Team 6
Test Plan

Revision N

Revision History

DATE	REV	AUTHOR	DESCRIPTION
11/17/2017	A	Maigan	Section 2.0 - 2.2.4
11/17/2017	B	Seiichi	Section 1.0 - 1.3
11/17/2017	C	Maigan	Section 2.0 - 2.2.4
11/18/2017	D	Seiichi	Section 3.2.3.2
11/18/2017	E	Jeremy	Section 3.2.4 - 3.2.4.3
11/18/2017	F	Maigan	Section 3.2.2
11/19/2017	G	Seiichi	Section 3.2.3, 3.0 - 3.1
11/20/2017	H	Jeremy	Section 3.1
11/20/2017	I	Maigan	Section 3.0, 3.2-3.21, 3.23, 3.2.2.1
11/20/2017	J	Jeremy	Section 3.2.2
11/20/2017	K	Seiichi	Section 3.2.2, format
11/21/2017	L	Seiichi	Section 3.2.2
11/21/2017	M	Jeremy	Section 3.2.1
11/21/2017	N	Maigan	Section 3.2.2

1.0 Introduction

Our team was given a checkers game application with seeded bugs, overly complex methods, little documentation and no existing test data.

We hope to debug, refactor and optimize the system all while creating and executing a successful test plan.

1.1 Goals and objectives

- Define activities and strategies required for Unit, Integration, System and Acceptance testing.
- Define specific testing procedures and requirements for all types of tests:
    - Test Cases

1.2 Statement of scope

In Scope:

1.2.1 Class Functionality

The first step is to ensure all of the methods inside of classes in the program run without bugs and defects.

1.2.2 Inter-Class Functionality

The classes within the application must make the correct calls to other classes and objects.

1.2.3 Game Functionality

The game should be playable from beginning to end with gameplay options.

1.2.4 GUI Functionality

All graphics should populate, button clicks should have consequences, moves should be shown, sounds should play at correct times.

Out of Scope:

Computer Opponent – Verification of difficulty level.

1.3 Major Constraints

- Lack of previous testing documentation: requires testing existing system from the ground up
- Lack of understanding of how the intelligent system was designed
- Lack of requirements specification

· Limited experience with system

2.0 Test Plan

As the introduction states, “We hope to debug, refactor and optimize the system all while creating and executing a successful test plan.” In order to be successful in providing a checker application that is bug free and is user friendly, it is necessary to spend an entire week on testing alone. The next few sections will describe our steps in testing and how we will be carrying out our plan for testing our program within the next week.

2.1 Software to be tested

2.1.1 The User interfaces

Start screen

- Make sure that when the “Start game” button is pressed, it goes to the game screen. Also make sure that the start screen looks user friendly.

Game Screen

- Test the New game button, insure that a new game actually is applied when the button is pressed. “A new game” implies that the current checker images will be removed from the board image. Also it will apply the selections that the user has picked (player mode, colour selection).
- Test undo button, that it undoes the last play when the button is pressed
- Test the Help button, when the button is pressed, a “how to” file will appear on the screen
- Test the difficulty selection menu, when a different difficulty is pressed, the playing of the game should be altered slightly depending on the difficulty. (Assuming that this only applies to 1-player mode, where the user is playing the computer.
- Test the Mode selector buttons, if 1 player is selected then the computer will play, this means the methods where the computer makes moves with the checker images must be tested extensively. If 2-player is selected, it is necessary to only be able to select one color checkers not the other color, depending on which player’s turn it is. Also only one mode can be selected.
- Test Color selector buttons, make sure only one color can be selected per a game. If a color is selected and New game is selected, then the color preference should be applied. When the color is applied, the player can only move that specified

color. If it is 2-player mode, the color selected will be player-1's color. Also no matter the color, yellow must always move first.

2.1.2 Classes

- CheckerFrame.java
- CheckerMove.java
- Checkers.java
- GameEngine.java
- GameWin.java
- Help.java
- IntelliCheckers.java
- PlaySound.java
- StartPanel.java

2.2 Testing Strategy

This section is dedicated to presenting our plan for testing our program. The different testing methods that we will use to test our checker's program are the following: unit testing, integration testing, system testing, and acceptance testing.

2.2.1 Unit Testing

It is expected that unit testing is carried out by the developer of the program. Although we are not the original developers of this checker program, we have developed a deep understanding of the code through the several assignments leading up to this assignment involving the code. Therefore we will be performing the unit testing.

