FIT3077 Software Engineering: Architecture & Design

Nine Men's Morris - Sprint 1

Group: CL_Monday6pm_Team38
Team Members: Rebekah Fullard, Nikola Mutic, Joshua Van Der Veen

Table of contents

Team Information

Team Name

HelloWorld development team

Team Photo

Team Membership

- *Josh Van Der Veen*
    - Email: jvan0048@student.monash.edu
    - Technical and professional strengths:
        - Experience in many programming languages, including Object Oriented
        - Good communication and collaboration skills
    - Fun Fact: I lived in Sweden for 18 months

- *Nik Mutic*
    - Email: nmut0005@student.monash.edu
    - Technical and professional strengths:

        - Communication & collaboration skills
        - Multi lingual (python/c/c++/java)
        - Fast learner
    - Fun fact: can solve a rubik's cube in under a minute

- *Rebekah Fullard*
    - Email: rful0003@student.monash.edu
    - Technical and professional strengths:
        - Good organisational skills
        - Experience in C++, Java & Python
        - Good writing and communication skills
    - Fun fact: Doing a minor in Archaeology and Ancient History

Team Schedule

- Weekly in-person meetings to assess progress and discuss goals (Tuesdays 2pm-4pm).
    - Additional meetings as needed near due dates.
- Frequent communication via a group chat to check in between meetings, and keep in contact, discussing collaboration and distributing and organising the workload.
- To distribute the workload we have determined that after each meeting we will delegate tasks to each team member so that everyone is doing their part. Each team member will also check in and add value to the other team member's work between meetings so that all team members are working collaboratively on all aspects of the project and are able to check each other's work.

Technology Stack and Justification

Chosen Stack

- We are planning to implement this project as a Java terminal application with the possibility to add a UI overtop if time permits.
- Justification: We have chosen Java because it is a popular object oriented language that each member of the team has prior experience with. Everybody in the team has completed FIT2099 (Object Oriented Programming) - a subject in which object oriented programming is taught using Java - so we are familiar with implementing object oriented solutions with this language.
- We have chosen to implement a UI in the terminal because the requirements of this project focus primarily on the back-end design. By paring back the UI we hope to be able to have more opportunity to focus on creating a more robust application.
    - If time allows, we are hoping to implement a more robust UI. The framework in mind for that is Java Swing which is included in the JDK, and has a support for drag drop implementation within Intellij
- Is a widely supported language, well documented with good support.

- Has a variety of libraries and frameworks that can be utilised in order to complete the project effectively if needed

