Chess Royale Final Report

By Group 5: Dan Hrubec, Julian Gonzales, Joseph Canning, Victor Fong

Spring 2021

Table of Contents

List of Figures

I Project Description

1 Project Overview

Chess Royale is a combination of the battle royale genre of games, and the classical game of chess. In battle royale style of games, players compete with each other to be the last one standing, as a map or board shrinks in forcing the players to interact with each other. In a similar vein, we combine this with the classical game of chess, where the players start out on a 24x24 chess board that shrinks in after each player makes 2 rotations of their moves. There are special tiles on the board denoted by green and red tiles that either upgrade the player's chess piece, or eliminate the player respectively. Once the board closes in a player is finally announced as the winner, allowing them to either play again or quit the game.

2 Project Domain

The project was created via the Unity game engine. Most of the game's features are written programmatically with the use of C# scripts that create the game, load in the players, allow for the game movement, interactions with the players and the respective tiles, amongst many other features. The testing of the game will be done through mostly physical testing by playing the game trying to stretch the limits to ensure that the code encompasses as many different scenarios as possible.

3 Relationship to Other Documents

The initial project ideas and conceptions were detailed in the final report from Group 15 of Fall 2020[1]. The development for this project was documented via Scenario I, Scenario II and Scenario II documents. These documents outline a summary of completion so far, and a summary of where we wanted to be by the end of the sprint. We would then try different methodologies of development for each sprint to see what worked best in each scenario. Everything was kept and logged via Clickup.

4 Naming Conventions and Definitions

4a Definitions of Key Terms

Most of the terminology from the game comes from different chess terms from the standard game of chess. Included are these terms as well as others that were defined in the scope of this project.

Board: Denotes the 24x24 game board that the players play the game on. Follows the standard chess board with alternating white and black tiles.

King: A game piece that the player starts the game as. The movement for this piece is identical to the classical game of chess only allowing the player to move one space in any direction including diagonals.

Knight: The next piece in the upgrade tier list. Again, this is a playable piece that the players can be referenced from chess. The player's movement as a knight is limited to

moving two in one direction, and then on in a different direction. For example, the knight can move two spaces up, then must move either one space to the left or one space to the right.

Bishop: The next piece in the upgrade tier list. A bishop is a playable game piece that the players or the AI can play as. The movement for the bishop is restricted to only moving along the diagonals, but having no restriction as to how far they can move along the diagonal.

Rook: The next piece in the upgrade tier list. Similar to the other pieces the players can play as. The movement defined for the rook is moving along the cardinal directions, meaning the rook can only move up or down, and left and right. Like the bishop, there is no restriction as to how far they can move.

Queen: The last and best piece in the game of chess. The movement for the queen is a combination of both the rook and the bishop. Being able to move along the diagonals as well as the cardinal directions. Likewise, there is no restriction to how far they can move along.

Red Tiles: Recolored tiles from the standard chess board. Landing on this tile will eliminate the player from the game. The red tiles are randomly generated onto the board when the game initially loads in.

Green Tiles: Recolored tiles from a standard chess board. Landing on this tile will allow the player to upgrade to the next best chess piece. These tiles are randomly generated around the board.

4b UML and Other Notation Used in This Document

The only notation used is UML. It follows the standard guidelines described in “UML Distilled” by Fowler.

4c Data Dictionary for Any Included Models

Sprites are loaded in via .png files from Sprite sheets. Sprite sheets are basically a .png file with sectioned out pictures that you can select certain areas for you to act as sprites. Some of them in the scope of this project include all the different chess pieces as well as the black and white correspondents.

The movement for each of the chess pieces all come from an abstract piece class. Where each class controlling the movement for that piece would inherit from this abstract class as each of the chess pieces have a different property for their movement.

II Project Deliverables

The development of the Chess Royale was split into three major releases, all lasting around 4 weeks of development, with the total ranging from 12 weeks of complete development. In each of the major releases, we tried various different development methodologies being a blend of scrum and kanban, XP, and TDD.

5 First Release

During the first major release, there was a lot to set up to make sure that we can scale our project up properly. In the first release, our initial focus was to make an underlying grid system that would assist us in many different game mechanics as well as make a board in which we can develop the UI on top of. Afterwards, we began to start loading in the randomly generated green and red tiles. We started off just being able to load them randomly on the board on game start, and worried about the functionality of the tiles in later releases. Afterwards, we began to start loading in different pieces onto the board. To start off we loaded everyone as a king piece. So we loaded in the player that we will control, as well as 3 CPUs and recolored their pieces to distinguish them properly. We then worked on just moving our player for the time, and ensuring that the movement was correct for the king, e.g. the king can move in any direction, only moving one tile at a time. Amongst that, we also create a basic main menu, where the game would initially load and launch the game.

For this release we used a blend of Kanban and Scrum. So we would plan out our tasks for each sprint so we know which one to work on, very similar to a standard Scrum system. We would then allow anyone to take tasks and start working on them. Then to implement the Kanban portion, after someone was done with a specific item, they would push it into an in review section. Where at least one other team member would have to take a look at the code and the functionality of the item, before approving it and moving it into the complete section.