For the unit test cases, we will look at the methods in the program individually. It is necessary to insure that each of the individual methods do their own duties bug free before looking at the entire system as a whole. With that being said we will test each method individually by looking at the data that is used in that method and the outputs that the method will return. Gathering that data will allow us to record what the expected results should be, then we can compare these expected results to the actual results and alter the method based on how the results are delivered. This process is known as white box testing because are testing the internal structures of the system and not the functionality at this step.

2.2.2 Integration Testing

In the integration testing phase, we will take the individual methods that we harshly tested individually in the unit testing phase and will combine their functions. In order for our checkers application to run properly, all of the methods need to work together, because they use each other to reach the extent of the game known as checkers. For integration testing we can gather the methods that use each other and run test cases on those groups of methods. Once the group or groups of methods are considered bug free then we can move on to the next phase.

2.2.3 System Testing

System testing is the phase of testing where the integrated software that is considered complete is tested together. There are many different types of system testing, the following are the types that we will use in our testing of the checkers program:

Performance Testing: For performance testing, we will make sure that the system knows exactly how to respond to unexpected actions. The following are types of performance testing:

Quality Testing: For this, we will test for the reliability, maintainability and availability of the system. This means that we will make sure that throughout the entire game of checkers the rules are always kept in check. The quality of the moves and images and sounds will always be maintained and never go out for an unknown reason.

Human Factors testing: For this we will test with end users. For the sake of this assignment, we will be the end users. But the reason for this testing is to insure that all of the buttons have a clear purpose and it makes sense to have them on the screen. Along with that, the "how to" file will need to be explicitly clear, so there is no question that the end user understands the rules of the game.

Overall we are looking to ensure that the system as a whole provides the expected functionality. We will heavily use the GUI in System Testing.

2.2.4 Acceptance Testing

Acceptance testing is our final step in testing and is used to demonstrate that our checkers program is fully functional and ready for friendly checker matches.

To accomplish this, we need to validate that all of our functions work as they are supposed to. If for any reason we find errors by this phase, we will implement a list of the errors and inform the users of these errors. Since this is the last phase, our goal should be to have very little, if any (preferably none) errors by this phase.

3.0 Test Procedure

The testing procedures for Unit, Integration, System and Acceptance Testing are detailed here.

Testing will begin with the tester performing unit testing. This will allow the tester to insure that the internal structures of each of the individual methods is functioning correctly, taking in the correct inputs and returning the correct outputs. Once the unit test cases are labeled successful the tester will move on to Integration testing.

In integration testing the tester will gather the appropriate information from all of the individual methods and put them in groups based whether the methods use each other or not. If the methods use one another then tests will be conducted to confirm that they function properly together and output the correct data.
From integration testing, the tester will perform system testing to confirm that the system as a whole does as expected successfully, this means that not only the methods will be tested together but also the user interfaces.

Lastly the acceptance testing will be conducted by playing the Checkers game multiple times, multiple different ways to insure that the game is successful, functional and ready for release.

3.1 Software to be tested

CheckerFrame.java
CheckerMove.java
Checkers.java
GameEngine.java
GameWin.java
Help.java
IntelliCheckers.java
PlaySound.java
StartPanel.java
The User interfaces -- See Section 2.1

3.2 Testing Procedure

3.2.1 Unit Testing

Unit testing will performed by using white box testing and looking at the internal structures of each of the methods that are used in the Checkers program. Some of the methods may not be tested at rigorously because they do not serve many functions and some may be tested to a long extent because of the severity of the functions that they serve. Each method will be given inputs that should provide a specified and expected output and the actual output will be compared to this expected output.

3.2.1.1 Procedure for Unit Testing

The tester will analyze the needed input and the returned output of each method and will give the individual method an input that should result in an expected output. As long as the expected output is returned from each of the individual methods, the unit testing should be successful.

3.2.1.2 Unit Test cases

See Software Test Specification Doc: Unit Testing

3.2.1.3 Expected results

The given test cases should return the results stated in the Software Test Specification Document. The expected results should prove that each method functions properly and as expected.

