MONASH University

FIT2101 S2 2022

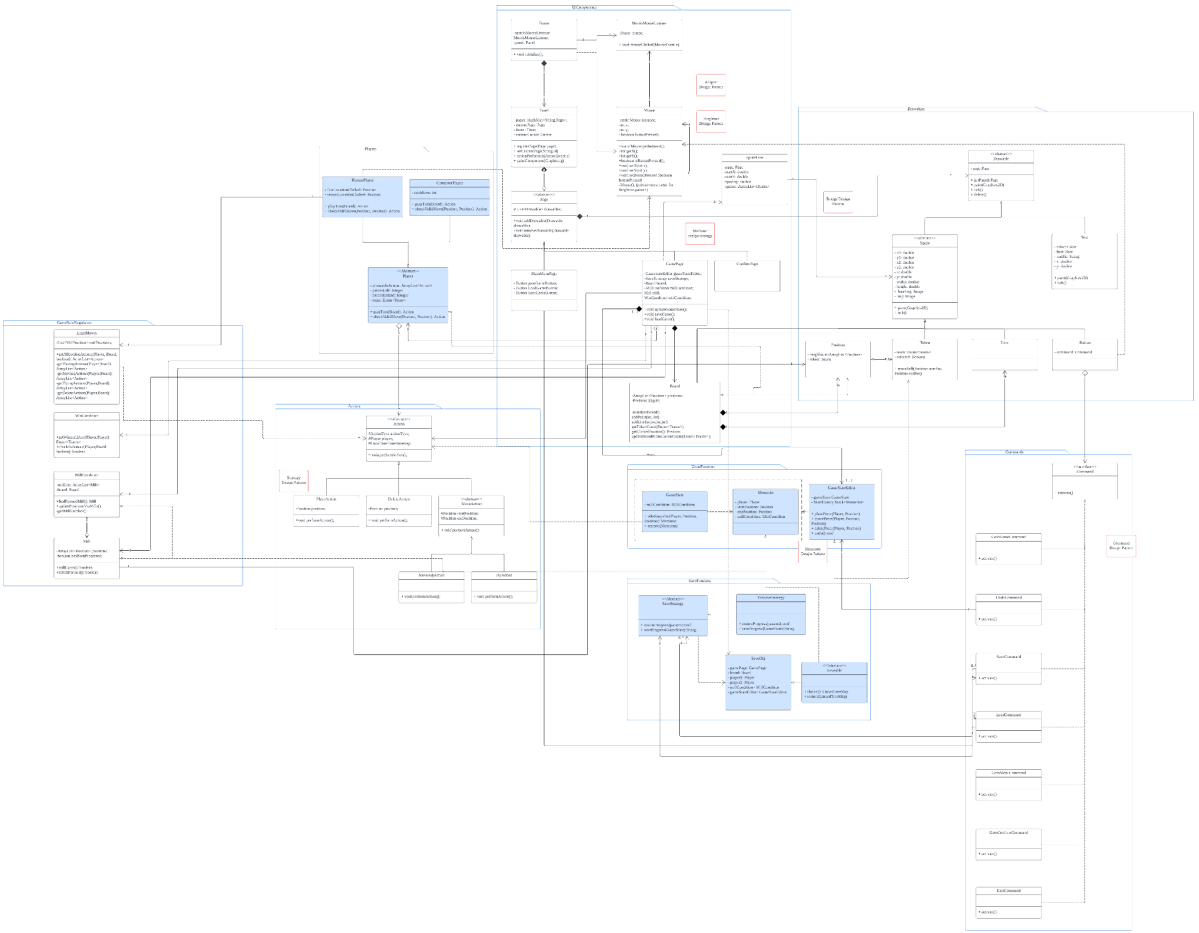
FIT 3077 Assignment (Sprint 4): 9 Men's Morris Game Application

Presented by:

Ong Chien Ming - 31861318
Syed Zubin Hafiz - 3227671
Garret Yong Shern Min - 31862616
Mario Susanto - 31103146

