21/11/2021

CSC207 Assignment 3

Sprint 0

Members: Alan Zilun Zhao, Jesse Han, Muzzammil Sultaan, Glyn Chen

Table of Contents

Strategy pattern
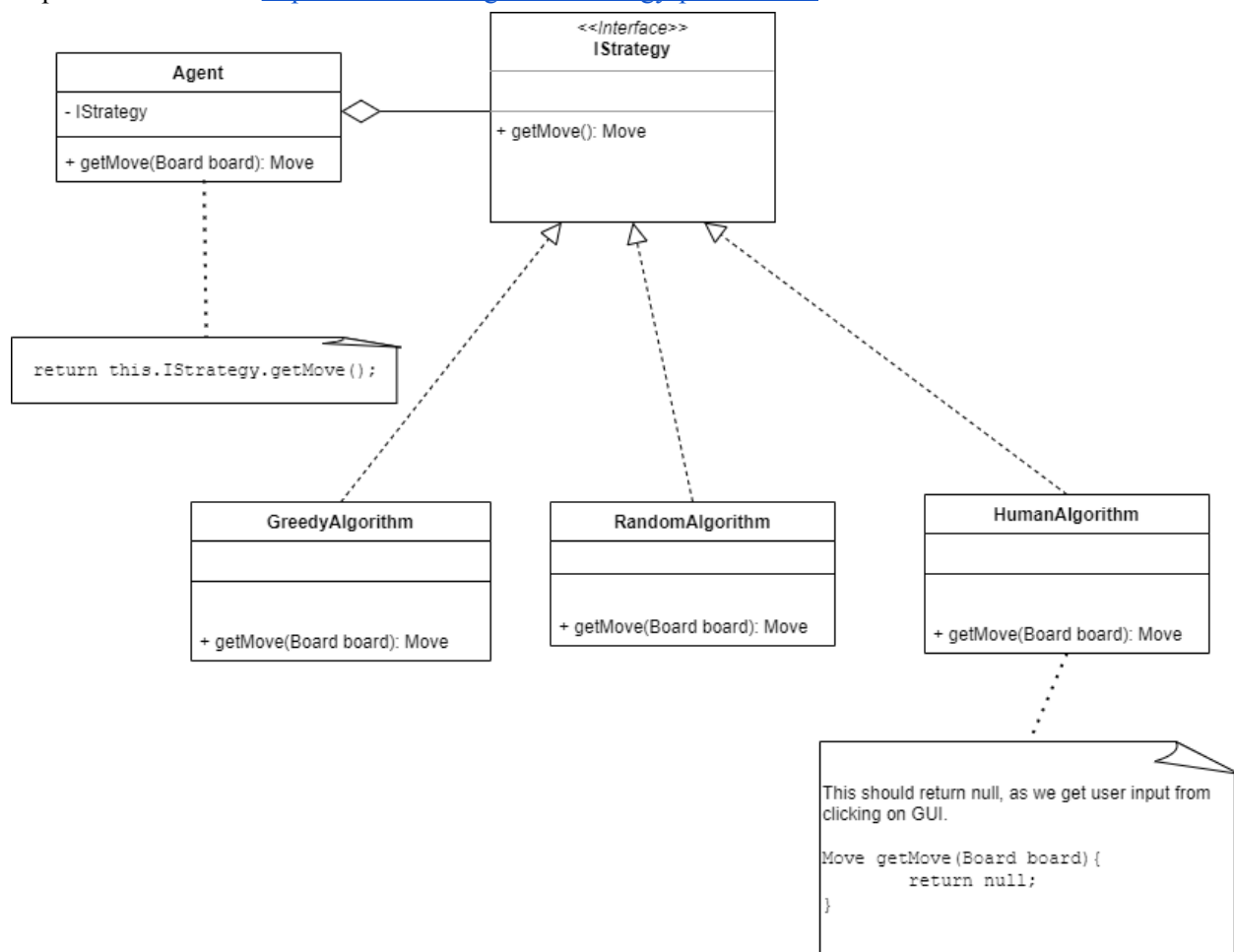
Command

Model, View, Controller

Memento

Builder

Factory

Strategy pattern

We think that the greedy and random agent is an example of strategy, because greedy and strategy agents differ only in their behaviour. In this case, I think that we could define an *interface* **<IBehaviour>** with a *method* **getMove()**, and three concrete classes **<RandomBehaviour>** and **<GreedyBehaviour>** (which have to make use of old code from RandomAgent and GreedyAgent classes respectively to get a move), and **<HumanBehaviour>**. We could then discard the RandomAgent, GreedyAgent and HumanAgent classes, and we could rewrite the Agent class to a concrete class and contain a **<IBehaviour>** field, and have our **getMove()** method in Agent would just return **getMove()** from some instance of a concrete behaviour **<RandomBehaviour>, <GreedyBehaviour>, or <HumanBehaviour>.**

The classes for the Greedy, Random and Human Agent differ solely in their behavior. For this case it is a good idea to isolate the GetMove() function into separate classes in order to have the ability to select different algorithms at runtime.

Inspiration took from https://www.oodesign.com/strategy-pattern.html.

Command pattern

Inspiration took from https://www.oodesign.com/command-pattern.html
We will implement the save function, undo function, and restart function with the command pattern. We can define a **<Command>** *interface* with the method **execute()**, and three concrete classes **<SaveCommand>, <UndoCommand>, <RestartCommand>**. The three concrete diagrams should extend JavaFX button class and implement the **<Command>** *interface*. Each of the concrete classes **execute()** method should call the receiver's corresponding method. We also need to define a **<Receiver>** *class*. This *class* should have **undo(), save(), and restart()** as methods that are implemented in the class. The client (which is the **View**.java *class*) will create a **<ConcreteCommand>** object and set its corresponding **receiver.** The JavaFX button is the invoker which asks the command to carry out the request.