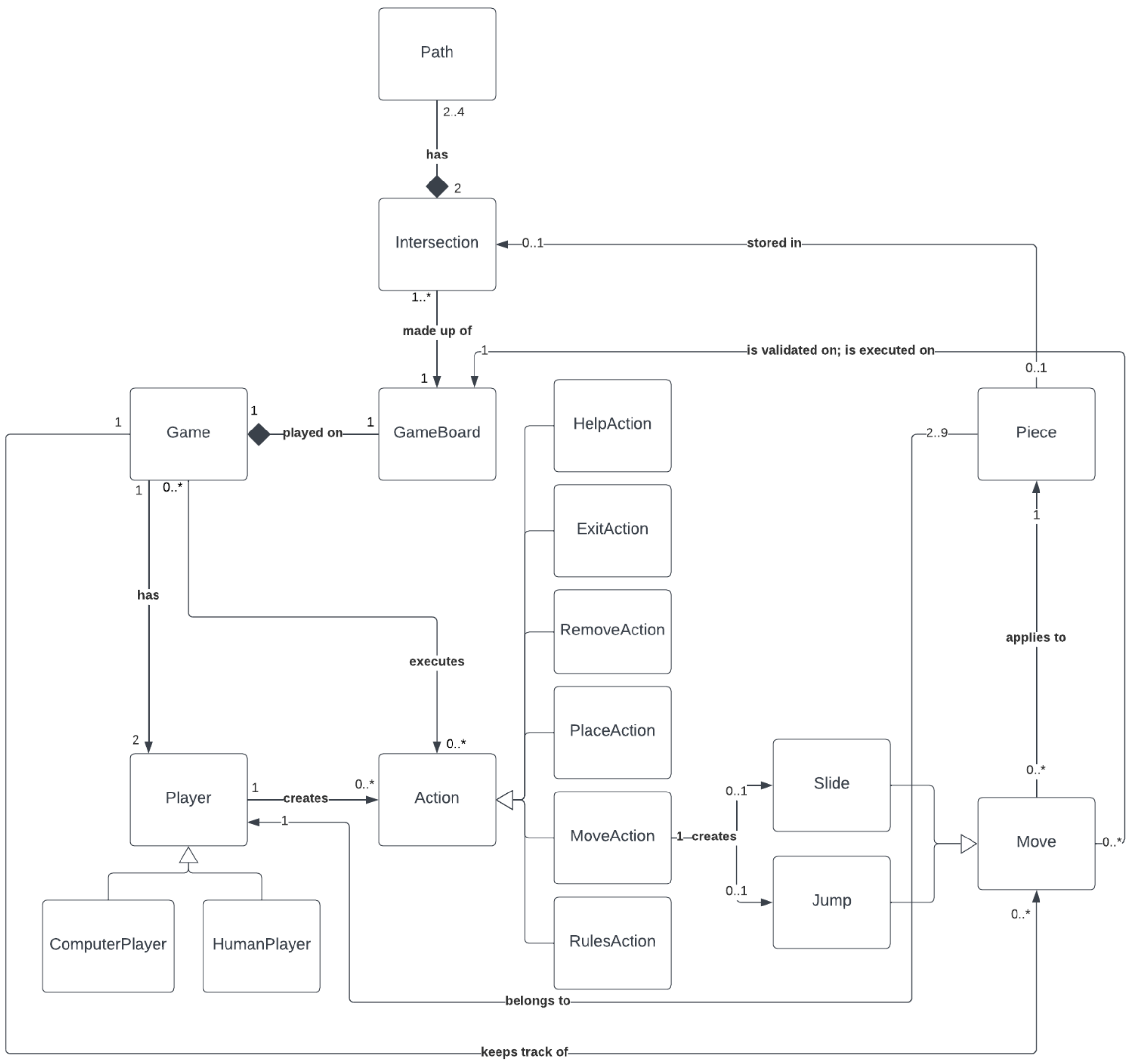
Discarded Alternatives

- Python - We are less comfortable as a group in Object Oriented python. Our general collective experience has been using python for algorithms/functional programming instead of object oriented, which is required for this assignment.
- C++ - This language is a lot more complicated than other Object Oriented languages, and we don't need any of the extra features it provides (eg. memory allocation and efficiency). We also don't want to have to manage memory ourselves :)

User Stories

- As a player who hasn't yet placed all of my pieces, I want to place a piece on the board, so that I can continue the setup phase of the game.
- As a player who has placed all of my pieces on the board, I want to slide a piece to an adjacent intersection, so that I can form a mill.
- As a player who has just formed a mill, I want to remove one of my opponents pieces from the board, so that I can gain a tactical advantage over them.
- As a player who has only three pieces left, I want to jump one of my pieces to any empty intersection, so that I can prevent my opponent from forming a mill.
- As a player about to start a game of nine men's morris, I want to be randomly allocated as either the player taking the first turn or taking the second turn, so that I can begin playing the game.
- As a player who hasn't yet placed all of my pieces, I want to know how many pieces I have left to place, so that I can strategise the best locations for those pieces.
- As a player who has lost a piece, I want to see how many pieces I have lost, so that I have visibility over the game.
- As a player who is playing the game, I want to have the ability to play a local 2 player game, so that I can play with a friend.
- As a player who is playing the game, I want to have the ability to play a computer controlled player, so that I can play by myself.
- As a player who has finished playing a game, I want the option to play again, so that I can keep playing.
- As a user who is unsure how to use the game, I want to be able to access a rules page, so that I can read the rules.
- As a user who is unsure how to control the game, I want to be able to access a help menu, so that I can familiarise myself with the game controls.
- As the game board, I want to alert players if they attempt to make an illegal move, so that I can enforce the rules of the game.
- As the computer player, I want to place or move a piece to a valid position on the board, so that I can create a challenge for the human player.

Basic Architecture



Rationales

Game Engine

We decided that we needed an engine to control the turn logic, updating states, deciding available interactions, etc. as well as implementing all the base classes. This will be created in a separate package to hold the parts of the game that are strictly closed for changes, as the rest of the game is depending on it (Open Closed Principle). In theory, the game engine will contain all necessary base classes to build any turn and board based game.

Game

Game is the class that keeps track of the game, and executes the turns. It handles the setup of the game, the turn by turn execution, the end logic, and any menus or displays that are in the game. The game will keep track of the game board, pieces (and their relationships) and players.