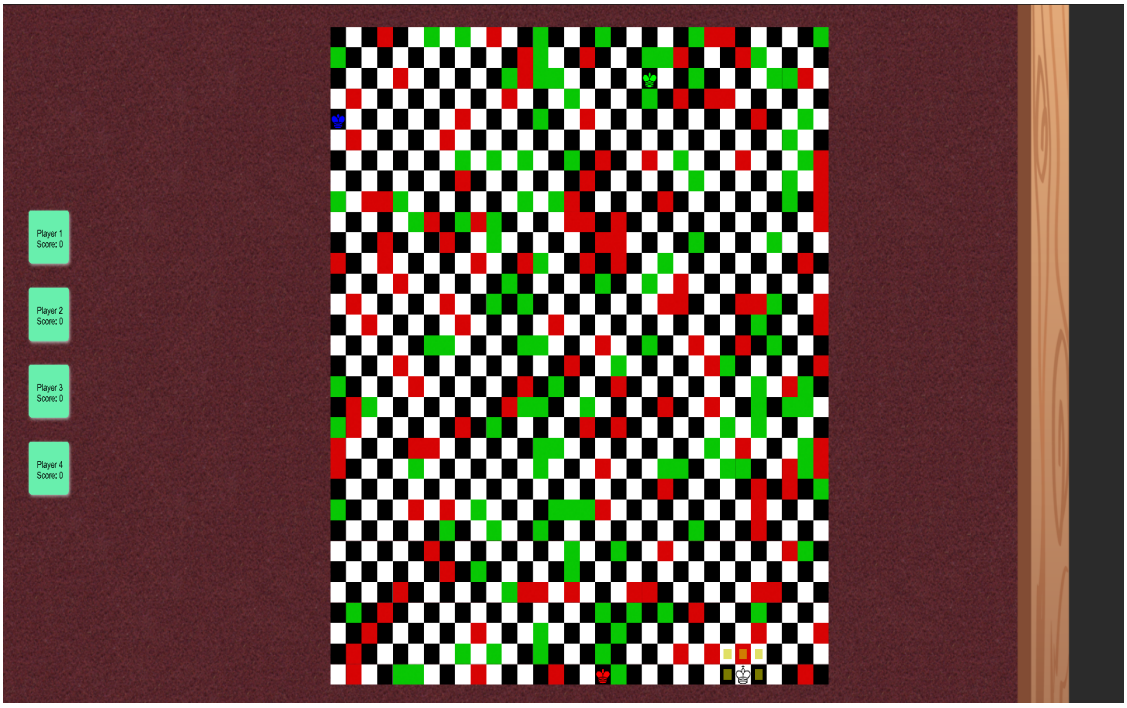
Figure 1 - Release 1 Screenshot

6 Second Release

During the second release of the game, we had a strong backbone to start building a lot of the game mechanics upon. The first mechanic we focused on was getting the rest of the players to start making their turn. After this was developed and completed, we started to move toward shrinking the board after two rotations. As per the original project design, and to implement the battle royale element of the game, the board needed to shrink by a certain amount every 2 rotations of players movement. Once this was created, we can then focus on another major point of this release being the difficulty selection menu. This would be after hitting play in the main menu, it would open up a second menu where you can select different parameters of the game that you can control, offering a different game experience. The three parameters that the player can control for a difficulty selection are the board size, the amount of green/red tiles as well as how much of the board shrinks. After having the difficulty implemented, we would then focus on some other game mechanics. We also implemented a way of showing all the valid moves when it was the player's turn. These were indicated by translucent yellow squares inside of the tiles that they can move to, making it very clear your movement options, as well as allowing new players who are not familiar with chess know how to play the game better. We then also added the option of capturing other players, removing them from the game, and having you transform into their piece.

For this release our primary development methodology was XP. We had a similar blend where we would just pool all of our tasks, and rank them in priority, trying to do the more important ones first to make sure that we can then work on any task. After trying a bunch of different practices in XP, the one we enjoyed using the most was Pair Programming. This was for a few different reasons, firstly, it was really easy to set up. We used discord as a means of communication, which had a simple share screen feature, so it was easy to follow along and assist with other members as they developed. If they were stuck at all, it was easy for the spectator to start looking up resources or ways to resolve the issue. It also made the overall workload a lot easier, as the work was less stressful as there was a good sense of communication and teamwork between the spectator and the coder.

Figure 2 - Release 2 Screenshot



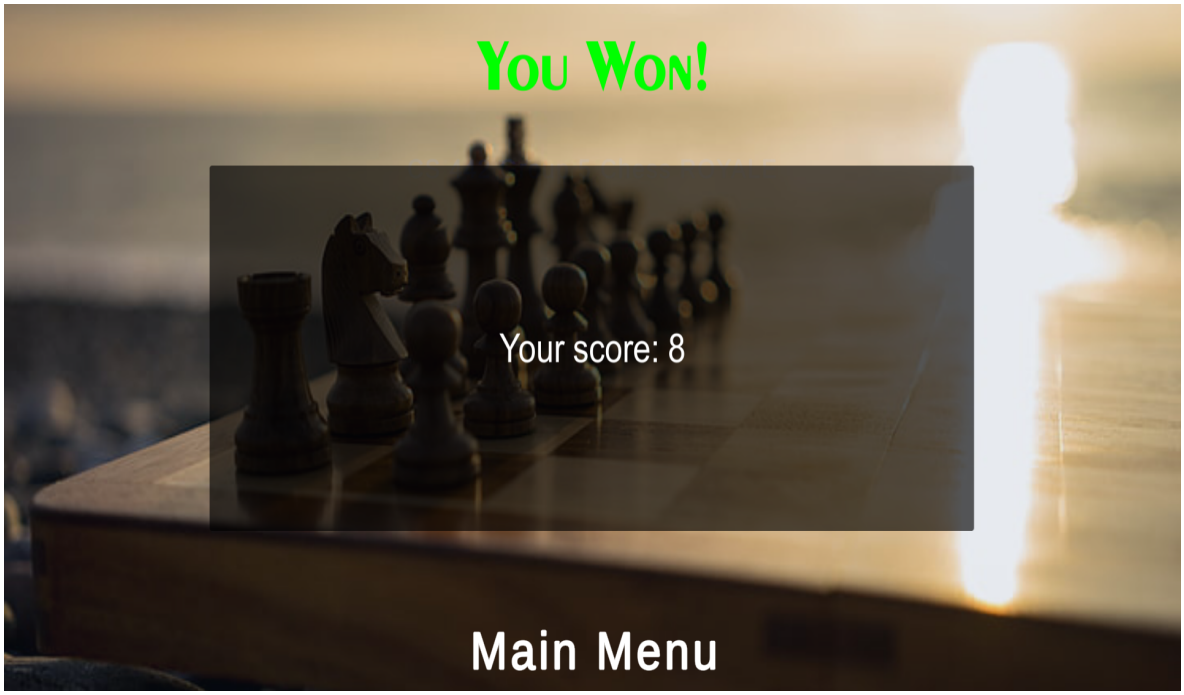
6b Third Release

During the third release of the project, a lot of game mechanics were implemented, but a lot of different bugs arose during the development of the last release. So the primary focus of this release was to patch up any and all underlying bugs, as well as create an end game scene, where the player can return to the main menu after reporting the end status. On top of this, we adjusted the options menu in the main menu to allow you to select either being a black or white chess piece, depending on whichever you selected in the options, with the default being the white piece. We also implemented a score system to be added in to also be reported in the end game scene. It is calculated by a few aspects. If you land on a green tile and upgrade your piece, your score is increased

by 2. If you land on a red space and downgrade your tile, your score decreases by 2. If you capture another player, your score will increase by 2. The score is then reported to the end game scene where you can see your overall performance for that game. After finishing all of the major game mechanics, we began patching a lot of different bugs that occurred during the development thus far. Some of the bugs that were patched included, the bishop movement being incorrect, the AI moving to the position that they are currently at, essentially skipping their turn, and the AI sometimes being able to move twice. We patched up these bugs, bringing us to the current point in which the Chess Royale game is currently at.

