Design Patterns of Three Musketeers

12.06.2021(final version)

Group Member (Javajaguars):

Zhanteng Zhang

Andrey Valkov

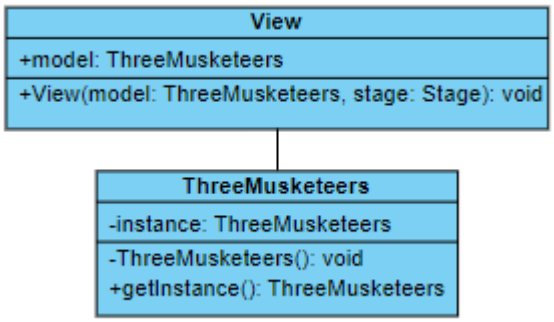
Chiung-Li Wang

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.

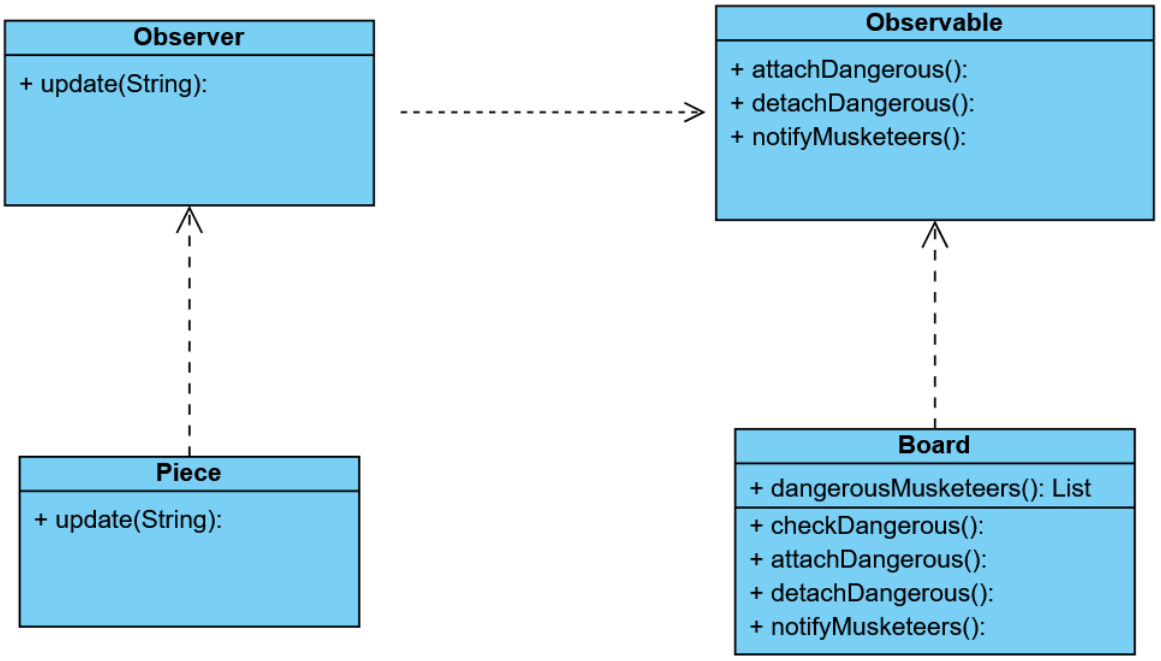


Observer Pattern:

We will use the observer Pattern to implement the sign change of Musketeers when there are exactly two musketeers on the same row or the same column. Because the sign 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 signs. And the notify function will remind the dangerous musketeers to update their signs 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.