Associations

- It is associated with the **Player** as the game will hold the active players, and will execute turns by iterating through them. The multiplicity shows that there must be 2 players to be able to play a game.
- It is associated with **Move** because the game will be able to keep track of moves, either for undo actions or checking whose turn it is if needed.
- It also has an instance of the GameBoard class to execute the move on so that the people playing the game can see what the move resulted in
- The association with

Game Board
The decision was made that as part of the Single Responsibility Principle design, it would manage the available moves instead of the game engine controlling that logic.
The gameboard also needs to keep track of the pieces available to place, pieces on the board, and pieces that have been taken. This is especially important since one of the

Associations

- The **GameBoard** is associated with **Intersection** because the decision was made to make the game board with a node/edge structure instead of something like a nested array

Move
Move was created for successfully validated and executed Actions. They're kept in the game to keep track of which player has moved, and who's turn it is.

- **Slide:** If they have less than three pieces, they will be able to move the piece anywhere that there is a free intersection.
- **Jump:** If they have more than three pieces they will only be able to move their piece to an adjacent free intersection.

Associations

- It is related to **GameBoard**, because of two processes: 1, the process of executing a Move, the move **Action** will need to be validated before it's confirmed. 2, the **GameBoard** needs to be updated with the successful **Move** and the new state needs to be stored.
- It is related to **Piece** because a move must contain a specific **Piece**

Player
The player class was created as an abstract class as there will need to be shared logic between the ComputerPlayer and the HumanPlayer.
Associations

- **Player** is associated with **Action** as it is how the player interacts with the game, so that they can make a move or use the different menu options

Action
Action is an abstract class that defines the structure of an action. It is abstract because it doesn't get directly interacted with, instead the child actions that extend the class get used since they'll have the extra functionality implemented.

Associations

- Has subclasses for the different types of actions

Help Action
This is a type of action that allows the player to access a help menu which tells them how to control the game on their turn.

Exit Action
This type of action allows a player to exit the game at any time.

Remove Action
This type of action allows a player to choose which of their opponents pieces to remove when they have formed a Mill.

Move Action
This type of action allows a player to select a piece that they want to move, and then to choose the intersection that they want to move it to.

Place Action
This type of action allows a player to choose where they wish to place a piece.

Rules Action
This type of action allows the player to view the rules of the game.

Intersection
Intersections represent the areas that pieces can be placed at on the game board.

Associations

- **Intersection** has a composition relationship with **Path**. If one of the intersections that the path is connected to ceases to exist, then the path itself will cease to exist. The concept for this relationship is the same as that between a Node and an Edge. So an Intersection can have 2..4 Paths because on the game board each valid intersection for a piece has at least 2 and at most 4 paths attached to it for pieces to move along.

Path
Edges represent the valid paths between the intersections on the game board that pieces can move along when a player is sliding their pieces (when they have more than 3 pieces left in play).

Piece
A piece is an object that the player moves to play the game. The **Piece** keeps track of the **Player** that it belongs to. Each Piece belongs to a player, and each player can have 2-9 pieces.

Associations:

- **Piece** is related to the **Player** as it needs to store the player it belongs to, to keep track of the players pieces in the game

Design choices

- The design choice to create the board as a node graph was made to help reduce the responsibilities of the engine, but also to create a better system for working out where different pieces could go.
    - The graph implementation will allow us to extend the game logic no matter the board size/dimensions.
    - Nodes will store information

Assumptions

- As we are not confident in Java UI designing and implementation, we are assuming that the domain model shouldn't need to be adjusted if we implement the UI, and it should fit seamlessly on top of it.

Discarded Alternatives

- We were thinking about using a nested array in the gameboard to store the intersections that the piece could be stored on, but we decided that the **GameBoard** shouldn't be responsible for the logic of allowable actions, so instead we created the Node/Edge model where the **Intersections** and **Paths** now store which **Intersections** are connected, and the move validity checking is handed over to the **Intersections** instead of the **GameBoard**. Nested arrays also do not offer many opportunities or comply to Object Oriented Practices.
- Initially we had thought of an implementation where the player stores its pieces. However, this does not comply to the Single responsibility principle.

Basic UI Design

List of screens:

- Placing a token
- No valid moves (Win condition)
- Moving a token
- Initial board (no pieces placed)
- Jumping a token
- Forming a mill (Win condition)
- Choosing your opponent (human or computer)
- Allocation of turn order
- Entering player names
- Rules screen
- Game controls screen