The rationale for implementing this pattern is that we would like to abstract/hide the implementation details of the classes, so that we only have to look into which commands are accessing the class. In addition, since we are leaving implementation details of the command, we can use a common method to instantiate the command.
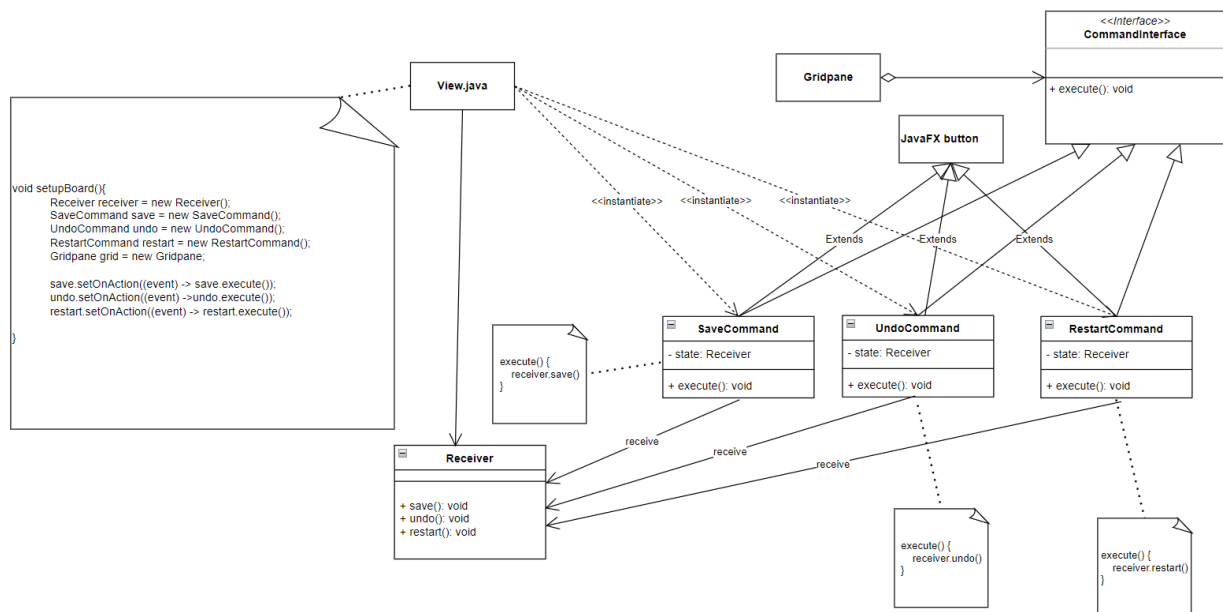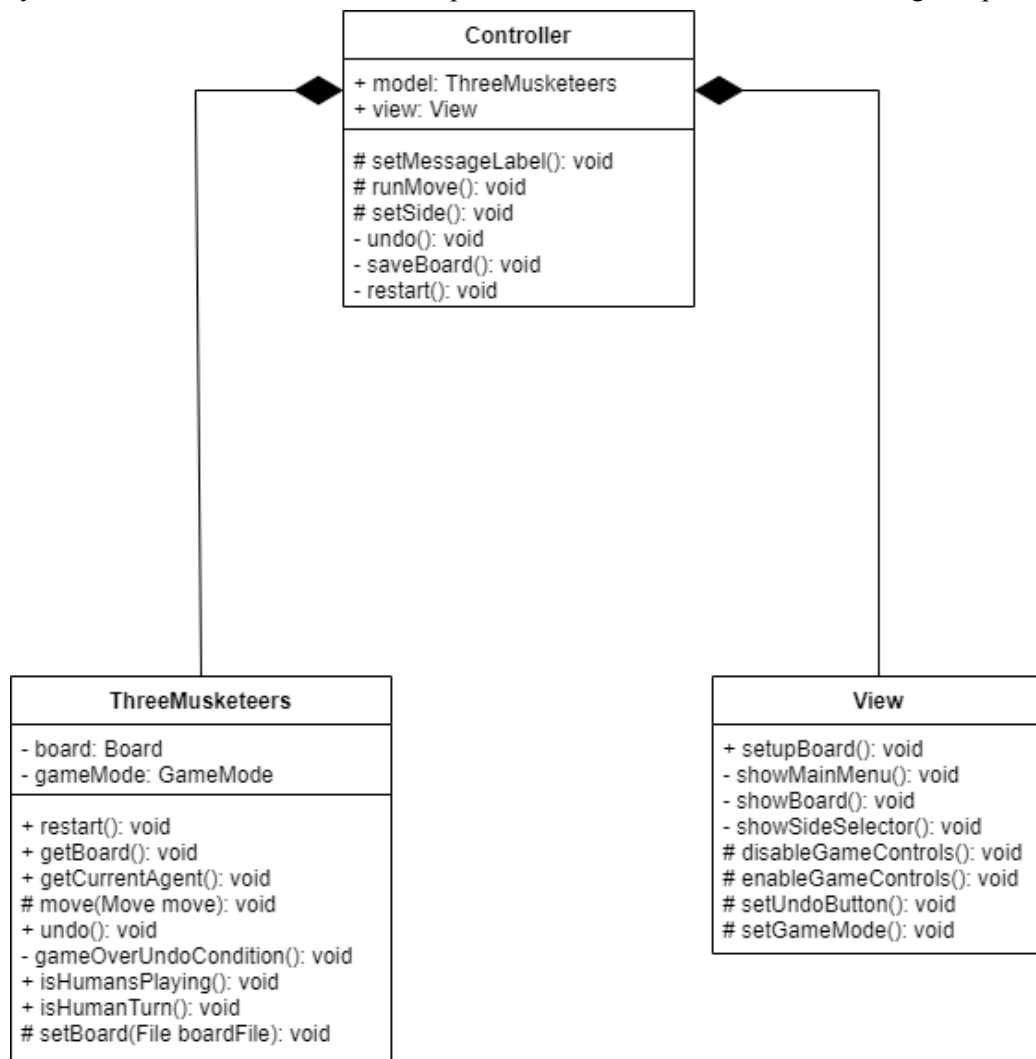
Model, View, Controller

(Lecture)