During the development of this sprint, we attempted to use TDD, but it was difficult to execute for the scope of our project. Since a lot of this release was bug fixing, we wanted to use NUnit testing to make sure that once we patch the bugs, that they were truly resolved, as well as potentially uncover any other bugs that we could not find during the physical testing of the game. But the framework was hard to implement for us, and we were overall inexperienced with it. In the future, we would like to look back at the framework and its documentation to be able to understand and use it better in the future.

Figure 3 - Release 3 Screenshot



7 Comparison with Original Project Design Document

This project was initially proposed by Group 15 from the Fall 2020 semester [1]. Many of the features proposed by the original documentation were implemented in our demo and release of the game, but some aspects of the original documentation had to be cut, having to be implemented in a full development of the proposal, as we lacked the time to fully implement every aspect mentioned in the documentation.

Having said that, we did implement many features of the game that were mentioned in the original documentation. Some of them include the map feature, difficulty selection, usage of green and red tiles with their respective functionality.

Some aspects either had to be cut short, or left for further development of the game. Some of these include an option on the green tiles to either teleport or upgrade your piece. In the current iteration of the game, it only allows the player to upgrade their piece. Another aspect of the game allows it to be online, using a client and server connection. Again, due to the shortage of time, we kept the game single player with AI until we can develop the rest of the mechanics of the game.

III Testing

8 Items to be Tested

All script files will be tested, each of them having their respective categories in which to be tested with.

Chess Piece Movement: Bishop.cs, King.cs, Knight.cs, Piece.cs, Queen.cs, Rook.cs

Options Menu Scripts: BlackScript.cs, WhiteScript.cs

Drop Down Menus: BoardSizeDropdown.cs, ShrinkSpeedDropdown.cs, TileSpawningDropdown.cs

Text/Button Scripts: PlayButtonScript.cs, StatusTextScript.cs, ScoreTextScript.cs.

Game Flow: InitGrid.cs, Grid.cs, GameEndScript.cs, GoBetween.cs,

Game Mechanics: MoveOnGrid.cs, CameraZoom.cs

Player Stats: PlayerStats.cs, PlayerStatsContainer.cs, PlayerStatsHandler.cs

9 Test Specifications

ID# - FT 1

Description: Play Button in the main menu loads the difficulty selection menu.

Items covered by this test: PlayButtonScript.cs

Requirements addressed by this test: When initially launching the game, the play button should load you into the difficulty selection before loading into the board.

Environmental needs: N/A

Intercase Dependencies: N/A

Test Procedures: The player will load into the game and be greeted by the main menu. From there, the player can hit the play button in the main menu and be brought to the difficulty selection menu.

Input Specification: The input depends on the player actually clicking on the play button in the main menu.

Output Specifications: The output should be that the menu changes from the standard main menu screen, and the should display the difficulty selection menu.

Pass/Fail Criteria: If the player clicks play and it loads the difficulty selection menu, the test will pass. Otherwise, it will fail.

ID# - FT 2

Description: Quit button functionality

Items covered by this test: PlayButtonScript.cs

Requirements addressed by this test: When initially launching the game, the quit button should simply exit the application.

Environmental needs: N/A

Intercase Dependencies: N/A

Test Procedures: The player will load into the game and be greeted by the main menu. From there, the player will hit the quit button to exit the application.

Input Specification: The input depends on the player actually clicking on the quit button in the main menu.

Output Specifications: The output should be that the application closes.

Pass/Fail Criteria: If the player clicks the quit button and application exits, the test will pass. Otherwise, it will fail.

ID# - FT 3

Description: The Difficulty Selection, correctly loads into the game, and works with all different menu selections.

Items covered by this test: BoardSizeDropdown.cs, ShrinkSpeedDropdown.cs, TileSpawningDropdown.cs, InitGrid.cs, Grid.cs.

Requirements addressed by this test: When loading into the game, there should be an option to select the difficulty level depending on different aspects of the game.

Environmental needs: N/A

Intercase Dependencies: FT 1

Test Procedures: The player will load into the main menu, following the difficulty selection menu. From there, the player will use the drop down menu to select their appropriate difficulty, and should load into the main game.

Input Specification: The input depends on the player actually clicking on the play button in the main menu, and then the drop down menus in the difficulty selection.

Output Specifications: The player should then load into the game board with the proper difficulty.

Pass/Fail Criteria: If the player can use the drop down menus to select each difficulty, and load to the game properly, with the proper changes, then the test will pass. Otherwise, it will fail.

ID# - FT 4

Description: Options menu let you change between black and white pieces on the game board.

Items covered by this test: BlackScript.cs, WhiteScript.cs

Requirements addressed by this test: After changing which color piece you want in the options menu, it should reflect in the main game.

Environmental needs: N/A

Intercase Dependencies: FT 1, FT 3

Test Procedures: The player will choose which color piece on the options, and it will reflect in the main game.

Input Specification: The input depends on the player clicking into the options menu and selecting their color. Then exiting the options, and clicking through the rest of the menus to start the game.

Output Specifications: The player should be able to see the color that they chose during gameplay.

Pass/Fail Criteria: If the player selects either color, it should reflect as your color piece during gameplay, then the test will pass. Otherwise, it will fail.

ID# - FT 5

Description: Movement for the king piece.

Items covered by this test: Piece.cs, King.cs, MoveToGrid.cs

Requirements addressed by this test: The movement should be different for each corresponding game piece.

Environmental needs: N/A

Intercase Dependencies: FT 3

Test Procedures: When the player is a king, the movement should be 1 in any direction, and should be displayed as viable moves.

Input Specification: No specific input required, so long as the current piece is a king.

Output Specifications: The player should see all viable moves displayed as the yellow squares, be accurate for a king piece.

Pass/Fail Criteria: If the movement for the king is displayed as 1 in any direction, and can move to any of those tiles, the test will pass. Otherwise, it will fail.

ID# - FT 6

Description: Movement for the knight piece.

Items covered by this test: Piece.cs, Knight.cs, MoveToGrid.cs

Requirements addressed by this test: The movement should be different for each corresponding game piece.

