RoboChess

TEST PLAN
VERSION 1.1
NOVEMBER 6, 2016

Prepared by: Team B5
LIN, YUCHEN | NELSON, JORDAN | PARK, RYAN | WANG, XINGENG | VAN HEERDE, WILLIE

Table of Contents

1. Introduction

For our testing document, we will be going into detail for the classes that are integral to the system such as the Player and Server classes. In our last document we specified the pre and post conditions for all the methods in each class so we can easily implement a diverse set of tests for each class. For testing interactions between vital classes we will develop separate classes that monitor the communication and state changes between the two classes, this will ensure that we know which class the bug originates from. Finally, we will utilize assertions while we are in the testing phase to make sure that no catastrophic errors occur. After we have finished and finalized the product and have a release build we will remove these assertions because the code should be robust enough that it is not needed. Finally, after group work sessions we will combine, link, and compile the current state of the system for smoke tests to record the state of the system (even if it is not complete). By doing frequent smoke tests we will be able to remove any bugs quickly for the next work session as well as know the origin of any new bugs that crop up.

2. Unit Testing

The following section outlines the tests for the major components of the RoboChess system.

2.1 Interpreter

The first unit test we have selected is the Interpreter class, which provides the A.I. functionality for the RoboChess system. Its main methods involve parsing FORTH scripts into FORTH words and translating FORTH words into its equivalent system actions (as defined below by the Player – Server Interface). It is critical that these methods are thoroughly tested, otherwise we risk losing the A.I. component of the RoboChess system.

For testing the FORTH interpreter, both the white box and black box testing methodologies will be utilized. Several mock FORTH scripts will be created for testing purposes. Each of these scripts will have a corresponding list of FORTH words that the interpreter should be able to replicate. These scripts will be parsed by the Interpreter and the resulting list of FORTH words will be compared to a static list of predetermined values. We will also need to evaluate the validity of the eval method, which handles the translation of FORTH words. Extensive parameter checking will ensure that only proper FORTH words are translated and any other case will be handled as an exception.

2.2 Server

The second unit test we have selected is the Server class. The server is the main class for the game – the model of the board and where most internal computations take place. The server acts as a model for the board, and thus it is important to ensure that it is working properly, otherwise the game potentially could be broken by a bug or simply unstable in general.

To test the server, we will use white-box unit testing, ensuring that every method is working properly. In the main function, we will write a testing scaffold to test the validity of each method, trying to break each by sending invalid parameters as defined by the pre conditions stated in the design document, as well as checking and testing boundary conditions.

2.3 Other Classes

For all remaining classes that we have not mentioned, we will be using our design document as a reference to write extensive unit tests for each class and place them in their respective static main method. In this way each test is only run when requested, saving computational resources during runtime. This allows us to easily run an extensive test when requested after new features are implemented or code is refactored for a class.

3. Integration Testing

The following section outlines the tests for interactions between the major components of the RoboChess system.

3.1 Player – Server Interface

The first interface we have selected to test is the interaction between the Player class and the Server class. The server is the main class for the game – the model of the board and where most internal computation takes place. The player class is the class where player commands are received and processed. This means that the server - client class interaction is absolutely vital to the performance and stability of the game, as it is the most critical interface interaction facilitating the main game simulation. It is therefore important that this interface is working flawlessly, as any errors or bugs within this interface will have a high potential to break the program by either disobeying a key rule of the game or executing a command different than that ordered by the player.

For testing this interaction, we will implement white box testing, by writing an additional class that creates a Server and several Player objects. The class will then send information from the player to the server, verifying that the server receives the correct information. The class will then do the opposite, verifying that the server's reply to the player is correct. The class will govern what this communication is through mocking, which will include move, fire, and end turn, focusing on when these commands are not valid, such as firing after you have already fired, moving to an out-of range-tile, etc. This verifies both human and robot script player interactions with the game, ensuring stability for both.

