Design Patterns

Six Unique Design Patterns Chosen By:

Javaholics

Ahmad Ghadban, Jessica Batta, Sakshi Jagtap, Zoubair Hamid

November 20 2021

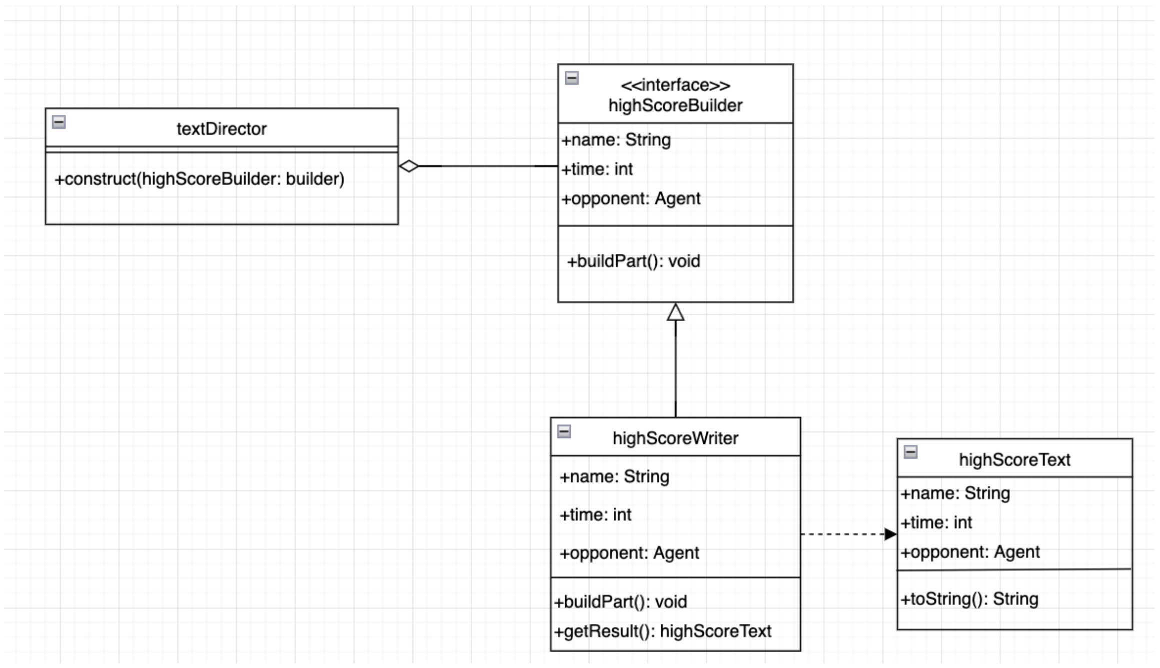
TABLE OF CONTENTS

Design Patterns

Builder Pattern

Description

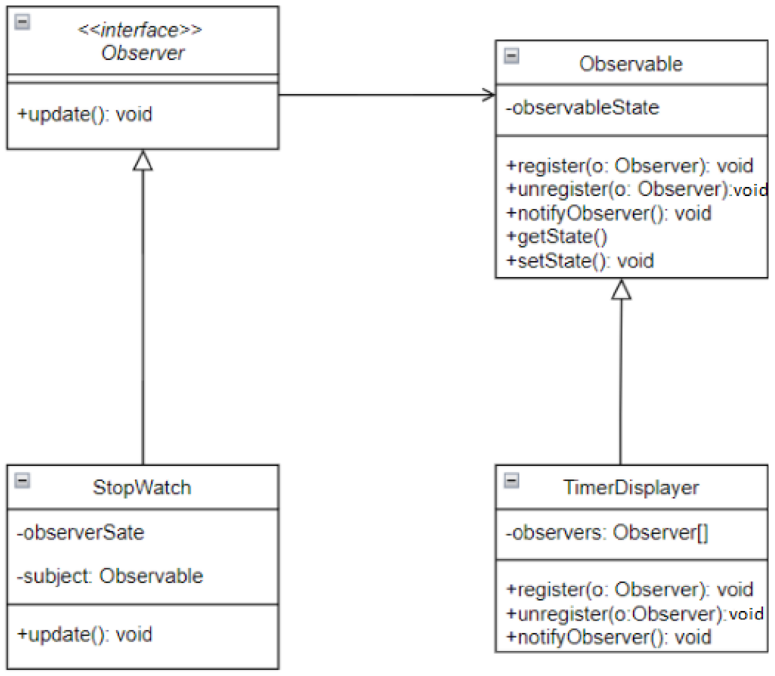
The builder pattern will be implemented by adding a question after a game is finished if they would like to save their time. The button will be the concrete builder since it will call the builder (ie building the string needed). The builder will then allow us to collect arguments one at a time. These arguments will include the user's name, time and opponent. Then the button will be able to get the product (ie the built string) and add it to the highscores table. When listing the highscores that we have implemented this could result in mistakes when taking in parameters for name, time and opponent. The builder pattern will solve any possible mistakes by collecting arguments one at a time. Increasing the quality of our code while handling possible mistakes.



