21/11/2021

CSC207 Assignment 3

Sprint 2

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

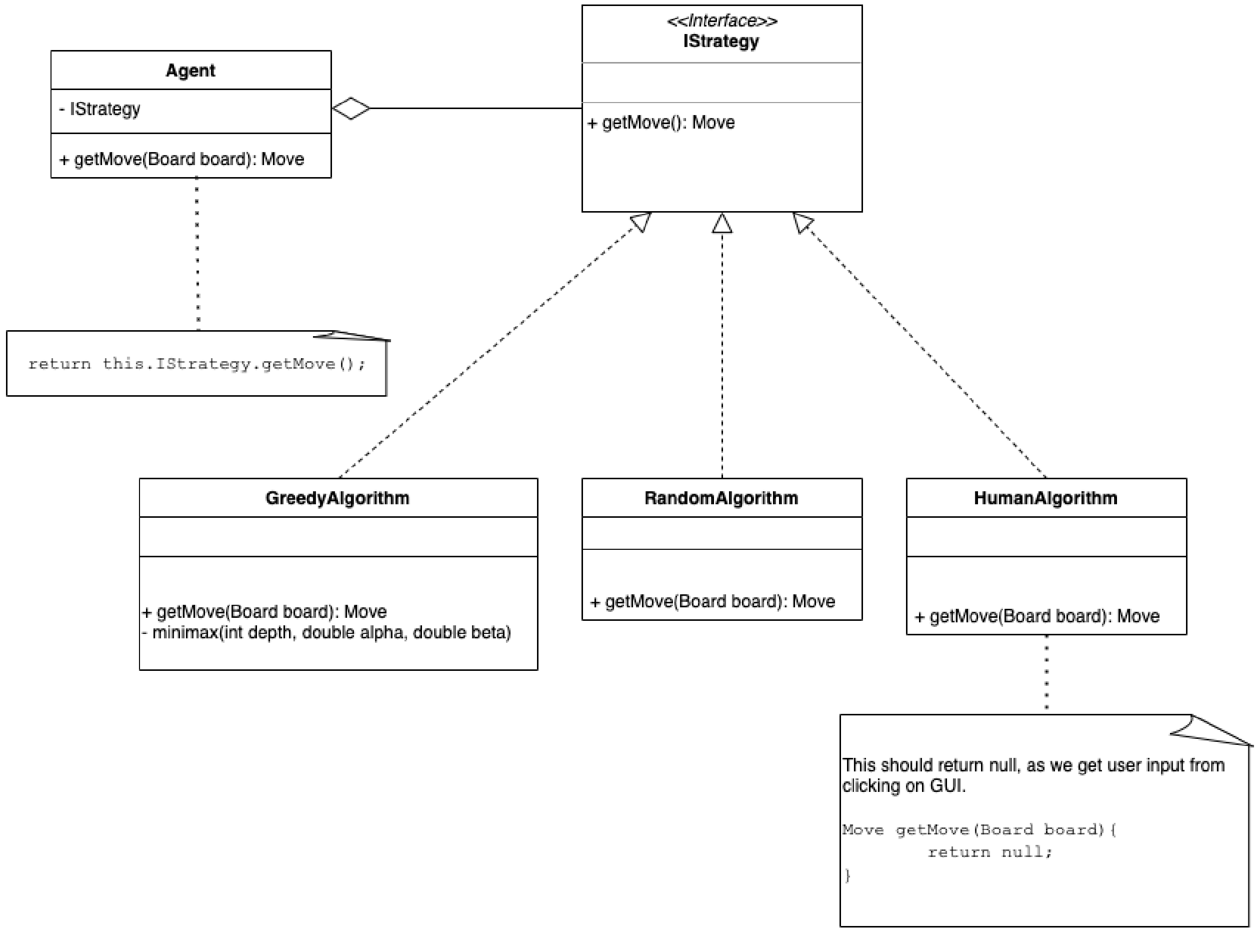
Table of Contents

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>.**

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.

