Design Patterns

The Four Musketeers: Ayesha Siddiqa, Guy D'Mello, Piyush Razdan, Maryum Raina

Table of Contents

Strategy Pattern

Description

The strategy pattern is used for a group of classes that differ only in their behaviour. It encapsulates different algorithms for the same behaviour in separate classes that implement a common interface. For the Three Musketeers game, the strategy pattern will be used to encapsulate the getMove() method for the HumanAgent, RandomAgent, and GreedyAgent.

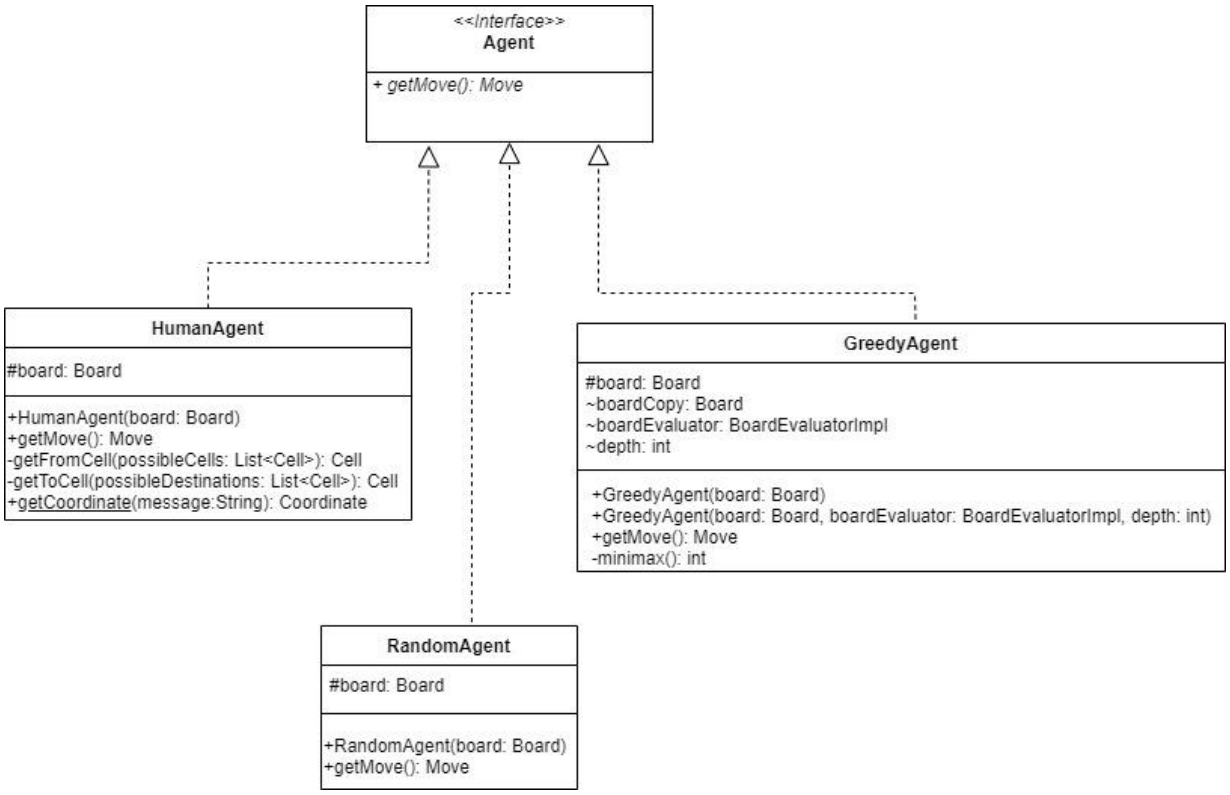
Implementation

There will be an interface called AgentMove. This is to avoid confusion with the Move class. The AgentMove interface will have one abstract method called getMove(). There will be 3 concrete classes that implement AgentMove called RandomAgent, GreedyAgent, and HumanAgent.