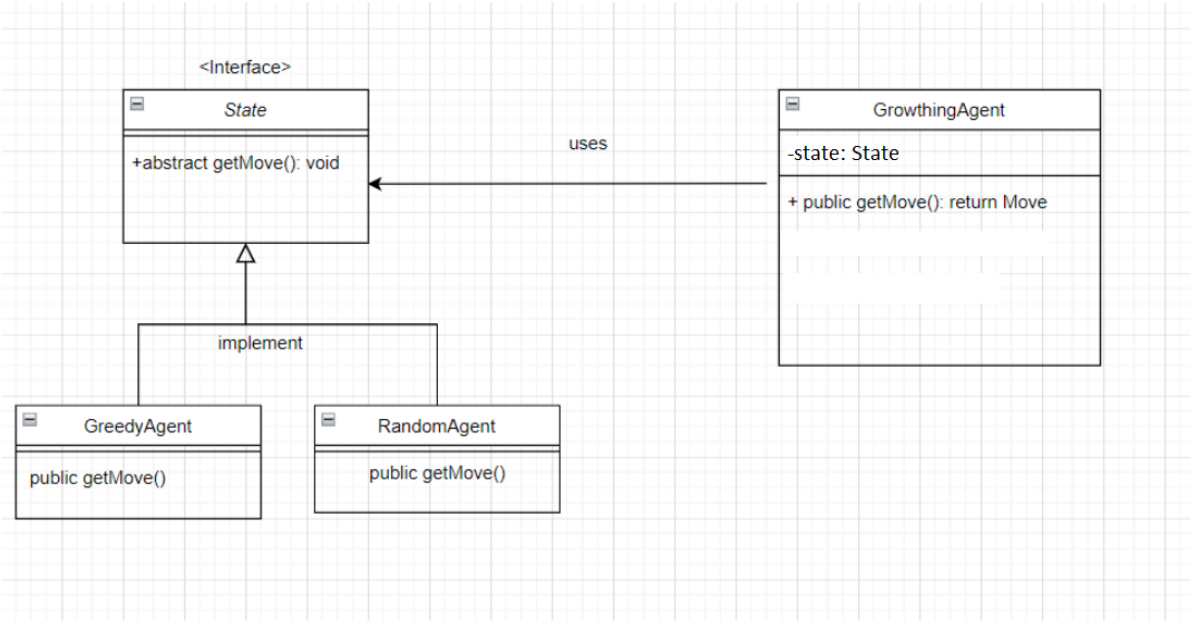


State Pattern:

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.

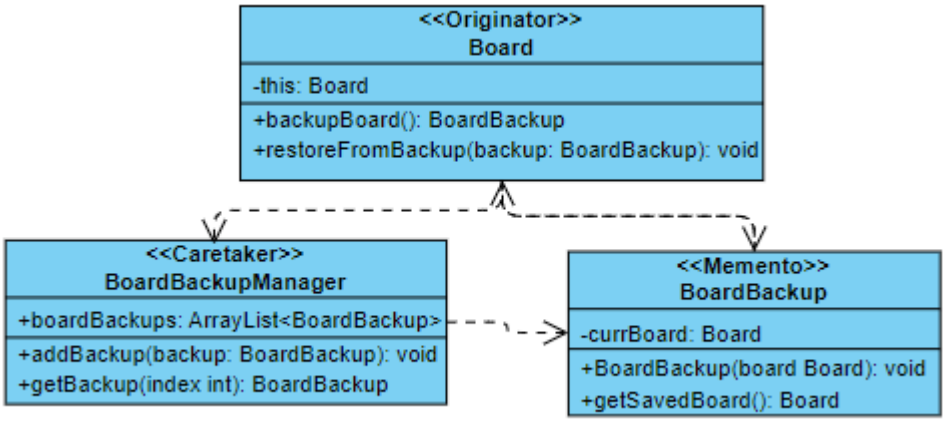
We use a count method to count the current step of the current game in the ThreeMusketeers class. And create an interface to implement the state pattern. If the player's round count is less than or equal to 1 time, the state will be the RandomAgent one which uses the getMove from the RandomAgent to be 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.



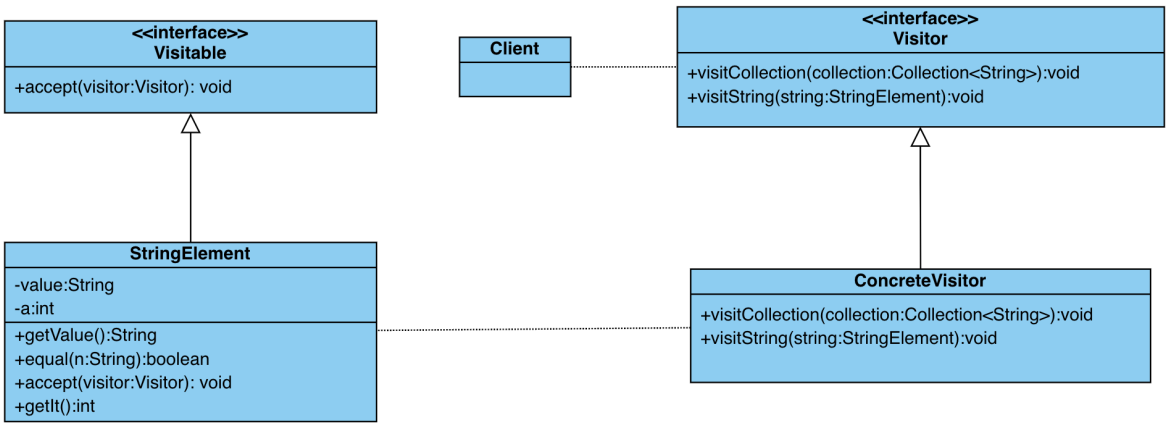
Memento Pattern

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 assigned username(user name:applepear, password:110), or just play as a visitor. If the user chose to log in, they will get their times of play. The source is recorded in a txt file name “source.txt”, the content of the source will be translated into a collection and visited by the ConcreteVisitor. Then visitString(string:String) will be called and so does the StringElement, the visitable will accept the vistitor and if equal returns true, int a will increase by 1. We get the times of play by the assigned user by call getIt():int.



Iterator Pattern

Every time a move is made, the current turn type is recorded in the ArrayList. When the user types “C”, a new object Count is created, with the arrayList of records as the source and guard as the target. The count function in class Count is then called to output the number of rounds currently in progress.
During this time, the iterator pattern is used for iterating the elements in the ArrayList. In the count () function, a new iterator is created by createIterator(), and the next () in ConcreteIterator and if conditions in count() determine if there are any more target elements in the list. Whenever there are, the int count is going to increment by one through the count function to indicate that we're adding one game.