Observer Pattern

Description

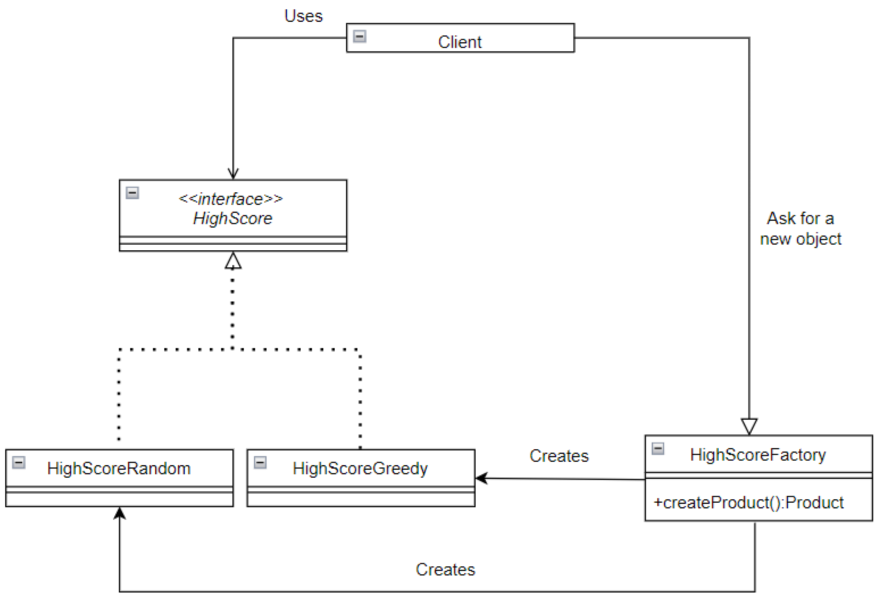
The observer pattern is used when an object is in need of updates in order to function productively and consistently. To specify, an observer pattern is necessary when a certain object is dependent on the state and changes occuring in a different object, in order to maintain consistency between both objects, and ultimately the framework. In our implementation of the Three Musketeers game, we will have a functioning timer that is displayed on the screen at all times. The timer will stop when the game has been completed, and will be saved. The saved times will be used to keep track of players who have won the game the fastest, and will be presented on a leaderboard that can be accessed through the main menu. The inclusion of this functionality can be incorporated through the use of an observer pattern. This pattern is necessary for this implementation as the object that displays the time on the screen, is dependent on constant updates from an object that counts and keeps track of the time that has gone by since the start of the round. A representation of our use of the observer pattern can be seen in the UML diagram below.



Factory Pattern

Description

The factory pattern design is recognized as one of the most commonly used design patterns. Factory pattern design is used in order to create new instances of objects with ease and without revealing the process behind the creation of the instance to the client. The usage of the factory design pattern is necessary in cases where one of numerous different classes may be instantiated, usually with all classes being children of the same parent class. However, the class that is being instantiated is unknown and should not be predetermined before the run time begins. Therefore, a factory pattern is necessary in order to determine which class should be instantiated, through a factory class. In our implementation of the Three Musketeers game, at the end of each game, the player's completion time will be stored in different classes in order to be displayed on a leaderboard. However, there are also different ways to play the game. Meaning, it would be unfair to have high scores for different game modes presented on the same screen, in the same way. In regards to this, different classes to represent different types of high scores for different game modes are required. Therefore, the implementation of the factory pattern is necessary in order to differentiate between each type of completion time and high score, in order to store and present them respective to their game modes.