The MVC design pattern is used within the GUI. In the case of our code, we have the **controller** in the "**View.java" class**, and the **model** is the **"ThreeMusketeers.java" class**. As a result, the **View + Controller** updates and moves the cells on the model. The **view** part of the MVC is the **GUI** and the **buttons** shown on the screen when playing the game. This, as earlier explained is in the **view class**. The **controller** part of the MVC the actions made on the move, when the buttons are **pressed**, which is again in the **view class**. Finally, the **model** of the MVC is the **ThreeMusketeers** file, which ultimately **controls the backend of the game** (how the cells on the board are moves and how the game is won/lost).

MVC is more of an architectural pattern, so we will use it to implement the UI in conjunction with the logic in the ThreeMusketeers game. MVC mostly relates to the UI and interaction layer of an application, which is perfect for us to select the appropriate pieces to move during the game. The logic will be handled by the model, so the MVC is a useful pattern for us to use in order to make a game paired with UI.

Controller

+ model: ThreeMusketeers
+ view: View

setMessageLabel(): void
runMove(): void
setSide(): void
- undo(): void
- saveBoard(): void
- restart(): void

ThreeMusketeers

- board: Board
- gameMode: GameMode

+ restart(): void
+ getBoard(): void
+ getCurrentAgent(): void
# move(Move move): void
+ undo(): void
- gameOverUndoCondition(): void
+ isHumansPlaying(): void
+ isHumanTurn(): void
# setBoard(File boardFile): void

View

+ setupBoard(): void
- showMainMenu(): void
- showBoard(): void
- showSideSelector(): void
disableGameControls(): void
enableGameControls(): void
setUndoButton(): void
setGameMode(): void

Memento

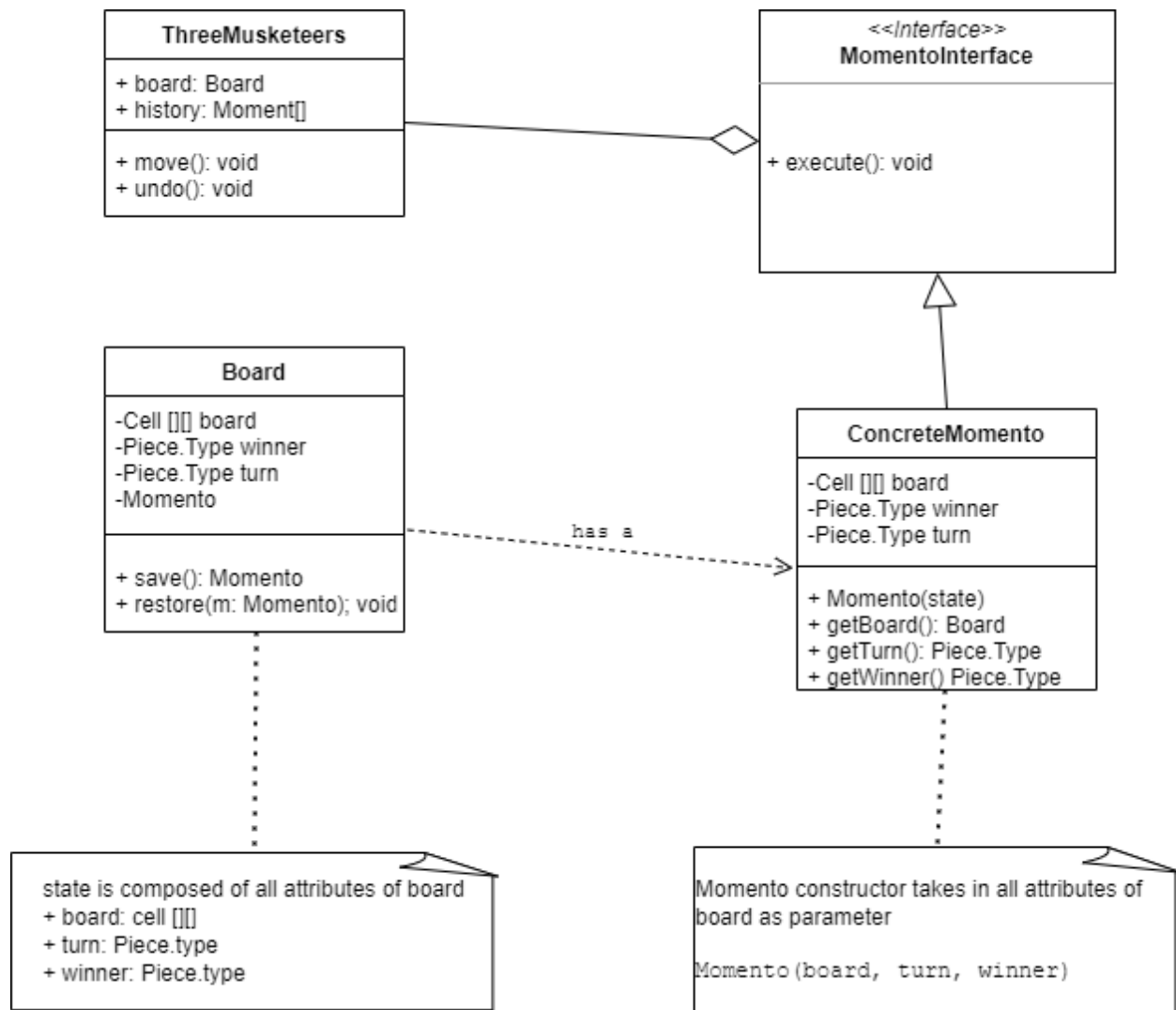https://refactoring.guru/design-patterns/memento

