FIT3077 - Software engineering: Architecture and design

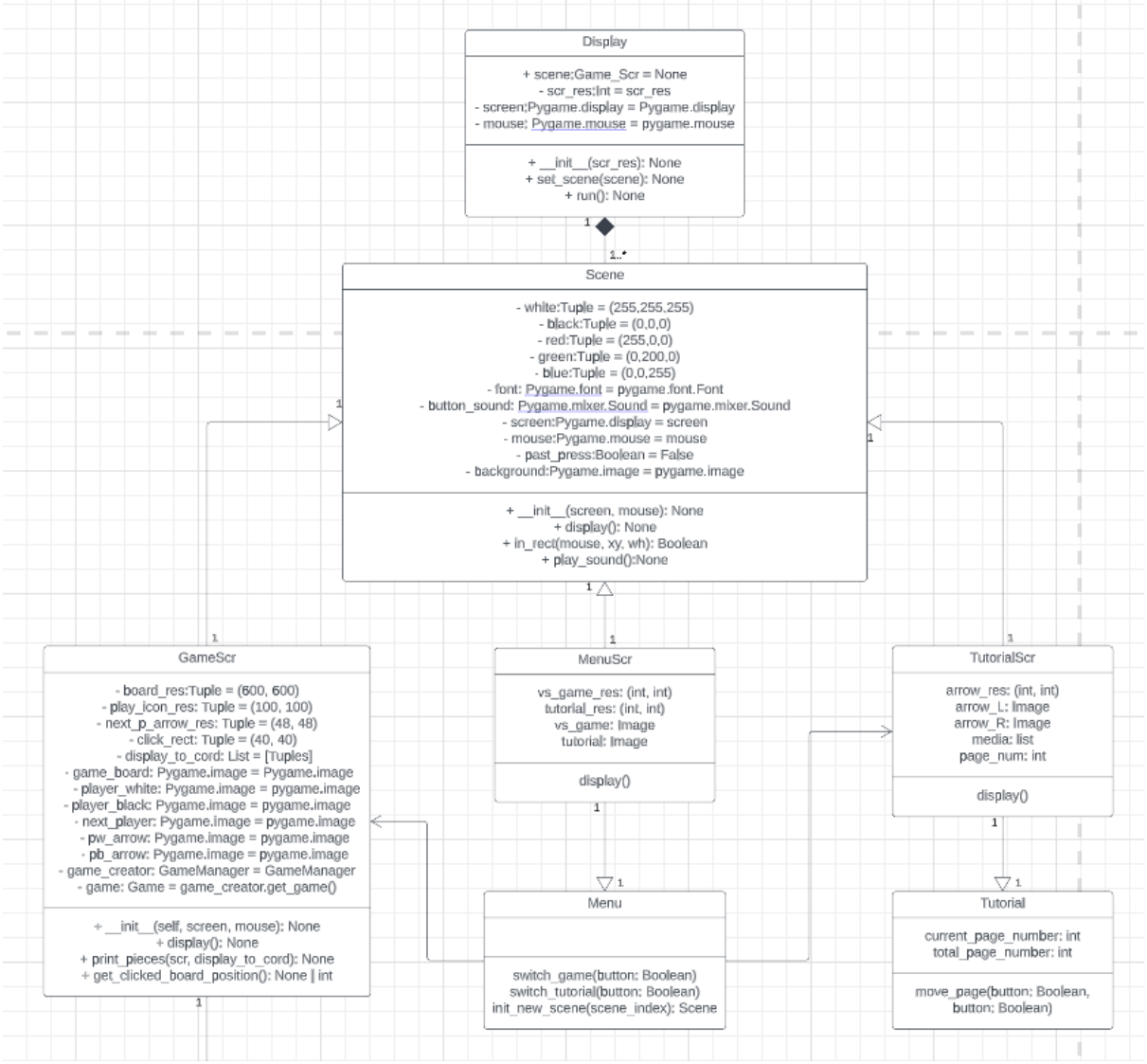
Sprint 2 - Rationale

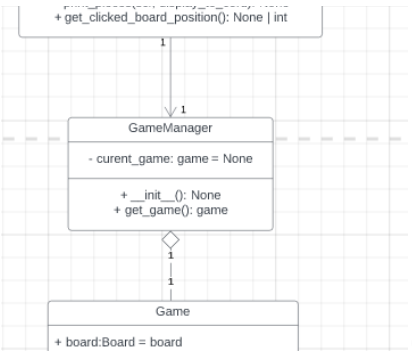
CL_Monday6pm_Team37

Team: HTML is an OOP Language

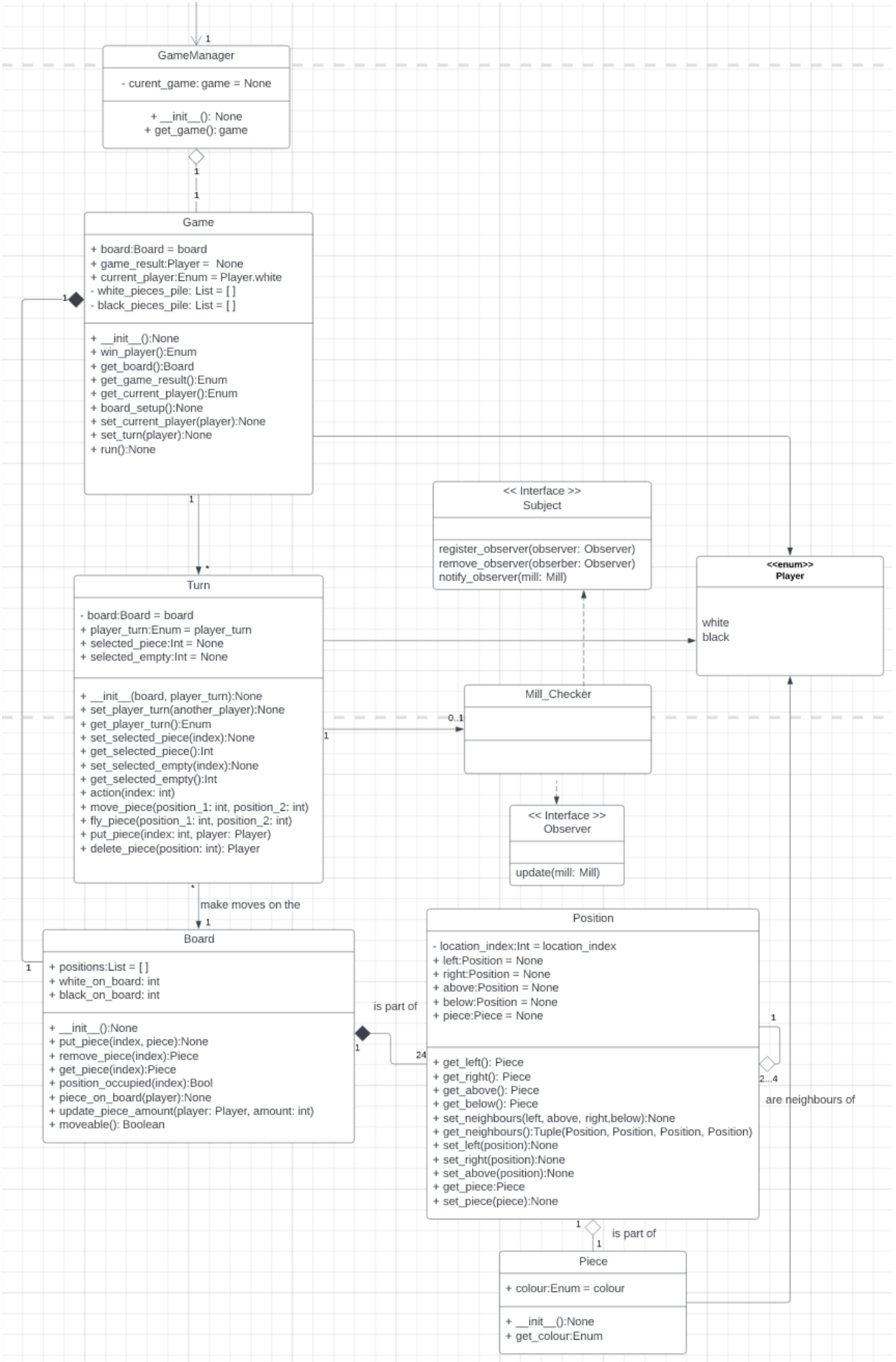
Class Diagram

Our diagram is too big so we separated them into multiple pictures.
If it is hard to read, we suggest go to our lucidchart tab: Sprint 2 to see the diagram.
https://lucid.app/lucidchart/0abd0599-3e18-4457-be1e-996aeab66dd3/edit?shared=true&page=0_0#