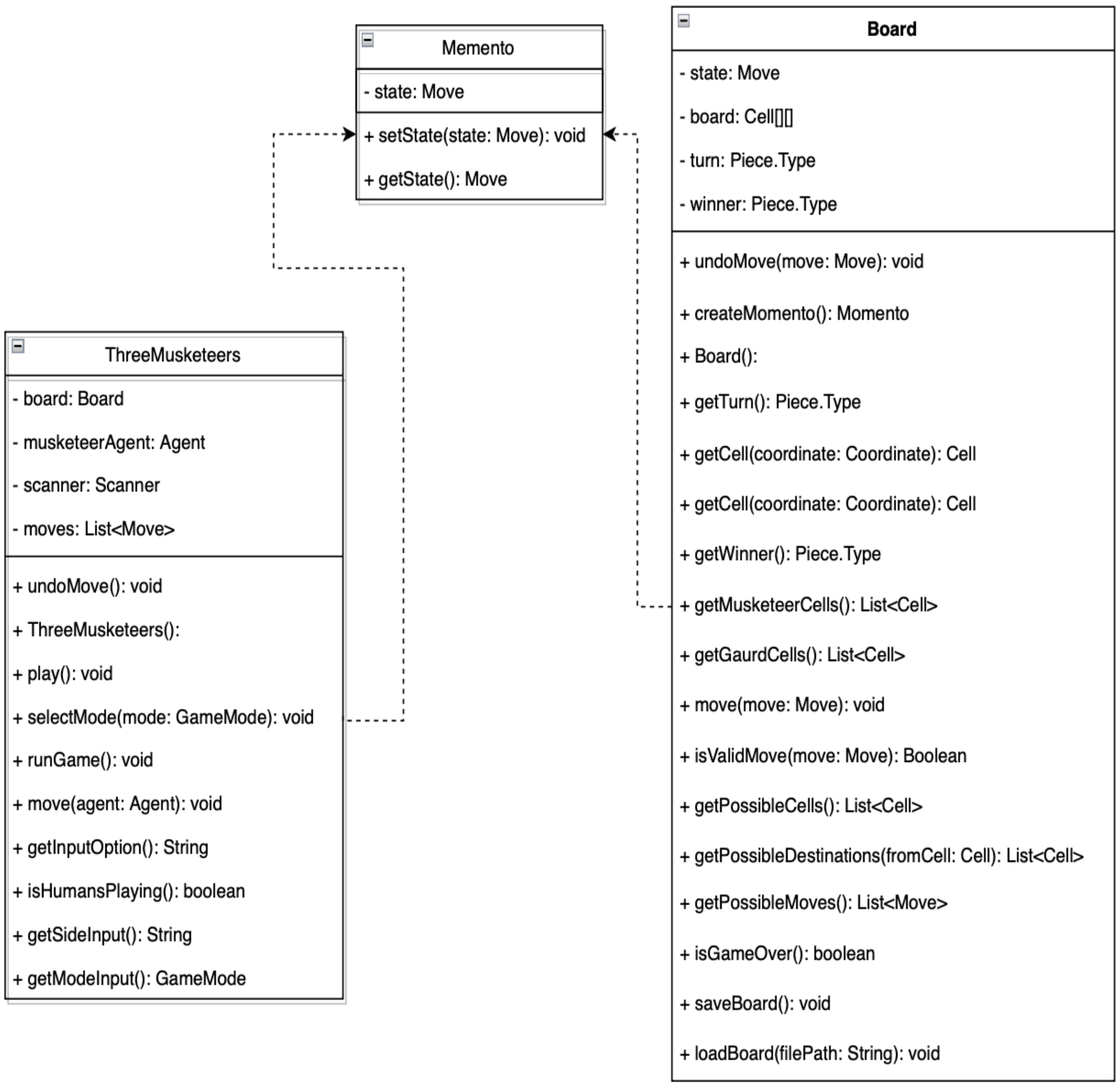
Memento Pattern

Description

The Meneto pattern is used when the client makes an error in their code, or when the client needs the past state of an object. Since errors are common in code, the memento pattern is a very usual way to solve this issue, typically made with an undo button or undo class.

Typically, in the memento pattern, there is a memento class, which stores the memory of the states internally, and is used to get the object again at a later time. One of the biggest advantages of this pattern is that it stores these objects without violating encapsulation. Then there is the originator, which creates the memento object and uses it to restore mistakes, and there is the caretaker, which keeps the memento object. Thus, pattern is used to save previous objects of a class, and is used to restore these states that were saved.