Environmental needs: N/A

Intercase Dependencies: FT 3

Test Procedures: When the player is a knight, the movement should 2 in any direction, then 1 in a different direction, and should be displayed as viable moves.

Input Specification: No specific input required, so long as the current piece is a Knight.

Output Specifications: The player should see all viable moves displayed as the yellow squares, be accurate for a Knight piece.

Pass/Fail Criteria: If the movement for the Knight is displayed as 2 in any direction, then 1 in a different direction, and can move to any of those tiles, the test will pass. Otherwise, it will fail.

ID# - FT 7

Description: Movement for the Bishop piece.

Items covered by this test: Piece.cs, Bishop.cs, MoveToGrid.cs

Requirements addressed by this test: The movement should be different for each corresponding game piece.

Environmental needs: N/A

Intercase Dependencies: FT 3

Test Procedures: When the player is a Bishop, the movement should be moving across the diagonals with no restriction on how far, and should be displayed as viable moves.

Input Specification: No specific input required, so long as the current piece is a Bishop.

Output Specifications: The player should see all viable moves displayed as the yellow squares, be accurate for a Bishop piece.

Pass/Fail Criteria: If the movement for the Bishop is displayed as moving across the diagonals with no restriction on how far, and can move to any of those tiles, the test will pass. Otherwise, it will fail.

ID# - FT 8

Description: Movement for the Bishop piece.

Items covered by this test: Piece.cs, Rook.cs, MoveToGrid.cs

Requirements addressed by this test: The movement should be different for each corresponding game piece.

Environmental needs: N/A

Intercase Dependencies: FT 3

Test Procedures: When the player is a Rook, the movement should be moving across the cardinal directions with no restriction on how far, and should be displayed as viable moves.

Input Specification: No specific input required, so long as the current piece is a Rook.

Output Specifications: The player should see all viable moves displayed as the yellow squares, be accurate for a Rookpiece.

Pass/Fail Criteria: If the movement for the Rook is displayed as moving across the cardinal directions with no restriction on how far, and can move to any of those tiles, the test will pass. Otherwise, it will fail.

ID# - FT 9

Description: Movement for the Queen piece.

Items covered by this test: Piece.cs, Queen.cs, MoveToGrid.cs

Requirements addressed by this test: The movement should be different for each corresponding game piece.

Environmental needs: N/A

Intercase Dependencies: FT 3

Test Procedures: When the player is a Queen, the movement should be moving across the diagonals or the cardinal directions with no restriction on how far, and should be displayed as viable moves.

Input Specification: No specific input required, so long as the current piece is a Queen.

Output Specifications: The player should see all viable moves displayed as the yellow squares, be accurate for a Queen piece.

Pass/Fail Criteria: If the movement for the Queen is displayed as moving across the diagonals or the cardinal directions with no restriction on how far, and can move to any of those tiles, the test will pass. Otherwise, it will fail.

ID# - FT 10

Description: Red tile spawning and functionality

Items covered by this test: InitGrid.cs, Grid.cs, MoveToGrid.cs

Requirements addressed by this test: The board should randomly generate red tiles across the map.

Environmental needs: N/A

Intercase Dependencies: FT 3

Test Procedures: The Grid.cs should randomly be able to spawn red tiles across the map. The MoveToGrid.cs file should have the proper functionality with interacting with them.

Input Specification: Landing/Clicking on a red tile.

Output Specifications: The player should be able to see all of the red tiles on the board. If interacting with them should either downgrade/eliminate them from the game.

Pass/Fail Criteria: If the red tiles are randomly generated on the board, with the proper functionality when interacting with them occurs, then the test will pass. Otherwise, the test will fail.

ID# - FT 11

Description: Green tile spawning and functionality

Items covered by this test: InitGrid.cs, Grid.cs, MoveToGrid.cs

Requirements addressed by this test: The board should randomly generate Green tiles across the map.

Environmental needs: N/A

Intercase Dependencies: FT 3

Test Procedures: The Grid.cs should randomly be able to spawn green tiles across the map. The MoveToGrid.cs file should have the proper functionality with interacting with them.

Input Specification: Landing/Clicking on a green tile.

Output Specifications: The player should be able to see all of the green tiles on the board. If interacting with them it should promote the player's chess piece.

Pass/Fail Criteria: If the green tiles are randomly generated on the board, with the proper functionality when interacting with them, then the test will pass. Otherwise, the test will fail.

ID# - FT 12

Description: Score increasing properly per each player.

Items covered by this test: PlayerStats.cs, PlayerStatsContainer.cs, PlayerStatsHandler.cs

Requirements addressed by this test: The game should keep track of every player's score.

Environmental needs: N/A

Intercase Dependencies: N/A

Test Procedures: The side bar should display each of the corresponding players and their scores. It should change depending on the game mechanic that you interact with.

Input Specification: Interacting with different game mechanics.

Output Specifications: The scores should increase in the sidebar.

Pass/Fail Criteria: If the player interacts with green/red tiles, it should change the score by +2/-2 respectively. As well as capturing another player, it should increase the score by +2. It is then reflected in the sidebar. If this occurs, the test will pass. Otherwise, the test will fail.

ID# - FT 13

Description: End game scene accurately able to report if you win.

Items covered by this test: StatusTextScript.cs, GameEndScript.cs, GoBetween.cs,

Requirements addressed by this test: The game should have a way of showing whether or not you won.

Environmental needs: N/A

Intercase Dependencies: N/A

Test Procedures: After reaching an end game state, the board should be unloaded, and you should be loaded into the end game screen.

Input Specification: Reaching an end game state.

Output Specifications: Displaying the end game scene.

Pass/Fail Criteria: If you are the last player standing, the game should unload, and load the end game scene, and report that you won. If this is the case, the test passes. Otherwise, the test will fail.

ID# - FT 14

Description: End game scene accurately able to report if you lose.

Items covered by this test: StatusTextScript.cs, GameEndScript.cs, GoBetween.cs,

Requirements addressed by this test: The game should have a way of showing whether or not you won.

Environmental needs: N/A

Intercase Dependencies: N/A

Test Procedures: After reaching an end game state, the board should be unloaded, and you should be loaded into the end game screen.

Input Specification: Reaching an end game state.

Output Specifications: Displaying the end game scene.

Pass/Fail Criteria: If you are not the last player standing, the game should unload, and load the end game scene, and report that you lost. If this is the case, the test passes. Otherwise, the test will fail.