User Stories

Revised Class Diagram



Link:
https://lucid.app/lucidchart/27de8e84-ffac-4296-8a15-8af769bc755a/edit?viewport_loc=-69%2C3481%2C1966%2C944%2Cak~XqipN5nIb&invitationId=inv_0671a230-e65d-416a-b124-591c7fd13cf0

Advanced Requirements Architecture

Undoing Moves

Throughout the development of the 9 Men's Morris game, various approaches to the undo function were considered. One approach was saving the game page every turn. Another approach involved only recording each action during a turn, along with any other information that might be necessary to restore previous game states.

The main advantage of saving the game page every turn is that it is simpler, and is therefore less likely to result in faults. This would improve the feature's readability, therefore also avoiding future errors. However, the main disadvantage of this approach is that an excessive amount of memory is used to store duplicate data each turn. While the application is small enough that this is unlikely to be problematic, it is nonetheless highly inefficient.

Due to the disadvantage of saving the entire game page, we opted to store the specific actions taken each turn instead. This means that the undo feature would be more space efficient and would likely work more quickly. Furthermore, saving specific actions would be easier to implement than saving the entire game page as it would take less attributes into account for each turn. It should be noted however, as not all features, such as mills, can be represented exclusively as an action, it is necessary to take snapshots of some additional classes each turn in order to ensure that the game state is correctly restored. While this would add complexity to the overall implementation, this disadvantage is nonetheless outweighed by the advantages of this approach.

Saving and Loading the Game

There were several considerations that were made when designing the functionality of the save and loading game functions. In order to reduce the time taken to implement the saving and loading of the game, we initially intended to use Gson, a java json library, to automatically convert a class to and from a json file. This has the advantage of theoretically being straightforward to implement, along with being easy to maintain as changes to other classes would not require the modification of the save functionality. Saving and loading would then be done within the TxtSaveStrategy, which inherits the SaveStrategy class so that other formats could be handled in the future.

Unfortunately due to unforeseen problems that arose when we attempted to implement the above structure, we had to instead consider other options as automatic conversions would require the code to be refactored significantly. At this point, we considered either having each class obtain and return the relevant data with the classes, or having setters and getters in all classes for relevant data so that the data can be directly obtained and restored by the SaveStrategy and the TxtSaveStrategy classes.

The main advantage to obtaining the data within each class is that it respects the Open Closed Principle (OCP) by avoiding the need for public access of attributes that need to be saved, allowing for the TxtSaveStrategy to simply call a method to obtain the necessary information. Furthermore, it avoids having the TxtSaveStrategy and SaveStrategy classes becoming bloated and having excessive dependencies.

In contrast, having getters for accessing data that needs to be saved would adhere more to the Single Responsibility Principle (SRP) as SaveStrategy and TxtSaveStrategy, which have the responsibility of handling the loading and saving of games, would handle obtaining the relevant data instead of having various other classes handle it. It is also advantageous because all saving and loading functionalities are located within the same class, meaning that readability of the code increases.

We opted to have each class handle the saving and loading of its own data. This is because we wanted to avoid the risk of data leaks or other problems that could occur from exposing otherwise private data to other classes, which could be detrimental to maintainability. Furthermore, we mitigated the potential of infringing the SRP by adding an interface, Saveable, that all classes with relevant data would implement. We furthermore added another class, SaveObj, that stored an instance of all classes with data that should be saved in order to aid readability, as it provides a view of the classes that need to be saved.

