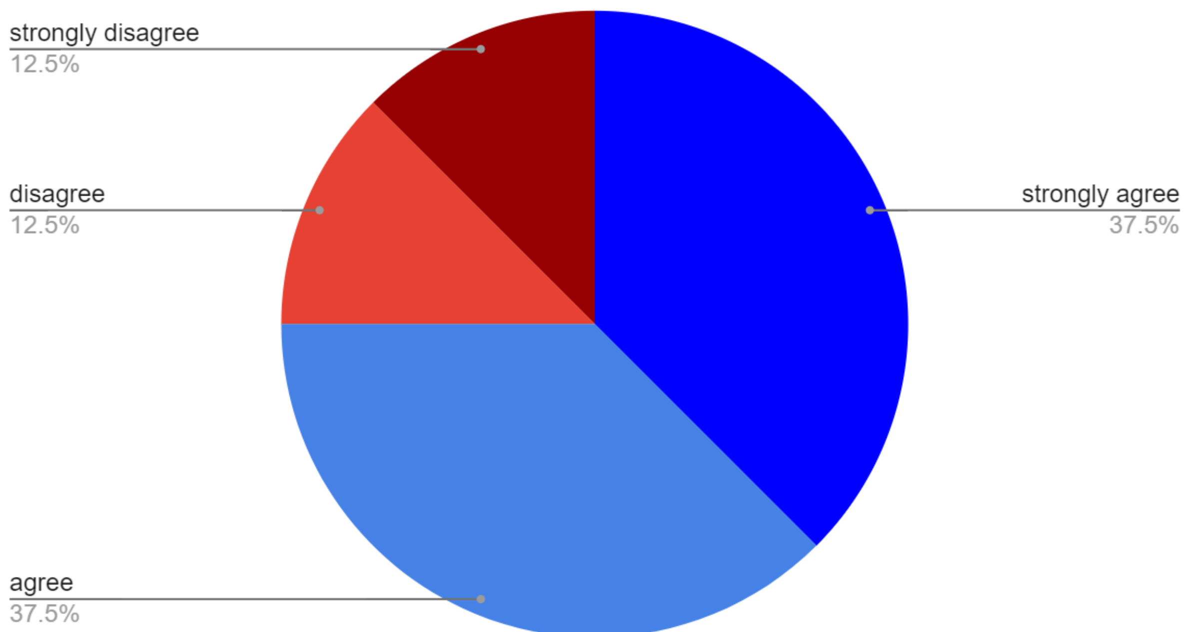


Figure 4: Percentages of participants who believed the Code Critiquer was easy to use.

Despite the bugs within the code, the majority of participants reported that the application was easy to use. However, all participants specified that they were mostly referring to the interface of the application and not the application as a whole. As such, the issues with usability are likely caused by current bugs in the application rather than the design itself.