ID# - FT 15

Description: End game scene accurately able to report your score regardless if you won/lost.

Items covered by this test: ScoreTextScript.cs, GameEndScript.cs, GoBetween.cs,

Requirements addressed by this test: The game should have a way of showing whether or not you won.

Environmental needs: N/A

Intercase Dependencies: N/A

Test Procedures: After reaching an end game state, the board should be unloaded, and you should be loaded into the end game screen, with the recorded score from the game.

Input Specification: Reaching an end game state.

Output Specifications: Displaying the end game scene, showing the score.

Pass/Fail Criteria: When the end game scene is loaded, it should report your score on the screen, being the score from the game. If this is the case, then the test passes. Otherwise, the test will fail.

ID# - FT 16

Description: End game scene should be able to return you to the main menu.

Items covered by this test: GameEndScript.cs, GoBetween.cs,

Requirements addressed by this test: The game should have a way of showing whether or not you won, giving you the option to go back to the main menu.

Environmental needs: N/A

Intercase Dependencies: N/A

Test Procedures: After reaching an end game state, the board should be unloaded, and you should be loaded into the end game screen, with a button to return to the main menu.

Input Specification: Reaching an end game state.

Output Specifications: Displaying the end game scene, showing the main menu button.

Pass/Fail Criteria: When the end game scene is loaded, you should be able to click the main menu button to return there. If this is the case, then the test passes. Otherwise, it will fail.

ID# - FT 17

Description: Implementing the game mechanic of the board shrinking, having the playable area decrease after 2 rotations.

Items covered by this test: Grid.cs, InitGrid.cs, MoveToGrid.cs

Requirements addressed by this test: The game should implement the aspect of the battle royale element of the game, having the board shrinking.

Environmental needs: N/A

Intercase Dependencies: FT 3

Test Procedures: Let each of the players make their respective moves twice. After that, it should trigger the function call to make the board shrink. The board shrinkage is determined by the difficulty selection.

Input Specification: Making 2 moves.

Output Specifications: There should be a ring of red tiles around the board that begins to start closing in.

Pass/Fail Criteria: Make two moves, waiting for each other player to finish making their moves as well. After the two rotations, there should be an increasing red ring surrounding the players. If this is the case, the test will pass. Otherwise, the test will fail.

ID# - FT 18

Description: Players should be able to capture other pieces.

Items covered by this test: Grid.cs, InitGrid.cs, MoveToGrid.cs

Requirements addressed by this test: The game should implement the aspect of chess where you can capture other pieces.

Environmental needs: N/A

Intercase Dependencies: FT 3

Test Procedures: Play into a game, and reach a scenario where you can capture another player.

Input Specification: Moving to the other player.

Output Specifications: The captured player should be eliminated from the game, as well as the player who captured should be their piece afterwards.

Pass/Fail Criteria: If the player captures a piece, the captured player should be eliminated from the game, as well as the capturing player should be the piece they captured. If this is the case, the test passes. Otherwise, it will fail.

ID# - FT 19

Description: The player should be able to zoom in and out using the scroll wheel.

Items covered by this test: CameraZoom.cs

Requirements addressed by this test: To make the game more well polished, the player should be able to zoom in and out using the scroll wheel.

Environmental needs: N/A

Intercase Dependencies: FT 3

Test Procedures: Use the scroll wheel to change the camera zoom.

Input Specification: Scrolling on the mouse wheel.

Output Specifications: The camera should zoom in and out.

Pass/Fail Criteria: If the camera changes in accordance with the scroll wheel, the test will pass. Otherwise, it will fail.

ID# - FT 20

Description: The Grid and Init Grid should be used to load up dynamic boards properly.

Items covered by this test: Grid.cs, InitGrid.cs

Requirements addressed by this test: To be able to load into the game properly, making sure these files work in tandem with each other.

Environmental needs: N/A

Intercase Dependencies:N/A

Test Procedures: Loading into the game changing the parameters required for the grid class creation.

Input Specification: Creating a game.

Output Specifications: Being able to see all aspects of the game, without any null references.

Pass/Fail Criteria: If the grid files are able to load into a board without any errors, the test will pass. Otherwise, it will fail.

10 Test Results

ID# - FT 1

Date(s) of Execution: 4/27/2021

Staff conducting test: Dan Hrubec

Expected Results: When pressing play, the difficulty selection menu opens.

Actual Results: I pressed play, and it loaded into the difficulty selection.

Test Status: Pass

ID# - FT 2

Date(s) of Execution: 4/27/2021

Staff conducting test: Dan Hrubec

Expected Results: When pressing quit, the application exits.

Actual Results: I pressed quit and it exited out of the program back to the desktop.

Test Status: Pass

ID# - FT 3

Date(s) of Execution: 4/27/2021

Staff conducting test: Dan Hrubec

Expected Results: Each of the different difficulties for each of the drop down menus will change the gameplay when loading into the board.

Actual Results: After selecting a difficulty for each, for every drop down menu item, the game loads, producing the appropriate results for the selected difficulties.

Test Status: Pass

ID# - FT 4

Date(s) of Execution: 4/27/2021

Staff conducting test: Dan Hrubec

Expected Results: Options menu will allow the player to select between the black pieces and the white pieces. When loading the game, your color should be the one you selected in the options, with white being the default.

Actual Results: When selecting nothing in the options menu, I was loaded in as a white king. When selecting between white and black, I was loaded as the correct sprites.

Test Status: Pass

ID# - FT 5

Date(s) of Execution: 4/27/2021

Staff conducting test: Dan Hrubec

Expected Results: Movement is correct for the king. As a king you are only allowed to move 1 in any direction. The valid moves indicator should reflect this, and you should not be able to move anywhere that is not highlighted.

Actual Results: The movement for the king is accurately displayed in the move indicator. You are not allowed to move to any tile that is not displayed by the indicator.

Test Status: Pass

ID# - FT 6

Date(s) of Execution: 4/27/2021

Staff conducting test: Dan Hrubec

Expected Results: Movement is correct for the knight. As a knight you should only be able to move 2 in one direction, then 1 in a different direction. The valid moves indicator should reflect this, and you should not be able to move anywhere that is not highlighted.

Actual Results: The movement for the knight is accurately displayed in the move indicator. You are not allowed to move to any tile that is not displayed by the indicator.