Computer Players

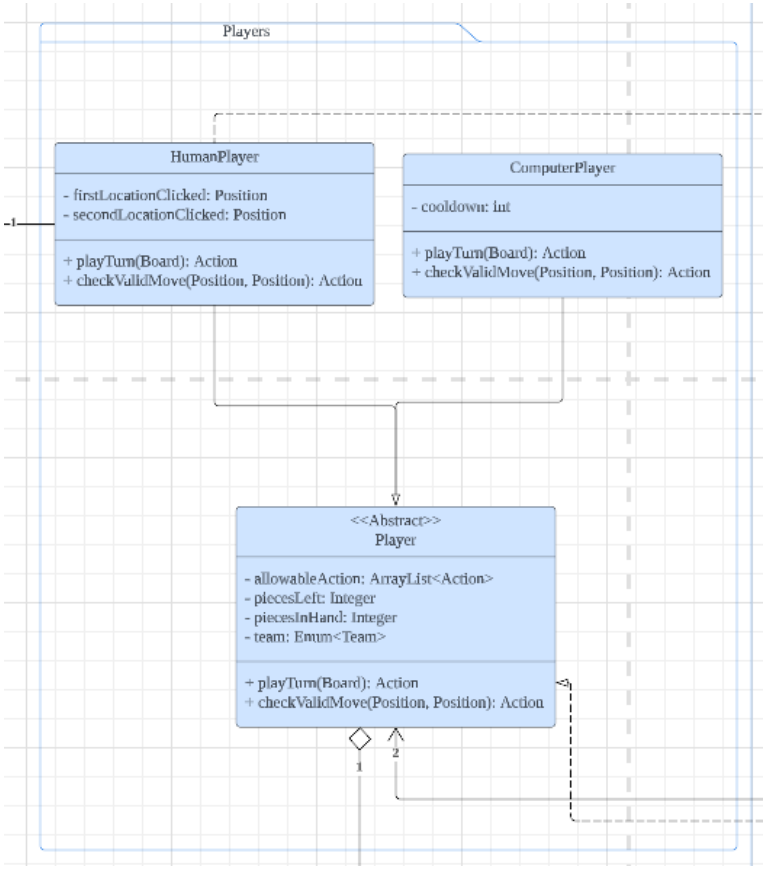
Computer players are the counterpart of human players, it allows players with no companions/ competitors to play Nine Man's Morris. This version of the game includes a very simple AI which was added to the game to make random moves.

The architecture of this additional requirement is quite simple as it builds upon the previously planned HumanPlayer class that extends the Player class. Following good programming practices and object-oriented design, it was established that there would be a Player abstract class that would act as a template for all future player-type classes to implement. Some might argue that there should only be about 2 player-type classes and this would mean that creating the player abstract class would be redundant, excessive or over-engineering. However, the team has landed on this design decision to allow for more player classes; human classes can be final but computer classes can vary in difficulty where each computer class will make moves based on varying algorithms/ heuristics and this would separate them enough from other difficulties of the computer players to justify other classes.

Next, to explain the inner workings of the computer class, there was not much debate about the design of the simple computer player as it was meant to mock a human player's constraints with none of the brains. The implementation of this was through a very simple concept where an abstract player class is developed as an outline for both the HumanPlayer and Computer player. Both human players and computer players follow the same flow whereby when a player class is first instantiated, it instantiates a game rule class which is responsible for determining what legal moves each player is allowed to make. From there, whenever a player is prompted to make a move/ player their turn, the game rule will run through and generate all possible legal moves that that player is allowed to make. When processing the human player this list of legal moves is used as a check to see whether the user's attempted input leads to a legal move. However, for a Computer player, a random number generator will be used to randomly pick a legal move from the list and select it as their intended action.

