Multiplayer Yahtzee

Test and Validation Plans

Zags Help Zags

By: Jesse Adams, Jackie Ramsey, Katie Imhof, John Stirrat

Course: CPSC 224 - Software Development
Instructor: Aaron S. Crandall

I. Introduction - Jesse

I.1. Project Overview

We will be performing software testing on all parts of our Yahtzee program such as the scorecard, the dice, and the main game loop/class. Testing is crucial for each of these main parts of our program since the player(s) ability to roll a hand of dice, their ability to mark and loop view their scorecard as well as their ability to play the game all rely on these main parts of the program. We will be performing unit testing to ensure proper behavior of key aspects of each part listed above, as well as integration and system testing to ensure that all parts of the program work together as planned and as expected.

I.2. Scope

The main purpose of this testing plan document is to lay out an outline for how we will implement testing into our project and what exactly will be tested and how. This document will help us stay organized and give us a detailed plan to follow to ensure that our program is thoroughly tested and works/behaves as expected. The scope of this document covers all code that we will write and test in this project as well as the testing methods that well will make use of in this project.

II. Testing Strategy

1. Identify the requirements and goals for the software's testing.
2. Identify the necessary tests to be implemented for each module.
3. Decide the configurations for all components of the operations.
4. Identify and document the expected results for the decided configurations.
5. Perform the unit tests with all the given configurations, results, and data.
6. Document the findings throughout this testing process.
7. After unit testing is completed for each unit, integration and system testing will occur.
8. If bugs are found in testing, an issue shall be created to report and resolve the problem. This issue will contain a description and location of the bug.
9. All documentation and reports for testing shall be submitted after being reviewed by the group.

III. Test Plans

1. First, unit testing will occur independently, and each individual will complete the tests for the units they are responsible for. Each person will also push their code to github for

everyone to be able to view in order keep track of issues, milestones, and for future steps. The unit tests will be conducted by testing instances of edge cases, input/output cases and others in order to verify the behavior of the unit of code.

2. Once all unit tests are finished, units that are within the same module will be grouped and integrated testing will occur. The members responsible for each module will be those whose unit tests are within the module of interest. The integrated tests will be conducted by grouping by actions and events that are important to our software.
3. Once integration testing is completed, we will meet up to discuss the system testing requirements. The details of further testing and planning will be decided then on how to complete the system testing and its components (functional, performance, user acceptance). The system testing will be conducted by verifying for acceptable user usability and checking for problems such as no new bugs that may have been caused during this development process.

III.1. Unit Testing

Since this part of testing will involve the smallest and most detailed components of our code, the unit test writing will be broken up by modules. Each person will be responsible for the testing of the module they have written because they will have the best understanding of how their code works. Progress of unit testing can be reviewed by checking the issues and milestones that are open or closed. This will indicate whether a unit test has been completed or is still in progress. This should allow for easy tracking of what remaining work is left in each module. Other team members will be able to view the output of the test harness when an individual has pushed their work to the repository for everyone to see.

III.2. Integration Testing

Yes we will be using a structured approach. As new code gets merged to the main branch, testing will be conducted to ensure that the new code integrates well with the original code. If there are tests that don’t pass, then the teammate that is merging the code and running the tests can fix it before it becomes a big problem.

III.3. System Testing

If there are system tests that do not pass, we will find the root cause of the error as a team, and then determine the appropriate next step. If it’s a specific teammate's code that they integrated, and if it’s simple enough, that teammate can figure it out, or we can collaborate for issue resolution.

III.3.1. Functional testing:

Due to unit testing, integration testing, and system testing prior to functional tests, there should be a high level of confidence that the functional tests will pass. This is from testing at bite size pieces and all together. If the functional tests were to not pass, then it may be easier to figure out the root cause because of the previous tests outcomes.

III.3.2. Performance testing:

We will test our program's non-functional requirements to make sure the game is behaving like we intended, and also to make sure that everything runs smoothly and does not take too long. We want to avoid creating a program that takes an unreasonable amount of time to play a game of Yahtzee, so once the major functional requirements are met we will see how these are performing. This will give us a sense of if we need to tweak our implementations of the major features in any way to improve our program. In terms of stress testing, we will stress test by going through the program and try to purposefully break the program. We will play the game correctly enough to make it through a game, but will try to find any places that cause unintended behaviors. If any bugs are found this way, we will address them, and will continue this process as needed.

III.3.3. User Acceptance Testing:

After the program passes all of our other tests and seems to be working as intended, we will all find a small sample size of other people to play the game. At first, we will let them play the game, and then ask them for feedback. For example, some of the things we will ask will be things like: Did the game behave as you expected? Did you run into any problems? Was there anything missing? This will also help us get an unbiased sense of whether or not our program actually meets the specifications of the project. We also will do some form of stress testing, where we will ask the users to try and do things that would potentially cause problems, like clicking buttons out of the usual order for example, and this will help us find potential bugs that we did not find in our own testing. After we get this feedback, if there are any issues we face, we will go back into our code and fix them, and then repeat this process as needed.

IV. Glossary

Root cause: A factor that caused a nonconformance and should be permanently eliminated through process improvement.

Root cause analysis: A collective term that describes a wide range of approaches, tools, and techniques used to uncover causes of problems.

Structured approach: A set of standards for systems analysis and application design.

Integration testing: The phase in software testing in which individual software modules are combined and tested as a group.

System testing: Testing conducted on a complete integrated system to evaluate the system's compliance with its specified requirements.

Functional testing: A type of software testing that validates the software system against the functional requirements/specifications.

Design requirements: Those specifications and design criteria contained in the Contract that specify the minimum acceptable technical standards and define the limits within which the design of the Project shall be developed and conducted.

Test harness: a collection of software and test data used by developers to unit test software models during development.

Unit testing: the phase in which a small unit of testable code is isolated from the rest of the code and tested for bugs and unexpected behavior.