Test Status: Pass

ID# - FT 7

Date(s) of Execution: 4/27/2021

Staff conducting test: Dan Hrubec

Expected Results: Movement is correct for the bishop. As a bishop your movement should only be along the diagonals, with no restrictions as to how far. The valid moves indicator should reflect this, and you should not be able to move anywhere that is not highlighted.

Actual Results: The movement for the bishop is accurately displayed in the move indicator. You are not allowed to move to any tile that is not displayed by the indicator.

Test Status: Pass

ID# - FT 8

Date(s) of Execution: 4/28/2021

Staff conducting test: Julian Gonzales

Expected Results: Movement is correct for the rook . As a rook your movement should be along the cardinal directions, with no restrictions as to how far. The valid moves indicator should reflect this, and you should not be able to move anywhere that is not highlighted.

Actual Results: The movement for the rook is accurately displayed in the move indicator. You are not allowed to move to any tile that is not displayed by the indicator.

Test Status: Pass

ID# - FT 9

Date(s) of Execution: 4/28/2021

Staff conducting test: Julian Gonzales

Expected Results: Movement is correct for the queen. As a queen your movement should be along the cardinal directions as well as across the diagonals, with no restrictions as to how far. The valid moves indicator should reflect this, and you should not be able to move anywhere that is not highlighted.

Actual Results: The movement for the rook is accurately displayed in the move indicator. You are not allowed to move to any tile that is not displayed by the indicator.

Test Status: Pass

ID# - FT 10

Date(s) of Execution: 4/28/2021

Staff conducting test: Julian Gonzales

Expected Results: The functionality of the red tiles should be working. Landing on a red tile will downgrade your piece. If you are already the lowest possible piece in the king, you will be eliminated from the game.

Actual Results: Aftering force spawning as a queen, being the highest possible piece, and landing on red tiles, we correctly downgrade to a king until we are ultimately eliminated by landing on a red tile as a king.

Test Status: Pass

ID# - FT 11

Date(s) of Execution: 4/28/2021

Staff conducting test: Julian Gonzales

Expected Results: Functionality for the green tiles should be working. Landing on a green tile will upgrade your current game piece. Landing on a green tile as a fully upgraded piece, being the queen, will no longer upgrade your piece, but still provide points to your score.

Actual Results: Upgrading through the chess pieces work by landing on the green tiles. Landing on a green tile as a queen does not upgrade your piece, but still gives the notification that it was upgraded.

Test Status: Fail

ID# - FT 12

Date(s) of Execution: 4/28/2021

Staff conducting test: Julian Gonzales

Expected Results: Score is being calculated properly. By landing on a green or red piece, it should update your score by +2 or -2 respectively. Capturing another piece will also increase your score by +2.

Actual Results: Testing both of the red and green tiles, it does calculate the score properly. As well as capturing another piece also increases the score properly.

Test Status: Pass

ID# - FT 13

Date(s) of Execution: 4/28/2021

Staff conducting test: Julian Gonzales

Expected Results: The end game scene should accurately report if you have won the game and exits only when all other players are eliminated.

Actual Results: By becoming the last piece standing, it loads up the end game scene and correctly reports that we have won the game.

Test Status: Pass

ID# - FT 14

Date(s) of Execution: 4/28/2021

Staff conducting test: Julian Gonzales

Expected Results: The end game scene should accurately report if you have lost the game, and exists when you are not the last player standing.

Actual Results: The end game scene does get loaded if we are not the last man standing and are eliminated early. It correctly reports that we have lost the game.

Test Status: Pass

ID# - FT 15

Date(s) of Execution: 4/29/2021

Staff conducting test: Joseph Canning

Expected Results: The end game scene should accurately report your score, regardless if you have won or lost.

Actual Results: The end game scene gets loaded and reports the players score from the game correctly. It works in both scenarios if the player has won or if the player has lost.

Test Status: Pass

ID# - FT 16

Date(s) of Execution: 4/29/2021

Staff conducting test: Joseph Canning

Expected Results: The end game scene should have a button, when pressed,

returns you to the main menu.

Actual Results: The end game scene gets loaded in, regardless of if the player has won or lost, providing a clearly visible main menu button. When pressed it brings the player back to the main menu.

Test Status: Pass

ID# - FT 17

Date(s) of Execution: 4/29/2021

Staff conducting test: Joseph Canning

Expected Results: The board should shrink after 2 player rotations of moves.

Actual Results: After playing the game multiple times, in most scenarios the map shrinks after 2 rotations of players. However, this will sometimes bug out, when an AI player moves multiple times during a rotation. This is detailed in OI #1.

Test Status: Fail

ID# - FT 18

Date(s) of Execution: 4/29/2021

Staff conducting test: Joseph Canning

Expected Results: If you land on another player, they should be eliminated from the game regardless of what piece they were, and the player that captured them should now become that respective chess piece.

Actual Results: During gameplay, it was possible to capture another player by landing on the same square that they were at. This successfully eliminates the other player, and updates the game piece to their respective one.

Test Status: Pass

ID# - FT 19

Date(s) of Execution: 4/29/2021

Staff conducting test: Joseph Canning

Expected Results: During gameplay, the player can use the scroll wheel to zoom in and out of the camera.

Actual Results: Using the scroll wheel changes the camera to be able to zoom in and out depending if you scroll up or down respectively.

Test Status: Pass

ID# - FT 20

Date(s) of Execution: 4/29/2021

Staff conducting test: Joseph Canning

Expected Results: The Grid.cs and InitGrid.cs should work together in order to produce a dynamic board size depending on the values sent in.

Actual Results: The script files properly generate boards depending on the different parameters sent in as arguments.

Test Status: Pass

11 Regression Testing

Considering the scope of this project, and the created tests previously mentioned, there was no need for regression testing to be done for this project.

IV Inspection

12 Items to be Inspected

The items to be inspected were previously mentioned in section 8. These were all the script files used to create the game.

All script files will be tested, each of them having their respective categories in which to be tested with.

Chess Piece Movement: Bishop.cs, King.cs, Knight.cs, Piece.cs, Queen.cs, Rook.cs

Options Menu Scripts: BlackScript.cs, WhiteScript.cs

Drop Down Menus: BoardSizeDropdown.cs, ShrinkSpeedDropdown.cs, TileSpawningDropdown.cs

Text/Button Scripts: PlayButtonScript.cs, StatusTextScript.cs, ScoreTextScript.cs.

