Design Patterns of Three Musketeers

11.29.2021

Group Member (Javajaguars):

Zhanteng Zhang

Andrey Valkov

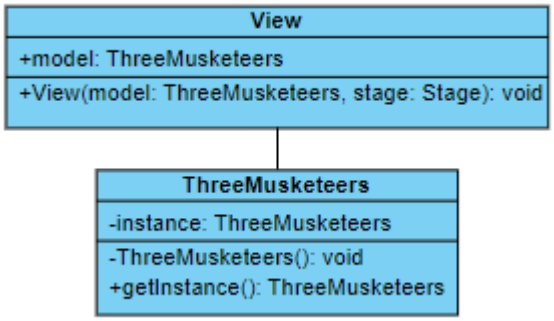
Chiung-Li Wang

Chutong Li

Table of contents

Singleton Pattern

This is a purely aesthetic pattern as we will operationally never be instantiating multiple instances of the ThreeMusketeers class, but we are nonetheless using this pattern to ensure that even if someone tries instantiating multiple of these classes only 1 will ever actually exist at a time.

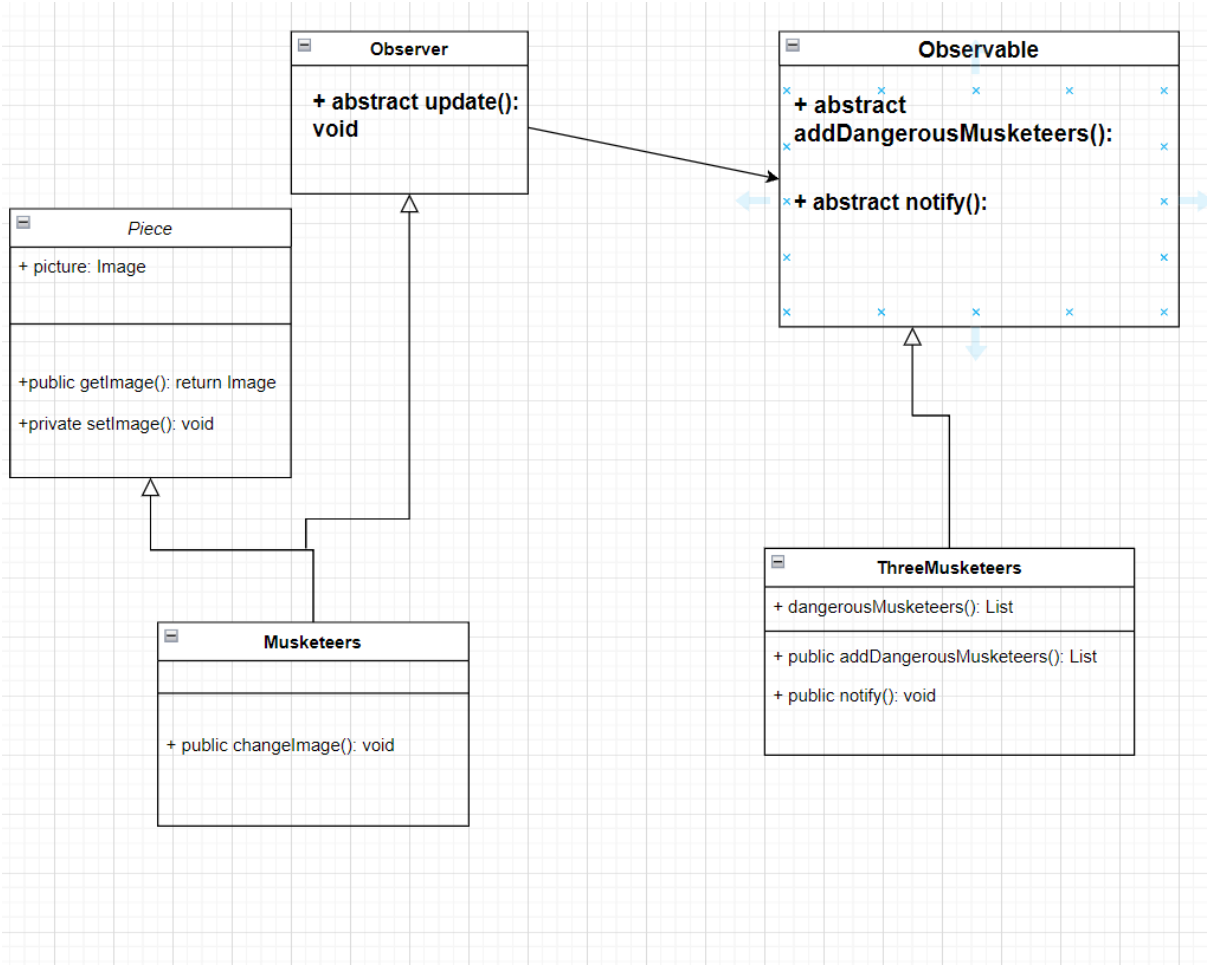


Observer Pattern:

We will use the observer Pattern to implement the picture change of Musketeers when there are exactly two musketeers on the same row or the same column. Because the picture of the musketeers is changed according to the current situation of the board game, we think the observer pattern is the best design pattern to implement this.

Once there is a new move happening in the game, the game will check if there are any dangerous musketeers on the board and if there is, it will add it to the dangerousMusketeers list. The dangerousMusketeers list helps us attract the musketeers that need to change their image. And the notify function will remind the dangerous musketeers to update their images at the next round of the game.

Observer patterns help us to follow the current situation of the board game once there is a new move and make our code more efficient and clean.