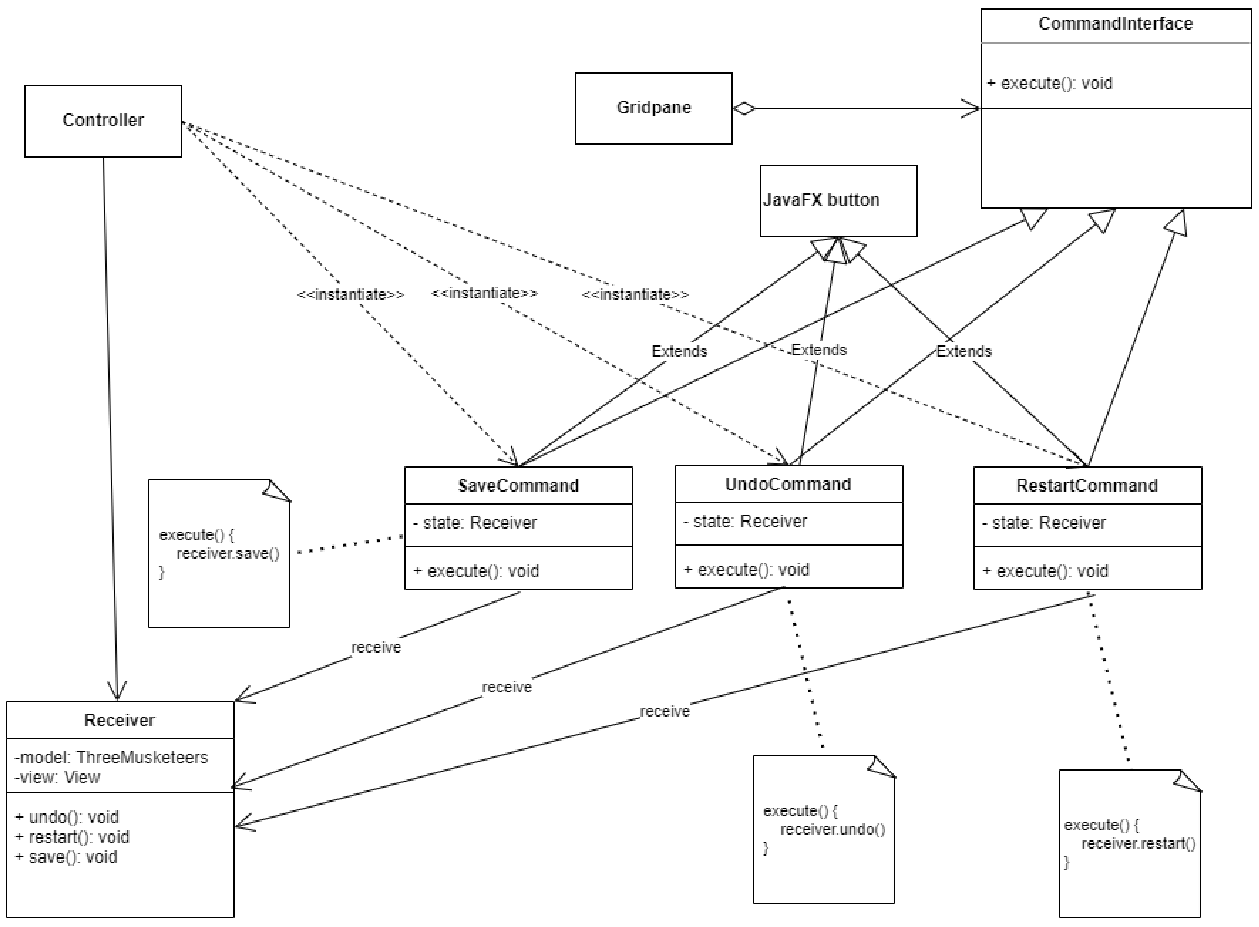
Inspiration took from https://www.oodesign.com/strategy-pattern.html.


Agent
- IStrategy
+ getMove(Board board): Move
<<Interface>>
IStrategy
+ getMove(): Move
return this.IStrategy.getMove();
GreedyAlgorithm
+ getMove(Board board): Move
- minimax(int depth, double alpha, double beta)
RandomAlgorithm
+ getMove(Board board): Move
HumanAlgorithm
+ getMove(Board board): Move
This should return null, as we get user input from clicking on GUI.
Move getMove(Board board){
return null;
}

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.

