Multiplayer Yahtzee

Test and Validation Plans

The A Team

Joshua Venable

Maalik Brown

Tyler Ciapala-Hazlerig

Course: CPSC 224 - Software Development

Instructor: Aaron S. Crandall

I. Introduction

I.1. Project Overview

The software that we are attempting to create is a multiplayer version of Yahtzee. It should be playable using solely a graphical interface and should allow the user to input multiple commands. These commands should provide a seamless implementation of the game, and correctly follows the specification of the Yahtzee Game. Major functionality that must be tested include the dice, the scoring of each player's score card, how the game progresses through each round, and finding possible scores. Key requirements are a working graphical interface, a backend that can be implemented easily and is scalable.

I.2. Scope

The purpose of this document is to design and field tests that will 'prove' the usability and 'correctness' of our code. It should describe and give examples of tests and testing strategies that will allow us to move from a pseudocode and written description, into a concrete code boilerplate. We shall also specify the types of testing that we will be conducting, as well as how we will be validating success. Our success will be measured by the passing of tests, and the modularity of the program, so that each individual should be able to work independently of the others, with exceptions of core functionalities.

II. Testing Strategy

Tests will be written as we make classes and progress through writing Multiplayer Yahtzee. If we find a bug in already written code that is not related to our current task than an issue will be created, marked as "bug", and assigned to whomever was originally assigned to that task. If it is a more general architectural issue, then an issue will be created and marked as for discussion and will be brought up at our next meeting. Whomever is eventually assigned to fix the bug will create a feature branch, fix the bug, submit a pull request, and attach the issue to the request. Once the issue is resolved a unit or integration test should be made, if possible, to prove the issue is resolved. The reviewer of the request should checkout the commit and verify the issue is resolved.

III. Test Plans

Every non-view class will have comprehensive testing verifying the functionality of nearly every method and function of the class. This will be done iteratively through the development process, so once a non-view class is created, a test will be made testing that class. Since the most specific classes will need to be written first, unit testing will be the first tests written. Once the base classes are written and tested, the integrated classes and integration tests will be made.

Once progress is made, the developers and a subset of the Gonzaga student body will test the program and list issues they come across.

III.1. Unit Testing

The creator of the code will write the test for it. To view progress with the unit testing, they will at least test for the expected output when the program runs and a low and high outlier to make sure it still works in a range of options. The other team members will look at the output of the test harness when the creator of the test is done review the test.

III.2. Integration Testing

First, we would do unit tests on each team members' component that will be tested together to make sure that they work individually. Then, one team member will merge their branch with the other/s, create a branch from the merged branch, and each team member will be able to run tests on the two or more components.

III.3. System Testing

Functional Testing: We will be doing functional testing on our program. To test this, we will most likely run the program and use the debugger to make sure that certain things are happening when they need to.

User Acceptance Testing: We most likely will be doing user acceptance testing for our program. To test this, we will ask our room/suitemates to test the game for us, receive their feedback on it, and adjust the game as needed.

Glossary

Class: A user defined blueprint for which objects are created from.

Unit test: Procedural code that verifies if a class's basic functionality.

Integration test: Procedural code that verifies a class's integrated functionality with other classes.

Branch: A pointer to a commit in a git repo.

Issue: A "to-do" item for a project.

Pull Request: A request to merge one branch in a git repo to another branch in a git repo.

Appendix-A

General Bug-Reporting Workflow:

1. Identify issue/bug in program.

2. Create GitHub issue and assign necessity tags.
    1. Bug
    2. Discussion

3. If tagged as bug, the issue will be assigned to whomever originally wrote that code.

4. If tagged as discussion, the topic will be brought up at next meeting and an assignee will be determined.

5. A bug-fix git branch will be created.

6. Once issue is assigned, the developer will fix the issue and create unit and integration tests sufficient to prove the issue is resolved.

7. The developer will submit a pull-request to merge with the working dev branch and a reviewer will verify the issue has been solved.

8. If the issue is solved, the feature branch will be merged with the dev branch and the issue is closed.