Nuffatafl

Final Report

Mark of the Chili Onions

Mark Reggiardo, Cash Hilstad, Orion Hess

Course: CPSC 224 - Software Development

I. Introduction and Project Description

Our team made a version of the strategy-based checkerboard game Tablut, a member of the Tafl family of games which were initially developed in fourth through 12th-century Northern Europe. The rules of Tablut are one of the most well-preserved amongst the Tafl game family but still have struggled to persist over the centuries [1]. We referenced the rule set found at brainking.com [2] as inspiration for our game.

The game consists of two teams, an attacking team and a defending team playing on a nine-by-nine grid similar to the eight-by-eight grid of chess or checkers. The defending team has a King and their goal is to get the King to one of the four corners of the board to win. The attacking team's goal is to surround the King on all sides and trap that piece to win. Each turn, each side gets a chance to move one piece either up, down, left, or right by as many contiguous spaces as are not occupied by another piece. If two pieces of the same team surround a piece of the opposing team on two sides (either top and bottom or left and right), that piece is eliminated from the board.

Nuffatafl was made in Java using the Java Swing GUI library (a sad excuse for a library honestly, but it's what we had). We implemented a full UI for the game and complete business logic for the rules of the game, which we brought to life through UI with MVC. This document will go over the process of creating Nuffatafl along with project requirements, solution approach, test plan, implementation description, and future work.

II. Team Members - Bios and Project Roles

Orion Hess is a first-year computer science student interested in software security, Linux, and info tech. His prior projects have included an evolution simulation, discord bots, and a rust-based text adventure game. Orion's skills include C++, Python, Java, C#, Linux, and Docker. His responsibilities during the development of Nuffatafl included some general-purpose screen design, settings functionality, settings persistence, and bug fixing.

Mark Regiardo is a Junior studying computer science and pursuing a bachelor's of science, who started at Gonzaga University this semester, transferring in from a community college in California after completing an A.A. degree. Mark has prior experience with C/C++, Javascript, and Swift, but only started learning Java this semester and has made hobby projects in SwiftUI for the Apple device ecosystem. Mark focused on implementing MVC, theming, and the gameplay screens for the program as well as creating unreasonably long pull requests on GitHub.

Cash Hilstad is a first-year computer science student interested in software development, particularly game development, evolution simulation, and embedded systems. His prior projects include numerous Godot games, an Android app, a personal website, and more. Cash's skills include GdScript, Java, Python, C++, and Rust. His responsibilities for Nuffatafl included the development and design of the backend, functionality testing, game icon, and bug fixing.

III. Project Requirements

This section details the functional and non-functional requirements for our project along with their importance and a description of how each requirement can be met. The functional requirements pertain to the mandatory features the game needed to implement and the nonfunctional requirements refer to the standards to which these features need to be implemented.

Functional Requirements

Feature	***Description***
Welcome Screen	High Priority: A welcome screen that appears when the program begins or before a new game that contains an image and allows a user to start a new game and view the rules
Gameplay Screen	High Priority: A screen on which the game is played that displays a 9x9 checkerboard, a button to view the rules, a button to view game settings, information about each player and which player's turn it is, and information on eliminated pieces and turn history
After Game Screen	High Priority: A screen that appears after a game concludes that displays which player won, how many of each player's pieces were eliminated from the game, and a turn history as well as a button that allows a user to start a new game
Rules Screen	Medium Priority: A screen that lists the rules and is accessible from the welcome screen and the gameplay screen
Settings	Medium Priority: A screen allowing a user to change the theme, toggle focused gameplay mode, and start a new game
Player Profiles	Medium Priority: Each player shall be able to set their name and an icon that represents them. In the absence of user input, users shall default to "Player 1", "Player 2", etc.

Board Layout	High Priority: The board shall initially be laid out according to Fig. 7
Turns	High Priority: Turns shall alternate between each player and shall consist of allowing a player to move one piece up, down, left, or right by as many contiguous spaces as are empty.
Eliminating a Piece from the Board	High Priority: If a piece is surrounded on two opposite sides, either left and right or up and down, it shall be eliminated from the board
Winning Mechanisms for Attacker and Defender	High Priority: The defender shall win if their King piece reaches any corner of the board or if every attacker is captured. The attacker shall win if they surround the attacker's King on all four sides. One of these sides can be the center tile.

Non-Functional Requirements

Feature	***Description***
Themes	The user should be able to change the background colors of the program, as well as the accent colors, and text colors for the program
Focus Mode	The user should be able to toggle whether the captured pieces view and turn history view are displayed on the gameplay screen
Highlighting Mode	The user should be able to toggle whether valid spots to move to are highlighted.
Persistent Settings	The user should have their preferred theme, focus mode, and highlighting mode saved and loaded for each startup of the game.
Auto Focus Mode	The program shall turn on focus mode when the window width becomes small enough that the board is hard to see and the program shall turn off focus mode when the window width becomes large enough again. If the user toggles the focus mode, auto focus mode shall not override their preference.
Board and Tile Resizability	The board and its tiles shall resize upon window resize and maintain a square aspect ratio in all scenarios.