3.2 Player Class – User Interface

The last interaction we have decided to test is the interface between the player class and the user. This includes both the GamePanel class and the ControlPanel class. The GamePanel class is the panel that displays the board, including hex tiles, robots, the fog of war, and various additional animations. This is by far the most complex class meant for GUI purposes, as it involves complex layering, many objects, various updates, and has to facilitate various user inputs needed to interact with individual hex tiles. Thus, it is important to test such functionality as it would be very easy for bugs to arise in these interactions. The ControlPanel class is the class that displays and takes input from the buttons along the bottom of the main GUI screen. These buttons include the move, fire, end turn, confirm, and undo buttons. This is part of the main way the user as an actor communicates with our system, and so needs to be tested for correct functionality as well as stability. The player - user interface is the way a human player interacts with the system, and so it is very important for the overall quality of the game, not just to be working properly, but also for it to be working and designed well.

When set up or updated, the GamePanel takes a reference to the current board state, and an ActionListener in order to interpret user input when a hex is clicked. These arguments will be tested with assertions. GUIs are very hard to test, especially automatically, and so we will test the GUI output by manually checking the output on the screen at runtime. We can do this by creating a mock board within our tests and verifying everything is in the correct spots. Input will be tested manually, by clicking in each hex, while making sure to check boundary conditions such as selecting outside of the board or on the boundary of a hex tile. When a hex is clicked, the hex's coordinates will be displayed to the log, and so we can also verify that each hex clicked is the correct hex intended.

We will write similar tests for the ControlPanel class, manually verifying both input and output. Similarly, a message will be displayed verifying the button clicked was the correct button intended to be pressed. These tests will also be backed by a mock, which will be in charge of displaying the messages to verify the correctness.

4. Testing Environment

This section presents the resources required for successful implementation of the RoboChess test plan.

4.1 Hardware

We will ensure our system works on the Linux lab machines, although the game is simple computationally speaking, and so should work on any remotely modern computer.

4.2 Software

The software environment components we are using are Eclipse as our IDE and GIT as our version control system. We are supporting the Linux operating system that is installed on the lab machines, but by the nature of the Java programming language our system should work on Linux, MacOS, and Windows.

5. Changes

In the GamePanel class, we have added the parameter ActionListener to the construction method +GamePanel(<Tile> board) and the method void Update(<Tile> board). These will then become GamePanel(<Tile> board, ActionListener listener) and Update(<Tile> board, ActionListener listener). We've added these arguments to these methods because we had neglected to add them in our design document, and they are necessary in order to have player input be possible within the GamePanel.

In the Player class, we also needed to remove the second attribute "interpreter: interpreter", as we already had an attribute called "interpreter: <interpreter>", so it is redundant to have two attributes to store the same information when one is enough in our system. This was simply a mistake that went unnoticed until now.

6. Summary

In a video game, bugs are not just annoying, they can potentially ruin a player's immersion or experience with the game. This is especially true when the bug breaks a game rule or ruins a player's turn to a game breaking degree. It is therefore imperative that our system works as flawlessly as possible, and thus extensive testing is needed.

We chose to test the Server class extensively because it is a class involved with every player and almost every other class. This means a simple bug in the Server class can quickly get out of control as it propagates throughout the system. The interpreter is also an important class to test as it single handedly provides complete A.I. player functionality to our system. Without the A.I., the product is heavily lacking in features, and can only be used with several humans at a time.

Testing within our system is done in a variety of ways, including everything from basic unit tests of trivial low-level objects to complex black and white box testing of high-level abstract communication between major subsystems within our game. In some places this is automated as much as possible, eliminating the need for human intervention unless a bug arises, but elsewhere it by necessity is done completely manually, where a human must verify each step.

This not only adds another layer of complexity to an already non-trivial software system, but adds a lot of extra work and time to a project already needing as much time as it can get. This

simply further illustrates how necessary testing is, not only to ensure as few bugs as possible slip through our tests, but to ensure as much stability and fairness for the users of the program, to give the most comfortable and enjoyable user experience.