In our implementation of the 3 musketeer game, we will have an undo function, which will restore the previous state of the board, which will be best implemented with the memento pattern. To implement this into the project, the ThreeMusketeer class will act as the caretaker class, which will store the memento object, in an attribute called moves. Then, the board class will act as the originator class, which returns the undo state, and performs the undo move. In the board class, this is done through the method undoMove(), which takes the move that was undone and performs it. Lastly, I would make a new Memento class, which saves the undo states, in attributes called state. A representation of the use of the memento pattern can be found below.


Memento
- state: Move
+ setState(state: Move): void
+ getState(): Move
Board
- state: Move
- board: Cell[][]
- turn: Piece.Type
- winner: Piece.Type
+ undoMove(move: Move): void
+ createMomento(): Momento
+ Board():
+ getTurn(): Piece.Type
+ getCell(coordinate: Coordinate): Cell
+ getCell(coordinate: Coordinate): Cell
+ getWinner(): Piece.Type
+ getMusketeerCells(): List<Cell>
+ getGaurdCells(): List<Cell>
+ move(move: Move): void
+ isValidMove(move: Move): Boolean
+ getPossibleCells(): List<Cell>
+ getPossibleDestinations(fromCell: Cell): List<Cell>
+ getPossibleMoves(): List<Move>
+ isGameOver(): boolean
+ saveBoard(): void
+ loadBoard(filePath: String): void
ThreeMusketeers
- board: Board
- musketeerAgent: Agent
- scanner: Scanner
- moves: List<Move>
+ undoMove(): void
+ ThreeMusketeers():
+ play(): void
+ selectMode(mode: GameMode): void
+ runGame(): void
+ move(agent: Agent): void
+ getInputOption(): String
+ isHumansPlaying(): boolean
+ getSideInput(): String
+ getModeInput(): GameMode