(Where it gets cut off)


GameManager
- curent_game: game = None
+ __init__(): None
+ get_game(): game
Game
+ board:Board = board
+ game_result:Player = None
+ current_player:Enum = Player.white
- white_pieces_pile: List = [ ]
- black_pieces_pile: List = [ ]
+ __init__():None
+ win_player():Enum
+ get_board():Board
+ get_game_result():Enum
+ get_current_player():Enum
+ board_setup():None
+ set_current_player(player):None
+ set_turn(player):None
+ run():None
<< Interface >>
Subject
register_observer(observer: Observer)
remove_observer(obserber: Observer)
notify_observer(mill: Mill)
<<enum>>
Player
white
black
Turn
- board:Board = board
+ player_turn:Enum = player_turn
+ selected_piece:Int = None
+ selected_empty:Int = None
+ __init__(board, player_turn):None
+ set_player_turn(another_player):None
+ get_player_turn():Enum
+ set_selected_piece(index):None
+ get_selected_piece():Int
+ set_selected_empty(index):None
+ get_selected_empty():Int
+ action(index: int)
+ move_piece(position_1: int, position_2: int)
+ fly_piece(position_1: int, position_2: int)
+ put_piece(index: int, player: Player)
+ delete_piece(position: int): Player
Mill_Checker
<< Interface >>
Observer
update(mill: Mill)
make moves on the
Board
+ positions:List = [ ]
+ white_on_board: int
+ black_on_board: int
+ __init__():None
+ put_piece(index, piece):None
+ remove_piece(index):Piece
+ get_piece(index):Piece
+ position_occupied(index):Bool
+ piece_on_board(player):None
+ update_piece_amount(player: Player, amount: int)
+ moveable(): Boolean
is part of
Position
- location_index:Int = location_index
+ left:Position = None
+ right:Position = None
+ above:Position = None
+ below:Position = None
+ piece:Piece = None
+ get_left(): Piece
+ get_right(): Piece
+ get_above(): Piece
+ get_below(): Piece
+ set_neighbours(left, above, right,below):None
+ get_neighbours():Tuple(Position, Position, Position, Position)
+ set_left(position):None
+ set_right(position):None
+ set_above(position):None
+ get_piece:Piece
+ set_piece(piece):None
are neighbours of
is part of
Piece
+ colour:Enum = colour
+ __init__():None
+ get_colour:Enum

Changes and new elements since sprint one

Brought back the "Board" class

We originally thought of the board itself as a network of positions. However, while implementing, some of the states checking for the "network of positions" have nowhere else to be. Such as checking if a position is occupied, one has to go through the network of positions to check. This is rather hard to implement and even if it was implemented it is confusing and difficult to maintain good quality of code. As an alternative way, we added back the Board class to handle all functions and resolved those problematic issues.

"Display" engine

Instead of making every class have a display as an attribute to display. We switched to make the Display that loads scenes, which could be any subclass of scenes. A scene included the resources needed for that section, such as images, logic. In the Display class, it registers which scene to load and goes running display() in that scene class/subclass. Doing so makes the display highly modular and easier for maintaining. If more scenes are needed, all we have to do is create a new subclass of Scene and define when it will be loaded. The main Scene class included the shared resources of the different scenes like background, button sounds, and basic colour palette. If the resources are not shared and would only be used in a subclass, it just needs to be extended. By adopting this method, we successfully achieved the Open-Closed Principle where the subclass of Scenes are open to extension, and Display and Scene are closed to modification.

Implemented Design Pattern

Singleton and Factory Method on "GameManager" class

Once a game is initialised, there will only be that game until the game is finished or terminated. So when we call the game objects, it needs to ask the GameManager if there is a Game that already exists. GameManager has an attribute to store the current game, whenever a game is in progress, that Game object is the one returned. Applying the Singleton design pattern, this streamed down our code to prevent creating other game objects. Also, with GameManager, it also acts as a factory method. It checks for any games that already exist, if not, create a new Game and set it to current_game. When a game is terminated, the game will be deleted. In general, GameManager handles all object calls for Game to ensure no other games are created and conflict with other codes.