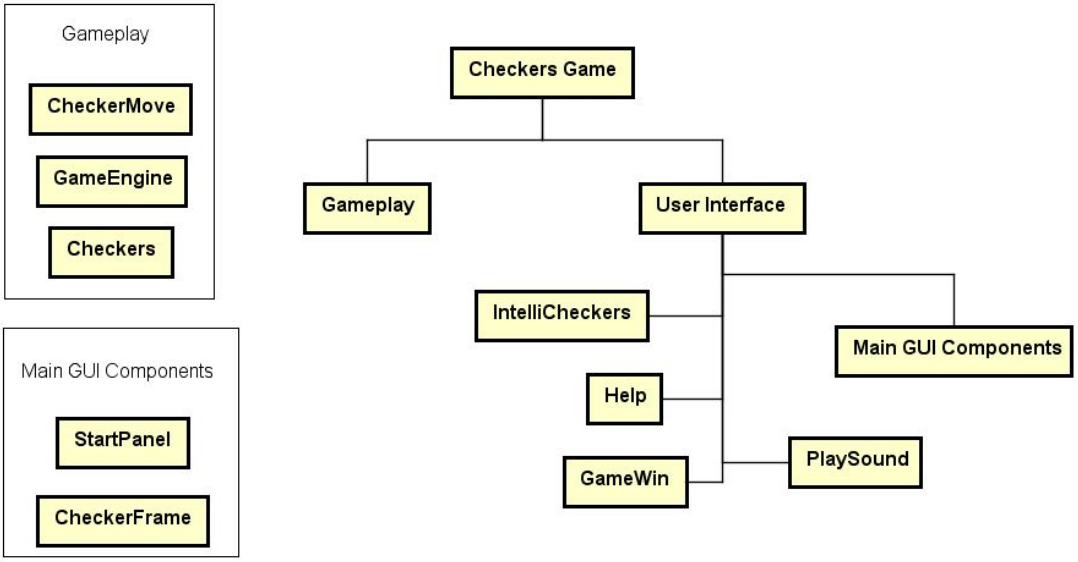
3.2.2 Integration Testing

Integration testing phase will insure that the checkers application runs properly, by checking that all of the methods need to work together, because they use each other to reach the extent of the game known as checkers.

3.2.2.1 Testing Procedure for integration

The tester will take the individual methods that we previously tested individually in the unit testing phase and will combine their functions. The entire system will be split into two main subsystems. The first consisting of all the classes involved with movement and actual gameplay algorithms (Checkers, CheckerMove, GameEngine). The second subsystem consists of all of the classes involved in the GUI. The second subsystem further splits into a Main Components Subsystem and additionally, all of the classes involved in the UI. The Main Components subsystem consists of the classes that create the two main windows

(IntelliCheckers, StartPanel and CheckerFrame). The rest of the classes in User Interface are: Help, GameWin, PlaySound.



3.2.2.2 Test cases and their purpose

See Software Test Specification Doc: Integration Testing

3.2.2.3 Expected results

The given test cases should prove that the interaction between subsystems functions as expected.

3.2.3 System Testing

System Testing will be conducted by examining the GUI functions as well as the methods functionalities as a whole. There will be functional and nonfunctional testing as stated in the system testing description (p. 5-6)

3.2.3.1 Testing Procedure

The System Testing will involve testing the entire functioning system via GUI making sure that full GUI functionality is present as well as gameplay/algorithmic functionality. A Test Case Specification table is provided.

3.2.3.2 System Test Cases

See Software Test Specification: System Testing

3.2.3.2 System Testing -- Expected Results

The Expected Results portray live events that the user can visually verify via GUI. System test cases resemble a walkthrough of the game window and functionality of the buttons and labels present in the application.

3.2.4 Acceptance Testing

Acceptance testing will be conducted simply by playing through multiple games of checkers under multiple configurations. Additionally, a friend or family member may be recruited to play the game in both 1 and 2 player configurations to gain further end-user feedback as to how acceptable the final product is as a for its intended entertainment purposes.

3.2.4.1 Procedure for Acceptance Testing

The tester will play an equal number of games on each difficulty level and with each color and record whether they win or lose. The tester will also play the same number of 2-player games with a second player.

3.2.4.2 Test cases and their purpose

See Software Test Specification Doc: Acceptance Testing

3.2.4.3 Expected results

Test cases on the easy and fairly easy difficulties should result in the player winning. The moderate difficulty setting should be (approximately) and even split of player wins vs computer wins. The bit difficult and tough settings should likely result in the player losing. If the player cannot seem to win after completing testing for the two easiest difficulty settings, the game engine for determining the computer player's move is inappropriate for general entertainment purposes and must be adjusted. The 2-player option should simply play through to completion with no errors. A passing acceptance test for the 2-player option is completely subjective to the opinions and skill level of the players.