The **Caretaker** *class* is our **<ThreeMusketeers>** *class* which has the **Originator(Board)** as a field, and a **Momento** array history (basically our stack). **move()** and **undo()** are two methods that need to be implemented by the **Caretaker** *class*. The **Originator** is our **<Board>** *class* which must implement **save(): Momento** and **restore(m: Momento)**, we also need to create a **<Momento>** *class* which can hold the board state (deep copy all fields of board state and create new objects). The **<Momento>** *class* is an object with the same fields as the **<Board>** class, also has constructor **Momento(State), getState(): Memento.**

The rationale for implementing the memento class is that it allows us to restore an object to its previous state. In this case, we would like to restore the board to its previous state in the event that we need to undo a move.

ThreeMusketeers

+ board: Board
+ history: Moment[]

+ move(): void
+ undo(): void

<<Interface>>
MomentoInterface

+ execute(): void

Board

-Cell [][] board
-Piece.Type winner
-Piece.Type turn
-Momento

+ save(): Momento
+ restore(m: Momento); void

has a

ConcreteMomento

-Cell [][] board
-Piece.Type winner
-Piece.Type turn

+ Momento(state)
+ getBoard(): Board
+ getTurn(): Piece.Type
+ getWinner() Piece.Type

state is composed of all attributes of board
+ board: cell [][]
+ turn: Piece.type
+ winner: Piece.type

Momento constructor takes in all attributes of board as parameter

Momento(board, turn, winner)

Builder

The **\<Controller\>** will act as our director and has a BoardPaneBuilder attribute with methods that decide on when and how the user's board will be created. The **\<Controller\>** has a **\<BoardPaneBuilder\>** which is an interface that has 4 methods; **\<buildBoard\>**, **\<buildCells\>**, **\<loadCells\>**, and **\<getBoard\>**. **\<TxtBoardPaneBuilder\>** and **\<CustomBoardPaneBuilder\>** are two concrete classes that implement the **\<BoardPaneBuilder\>** interface. **\<TxtBoardPaneBuilder\>** will be called when the user wants to load the game through a txt file that's already created. **\<CustomBoardPaneBuilder\>** will be called when the user wants to create their own custom board. However, the board will still follow the properties of a normal game such as the max pieces of each type. Both will implement the methods in **\<BoardPaneBuilder\>** in their own respective rights in order to create and load the necessary components for the board to operate.