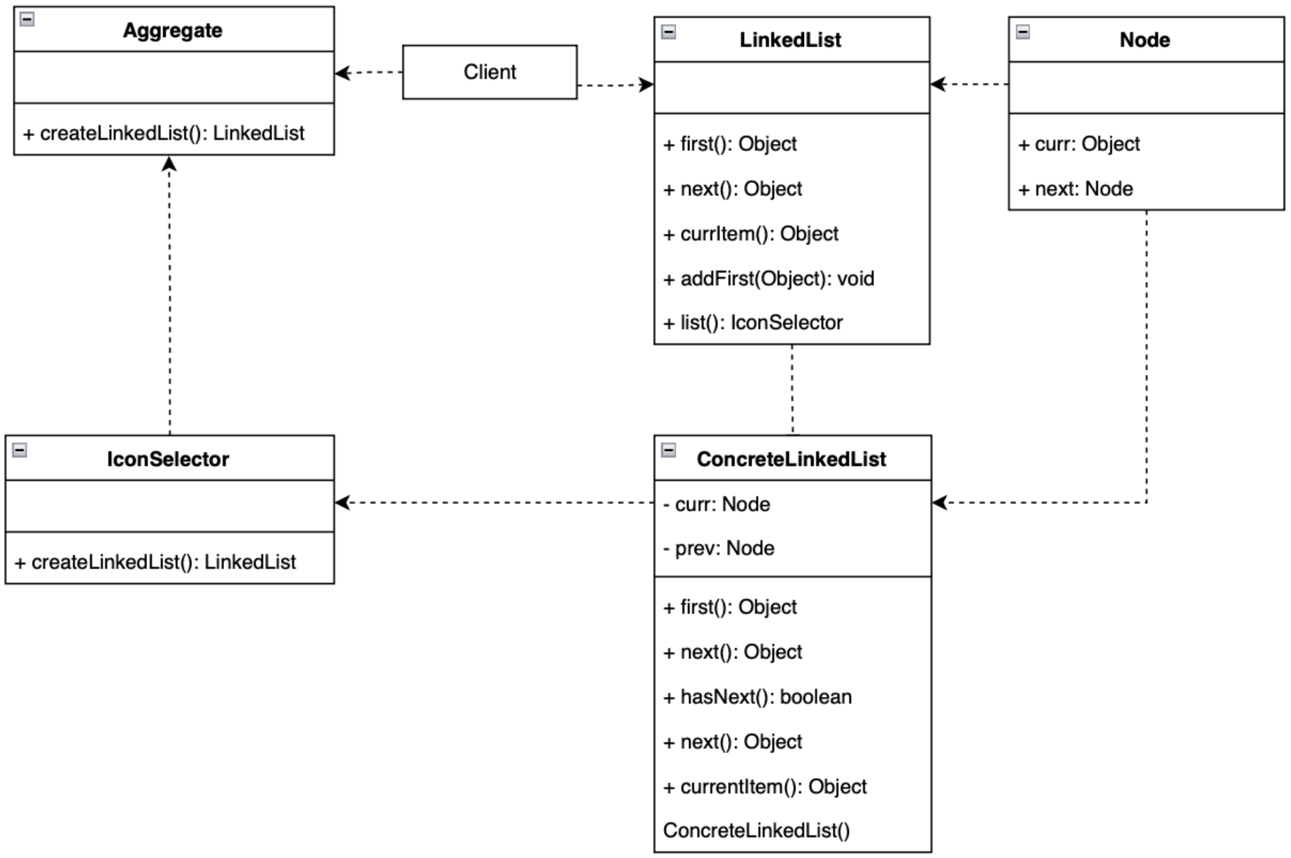
Iterator Pattern

Description

The Iterator pattern is one of the most frequently/commonly used patterns. It is often used when it is required to traverse through a collection of objects in a non destructing way. Some examples of collections are ArrayList, HashMap, PriorityQueue, TreeSet and LinkedLists. There also can be user defined collections that can be iterated through such as a list of songs. The goal is to be able to use some abstraction that is going to encode behaviours that are in common across everything that needs to be iterated over. In short, it leverages an interface that encodes the kind of behaviors that are common across everything that must be iterated. This is beneficial as we can use that same syntax to iterate over any collection of any object we want. They allow the implementations to be seamlessly invisible so they facilitate object hierarchies. One can make specialized iterators to fit the features of the objects you have to iterate over. The collection is going to depend on the iterator interface.

The iterator has an iterator that points to the beginning of the collection and has a next function that iterates over the objects in that collection. The iterator pattern has abstract classes for the iterator and Aggregate class. Next it has the concreateIterator class as an implementation of the iterator interface. Lastly, it has the concreteAggregate class as an implementation of the aggregate interface.

In our ThreeMusketeer game, we plan on using the iterator pattern to let the user select an image to play the game in. We plan on using a collection of images the user can select from, where they use LinkedList to traverse through the icons, and select one. Thus we will first make an iterator interface, called Linkedlist. Next we will make the ConcreateIterator, which we will call ConcreteLinkedlist. Next we will make the Aggregate interface, and then make the ConcreateAggregate which we will call IconSelector. Lastly, to store the images, we will make an additional node class. A representation of the use of the iterator pattern can be found below.


Aggregate
+ createLinkedList(): LinkedList
Client
LinkedList
+ first(): Object
+ next(): Object
+ currItem(): Object
+ addFirst(Object): void
+ list(): IconSelector
Node
+ curr: Object
+ next: Node
IconSelector
+ createLinkedList(): LinkedList
ConcreteLinkedList
- curr: Node
- prev: Node
+ first(): Object
+ next(): Object
+ hasNext(): boolean
+ next(): Object
+ currentItem(): Object
ConcreteLinkedList()