Observer for checking Mills

For the game to operate, we designed it to work by creating turn objects. The game's responsibility is to manage the state of the game. Turns are created inside the Game's loop as a part of the game, and the Player interacts with the board via turns. Once the player has made their actions in turn, Turn invokes MillChecker for any newly created Mill once the piece has been placed/moved, but not when a piece is removed since it will not result in removing a piece. The result of MillChecker will return to Game for determine the next turn. By applying the Observer, it helps with handling mill checking problems which would be clumsy.

Discussion made to determine the design

○ **Explain two key classes that you have debated as a team, e.g. why was it decided to be made into a class? Why was it not appropriate to be a method?**

Initially we did not have a Board class, but when we came around to actually start coding the game, we realised that it would be both easier to code and fundamentally more sound to do so - giving each class an appropriate amount of responsibility. Originally we had a method that would initialise the 24 positions(which are now on the board), but with the Board class we can assign each position a unique index therefore making them immediately accessible and have the positions be managed by the Board class. Having Board as its own separate class also means that we can have different board states (for different turns) and manipulate the boards - comparing them against each other or saving them as board states (should we choose to implement an undo move function or load game function). It also made our code a lot cleaner (more readable) instead of having many many lines of code doing heaps of random things inside the main Game class (kind of like a God class).
It would not be appropriate as a method, as positions currently only know their neighbours, some class above positions (and pieces) needs to be able to get the location and occupancy of positions/pieces so that we can implement remove and fly moves.

We also extensively debated how we would check for Mills, which on their creation would trigger a turn action for the player who created it (the move to remove an opponent piece). Originally we thought of throwing it inside turn, however upon closer inspection of the rules, the pieces that were in mills needed to be stored somewhere and managed/calculated so that we would know that those pieces cannot be removed by the remove action turn. We concluded that certain types of Turns would run MillCheck which would check the previous and new position of the last/currently moved piece and delete and or create mills from the list of mills (to be stored in Board) as fit.

○ **Explain two key relationships in your class diagram, e.g. why is something an aggregation not a composition?**

Position -> Board
The positions are on the board or contained within the board, that is the positions cannot exist independently without the board (ruling out aggregation). This means that this relationship is composition. The board is also not much of a board (useless/redundant) if it doesn't contain positions.

Piece -> Position
This relationship is aggregation and not composition. This is because Positions can exist without Pieces. Pieces can also exist outside of a position, 9 of them are created at the start of the game as a part of each players' starting pile.

○ **Explain your decisions around inheritance, why did you decide to use (or not use) it? Why is your decision justified in your design?**

We used inheritance for the Game_Scr, Menu_Scr, and Tute_Scr, which inherit from the Scene class. This is because they all use methods from the Scene to control and display things on the screen as well as use the attributes from the Scene class as assets (sounds and images).

○ **Explain how you arrived at two sets of cardinalities, e.g. why 0..1 and why not 1…2?**

Turn -> mill checker ended up being 0..1, because there are certain types of turns (that we plan on implementing) that do not require mill_checker to be invoked. Placing (first 9 tokens), moving, and flying pieces all require checking if mills are created, but removing a piece does not.

Position -> position ended up being 2..4, because each position knows the locations of its immediate neighbour positions. Some of them have only a left and right neighbour for example, but some of them have 4 neighbours.

○ **Explain why you have applied a particular design pattern for your software architecture**

We chose to implement observer design pattern for the mill checker class as it most closely aligned with the principles outlined in Refactoring Guru regarding the observer pattern: "*Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.*" (Shvets, unknown) In our case, the members of a mill may be formed dynamically based on the board state, and any piece can be part of a mill, especially during flying phase. It also allowed us to maintain the open close principle if any modifications were needed in the future.

○ **Explain why you ruled out two other feasible alternatives?**

We decided against using the visitor pattern and the state based pattern for mills. The visitor pattern was posited as being primarily used when traversing tree nodes (Shvets, unknown), and was also helpful when needing to store sub-information during such a traversal. However, we did not implement a tree type data structure, and we do not foresee different types of pieces being added to the game in the future, therefore it was not used.

As for the state based pattern, we had already implemented a variation of this pattern in the turn types being used to interact with the board, and it was thus redundant to implement it again at the lower level of the mill checker.

~ End ~

References:

1) Shvets, A. (n.d.). *Observer*. Refactoring.Guru. Retrieved April 27, 2023, from https://refactoring.guru/design-patterns/observer
2) Shvets, A. (n.d.). *Visitor*. Refactoring.Guru. Retrieved April 27, 2023, from https://refactoring.guru/design-patterns/visitor