Sprint 2 - Specifications

Euan Lim, Zoe Tay Shiao Shuen and Lim Zheng Haur

Faculty of Information Technology, Monash University

FIT3077: Software Engineering Architecture & Design
Dr Jean-Guy Schneider

29 April 2023

Table of Contents

Team Schedule

Meeting	Start Date	Content
1	24/04	● Review of Sprint 1 Feedback ● Task allocation of Sprint 2 ● Discussion of software architecture
2	26/04	● Review base source files ● Class diagram discussion ● Task allocation to be completed before next meeting
3	27/04	● UML class diagram review ● Front-end deliverables review ● Task allocation to be completed before next meeting
4	28/04	● Back-end deliverables review
5	29/04	● Integration of front and back-end deliverables ● Final review of all components

Architecture:

Decision on Model-View-Controller

Model-View-Controller is a software architecture where the software is separated into 3 distinct components which are the Model, View and Controller. Each component has its own responsibilities such as:

- Model is responsible for being the central component handling the logic and data such as computing the possible actions by each player or the win/lose conditions are met during gameplay.
- View responsible for presenting the data from the Model to the user such as displaying the positions of each piece and the piece counts of each player on the screen, while allowing interaction from the end user
- Controller being able to handle user input and manipulates the Model accordingly such as passing the inputs from the view into the backend model to move position of pieces.

We believe this best suits a project with interaction from a user to a board, and for the moves to then be checked in the backend, and the board updated depending on the checks.
Below is a diagram of how MVC architecture functions.

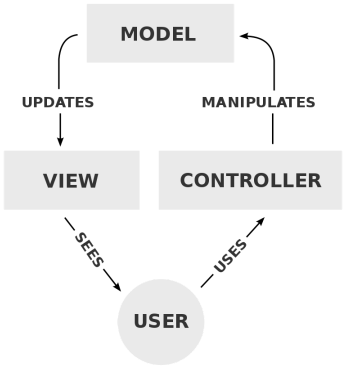


Diagram of interactions within one possible take on the MVC pattern. Retrieved from Wikipedia 2022.

We have collectively decided to implement our software using the Model-View-Controller architecture as we believe that it is the most suitable for our constraints and familiarity. In the context of our design rationale, we will use React to implement the user interface making requests to the backend involving Java Spring Boot, segmented into controller and model

handling the logic and data. Furthermore, breaking the architecture into 3 independent components allows our team to break down the tasks that improves our collaboration and make a more cohesive working environment for us. In addition, some other benefits that Model-View-Controller architecture brings us is the separation of concerns that reduces the complexity of the software and makes it easier for us to extend or modify different components if required.

In terms of the codebase the View is mainly responsible for the interfaces and prompts that users can interact with, which is sent to the controller that routes to the necessary backend logic handling move logic, turn logic etc, and then sent back to the controller ending with on the View, displaying to users their alterations.

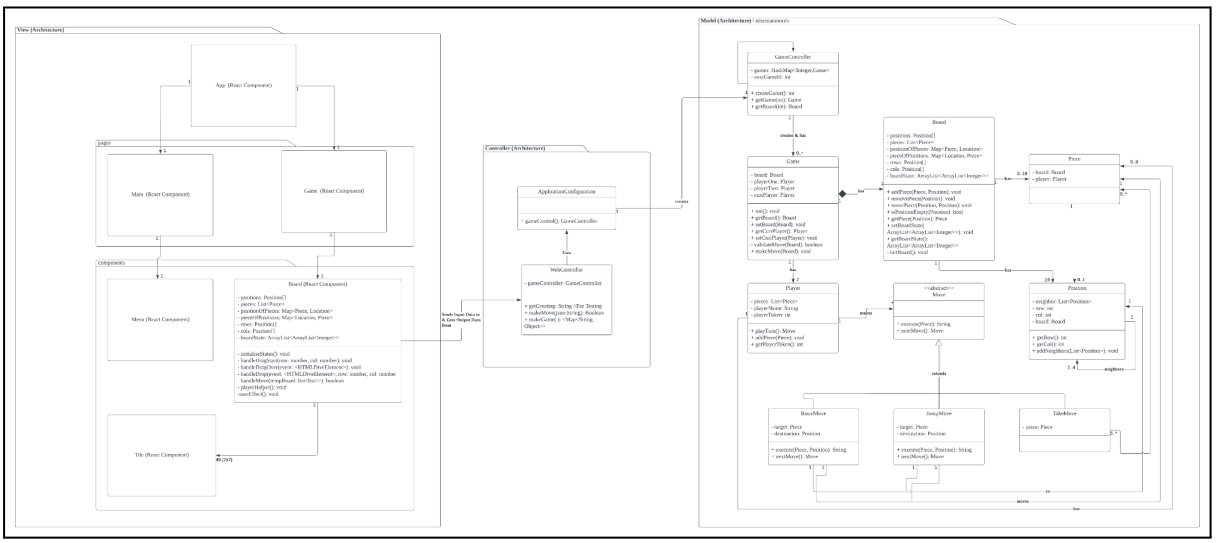
Alternatives to Model-View-Controller Architecture

Our team has compared multiple architectures prior to making our collective decision to Model-View-Controller architecture. Firstly is the monolithic architecture which is a practice of implementing softwares by tightly coupling them in a single codebase. Although this is a viable option as it makes development much quicker and simpler, there is a drawback where the software could not be easily extended. This causes the project to be not scalable and less flexible. In the long run, we may end up with longer development cycles (Harris, 2023). We also determined as possible additional features would be required to be added, a monolithic would not be the best decision.

