21/11/2021

CSC207 Assignment 3

Sprint 0

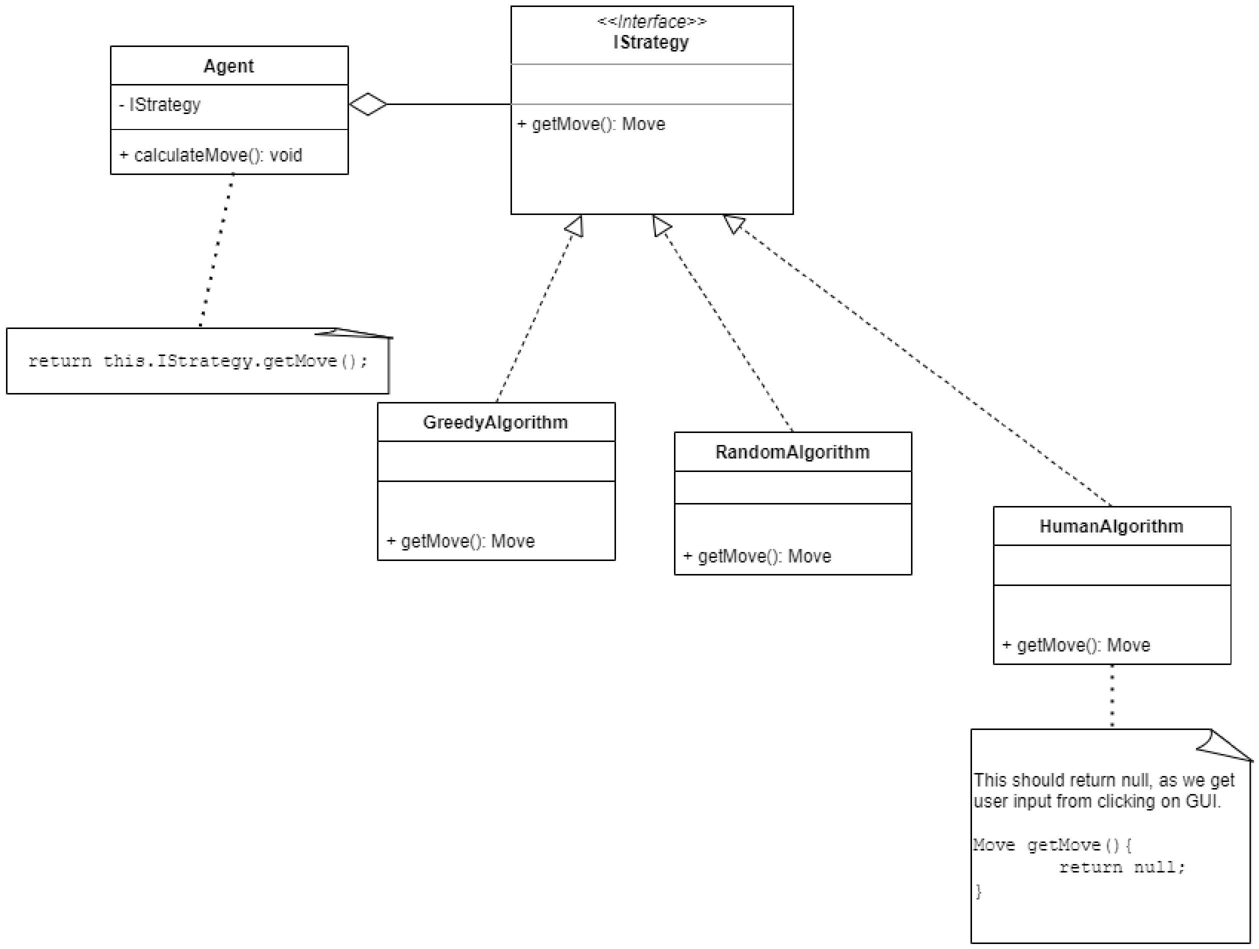
Members: Alan Zilun Zhao, Jesse Han, Muzzammil Sultaan, Glyn Chen

