Arimaa Final Report

Tayler How, Luke Miller, and Jesse Shellabarger

# State of Features

We implemented nearly every feature that we mentioned in our problem statement. The basic features of the game are in place. You are able to move pieces, push and pull them, and pieces can be removed from the board on death. The only feature that we ended up dropping was turn timer banks, which would store excess time at the end of a players turn. This excess time would be able to be used if a player ran out of time. We decided that this feature added very little to the game and to the goals of the project.

Game states can be loaded and saved at any point of the game. This is accomplished by writing the current state of the game to a text file that the user specifies. The user is then able to load any game that has been previously saved.

The game checks for all win conditions at the end of every move. These conditions are having a rabbit in the opponent’s home row, the loss of all the opponent’s rabbits, and the inability of the opponent to move.

Multiple players are able to play the game using “hot-seat” multiplayer on the same machine. It switches control of the pieces every four moves.

The game uses the user specified turn timer setting and then counts down in real time. If the turn timer expires, the user whose turn it is loses.

The board state is saved at the end of every user’s move, and the user is able to undo his moves. This reverts the board state to exactly how it was at the beginning of that players turn. This is useful in the case of a piece accidentally killed or an unwise move made.

The GUI of the game shows both player’s names, allows the user to start games or load games. It also displays whose turn it is and how much time is left in their turn. Finally, it shows how many move are left in the current turn.

# Testing Strategies

Since the focus of this project was the test-driven development process, the primary testing style we used was scripted automated unit testing. All of the game logic functions were easily unit tested, however the GUI could not be unit tested effectively. The unit tests for the save and load functions required a dependency injection to account for reading or writing to a text file. An example of our basic unit tests is testBasicPullLeft. For this test, we create an instance of the game with custom board that has pieces placed to execute a pull. Next, we pull the appropriate pieces and verify that the pull was executed successfully. We assert that the pull command returned true and the pieces are in the correct spots. We repeated this process for all possible directions, as well as for other caveats (i.e. can’t pull another of your own pieces, can only pull pieces strictly weaker than you).

Another testing style we used was big-bang integration testing. There were only had three parts to integrate: the game model, the GUI, and .txt files for saving and loading games. We chose the big-bang approach because the complexity of our integrations was relatively low. It was very easy to debug any problems that occurred during integration. To perform integration testing, we simply ran our game with the GUI and hoped everything worked.

Since the ultimate goal of the project focuses on the user’s experience playing the game, we did a lot of exploratory manual black-box acceptance testing. Testing the game from a user’s perspective helped us find and fix any bugs that we may have overlooked during unit testing and early development. To perform acceptance testing, we played through whole games countless times. We made sure to test all of our game model and GUI functionality, focusing on edge and corner cases. Towards the end of the project, we also tested our game through independent verification and validation. We had multiple third parties play the game and report any bugs or issues with their gameplay experience.

We also did independent verification and validation of our project by sending it to multiple third parties to play the game and report any bugs or issues with the gameplay experience.

# Testing Thoroughness

Our testing was extremely thorough. We used CodeCover and the Eclipse Metrics plugins to measure our testing comprehensiveness. CodeCover reported 98.1% statement coverage over the entire project, excluding the Graphical User Interface, and 96.5% term coverage. All of our metrics were within acceptable boundaries. Our McCabe Cyclomatic complexity was above the plugin’s recommended setting, but we were able to decrease that from 61 to 19, so we felt it was ok to be a little above the recommended value.