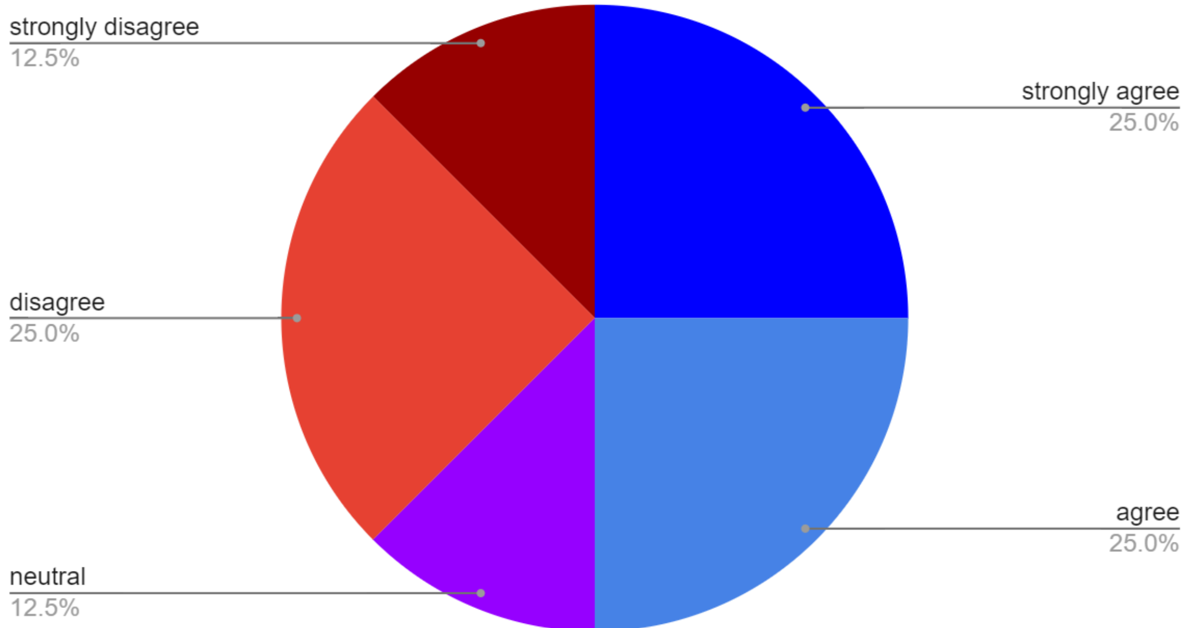


Figure 5: Percentages of participants interested in the Code Critiquer after usability testing for future assignments (top) and outside of school (bottom).

Using the application itself had a clear impact on the participants' interest in the application. While the majority of participants agreed that they would use the application for school, there were participants who became less interested in using the application in the future. The largest complaint from all participants whether they were more or less interested was fixing the bugs. Participants who answered disagree for these two questions also mentioned that fixing the current bugs would make them reconsider.

Interestingly, one of the students who answered "no" to programming outside of class still answered "strongly agree" to whether they would use this application outside of class. This may suggest that having this application available could encourage students to begin programming independently.

Conclusions

As only students participated in usability testing, no data was gathered from the second group of primary users, independent learners. This group would need to be included in future testing. However, while no exclusively independent learners could be tested, students who also wrote programs outside of school could still be tested. These students' feedback could be similar to these independent learners.

In addition to fixing the bugs discovered during testing, new features could be added to the critiquer to resolve the issues encountered by users. More information on the detected antipatterns should be provided to improve the code critiques utility as a learning tool.

For example, some of the messages given by the code critiquer are not errors. Instead, they are marking code that is indicative of bad habits. By providing a description of why these habits could lead to errors in the future, users will better understand why to avoid making the same mistake in the future. In the critiquer's current state the users will only know that there is a problem with the code but not why there is a problem with the code. If they do not understand the problem with their code, users may simply repeat the same behavior on other programs regardless of feedback from the application.

Providing examples of better practices could clarify issues with code as well. This is especially important with messages like "Comment Header Block", which appear to require specific formatting not explained by the application. Knowledge of specific criteria would assist usability testing as well, as potential false positives in error detection would be more easily identified.

The Code Critiquer is designed to be an educational tool in addition to assisting programmers. However, to fulfill this function, it will need to include more than notifications of errors and antipatterns. If these recommendations are followed, users will better understand what mistakes they are making and how they can improve their code.

Appendix A: Team Member Attendance

4/7/2024 13:00
Members Present:
Andrea Lee
Connor Ward
Johnathan Oestringer

4/8/2024 16:30
Members Present:
Andrea Lee
Nick Zimansky
Chris Torrey

4/9/2024 16:00
Members Present:
Andrea Lee
Connor Ward

4/10/2024 16:30
Members Present
Andrea Lee
Pan Prathongkham
Noah Yacoub

4/11/2024 16:30
Members Present
Andrea Lee
Chris Torrey
Johnathan Oestringer

4/12/2024 16:30
Members Present
Andrea Lee
Pan Prathongkham
Johnathan Oestringer

4/13/2024 16:30
Members Present
Andrea Lee
Johnathan Oestringer

Devon Gosnick

4/14/2024 16:30
Members Present
Andrea Lee
Devon Gosnik
Pan Prathongkham

Appendix B: Bug Report

Number	Name	Description
1 Instances: 7	False syntax error	syntax error when none is present
2 Instances: 6	Incorrect function	Submit code checks code instead
3 instances:4	No scroll	Scrolling occasionally stops appearing
4 Instances: 19	Redundant lines	Lines created when running code
5 Instances: 2	Close window deletion	Closing pop-up window clears program window
6 Instances: 13	Error misalignment	Syntax error on wrong line
7 Instances: 3	Doubled line numbering	Two numbers on one line
8 Instances: 2	line number misalignment	Indented line number
9 Instances 7	Repeated error	Repeated error message
10 Instances 1	Unknown text	Mysterious string of letters at end of program
11 Instances: 16	Error text merging	Error messages become lines of code. Require deletion

Appendix C: Challenges

Number	Name	Description
Instances: 2	Comment header block is always an issue	Comment header block is always an issue
Instances:2	i	Interator variable labeled antipattern if named i
Instances: 5		File upload text is white on white
Instances: 7	Vague messages	Antipattern messages are too vague
Instances: 1		Participant could not think of test program