Test Plan

for

Chess Chaos

Version 1.0 approved

Prepared by Drew Grubb

Texas State University

Committed March 30

CONTENTS **2**

1. Introduction

The following pages will serve as a Test Plan overview for Chess Chaos.

1.1 Test Plan Objectives

The purpose of this test plan is to formulate an efficient way to make sure that the requirements laid out in the SRS have been fulfilled by Chess Chaos. Should errors or bugs occur during test runs, the SRS, SDD, and source code will be inspected and modified. By the end of this test plan execution, Chess Chaos should be ready for the maintenance and end of life phase.

The verification aspect of Chess Chaos is successful. The program runs as intended, navigating through a series of menus and game states without crashing in all the ways that have been tested. Every game state is rendered successfully with the correct BufferStrategy, and timers work based on real time.

The validation aspect of Chess Chaos works as intended. The user can play a game of chess against another player, play against an Artificial Intelligence, or watch 2 AI face off against one another.
Game states are rendered as intended, with all the associated buttons and timers required.

Due to the nature of this project being a stand-alone program, there are many testing strategies that will be empty, but will be written in anyways for the learning experience.

2. Test Strategy

2.1 System Test

Chess Chaos is not a part of a system, due to being a stand-alone Java project. Therefore, a system test is purely program testing. See 4. Functions to be Tested.

2.2 Stress/Performance Test

Chess Chaos does not change object positions fast enough for lag caused by algorithms on slower machines to be visibly noticeable. However, it is possible that slower machines will have some brief input delay in between specific algorithms.

There are several algorithms that need to be tested on slower machines:

- updatePossibleMoves (Board)
- Switchturn (PlayState)
- CalculateMove (Various AI)

2.3 Security Test

Chess Chaos does not involve any networking, authentication, or authorization. Therefore, security tests are not required.

2.4 Automated Test

Due to the simplicity of Chess Chaos, automated tests are not a part of this test plan. Any testing will be done by Unit Testing or beta testing.

2.5 Recovery Test

Chess Chaos handles errors using the built in java exception handling. There are very few try-catch blocks throughout the source code of chess chaos, but the few that exist work to stop the program from crashing. Recovery testing will involve forcing these exceptions to be thrown and working with the program accordingly.
However, if an error is not expected and it is thrown, the JVM may crash entirely, and the user will have to restart the program.

2.6 Beta Test

The extent of beta testing within the Chess Chaos project will be testing by the developer. I will click every menu button to create the different desired results, and react accordingly within the source code if the result is not correct. I will also play multiple AI vs AI games and Player vs. AI games to test all piece functionalities and game mechanics.
Junit coverage testing and the GUI allows most bugs to be caught manually, allowing me to fix bugs as I test the program.

3. Environment Requirements

3.1 Environment 1

Chess Chaos was programmed and tested on a single machine and experienced no performance issues:
Intel Quad Core i5-6600K 3.50GHz
Windows 10 x64
16GB RAM

4. Functions to be Tested

4.1 Unit Testing

Unit tests should cover various positive/negative tests.
The following functions need to be tested

4.1.1 Board updatePossibleMoves()

There will be unit tests created for each piece's updatePossibleMoves method for a total of 8. The order of the unit tests goes as follows:
Initialize Board
Add a Piece on a specific spot on the board
Create a negative case and a correct case
Assert =/= negative
Assert == positive

4.1.2 isInCheckmate()

There will be one unit test created for the isInCheckmate method. The order of the unit tests goes as follows:
Initialize Board
Add pieces on board in Checkmate position
UpdatePossibleMoves()
Assert isInCheckmate == correct position

4.1.3 isInStalemate()

There will be one unit test created for the isInCheckmate method. The order of the unit tests goes as follows:
Initialize Board
Add pieces on board in Stalemate position
UpdatePossibleMoves()
Assert isInStalemate == correct position