Table of Contents

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>.**

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



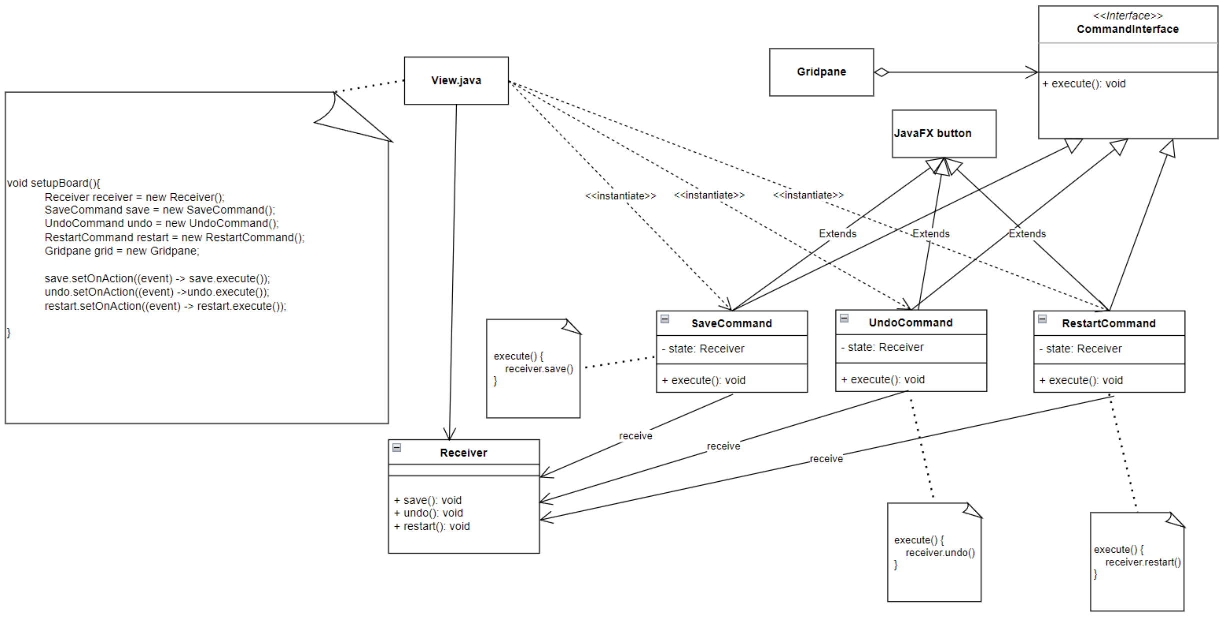
```
return this.IStrategy.getMove();
```

This should return null, as we get user input from clicking on GUI.

```
Move getMove(){
        return null;
}
```

Command pattern

Inspiration took from https://www.oodesign.com/command-pattern.html
We think that the save function, undo function, and restart function is an example of 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.



```
void setupBoard(){
        Receiver receiver = new Receiver();
        SaveCommand save = new SaveCommand();
        UndoCommand undo = new UndoCommand();
        RestartCommand restart = new RestartCommand();
        Gridpane grid = new Gridpane;

        save.setOnAction((event) -> save.execute());
        undo.setOnAction((event) ->undo.execute());
        restart.setOnAction((event) -> restart.execute());

}
```