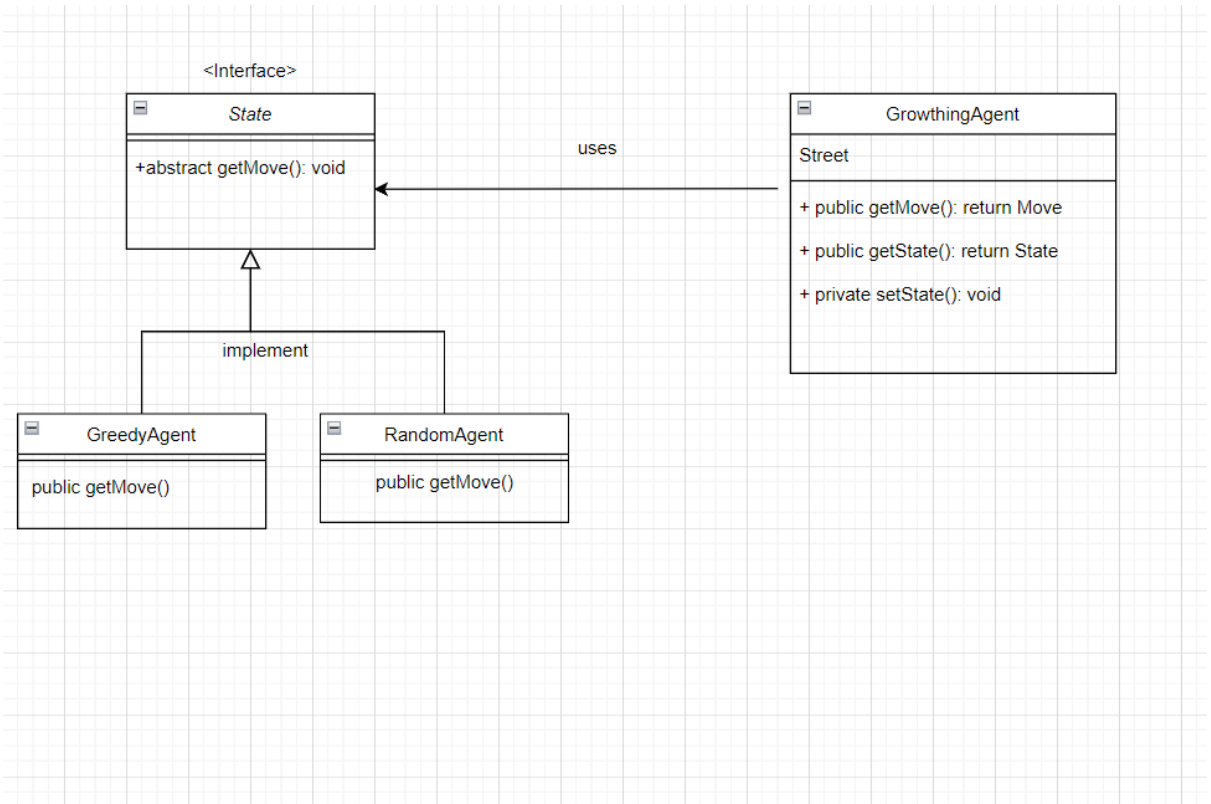


State Pattern:

Instead of a random agent and greedy agent, we make a new agent which is called the growthing agent. If you win more rounds of the game, the stronger this agent will be, but it is not unlimited, it is still weaker than the greedy agent.

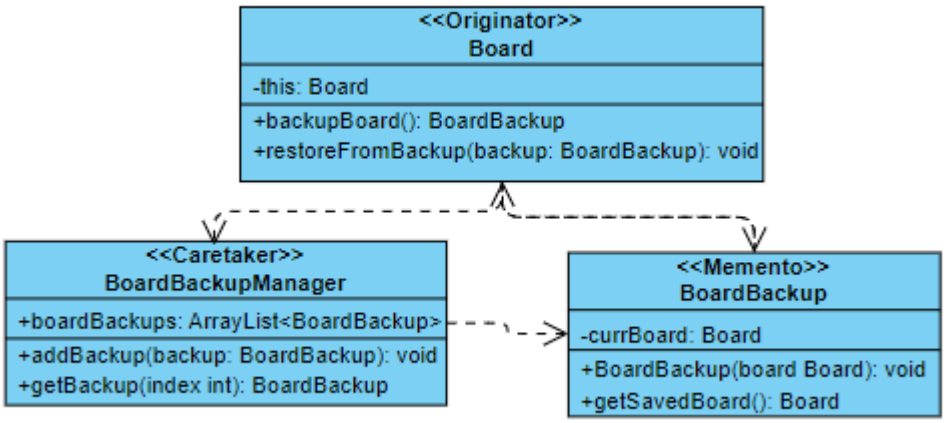
We use a count method to count the current step of the current game in the ThreeMusketeers class. And create an interface to implement the state pattern. If the player's round count is less than or equal to 1 time, the state will be the RandomAgent one which uses the getMove from the RandomAgent to be friendly. Then if the round count is less or equal to 10 times, randomly set the GrowthingAgent state as half RandomAgent and half GreedyAgent, which means there is 50% using the getMove() from the GreedyAgent or RandomAgent. Finally, if the round count is larger than 10 times, it will have a 80% chance to use the getMove() from the GreedyAgent.

We use this way to make the player feel like the robot agent seems like learning from the game, but this is not real. And the difficulty of the board game should be between the Random and the Greedy.