This pattern is being created to address the user request of creating a custom board and being able to start the game however they wish. With the Builder pattern it allows us to create the board with either a txt file or a custom board with the MVC GUI while making sure that they both provide the required end result which is a Board.
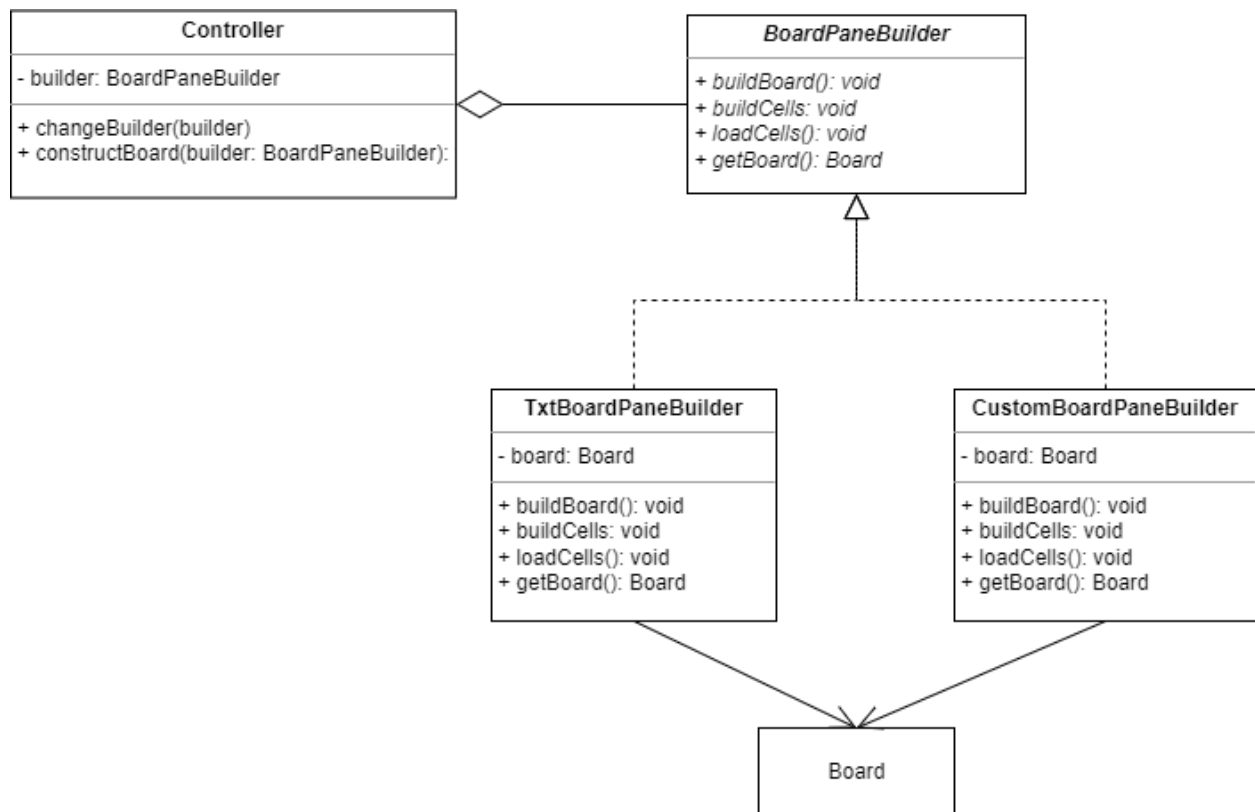
Factory

The factory pattern will be used when implementing how the pieces will be created in the board editor. An interface **<Product>** will be created with concrete classes **<Musketeer>** and **<Guard>** while a class **<PieceCreator>** will be made with concrete classes **<MusketeerCreator>** and **<GuardCreator>**. It will return the required pieces that need to be created on the board editor.

This pattern is being implemented to assist with addressing the user request of being able to start the game in any position they want. With this pattern it allows the pieces to be built on demand which allows for custom board builds. Also with Factory pattern it allows for the creation of more pieces in the future without unnecessary disruption of other classes.