A3 - Sprint 0
Team Null

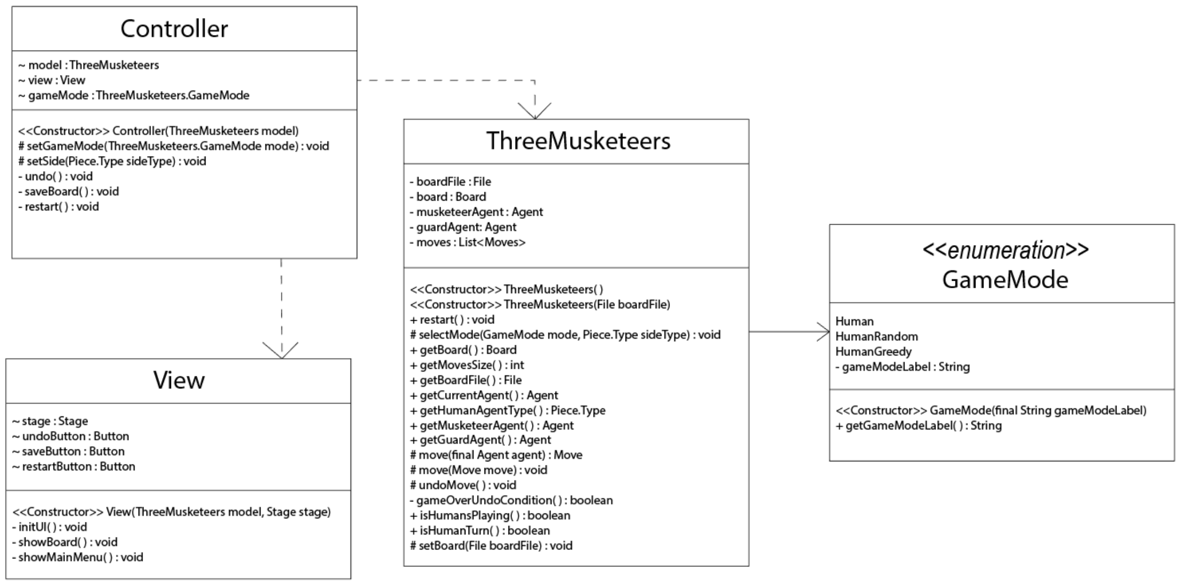
Junaid Khan
Behrouz Akhbari
Jahangir Minhas
William Chik