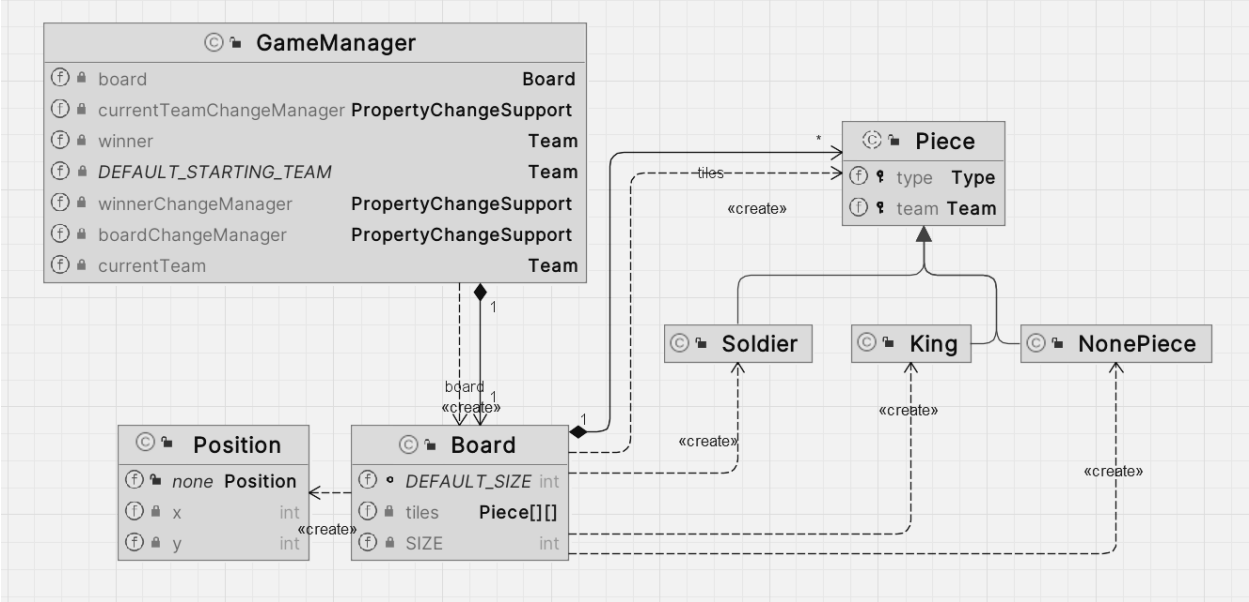
IV. Solution Approach

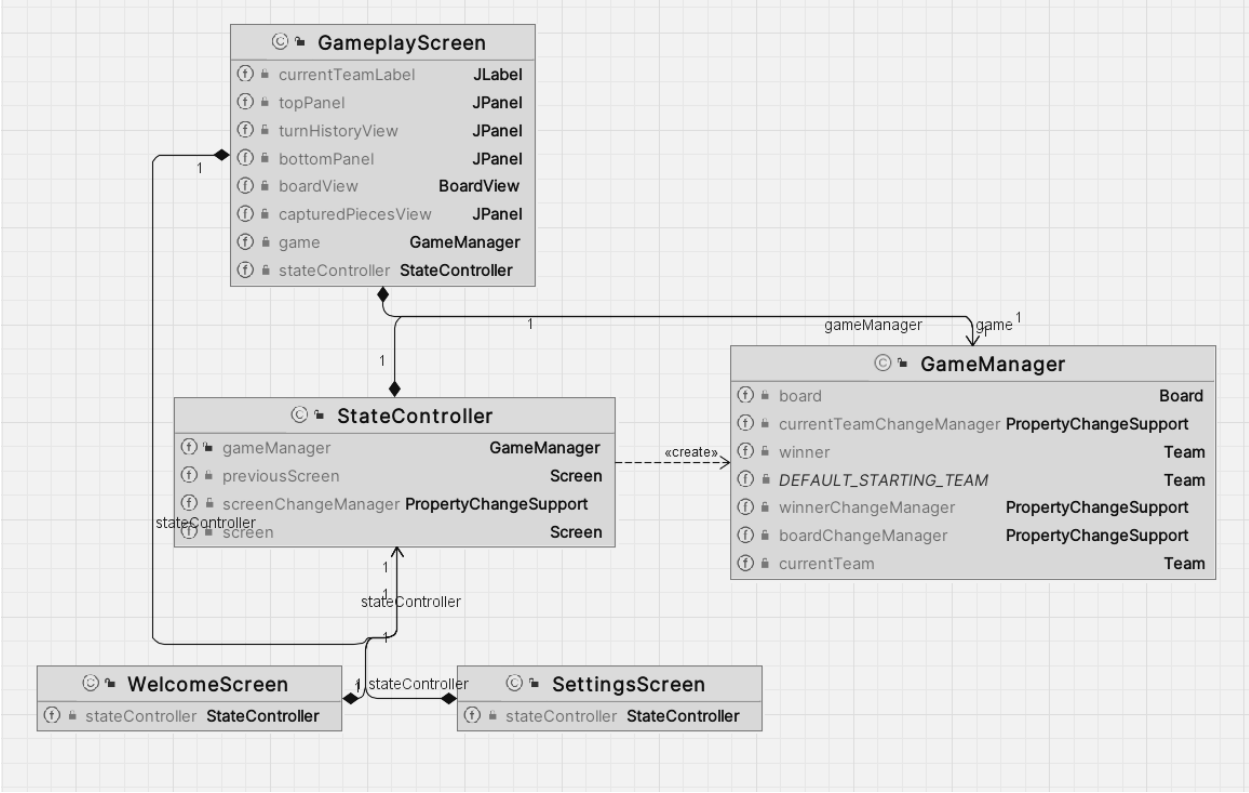
Our project used OOP and MVC to implement a GUI. But seriously, enough with the initialisms, how did we make this work? To avoid a circular reference, we put our controller (The StateController and the GameManager contained within) inside the class for our main view (MainView). We located our (Board) inside the GameManager to keep everything in one place. The StateController uses the observer pattern and an enum called Screen to keep track of the current and previous screen of the program. GameManager is a controller that focuses on controlling the game logic and interfacing between the views and the Board model. Our main view was also saved to a static member variable of our main class to be easily referenced by JDialogs later in the program.