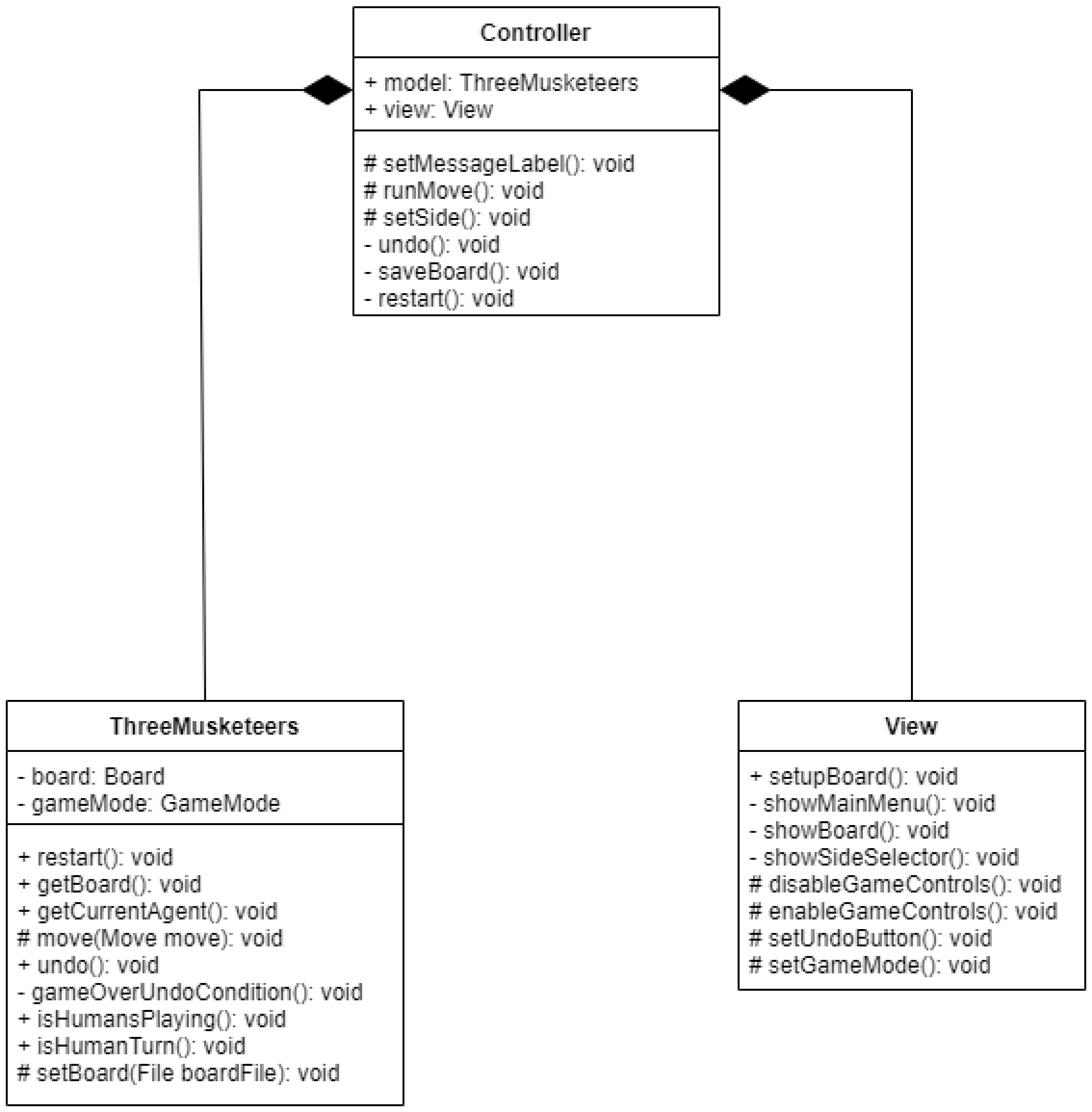
```
execute() {
    receiver.save()
}
```

```
execute() {
    receiver.undo()
}
```

```
execute() {
    receiver.restart()
}
```