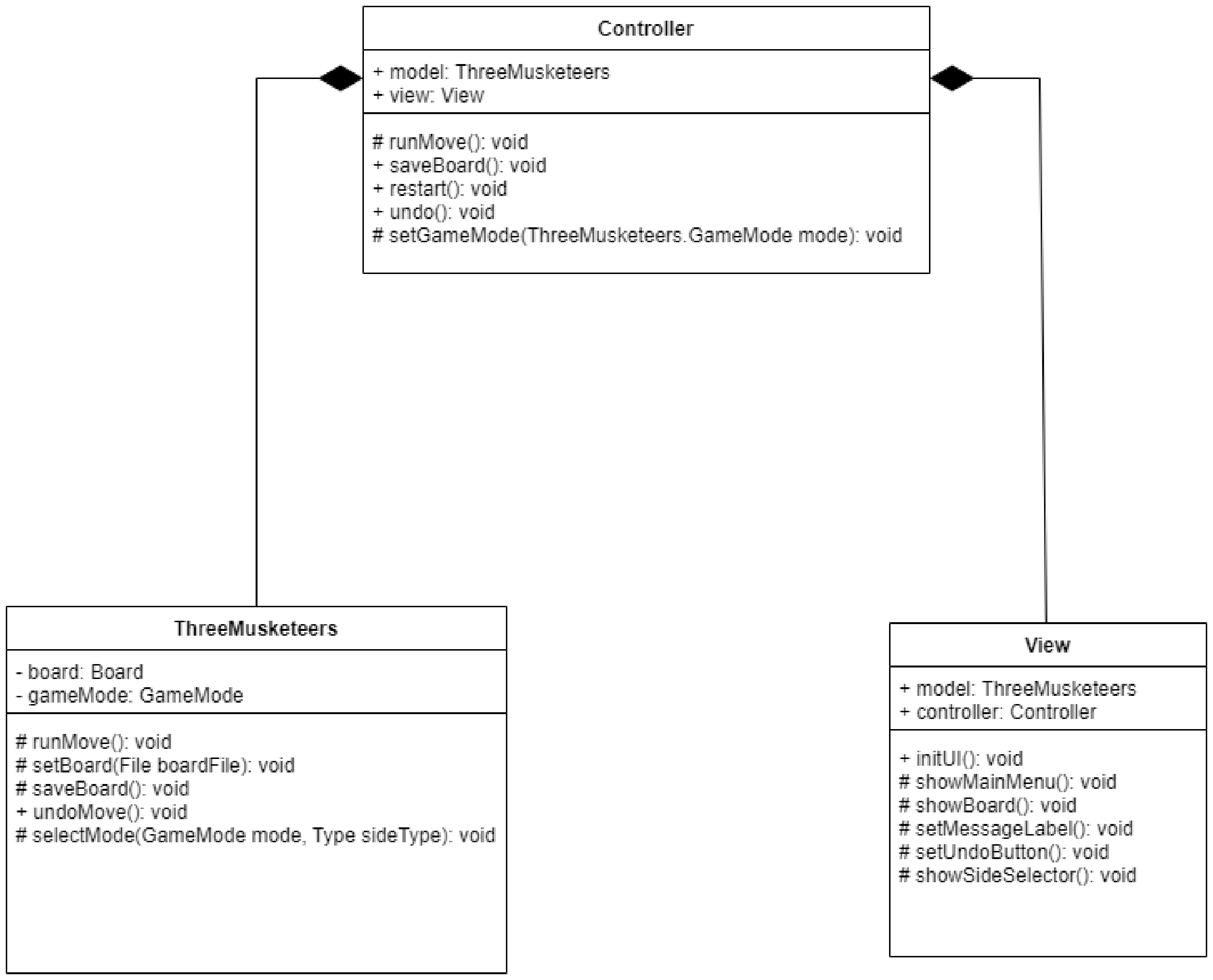
The rationale for implementing this pattern is that we would like to abstract/hide the implementation details of the classes, so that we only have to look into which commands are accessing the class. In addition, since we are leaving implementation details of the command, we can use a common method to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