Our views consisted largely of subclassed JFrames for each “screen,” as well as other JComponents housed within. To accomplish theming, a Theme class was created with a little bit of clever trickery to make theming less of a nightmare than Java Swing would have liked it to be. Theme contains a static member called current which acts almost like a singleton in that it keeps one theme in a centralized location. Each Theme object contains a ColorKey which is an interface that defines a lambda expression which takes a ThemeKey enum, which defines a color for each theme to implement (such as background2 or text), as an argument and returns a color. This allowed us to implement each theme succinctly using a constructor that took a string for the name and a lambda expression which returned the correct color based on a switch over the input enum. We then created static methods which took a JComponent and a ThemeKey to set the background or foreground color for the given JComponent and update it each time the current theme changes using the observer pattern. To simplify code for ourselves, we created some ThemeComponents like ThemeLabel and ThemeButton which implement theme conformance of your favorite JComponents without any additional code, yay!

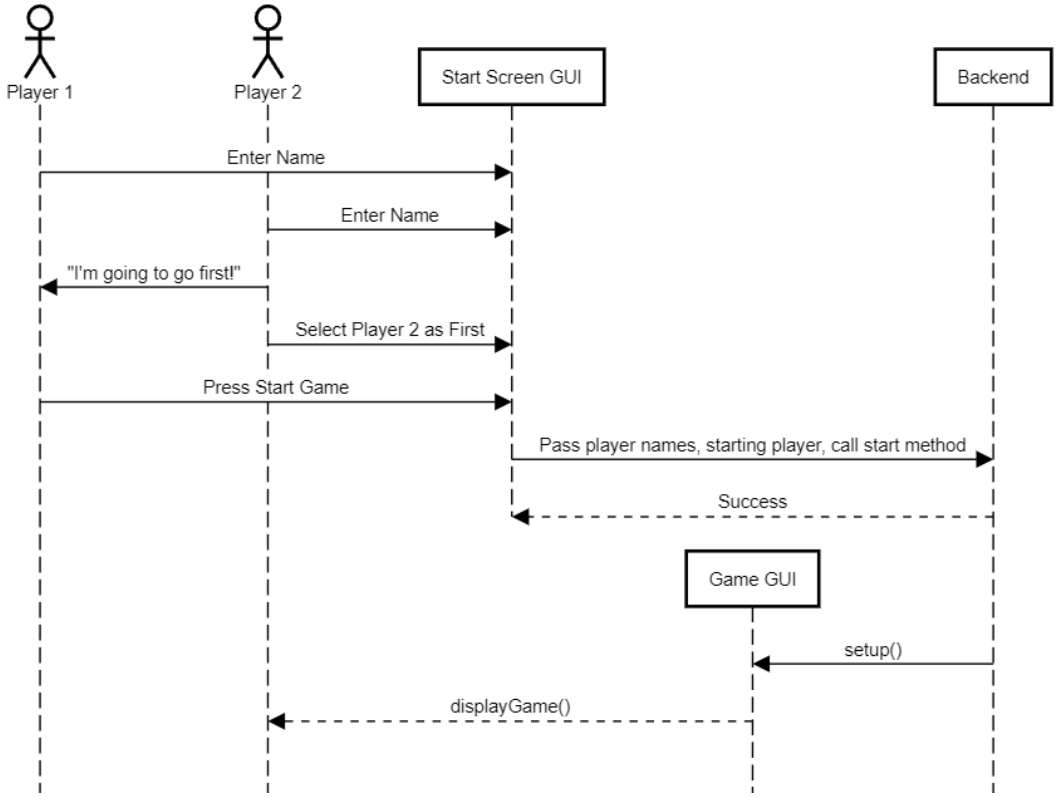
To handle the backend game logic, we created a Board class. At the center of its functionality is a two dimensional array of Pieces called tiles. This is where the entire board’s state is stored; all piece locations are on this array. As it holds all the data, the Board also contains all of the functions necessary to implement the rules, such as capturing and moving rules. Piece is an abstract class, and Soldier, King, and NonePiece implement it.

Two things we implemented to enable easy communication from all parts of the project was the Team enum and the Position class. Team was a simple public enum that represented a given team, attacker, defender, or none. Position was a simple class that represented a 2D position. It lets us easily communicate spots on the board without needing to pass two separate coordinates. These two features helped to make everything mesh well together.