As we are working as a team, and as independent coders at times, MVC offers a much better separation of concerns, lowering the barrier for initially getting started when working on the code base. Again this comes back to concerns of having a monolithic architecture being more difficult as understanding every component intimately before working on further implementations takes up alot of time, of which we are limited; despite the low start time, we believe the long term effects of maintaining the codebase outweighs the benefits of the initial ease in creation regarding following the monolithic architecture.

Additionally, there is also the microservices architecture where softwares is broken down into small independent services. This is an architecture that supports greater scalability. However, it comes with a drawback of being more complex when developing and testing which our team presume to be unnecessary and adding to the potential complexity. Hence, our group has decided that Model-View-Controller architecture is the most suitable architecture to be implemented in this 9 Man Morris game.

Class Diagram:



Design Rationales

1. Cardinalities

a. Why is the cardinality from BasicMove class and JumpMove class to Position class 1..1 rather than 1..2?

The cardinality from BasicMove class and JumpMove class to Position class 1..1 is because a BasicMove and JumpMove has two attributes of 1 Piece instance and 1 Position instance. It moves the associated Piece to the instance of the associated Position. The reason it does not require 2 distinct Position instances for both its original position and new position is because the Board stores the initial position of the associated Piece. Hence, including the “from” Position instance is redundant and unnecessary.

b. Why is the cardinality

2. Relationships

a. Why is there no relationship between TakeMove class and Position class?

There is only a relationship from TakeMove class to Piece class instead because the position of the Piece is stored within a HashMap in the Board instance which is also an attribute of the Piece. Hence, we could access the Position of the Piece through the Board attribute rather than needing a relationship between the Position class and TakeMove class.

3. Debate on Classes

a. Player

One of the classes we had to really evaluate and discuss on whether we needed it was the Player class. The purpose of creating a separate class for Player is firstly for extensibility; for instance, future implementations might involve storing history of past moves, etc. However, it could also have not been included as currently it is mainly used to store simple data and there would definitely be a way to implement the game without a specific Player class. Despite that, the creation of a Player class is a better decision due to its creating more structure as well as the application of the Single Responsibility Principle (SRP). We additionally decided on Player class as storing **a past move set** could be done so via the Player class. This would extend to the capability of rewinding and undoing past moves. Additionally, without the inclusion of a Player class, there would exist potentially floating magic values that might require an enum to maintain.

b. Game

The reason Game class was also a debate was due to being able to use the Board class to handle most of the responsibilities such as running the main game loop, switching between turns and validating moves along to the rules of the game. However, since multiple components would be used together and the alternative was that the board can only be used to store the grid and token

positions, making a Game class to handle the bringing of multiple components together to enforce the rules of the game seemed a more organised way to distribute the roles between the classes.

4. **Modelling of Grid/Board**

We went through multiple potential options of how to model the game board; mainly deciding whether to use a matrix with digits to represent states of each position on the grid, or having a grid full of Location class objects to represent the positions of the tokens. A matrix would be easier to implement, as well as more simple to work with and update as the game progresses. However, a Location class modelling the positions on the grid would be more robust. For the simplicity of the implementation for the current sprint submission, we are going with the matrix. For future implementations however, we are planning to maintain the matrix form of storage but populating it not with digits but Location objects instead as that provides more extensibility and fluidity in data storage. As the location objects themselves would store the values and offer the ability for further extensibility.

5. **Inheritance**

.

6. **Additional Rationale**

- Decision on having a Move class and additionals

- For the Move class, the alternative was that the role of making moves could be put into Game which essentially manages the flow of the entire Game, or even Board since it deals with the state of the Board, but due to the game rules that there are different move rules for different situations, implementing an abstract Move class was what we decided on. Each type of specific move would be a separate class on its own (for instance BasicMove and JumpMove) that inherits from Move abstract class due to them having different properties but also some similar ones. Currently for the simplicity of this sprint's requirements not including the implementation of game rules, a simple move method was put into the Game class. However when actually implementing the rules in future sprints we would be using Move class and its subclasses. That would be an improvement because overridden methods can reuse code

and implement different properties of similar components in a way that is organised and encapsulated to their own respective classes.

-

- Decision on including a Game Controller Class

-The game would still be playable without a Game Controller Class, however we felt that the inclusion of a Game Controller class would enable for managing of multiple instances of games concurrently, make method calls towards features easier, and after the decision on MVC pattern, it became evident that the GameController class was necessary over the potential decision to exclude. Currently, it carries a hashmap populated with game id's as keys and game instances as values, allowing for multiple games concurrently to be accessed and played.

- Decision on Singleton for our Game Controller Class

- Alongside the addition of the GameController Class, came the design decision of a singleton / factory method for interacting with and creating the Game Controller. This fit perfectly with the use case, as only a single GameController is required, and should be used to manage all of the Game Instances.

References

1) Wikipedia, 2022. *Model-View-Controller*. Wikipedia. Retrieved from https://en.wikipedia.org /wiki/Model%E2%80%93view%E2%80%93controller
2) **CHANDLER HARRIS, Atlassion, 2023** Microservices vs. monolithic architecture https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith#:~:text=A%20monolithic%20architecture%20is%20a,monolith%20architecture%20for%20software%20design.