Table of Contents

List of Patterns used

1. MVC Pattern
2. Strategy Pattern
3. Builder Pattern
4. Observer Pattern
5. Command Pattern
6. Composite Pattern

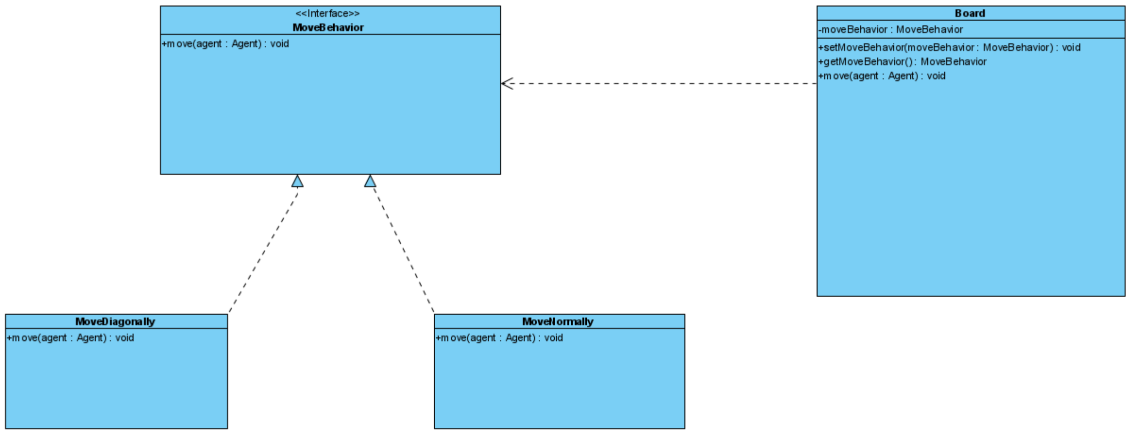
MVC Pattern


Controller
~ model : ThreeMusketeers
~ view : View
~ gameMode : ThreeMusketeers.GameMode
<<Constructor>> Controller(ThreeMusketeers model)
# setGameMode(ThreeMusketeers.GameMode mode) : void
# setSide(Piece.Type sideType) : void
- undo( ) : void
- saveBoard( ) : void
- restart( ) : void
View
~ stage : Stage
~ undoButton : Button
~ saveButton : Button
~ restartButton : Button
<<Constructor>> View(ThreeMusketeers model, Stage stage)
- initUI( ) : void
- showBoard( ) : void
- showMainMenu( ) : void
ThreeMusketeers
- boardFile : File
- board : Board
- musketeerAgent : Agent
- guardAgent: Agent
- moves : List<Moves>
<<Constructor>> ThreeMusketeers( )
<<Constructor>> ThreeMusketeers(File boardFile)
+ restart( ) : void
# selectMode(GameMode mode, Piece.Type sideType) : void
+ getBoard( ) : Board
+ getMovesSize( ) : int
+ getBoardFile( ) : File
+ getCurrentAgent( ) : Agent
+ getHumanAgentType( ) : Piece.Type
+ getMusketeerAgent( ) : Agent
+ getGuardAgent( ) : Agent
# move(final Agent agent) : Move
# move(Move move) : void
# undoMove( ) : void
- gameOverUndoCondition( ) : boolean
+ isHumansPlaying( ) : boolean
+ isHumanTurn( ) : boolean
# setBoard(File boardFile) : void
<<enumeration>>
GameMode
Human
HumanRandom
HumanGreedy
- gameModeLabel : String
<<Constructor>> GameMode(final String gameModeLabel)
+ getGameModeLabel( ) : String

What it does:
The MVC pattern works in a way such that every time game data needs to be modified, the controller gets the data from the model and sends it to the view to be updated (for example: to switch a turn, the controller calls the method from the model and the turns are switched. Once that is done, the controller asks the view to update the label displaying the turns accordingly).

What it solves:
This pattern solves the issue with typing commands through the console, which makes the interaction awkward and a countdown timer could not be implemented nicely. The MVC pattern allows us to implement a GUI to the game, which would allow the user to interact with the game without having to type in commands into the console itself. Making the game more user friendly and making the view less cluttered. With the use of a GUI, a timer can also be shown ticking down to visualise it without lines of code being printed every second on the console.

Strategy Pattern

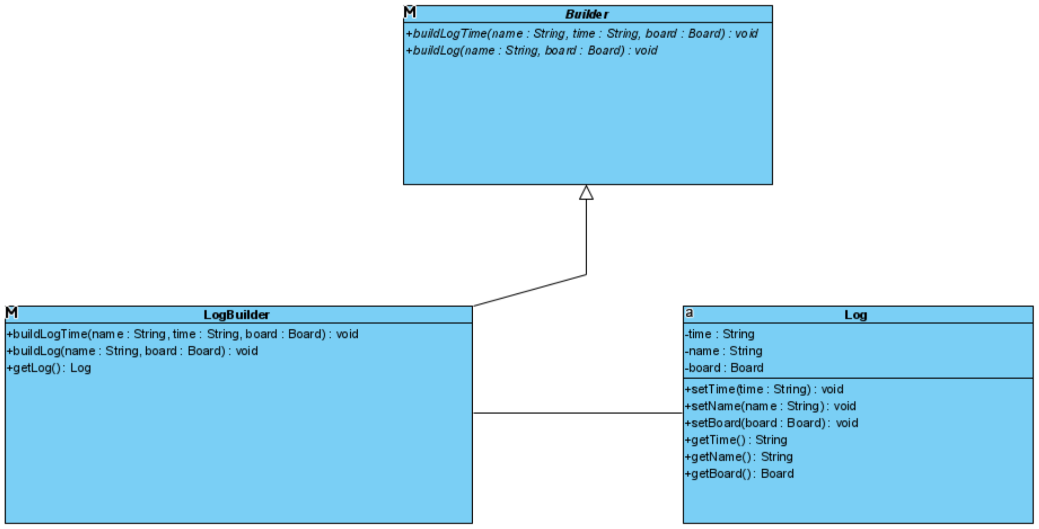


<<Interface>>
MoveBehavior
+move(agent : Agent) : void
Board
-moveBehavior : MoveBehavior
+setMoveBehavior(moveBehavior : MoveBehavior) : void
+getMoveBehavior() : MoveBehavior
+move(agent : Agent) : void
MoveDiagonally
+move(agent : Agent) : void
MoveNormally
+move(agent : Agent) : void

What it does:
The strategy pattern defines 2 different move behaviours, encapsulates each one and makes them interchangeable. We can set the strategy attribute, giving the board a certain behaviour, and the board refers to the behaviour every time it makes a move. In this particular example, the board can either have a diagonal moving behaviour which allows the pieces to move diagonally in conjunction with their regular moves. Or, with the MoveNormally class, the board would make the pieces move normally to orthogonal squares.