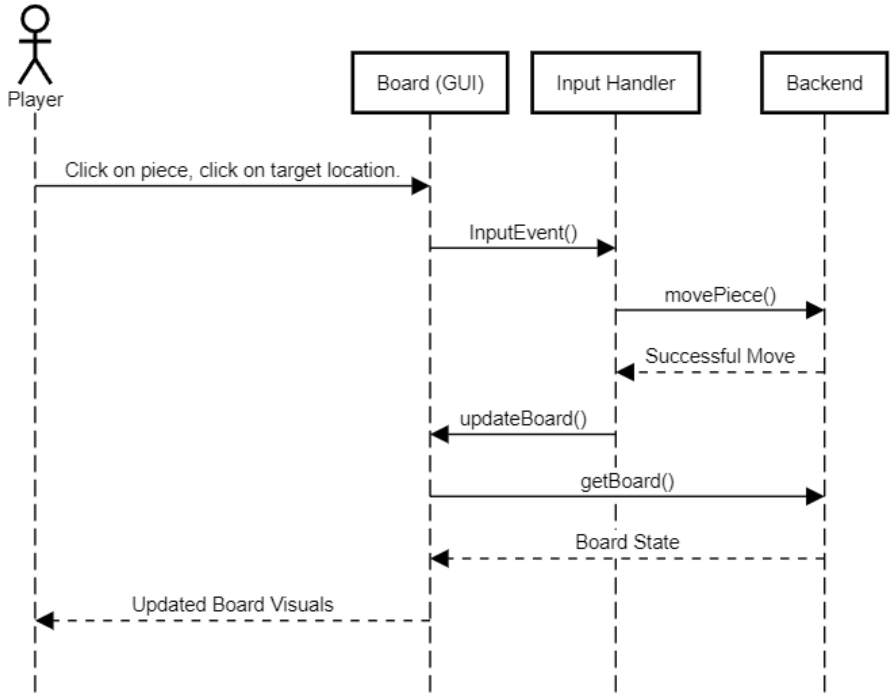

GameManager
board Board
currentTeamChangeManager PropertyChangeSupport
winner Team
DEFAULT_STARTING_TEAM Team
winnerChangeManager PropertyChangeSupport
boardChangeManager PropertyChangeSupport
currentTeam Team
Piece
type Type
team Team
tiles
«create»
Soldier
King
NonePiece
board
«create»
Position
none Position
x int
y int
Board
DEFAULT_SIZE int
tiles Piece[][]
SIZE int
1
*


GameplayScreen
currentTeamLabel JLabel
topPanel JPanel
turnHistoryView JPanel
bottomPanel JPanel
boardView BoardView
capturedPiecesView JPanel
game GameManager
stateController StateController
gameManager
game
GameManager
board Board
currentTeamChangeManager PropertyChangeSupport
winner Team
DEFAULT_STARTING_TEAM Team
winnerChangeManager PropertyChangeSupport
boardChangeManager PropertyChangeSupport
currentTeam Team
StateController
gameManager GameManager
previousScreen Screen
screenChangeManager PropertyChangeSupport
screen Screen
«create»
stateController
WelcomeScreen
stateController StateController
SettingsScreen
stateController StateController
1

Starting a Game



Moving a Piece



V. Test Plan

To ensure things worked properly, we implemented many unit tests. Below is one example of our tests running, and more examples are available in Appendix A. To summarize, the tests checked for major features and if they were working. For example, the board tests made sure that the game's backend logic functioned properly. These tests allowed us to feel reasonably confident in our project's progress. From the unit tests, we were able to ensure that the following requirements were fulfilled:

> Board Layout, Turns, Eliminating a Piece from the Board, Winning Mechanism for Attacker, Winning Mechanism for Defender, Center Tile Functionality, Welcome Screen, Gameplay Screen, After Game Screen, Rules Screen, Settings (Almost all)

Additionally, other features were tested through user testing. Once we had most of the project implemented, we played through the game many times, making sure to test different situations. We checked that themes and player customization worked, and that rules were behaving as expected. Through this we were able to catch a number of visual bugs, which were fixed and then tested once again. Overall, our testing methods allowed us to get a fairly feature complete version of the project.

VI. Project Implementation Description

https://github.com/GU-2024-Spring-CPSC224/final-team-project-the-mark-of-the-chili-onions

Every system proposed in the original design document got implemented, along with numerous extras and QoL changes.

Following are screenshots of the programs main screens in action.

Fig 1. Welcome Screen (Breezy Blues Theme)

Fig 2. Game Screen (Breezy Blues Theme, Focus mode disabled, Highlighting mode enable)

Fig 3. Game Screen (imaducklol Theme, Focus mode enabled, highlighting mode enabled)

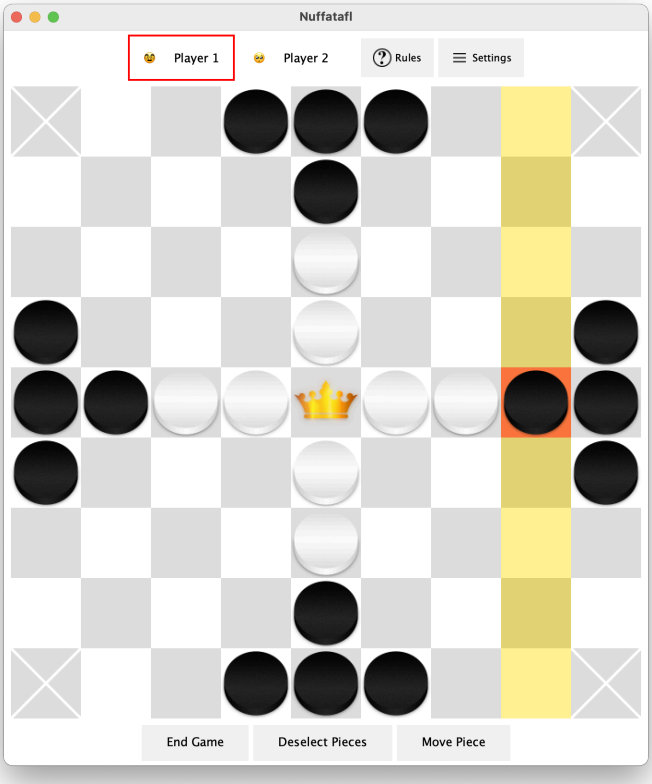


Fig 4. Settings Screen (imaducklol Theme)