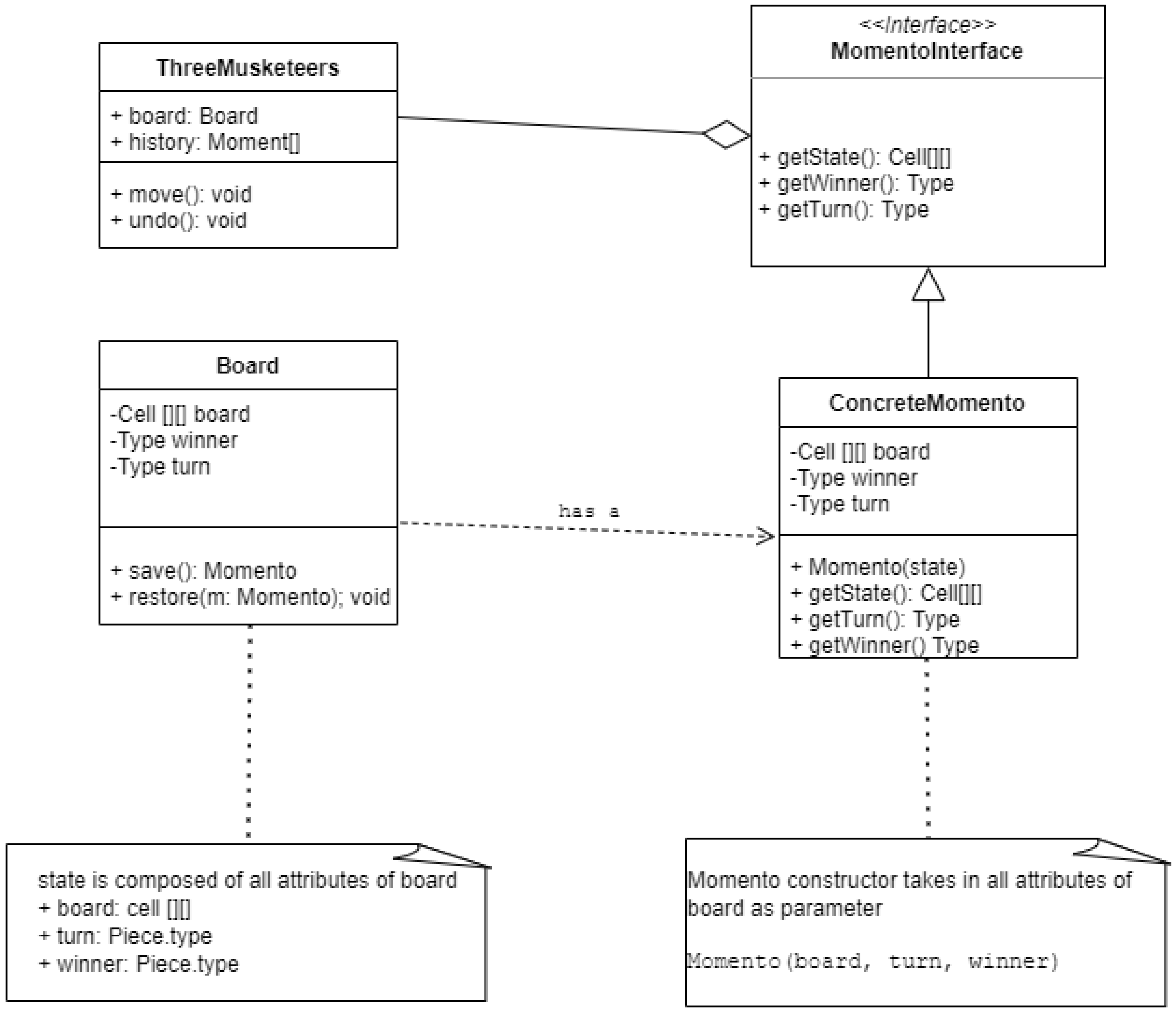

Controller
+ model: ThreeMusketeers
+ view: View
# runMove(): void
+ saveBoard(): void
+ restart(): void
+ undo(): void
# setGameMode(ThreeMusketeers.GameMode mode): void
ThreeMusketeers
- board: Board
- gameMode: GameMode
# runMove(): void
# setBoard(File boardFile): void
# saveBoard(): void
+ undoMove(): void
# selectMode(GameMode mode, Type sideType): void
View
+ model: ThreeMusketeers
+ controller: Controller
+ initUI(): void
# showMainMenu(): void
# showBoard(): void
# setMessageLabel(): void
# setUndoButton(): void
# showSideSelector(): 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.