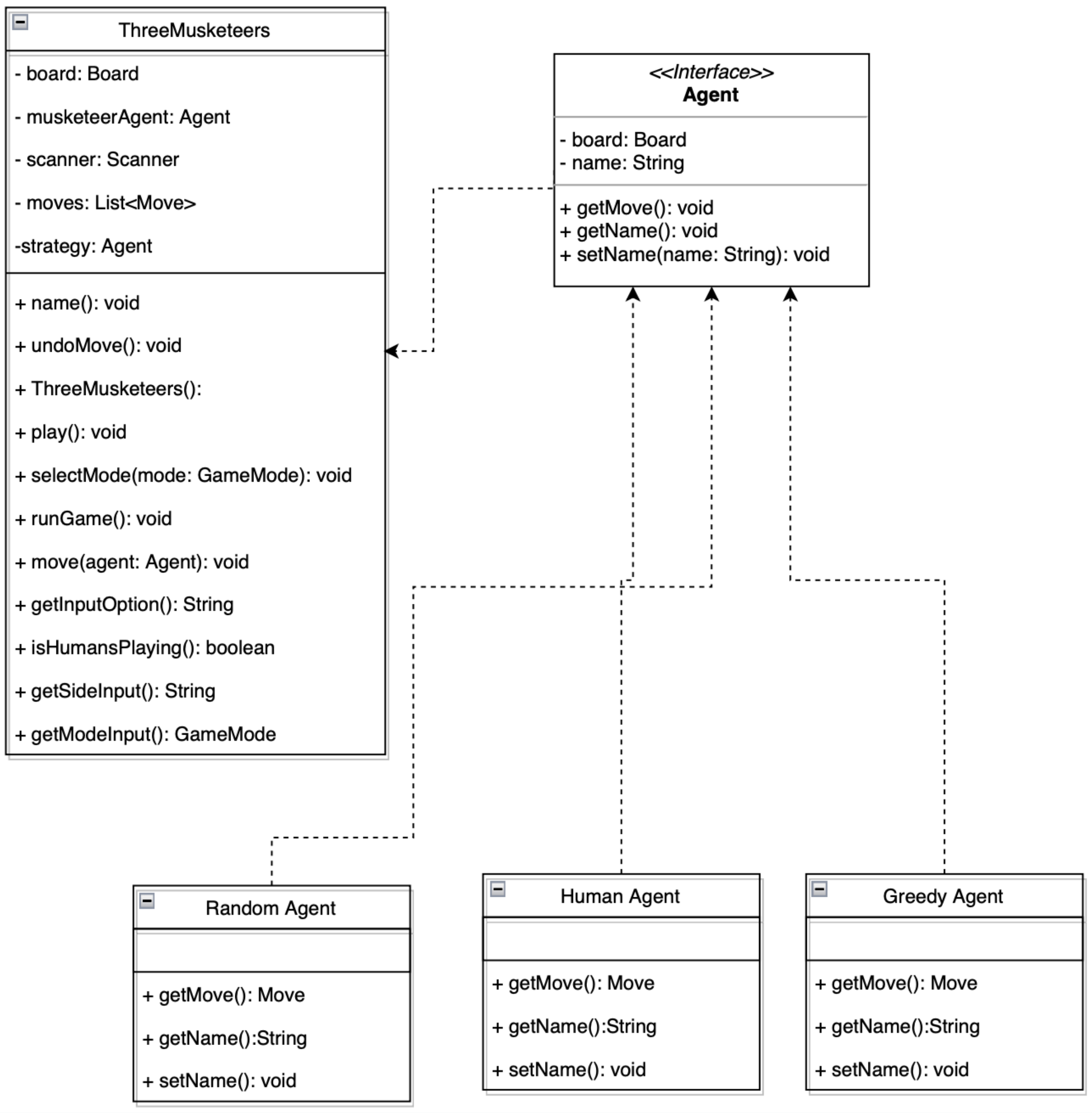
Strategy Pattern

Description

The strategy pattern is used when there are algorithms that differ by their behaviors, and an algorithm is chosen at runtime. Thus there is one interface class, which will have concrete classes that will implement these behaviors. In this pattern, it is better to divide up classes so each one has a different behavior when the code is run.

When implementing the strategy pattern, we have the strategy class, which is the interface which is shared by the supporting classes. Next, we have the concreteStrategy classes, which implement the strategy classes.

In our 3 musketeers game, we decided to use this pattern to set/change the name of the player. If it is humanAgent selected, then the client gets an opportunity to select their side's name. Otherwise, the opposing side will get a name depending on the mode selected. Thus, the humanagent, greedy agent, and the random agent were the concreteStrategy classes, since they were all agents, but have different implementations for getting the names method that was calculated. Furthermore, the Agent class would be the interface, where the method each concreteStrategy would have was getName(), setName(). The strategy pattern is the best for selecting the agent's name because this pattern makes it easier to switch between the classes during runtime, and provides us with cleaner code. A representation of the use of the strategy pattern can be found below.


ThreeMusketeers
- board: Board
- musketeerAgent: Agent
- scanner: Scanner
- moves: List<Move>
-strategy: Agent
+ name(): void
+ undoMove(): void
+ ThreeMusketeers():
+ play(): void
+ selectMode(mode: GameMode): void
+ runGame(): void
+ move(agent: Agent): void
+ getInputOption(): String
+ isHumansPlaying(): boolean
+ getSideInput(): String
+ getModeInput(): GameMode
<<Interface>>
Agent
- board: Board
- name: String
+ getMove(): void
+ getName(): void
+ setName(name: String): void
Random Agent
+ getMove(): Move
+ getName():String
+ setName(): void
Human Agent
+ getMove(): Move
+ getName():String
+ setName(): void
Greedy Agent
+ getMove(): Move
+ getName():String
+ setName(): void