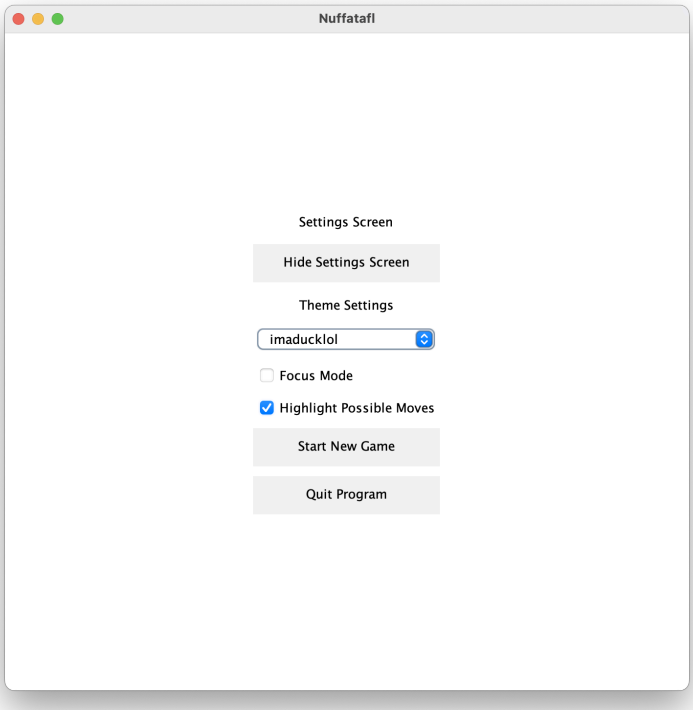
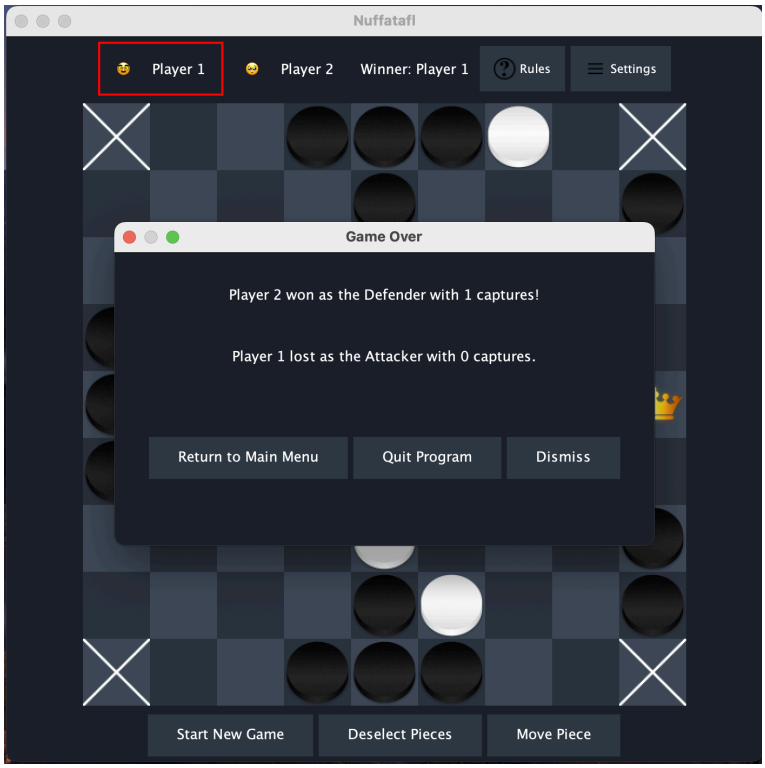


Fig 5. Game Over Dialog (Midnight Theme)



We heavily used GitHub Pull Requests along with GitHub Issues throughout the development of Nuffatafl. These services made cooperation simple and straightforward, even with three people programming in parallel. In addition to GitHub services, we used Discord for general communication, in person planning, and other discourse.