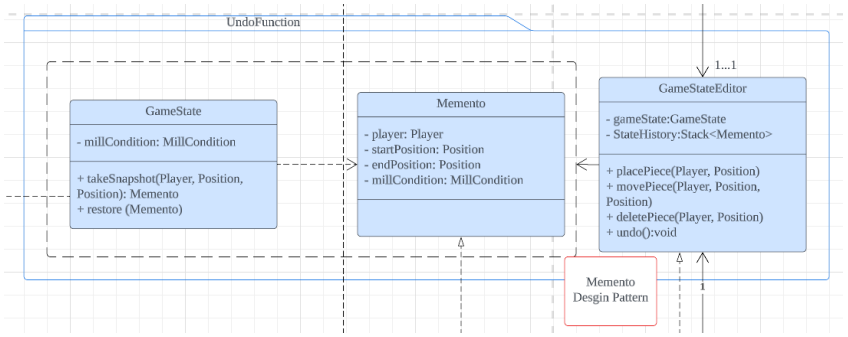
Architecture Revision Rationale

Player, HumanPlayer, and ComputerPlayer



During the process of implementing the ComputerPlayer class, various redundant variables needed to be cut out of the design as we realized that we had variables in the Player class that the ComputerPlayer did not have any use of. As such, the current design only has a cooldown attribute in ComputerPlayer to add a small delay to a ComputerPlayer's turns, along with two variables to track the selected locations for the HumanPlayer, with all shared attributes remaining in Player.

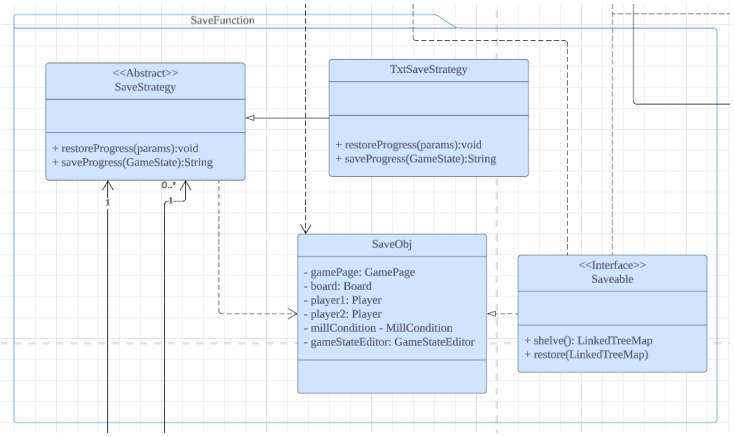
GameStateEditor, GameState, and Memento



The design of the Undo feature has been restructured to support the storage of all necessary attributes for the feature to function well. This is done by adding additional attributes to the Memento class,

along with functions in the GameStateEditor to be able to generate Memento instances that correspond to the actions taken by the players.

SaveStrategy, TxtSaveStrategy, SaveObj, and Saveable



The design for the save feature has been modified extensively with the addition of the Saveable interface and the SaveObj class. The Saveable interface was added to represent all classes that needed to be saved, ensuring that the data for such classes can be stored and restored consistently, thus improving maintainability. The SaveObj class meanwhile was added to more easily keep track of objects that needed to be saved, thus improving the modifiability and readability of the code and allowing the GamePage to delegate some aspects of saving the game to the SaveObj. Finally, there were redundant methods within SaveStrategy and TxtSaveStrategy, which have now been removed.

Difficulty of Advanced Requirement Implementation

Undoing Moves

A litany of problems and complications were encountered throughout the process of attempting to implement the system for undoing moves. This is because while we initially had a general idea of how best to implement the undoing feature, which we intended to keep in mind throughout the development process, we realized that there were implementation details that made implementing the feature more difficult, along with problems caused previous changes to the design in sprint 2 that didn't sufficiently take the undo feature into account.

One problem that arose was that we realized how storing the positions and the team that moved, placed, or deleted the token isn't sufficient. This can be seen by how each player had its own attribute representing the number of tokens in hand, which we needed to update when a place action is undone. This furthermore required us to refactor the structure of actions to some extent by having an Action class take in a GameStateEditor class instead of a GameState class so that we could distinguish between tokens being acted upon by players and tokens that were being reversed, allowing us to better represent how to modify the number of tokens in a player's hand.

The main difficulty of handling the undoing of moves is handling mills. This is largely because we had to substantially change how mills were to be implemented as we realized that our original plan wouldn't have worked, which added complications to undoing moves when mills were to be involved. This meant that we couldn't fully implement a system that stores information without any duplicates, forcing us to instead save the status of all the mills every turn. While this results in inefficiencies, it nonetheless is still more efficient than storing the entire game page each turn.