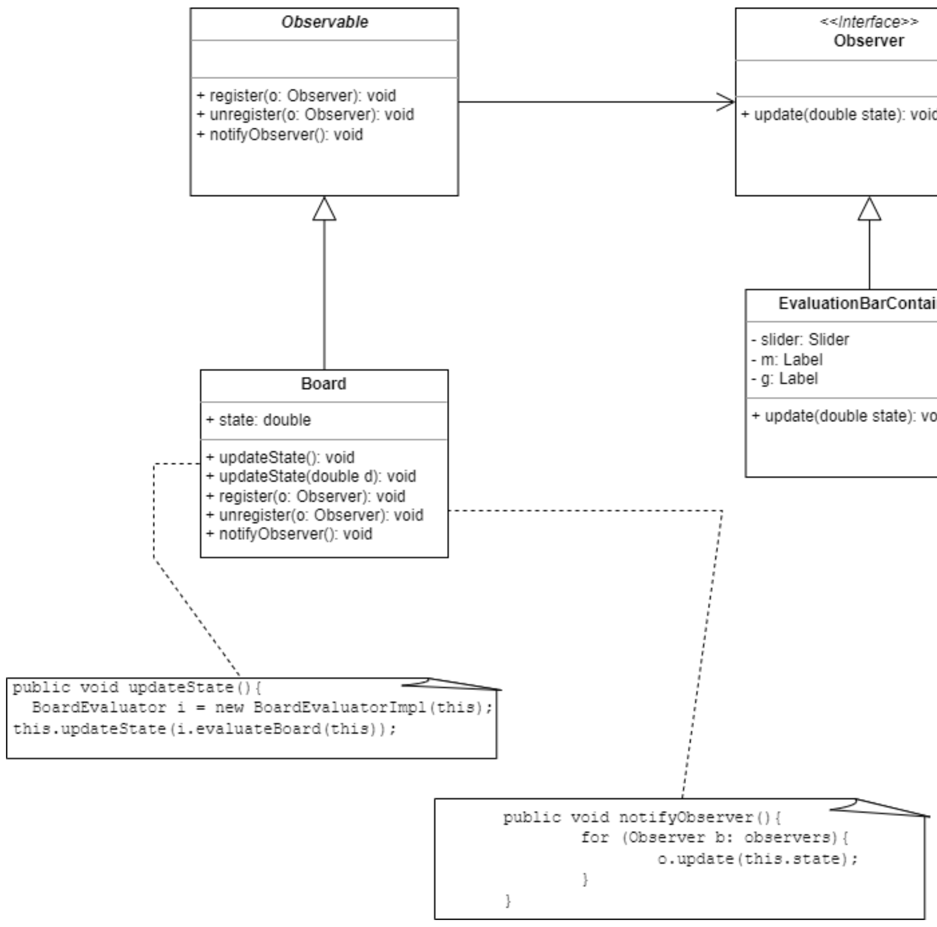


Observable

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

The **Board** *class* extends the **<Observable>** class which has the state that we need (in this case, state is a double value that represents the likelihood of the musketeer and guard winning). The **EvaluationBarContainer** *class* implements our **<Observer>** interface. It contains a slider that is used to represent which side is winning. The updateState method in our **Board** *class* instantiates a **boardevaluatorimpl** from A1, and updates the state, notifying all observers the new state value representing how likely the musketeer is winning.

The intention of an observer pattern is to define a dependency between objects such that when one object changes state, all its dependents are notified and updated automatically. As we wanted an evaluation bar, the observer pattern can be applied as we can register the evaluation bar as an observer to receive state changes when a move is made on the board and the new score is calculated.