Fig 6. A pull request in action

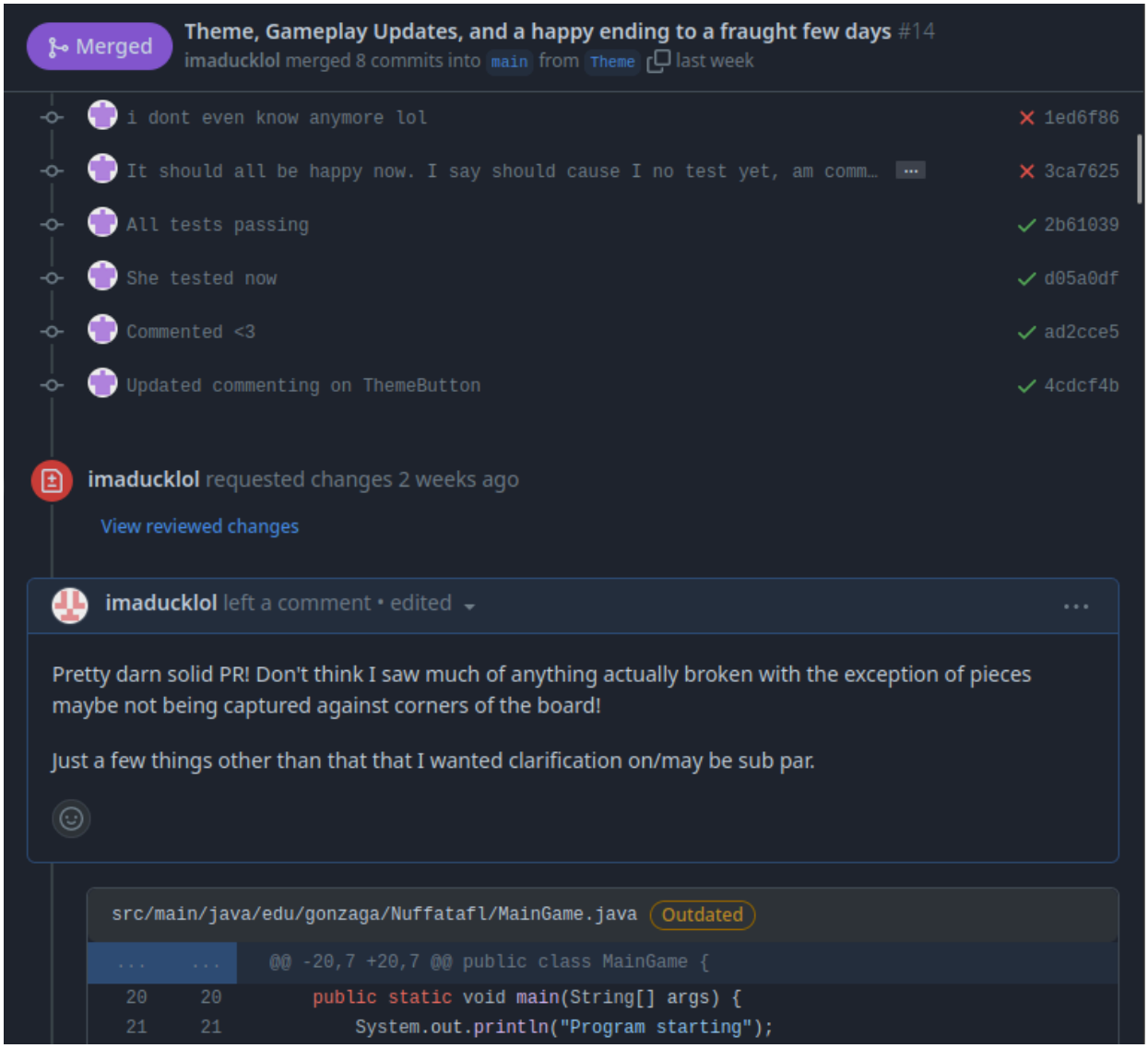


Fig 7. A screenshot of our GitHub issues page

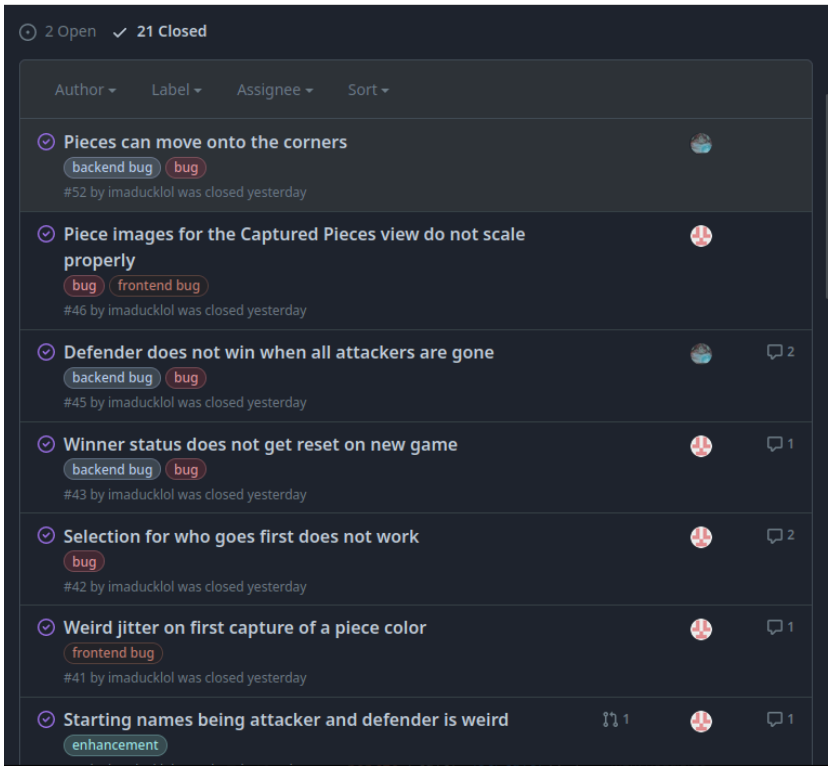


Fig 8. Pain in action, as documented by Discord

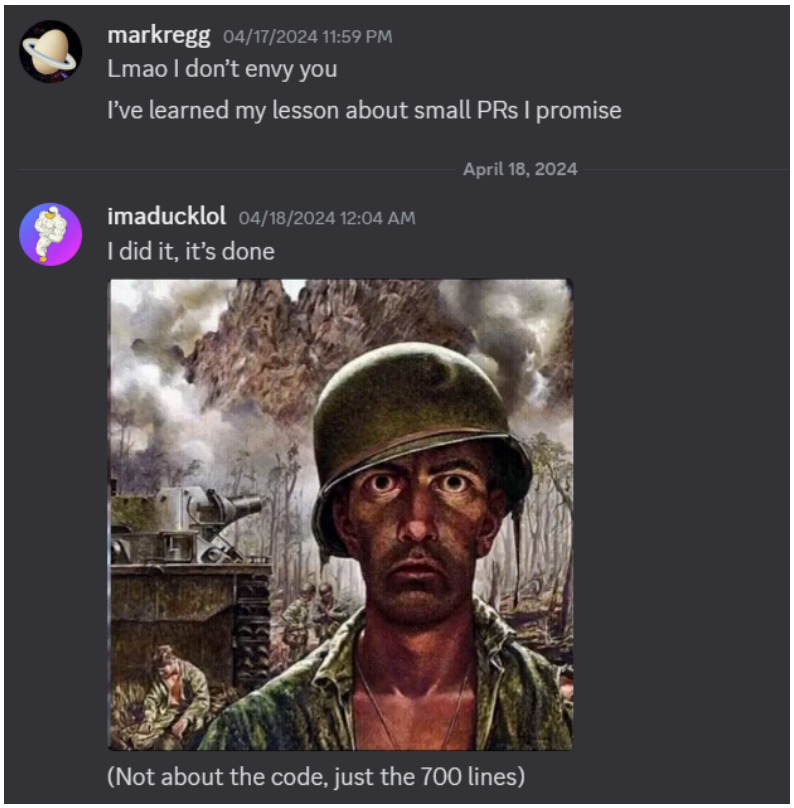


Fig 9. Default Board Setup Printed to Console, used for early testing

```
---AAA---
----A----
----D----
A---D---A
AADDKDDAA
A---D---A
----D----
----A----
---AAA---
```

Fig 10. Icon *(iconic)*

VII. Future Work

In the future, there are numerous non-functional features that would be beneficial to the program.

- Refined User Interface and User Experience
- A player v. computer mode
- Networked play
- Refactoring of backend for easier future work
- Lootboxes for new themes

Additionally, we would like to look into making the project open source. Before doing so, we would likely want to clean up the source code and make sure everything is documented as much as possible.