Model, View, Controller

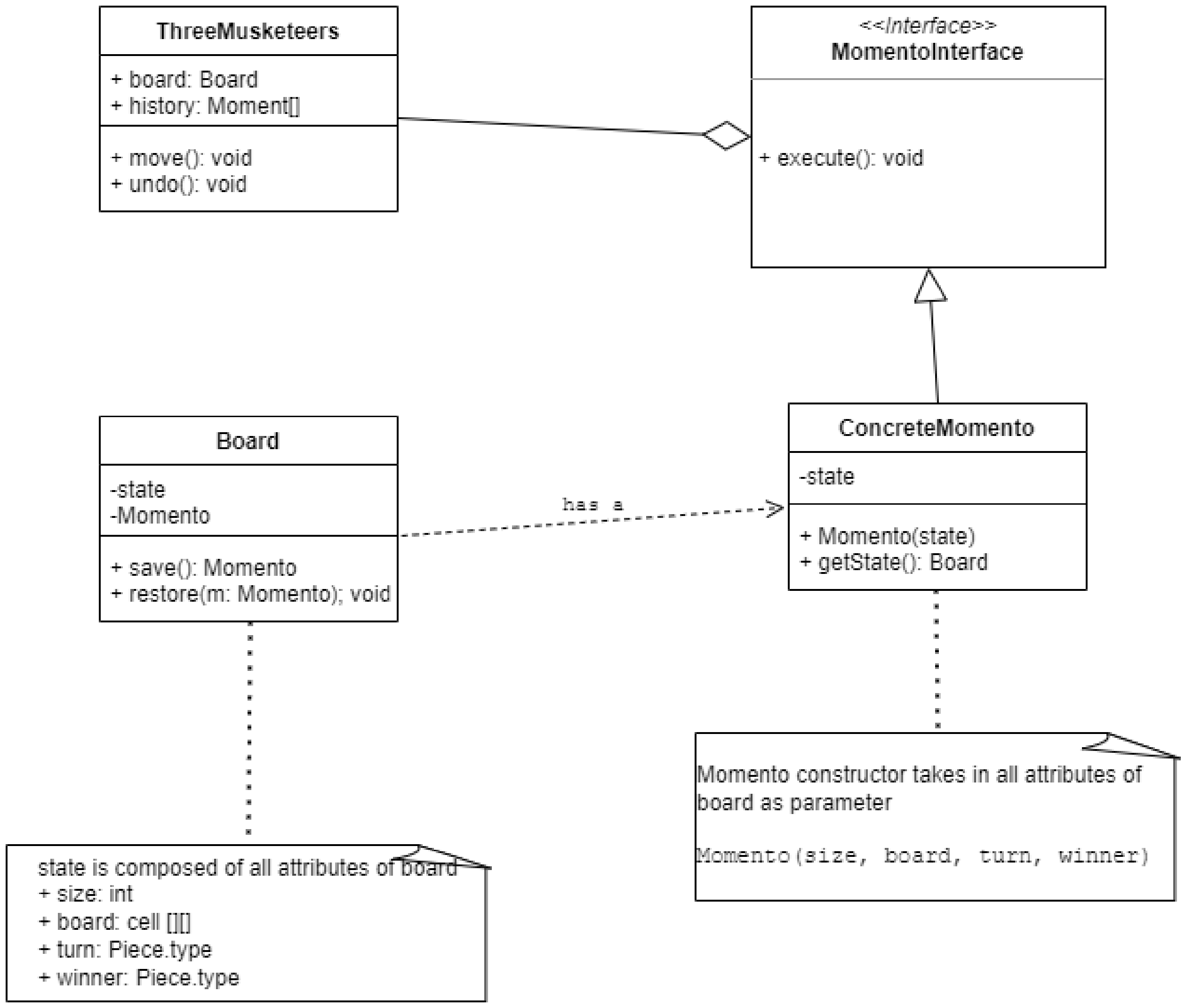
(Lecture)
The MVC design pattern is used within the GUI. In the case of our code, we have the **controller** in the "**View.java" class**, and the **model** is the **"ThreeMusketeers.java" class**. As a result, the **View + Controller** updates and moves the cells on the model. The **view** part of the 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).