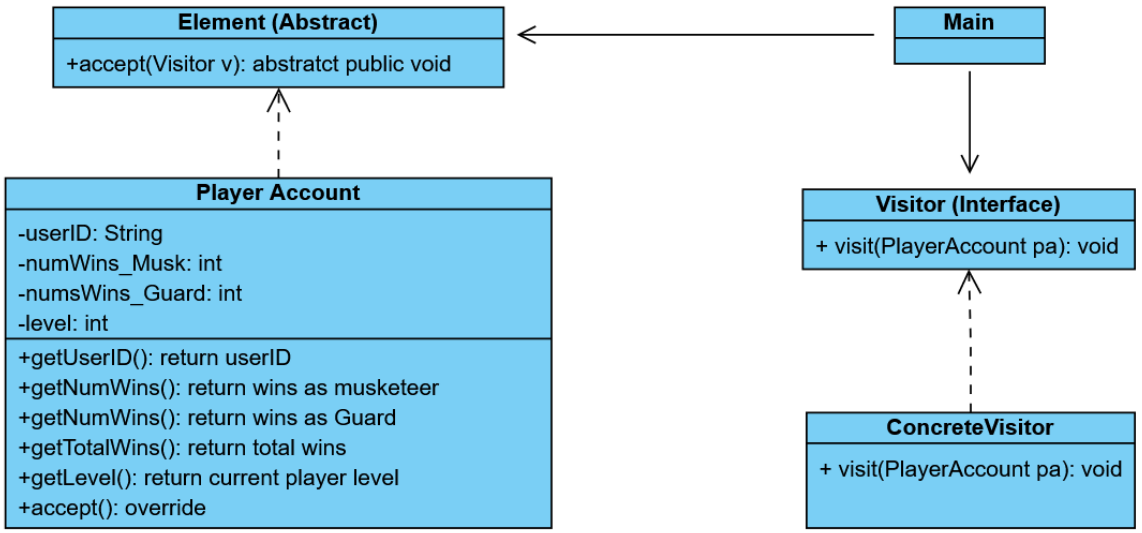
Memento Pattern

This design pattern provides a canned approach to saving states of any given object and can be implemented more easily than a set of ad-hoc arraylists. Essentially, each time a move is to be saved the Board class will create a new BoardBackup object and append said object to an arraylist contained within a BoardBackupManager object. Whenever a move is to be undone, Board will obtain the BoardBackup object from BoardBackupManager and make its current instance ("this") address point to that of the saved BoardBackup. This is not as efficient as storing a simple arraylist of moves and alternating whose turn it is, but it is a much more easily scalable approach to saving states.



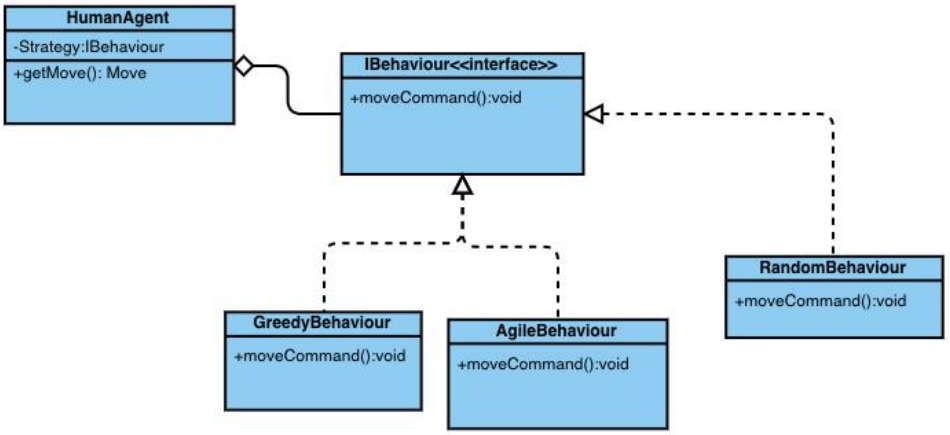
Visitor Pattern

By using this design pattern the user would be able to choose to login the game with their username, or start up with a new user ID. In each of the accounts there would be records of the times of how many games the player has won and lost and the level where they are at right now. Everytime after one game is over, one related data will be added up by one (NumWins or NumLose). If the user chooses to start with a new account, all the data will be set up initially as zero (e.g. numWins_Musk, numWins_Guard, totalWins...). Other than this case, if the user starts as a guest, all data will be set up as zero and will not change over time. It is important to note that the game class which implements the element interface will be able to accept the player's [data storage] class and more easily internalize its data as opposed to having to make all of its methods public so that the player class could call them externally.



Strategy Pattern

This design pattern will be implemented on choosing different strategies for human users. It helps the user to make a turn without thinking, in other words, it helps the user to decide a turn. Whenever a human user doesn't know how to make a suitable turn, the user will use this strategy and choose one of those three behaviours(Greedy, Agile, Random) to help him to decide.



Iterator Pattern

This design pattern will be implemented on counting the rounds of a game. It helps the user to count the rounds when the user want to know how many rounds he has played. With each move, the arraylist in Count will be added an element, and when the user wants to know the current number of games, count() will be called to output the current rounds of games.