VIII. Glossary

Nuffatafl - Our version of Tablut, a Tafl family game

OOP - Object Oriented Programming

MVC - Model View Control Architecture

Java Swing - Built in GUI Library for Java

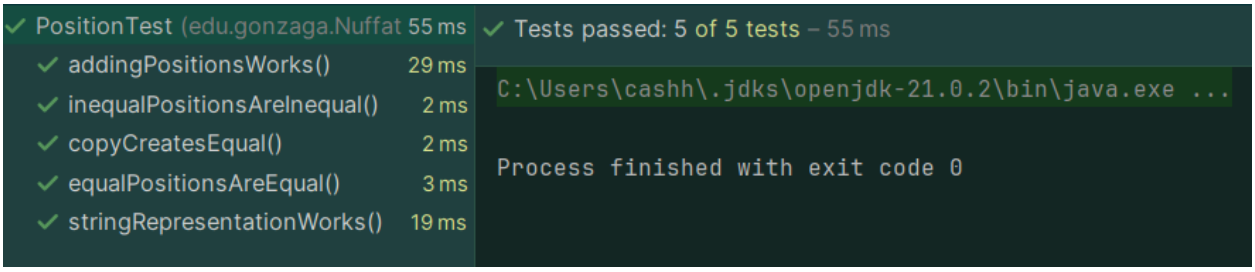
QoL - Quality of Life

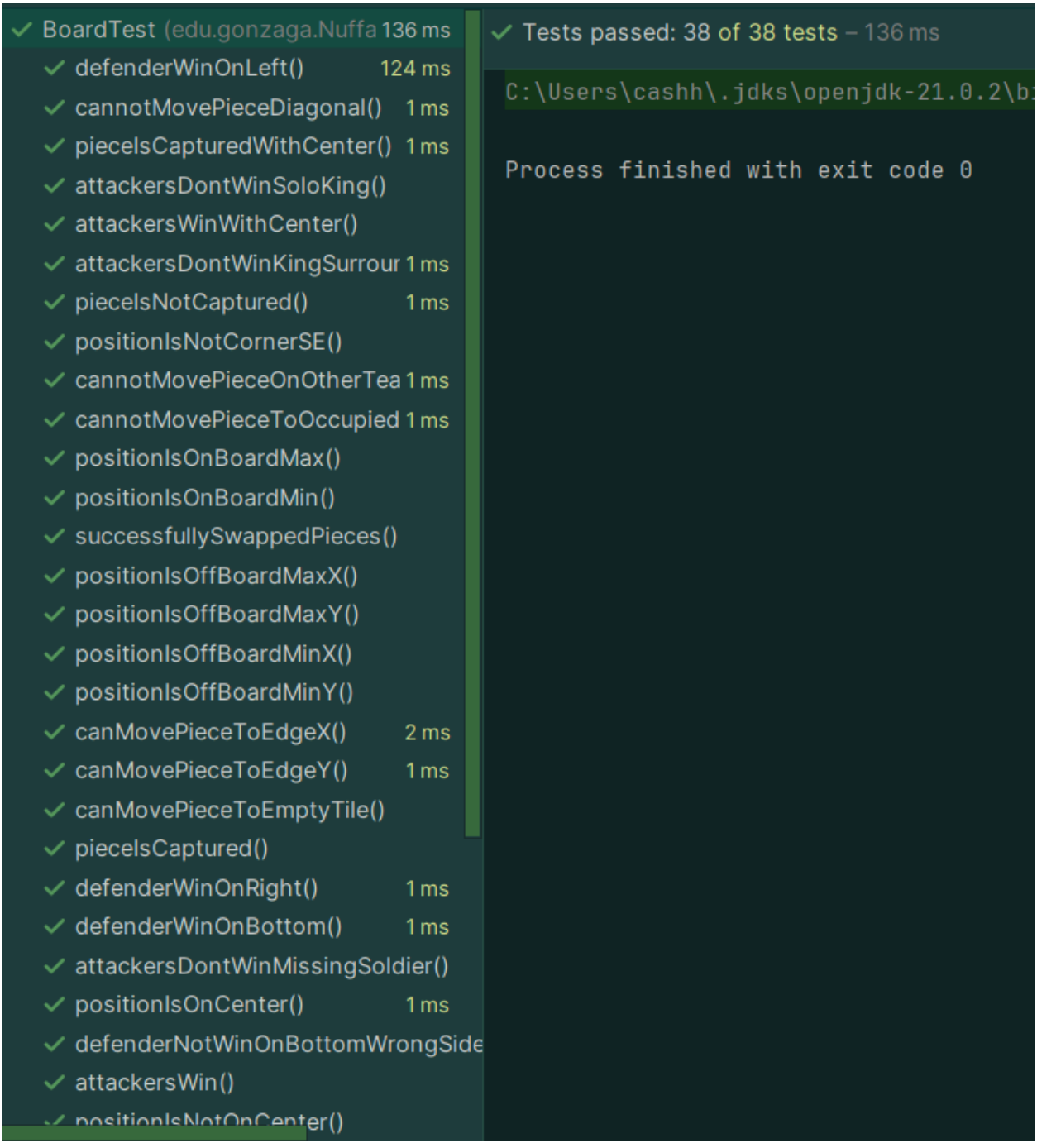
IX. References

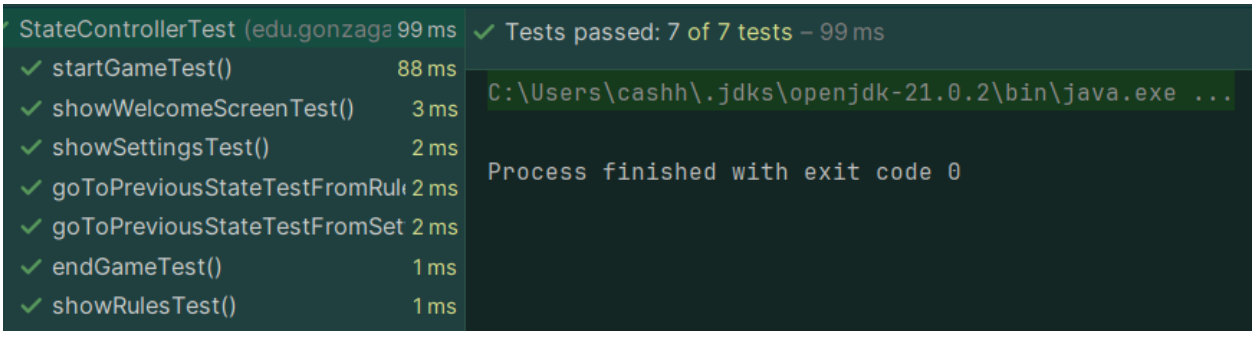
[1] F. Rachunek, “BrainKing - Game rules (Tablut),” brainking.com. Available: https://brainking.com/en/GameRules?tp=19. [Accessed: May 05, 2024]

X. Appendices

Appendix A - Unit Tests




BoardTest (edu.gonzaga.Nuffa 136 ms
defenderWinOnLeft() 124 ms
cannotMovePieceDiagonal() 1 ms
pieceIsCapturedWithCenter() 1 ms
attackersDontWinSoloKing()
attackersWinWithCenter()
attackersDontWinKingSurrour 1 ms
pieceIsNotCaptured() 1 ms
positionIsNotCornerSE()
cannotMovePieceOnOtherTea 1 ms
cannotMovePieceToOccupied 1 ms
positionIsOnBoardMax()
positionIsOnBoardMin()
successfullySwappedPieces()
positionIsOffBoardMaxX()
positionIsOffBoardMaxY()
positionIsOffBoardMinX()
positionIsOffBoardMinY()
canMovePieceToEdgeX() 2 ms
canMovePieceToEdgeY() 1 ms
canMovePieceToEmptyTile()
pieceIsCaptured()
defenderWinOnRight() 1 ms
defenderWinOnBottom() 1 ms
attackersDontWinMissingSoldier()
positionIsOnCenter() 1 ms
defenderNotWinOnBottomWrongSide
attackersWin()
Tests passed: 38 of 38 tests – 136 ms
C:\Users\cashh\.jdks\openjdk-21.0.2\b
Process finished with exit code 0


StateControllerTest (edu.gonzaga 99 ms
startGameTest() 88 ms
showWelcomeScreenTest() 3 ms
showSettingsTest() 2 ms
goToPreviousStateTestFromRul 2 ms
goToPreviousStateTestFromSet 2 ms
endGameTest() 1 ms
showRulesTest() 1 ms
Tests passed: 7 of 7 tests – 99 ms
C:\Users\cashh\.jdks\openjdk-21.0.2\bin\java.exe ...
Process finished with exit code 0