Memento

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

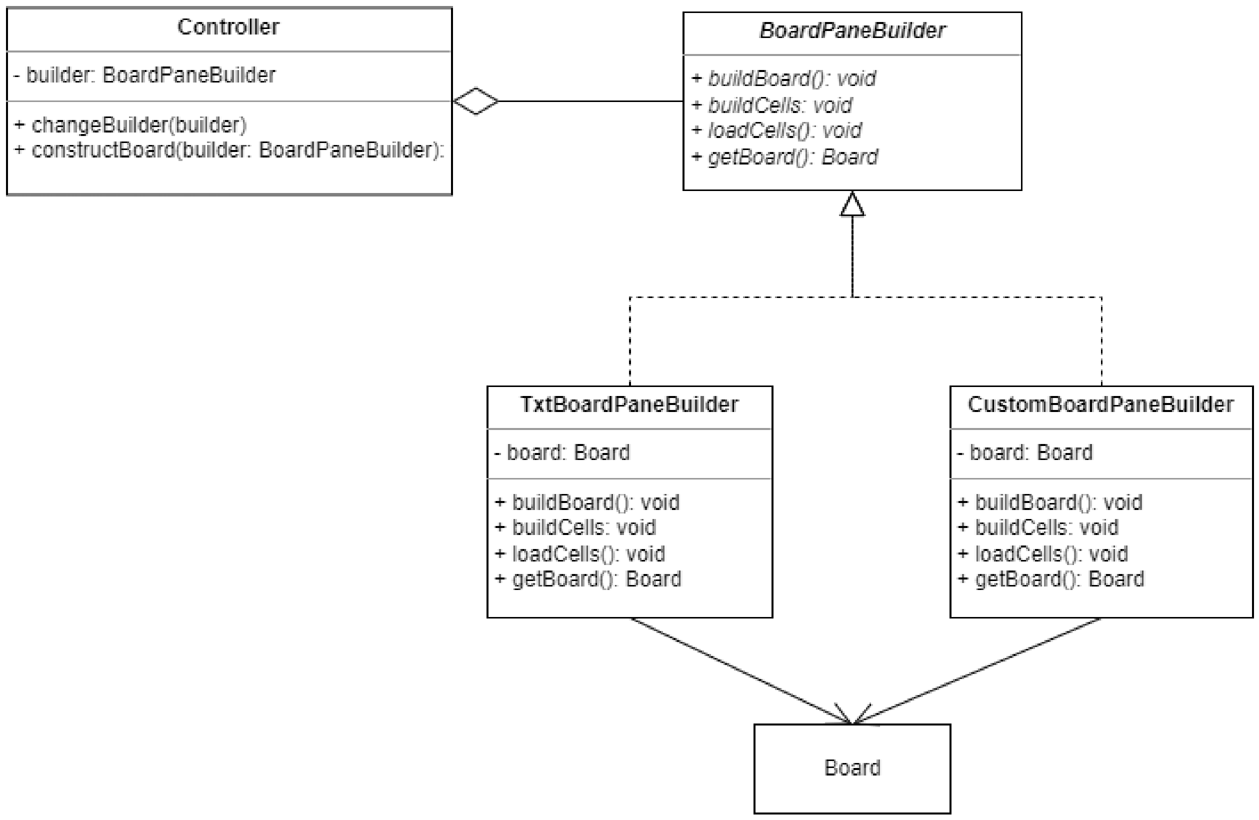
The **Caretaker** *class* is our **<ThreeMusketeers>** *class* which has the **Originator(Board)** as a field, and a **Momento** array history (basically our stack). **move()** and **undo()** are two methods that need to be implemented by the **Caretaker** *class*. The **Originator** is our **<Board>** *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(): Momento.**