What it solves:
It solves the issue of the game being too stale, and the moves being too repeated after multiple games. The strategy pattern allows us to create a new move for the pieces, which would in turn create a different dynamic in the game, diversifying the gaming experience.

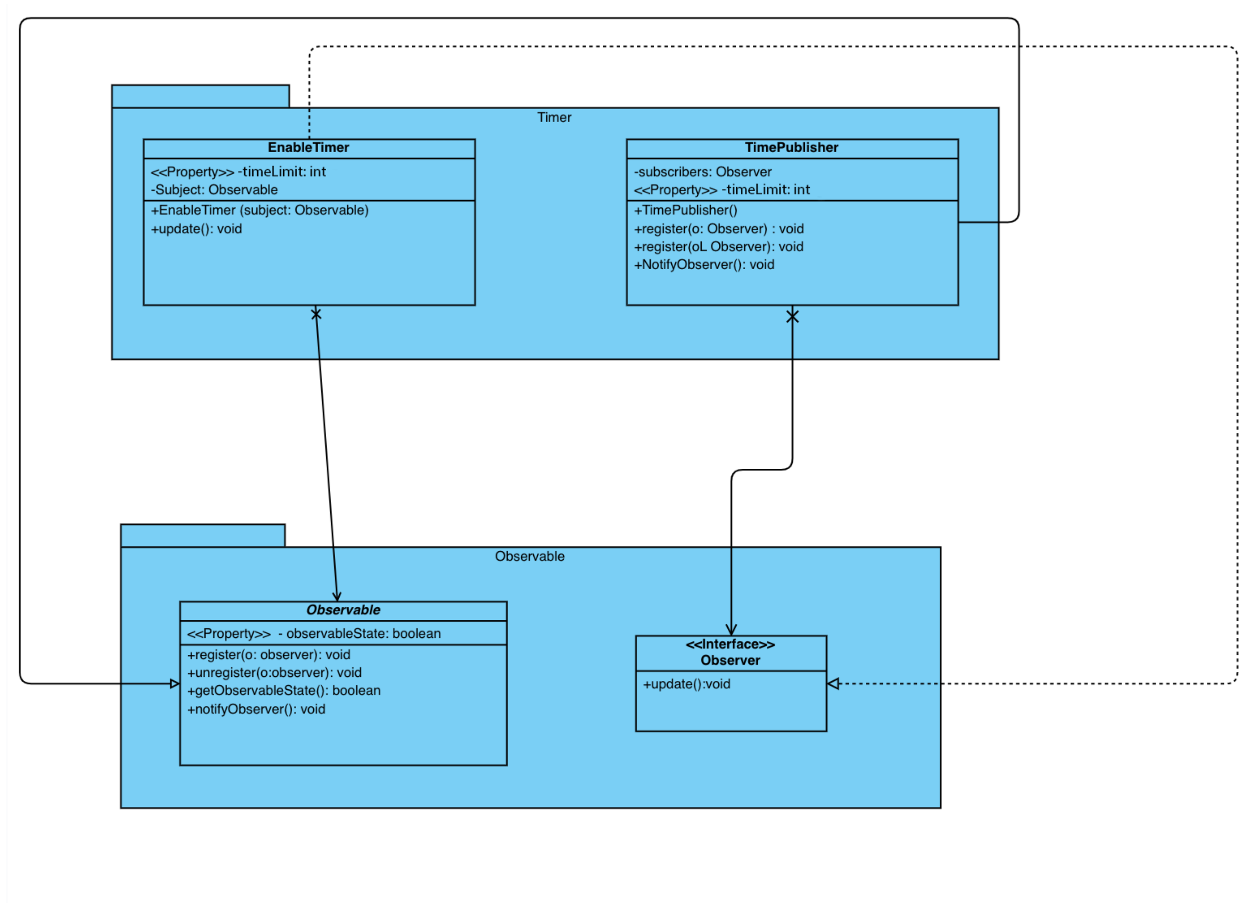
Builder Pattern


Builder
+buildLogTime(name : String, time : String, board : Board) : void
+buildLog(name : String, board : Board) : void
LogBuilder
+buildLogTime(name : String, time : String, board : Board) : void
+buildLog(name : String, board : Board) : void
+getLog() : Log
Log
-time : String
-name : String
-board : Board
+setTime(time : String) : void
+setName(name : String) : void
+setBoard(board : Board) : void
+getTime() : String
+getName() : String
+getBoard() : Board

What it does:
The Build Pattern implementation builds a log that allows a user to store certain points of the game in it, allowing for them to return to or save boards in a directory. The LogBuilder class has a feature that allows for it to store the time the board was saved if the user so desires. LogBuilder inherits from the builder and as the pattern demands, it separates the log construction from its representation and it refers to a newly created log through a common interface.

What it solves:
The LogBuilder collects arguments one at a time which avoids the problem of clutter in the constructors which helps avoid errors. Additionally, code quality is improved with this pattern in use as we program towards an interface and follow important principles such as the Open-closed.