Game Flow: InitGrid.cs, Grid.cs, GameEndScript.cs, GoBetween.cs,

Game Mechanics: MoveOnGrid.cs, CameraZoom.cs

Player Stats: PlayerStats.cs, PlayerStatsContainer.cs, PlayerStatsHandler.cs

13 Inspection Procedures

For inspection of our code, we are reusing an old reference to a Java code checklist provided by the University of Toronto[2]. Obviously, there will be some differences between the usage of the checklist with respect to our code being written in C#, but as the languages are similar in many ways, we ultimately decided to use it again.

Since a majority of our code was written collaboratively, it was hard to pinpoint which person wrote each section of code, so we met collaboratively over discord and went over each file, using the checklist to determine how well it was written. We would all look at the individually first, then collaboratively go over different talking points that we have found during our individual inspection.

14 Inspection Results

Movement Scripts: Bishop.cs, King.cs, Knight.cs, Piece.cs, Queen.cs, Rook.cs

Inspectors: Julian Gonzales, Dan Hrubec, Joseph Canning

Date of Inspection: 4/30/2021

Result: Passing

A lot of this code was the usage of an abstract class, with each of the individual chess piece classes inheriting from it. It would then be used in a lot of interaction and movement checking. Overall the structure of this idea was well implemented, and executed well in the scope of our project.

However, there was definitely a lack of documentation within the code, so to outside readers it can be confusing as to what the purpose was to each of the functions and why it was written in the way that it was.

The main takeaway was that for us, was that our initial inspection was good, that it was well implemented and executed. But we need to take into consideration documentation of our code through use of comments to really explain what exactly was happening at each section.

Miscellaneous Small Scripts : BlackScript.cs, WhiteScript.cs, BoardSizeDropdown.cs, ShrinkSpeedDropdown.cs, TileSpawningDropdown.cs, PlayButtonScript.cs, StatusTextScript.cs, ScoreTextScript.cs, GameEndScript.cs, GoBetween.cs, CameraZoom.cs.

Inspectors: Julian Gonzales, Dan Hrubec, Joseph Canning

Date of Inspection: 4/30/2021

Result: Passing

The reason for having such a big grouping like this in our inspection was that individually there was not much to talk about them, since the content within each of these script files were only around 50 lines of code or less. So combining them together in a group like this aided our inspections because they were all very similar.

Since a lot of these script files were small, this means that they only controlled small aspects of the game individually. Only controlling one aspect of the game per script file, so each of them all performed their duty. So the content and organization of it was somewhat well done. We could have done further organization by sectioning them in folders depending on different aspects of the game.

Little to no documentation was included in a lot of these script files, but since they were so small, it is not entirely the worst thing. Given the descriptive name of the scripts, it was easy to tell which game aspect the script was controlling and what it was there to accomplish.

Game Mechanics: MoveOnGrid.cs

Inspectors: Julian Gonzales, Dan Hrubec, Joseph Canning

Date of Inspection: 4/30/2021

Result: Fail

The MoveOnGrid script is a large file that controls and checks for a lot of different interactions in the game. During the development and usage of this file, it became more and more disorganized, and harder to follow. There were some good elements within it, but with little documentation, it was still difficult to understand if reading it for the first time.

We did refactor some of the code to be able to split it into its own functions and make the functions calls be the usage in the main Update() function that was the prebuilty unity function. But many things were inefficient to have to split into respective functions, so the main Update() function was unorganized. Despite the unorganized nature of the script, it still worked well, and all interactions were accounted for.

Moving forward, an overhaul and full refactoring of this script file will be much needed to make the file more readable, and easier to work with for further development.

Game Flow: InitGrid.cs, Grid.cs,

Inspectors: Julian Gonzales, Dan Hrubec, Joseph Canning

Date of Inspection: 4/30/2021

Result: Pass

For the InitGrid and Grid.cs, this controls the backbone of the project, creating the underlying grid system, to allow and check if many interactions have occurred. The file loads in all potential sprites, sets up the color scheming, the players, drives the board shrinking amongst many other features.

Overall the code was easy to follow, being our most documented script file, it was easy to follow and to understand the usage of each element of the file. The functions and variable names were well defined, making it easy to know which variable controls what aspect of the game. The only addition to the documentation would be preconditions and postconditions for each of the different functions within it. Considering all of this, this file passes the inspection, but still has room for improvement.

Player Stats: PlayerStats.cs, PlayerStatsContainer.cs, PlayerStatsHandler.cs

Inspectors: Julian Gonzales, Dan Hrubec, Joseph Canning

Date of Inspection: 4/30/2021

Result: Pass

These script files control all of the players and their stats. It functions properly with no current bugs noticed. As with many of our other problems the documentation seems lacking in our code, making it easy for us the developers to understand as we are the authors of the code, but can be difficult to understand to any future developers, or readers.

Going back and documenting the purpose of the code, with preconditions and postconditions for each the functions is needed. The code functions well, and is structured properly, so not much refactoring of the code will be needed.

V Recommendations and Conclusions

Chess Piece Movement: Bishop.cs, King.cs, Knight.cs, Piece.cs, Queen.cs, Rook.cs

Testing: Passed

Inspection: Passed

Future Actions: More documentation.

Options Menu Scripts: BlackScript.cs, WhiteScript.cs

Testing: Passed

Inspection: Passed

Future Actions: None

Drop Down Menus: BoardSizeDropdown.cs, ShrinkSpeedDropdown.cs, TileSpawningDropdown.cs

Testing: Passed

Inspection: Passed

Future Actions: None

Text/Button Scripts: PlayButtonScript.cs, StatusTextScript.cs, ScoreTextScript.cs.

Testing: Passed

Inspection: Passed

Future Actions: None

Game Flow: InitGrid.cs, Grid.cs, GameEndScript.cs, GoBetween.cs,

Testing: Passed

Inspection: Passed

Future Actions: Precondition and postcondition documentation in the Grid.cs file.

Game Mechanics: MoveOnGrid.cs, CameraZoom.cs

Testing: Passed

Inspection: Failed

Future Actions: An overhaul and refactoring needed on MoveOnGrid.cs. Code is all

over the place and hard to follow. Will need updates before further development to prevent future miscommunications and bugs.

Player Stats: PlayerStats.cs, PlayerStatsContainer.cs, PlayerStatsHandler.cs

Testing: Passed

Inspection: Passed

Future Actions: More documentation within the three classes.