Observable
+ register(o: Observer): void
+ unregister(o: Observer): void
+ notifyObserver(): void
<<Interface>>
Observer
+ update(double state): void
EvaluationBarContainer
- slider: Slider
- m: Label
- g: Label
+ update(double state): void
Board
+ state: double
+ updateState(): void
+ updateState(double d): void
+ register(o: Observer): void
+ unregister(o: Observer): void
+ notifyObserver(): void

```
public void updateState(){
  BoardEvaluator i = new BoardEvaluatorImpl(this);
this.updateState(i.evaluateBoard(this));
```

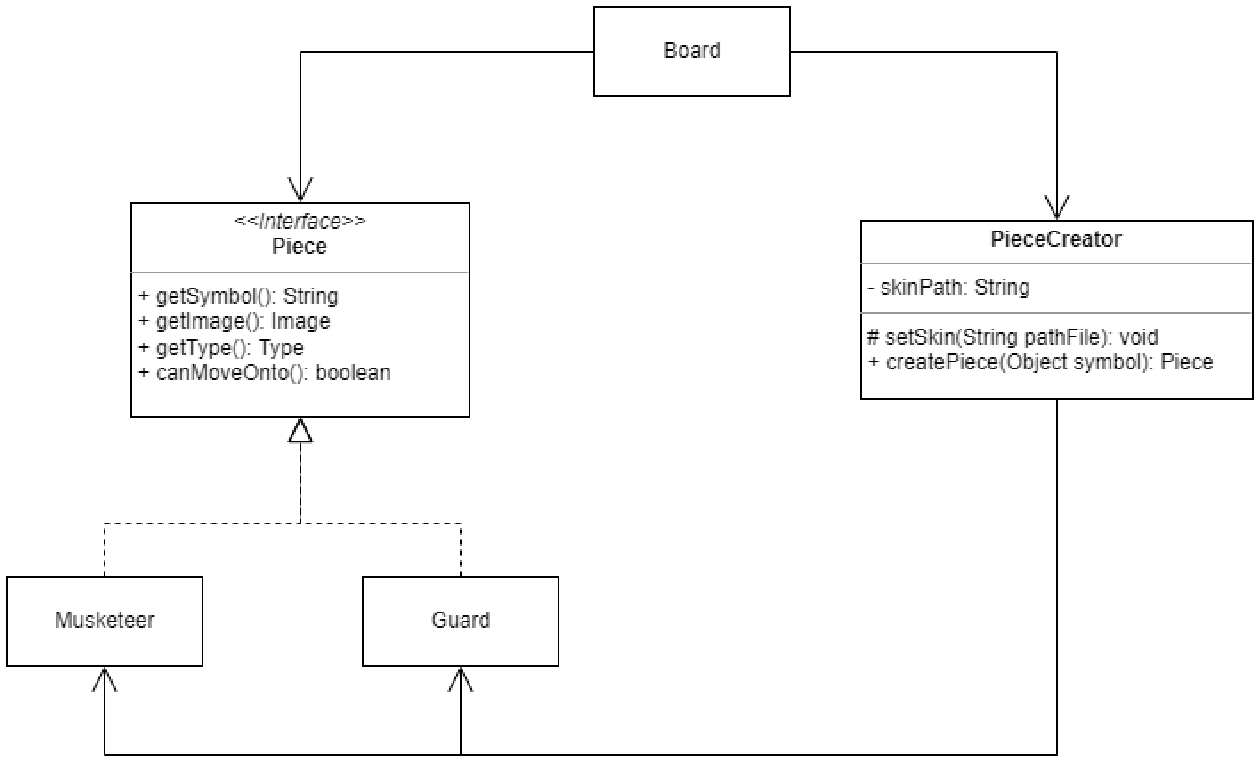
```
        public void notifyObserver(){
                for (Observer b: observers){
                        o.update(this.state);
                }
        }
```

Factory

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

The factory pattern will be used when implementing how the pieces will be created for the board. An interface **<Piece>** will be created with concrete classes **<Musketeer>** and **<Guard>** while a class **<PieceCreator>** will also be created. It will check and return the required pieces that need to be created on the board.

The Factory pattern is being implemented to create custom pieces when the board requires them. This allows the pieces to be created with custom designs and if wanted, it also allows for the easy addition of a new type of piece such as a guard.


Board
<<Interface>>
Piece
+ getSymbol(): String
+ getImage(): Image
+ getType(): Type
+ canMoveOnto(): boolean
PieceCreator
- skinPath: String
# setSkin(String pathFile): void
+ createPiece(Object symbol): Piece
Musketeer
Guard