Builder

https://www.oodesign.com/builder-pattern.html

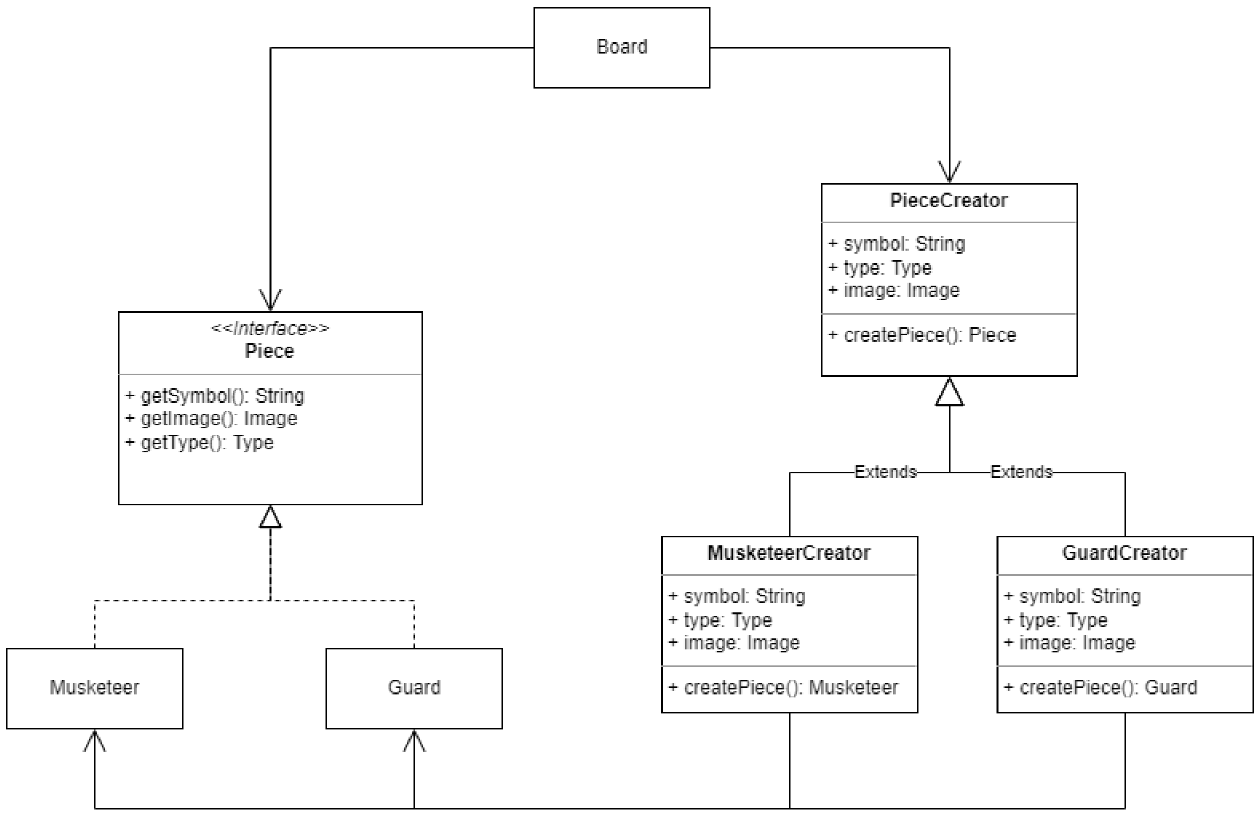
The **<Controller>** will act as our director and has a BoardPaneBuilder attribute with methods that decide on when and how the user's board will be created. The **<Controller>** has a **<BoardPaneBuilder>** which is an interface that has 4 methods; **<buildBoard>**, **<buildCells>**, **<loadCells>**, 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.



Factory

https://www.oodesign.com/factory-pattern.html

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.


Board
<<Interface>>
Piece
+ getSymbol(): String
+ getImage(): Image
+ getType(): Type
PieceCreator
+ symbol: String
+ type: Type
+ image: Image
+ createPiece(): Piece
Extends
Extends
MusketeerCreator
+ symbol: String
+ type: Type
+ image: Image
+ createPiece(): Musketeer
GuardCreator
+ symbol: String
+ type: Type
+ image: Image
+ createPiece(): Guard
Musketeer
Guard