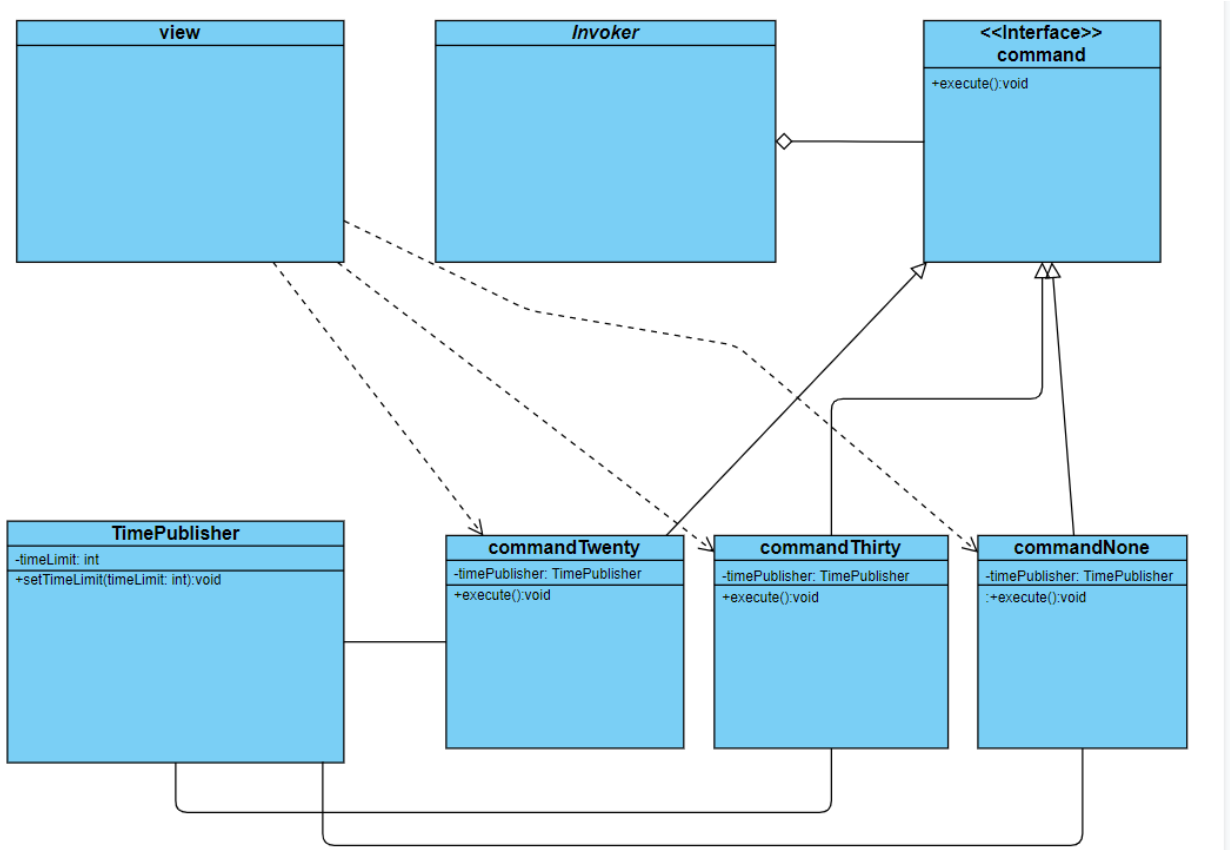
Observer Pattern


Timer
EnableTimer
<<Property>> -timeLimit: int
-Subject: Observable
+EnableTimer (subject: Observable)
+update(): void
TimePublisher
-subscribers: Observer
<<Property>> -timeLimit: int
+TimePublisher()
+register(o: Observer) : void
+register(oL Observer): void
+NotifyObserver(): void
Observable
Observable
<<Property>> - observableState: boolean
+register(o: observer): void
+unregister(o:observer): void
+getObservableState(): boolean
+notifyObserver(): void
<<Interface>>
Observer
+update():void

What it does:
In this Observer Pattern, the TimePublisher notifies EnableTimer so that the timer can start. TimePublisher also contains a newTime attribute which contains the time allowed for each player to move every turn. TimePublisher registers EnableTimer as an observer at the start of the game and will update EnableTimer after every move is made.

What it solves:
This pattern allows us to retrieve and update the time we have for each side without interfering with the timer itself. The timer can continue counting in the background and only notifies the players when it is called. The timer feature solves the issue of players taking too long for each turn unnecessarily, making the game unfair if one side decides to take a long time for each move. The feature also allows players to compete on their ability to think quickly as the timer limits the time each player has to think before each move, adding to the difficulty of the game.

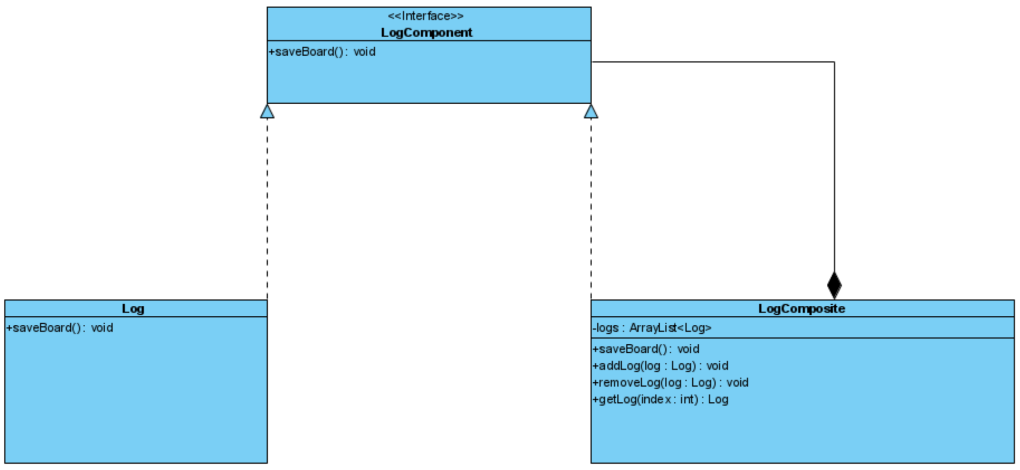
Command Pattern



What it does:
The UI offers the player the option to choose what the time limit will be on every turn. The view will be in charge of receiving user input and issuing commands based on the user's choice. Upon receiving the user's command, the pattern will execute the command by setting the appropriate time limit to TimePublisher.

What it solves:
This pattern solves the issue of allowing us to set the various different times that we have without affecting any of the other classes, namely Timer. Furthermore, it has the added benefit of freely adding more commands without changing or modifying any of the other code, which is a massive improvement in the quality of the code as it is more organized. Additionally, the feature itself is beneficial in that it allows for different time modes, which could help players think on the fly if they would like to play on a shorter time limit or help a paced game if they would like to take a bit longer.

Composite Pattern



What it does:

The Log class is capable of saving the board copy it holds in the Boards directory. LogComposite allows for all the logs the player has made to be saved into that directory. This pattern gives the user the ability to choose whether they want to make a single save or save the game at multiple stages.

What it solves:

It provides convenience for the user. Rather than saving the file individually each time, the user can keep making save points and different stages of the game and save them into a directory all at once.