Overall, while the issues outlined above were eventually resolved, handling the undoing of moves was nonetheless difficult as some parts of the code needed to be refactored to account for the changes.

Saving and Loading the Game

While we initially assumed that saving and loading the game would be straightforward, we eventually realized that there would be more issues with implementing this feature than expected. Our original plan was to automatically convert the GamePage class, which stores all relevant data, into a JSON file and back. However, we found it difficult to find such a library, so we spent a non-negligible quantity of time doing research for relevant java libraries. Furthermore, after finding such a library, Gson, we discovered that it would not be possible to automatically convert the GamePage class into a storable string without significant refactoring. This is because we had plenty of situations where two instances of the same or different class store each other, such as each Drawable storing their associated Page and each Position storing its neighbours. This caused an endless loop that caused the java library to fail. We also discovered that the Gson library was unable to handle some classes from the Java Swing library, meaning that we would have to figure out how to factor those classes out from the GamePage. As we determined that this may not be feasible, we had to rethink our overall design for saving and loading the game.

We instead opted to manually extract the data from each class and to return this data to the TxtSaveStrategy class. We faced some challenges with this approach as we had to manually identify what data we needed to save and how to restore this data after the fact. This was especially problematic for restoring the GameState, as we needed to figure out how to correctly identify the positions and players associated with a particular turn. Nonetheless, we managed to resolve these issues to ensure that this functionality works.

Computer Players

There were little to no issues when it came to completing the computer player thanks to some good practices and the simplicity of the task.

The main thing that made this additional requirement easy was good planning. Putting design principles aside, we were given ample time to think about the integration of the computer player in detail. If we were not given the opportunity to view and select the additional requirements in advance, planning would have been slightly harder as we would have had to rely solely on good design principles. One point of debate was when we were discussing the feasibility of the computer player and we came to the conclusion that it would be easier to scan for legal moves and have the computer

player select from the pool of legal moves. This logic has also been attributed to the design of the human player and the game rule classes.

The second factor that helped play a part in designing the additional feature was good design practice and conformance to the good design principles. Keeping in mind good design principles helped facilitate the conception of the game rules classes which were responsible for the identification of legal moves. Besides that, having implemented an additional player abstract class was very useful in setting up templates for future classes (in this case the computer player).

User Stories

Outside the game
As a user, I want a menu screen so that I can select between playing alone or with someone else
As a user, I want to be returned to the main menu after a game ends so that I can start a new game
As a user, I want a single player mode in the menu so I can play against the computer
As a user, I want the computer to automatically make moves after my turn so that it can act as a proper 2nd player
Starting the game
As a user, I want to start the game with 9 tokens so that I can place them on the board
As a user, I want to be able to undo my actions so that I can undo my mistakes
As a user, I want to form a mill when I align 3 men along a line so that I can retire one of the opponent's men
As a user, I want to only be able to make legal moves so that no one can cheat and the game flow is smooth
As a user, I want the starting player to be randomised so that each player has a fair chance to start first
Moving Phase
As a user, I want to be able to move my pieces to adjacent positions after I place all my men so that I can form additional mills

As a user, I want to be able to move my pieces to any empty position when I have three pieces left so that I have a chance to win despite my opponent's numerical advantage
As a user I want to be able to undo a move, so that I can make a better decision in how I want to play the next move and have a better chance of winning
As a user I want to be able to save the game, so that I can jump right back into my last game which I couldn't finish due to other priorities.
As a user, I want to be able to load from a previous game state to ensure that I can resume playing from where I saved previously because I was not finished with the saved game
Ending Game
As a user, I want the game to end once one player has lesser than 3 men so that I can move on to the next game.
As a user, I want the player with lesser than 3 men to lose so that the game can end and a winner can be declared

Format: As a **[WHO]** I want **[WHAT]**, so **[WHY]**.