VI Project Issues

15 Open Issues

OI#1 - Random multiple movements

Content: Even though the random movements bug was drastically reduced in our scenario 3 it sometimes randomly allows an AI to move twice.

Motivation: This needs to be fixed in order to provide a better experience for the user and to also make the quality and reliability of the game much better.

OI#2 - Restart at end screen

Content: Currently the player is unable to restart the game when it is finished they have to go to the home menu and then input the same settings to get another iteration of the same game.

Motivation: Fixing this issue would motivate the user to play the game again since there are less options involved in order to play the game again.

OI#3 - Scoreboard not dynamic to multiple screen sizes

Content: The score board currently changes in size depending on how big the screen size is. At times making it difficult for users to read what score they have

Motivation: Since users will be looking at the score frequently it is essential that it does not interfere with the gameplay and making it dynamic will insure that the user can easily read their score and get back to playing.

OI#4 - Local multiplayer gameplay

Content: The current game does not support the option local gameplay

which could be another interesting way to play the game.

Motivation: The addition of online will attract more users since they enjoy playing with friends and family. It will also prevent them from eventually getting bored from playing with an AI.

OI#5 - AI difficulty

Content: The game currently supports only one easy difficulty. The addition of more difficult AI needs to be added and the ability for the AI to win the game.

Motivation: This is needed to keep the user challenged and prevent them from getting bored from the game. This will also allow multiple types of users to play the game from children who play against the easy AI to adults who play the harder one.

OI#6 - Background Music

Content: Play music in the background to keep the user focused also adding the ability to mute the music during gameplay.

Motivation: Some players like when their is noise in the background and it also gives it a more finished feel to the game

OI#7 - Finished options menu

Content: Give the user the ability to adjust options like sounds and themes.

Motivation: Finishing this will also give the user a more finished feel to the game.

OI#8 - Dynamic number of players

Content: Currently the game only allows 4 players to play the game. It should be set to where the player picks the amount of AI or a lobby decides how many players are in the game. Up to a certain maximum

Motivation: This will allow the user to create more difficult games with more players. Allowing the user to become more entertained.

16 Waiting Room

WR#1 - Online multiplayer gameplay

Content: In the future online gameplay should be added to allow players to compete with other players also playing the game

Motivation: This will allow players to play competitively and possibly

create some type of ranked system. This will motivate players to keep playing the game and keep them further entertained.

WR#2 - Leaderboards

Content: Adding a leaderboard to create more competition against players

Motivation: Users that play the game more competitively will want a system to show how well they are doing against other players.

WR#3 - Themes for boards and pieces

Content: Give the player the ability to either purchase or create their own themes

Motivation: This can create revenue if the player would like to buy a theme that they like. Also maybe winning themes can be a way the user can be motivated to play the game more often

WR#4 - In game currency

Content: Like many other games a form of currency is usually used to create revenue. This can be used to spend on themes.

Motivation: The currency can be used as a reward for winning the game which could be then spent on themes for the users board and pieces. The currency can also be purchased to create profit from the game

17 Ideas for Solutions

IS#1 - Random multiple movements

Content: This could be fixed by refactoring and possibly rebuilding the MoveToGrid.cs file. The algorithm could be improved and the bug could be found in the process. Refactoring will also allow for more expansion in the future.

Motivation: Refactoring will provide future implementations to become much easier as currently it is quite messy. This will also make the process more efficient and less time consuming. Possibly eliminating future bugs.

IS#2 - Restart at end screen

Content: This can be done by maybe saving the start settings in an object and running the main grid script with these settings once again. Then moving the player from the end scene to the game scene

Motivation: Fixing this issue would motivate the user to play the game

again since there are less options involved in order to play the game again.

IS#3 - Scoreboard not dynamic to multiple screen sizes

Content: One way this can be fixed is by reading the screen size and creating multiple general profiles for different screen size ranges. Whenever the user loads up the game a specific profile will be used.

Motivation: By doing it this way other UI can use these profiles in order to adapt its own UI to different sizes

IS#4 - Local multiplayer gameplay

Content: It can be implemented by using multiple player objects instead of AI. Allowing multiple pieces to make moves. Similar to how the first player moves.

Motivation: Doing it this way allows the ability to make the amount of players dynamic. Since a simple addition of a player object will add another player.

IS#5 - AI difficulty

Content: Creating different algorithms that the AI player object can be injected with is a good way to implement different move types. Maybe creating a shortest path algorithm to the closest player can be a medium and then hard can be shortest path plus setting up for an elimination once it is close

Motivation: Using dependency injection is a simple way to allow the player object to move in different ways. If a better algorithm is developed in the future it can easily be injected into the player object without changing anything else

IS#6 - Background Music

Content: This is a simple fix and it can be done by attaching a sound file to one of the objects on the screen maybe the camera since it is always present

Motivation: This is the simplest way of doing it which can also be toggled for the mute ability

IS#7 - Finished options menu

Content: This can be done similar to the difficulty menu that was already created. Just changing the layout and setting up the scripts for each menu option.

Motivation: Straight forward way of adding options also allowing the

options menu to expand if needed in the future

18 Project Retrospective

PR#1 - Map Creation

Content: The map was able to be dynamic quite easily and allowed for multiple sizes. Which then allowed the adjustment of difficulties.

Motivation: This removes the need to create more code for each board size and with simple parameters it can be adjusted

PR#2 - Random red and green tile interaction

Content: The generation of tiles was very random and at times making it quite difficult to eliminate players directly which is the goal. They always generated densely when needed and also throughout the board

Motivation: This created a better experience for the player since it was more challenging with more dense red and green tiles

PR#3 - Displaying Movable Spots

Content: The ability to see where a player can move to was a good addition to the game. It kept the user from having to guess where they could move to.

Motivation: It created a better experience for the player allowing them to see where they could move made the interaction more inviting.

PR#4 - Piece move set classes and super class

Content: The inheritance from a base piece class and creating piece movements from that base class was very fluid and allowed an easy creation of piece movesets.

Motivation: It was done this way to allow the creation of more movesets if needed. Also allowing a better algorithm to easily be swapped in for the old one.

VII References / Bibliography

[1] Zephania Lema, Navya Reddy, Hasan Sehwail, Sumayya Siddiqui,Chess Royale Project Report

[2] University of Toronto, Java Inspection Checklist

https://www.cs.toronto.edu/~sme/CSC444F/handouts/java_checklist.pdf

[3] M. Fowler, UML Distilled, Third Edition, Boston: Pearson Education, 2004.