What It Solves

Using the strategy pattern to encapsulate the getMove() will ensure that the program favours composition over inheritance. There is no need to have a superclass for all the children of the Agent class since they only share one method in common. Their implementations of the method and even their instance variables are different. Using the strategy pattern in this way removes unnecessary hierarchies and allows for more flexibility in the distinct behaviours of the various agents.

UML Diagram For Strategy Pattern



Builder Pattern

Description

The builder pattern is used to separate object construction from its representation. For the Three Musketeers game, the builder pattern will be used to separate the construction of Cell objects by making use of a builder class, CellBuilder. The Director will be established to be the Board class that makes use of the builder class.

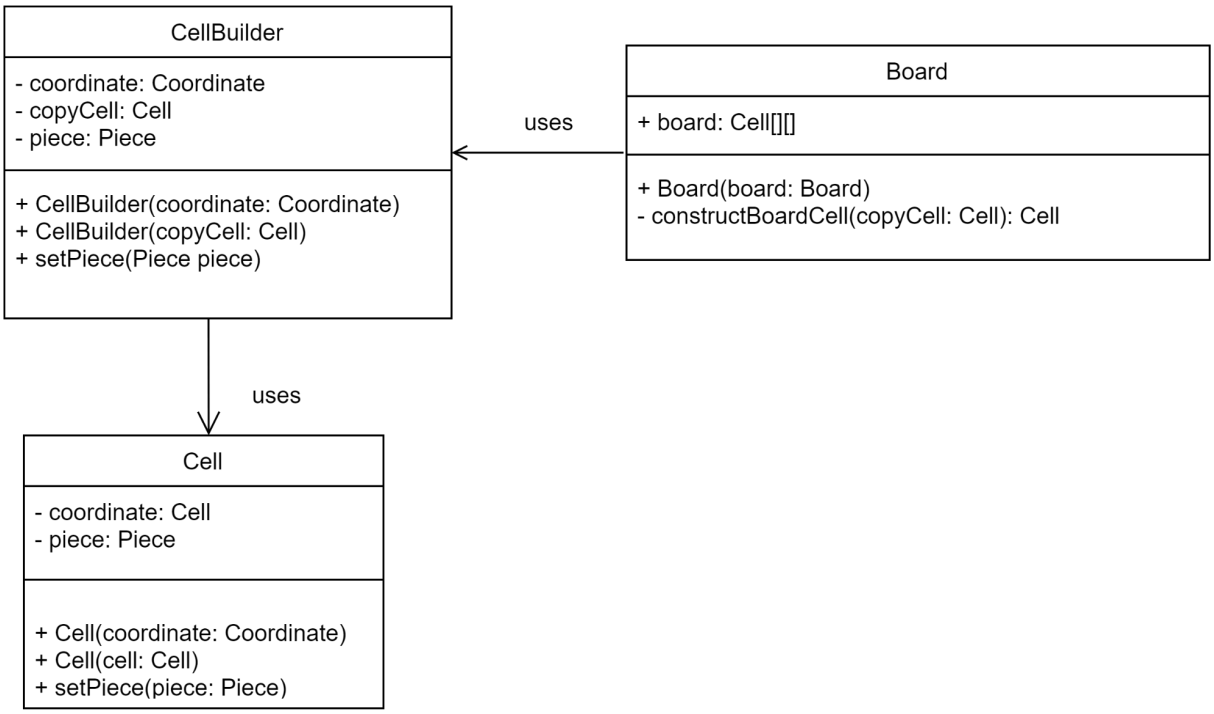
Implementation

The builder pattern is going to be used when first creating the cells of the Board class when calling the board constructor. In the board constructor, we will utilize a private method called ConstructBoardCell() which will return a product created by the builder class, CellBuilder. CellBuilder will return a Cell product based on specified attributes all the while using the Cell class.

What It Solves

The same construction to create a board cell in the Board class is applied each time the CellBuilder is called. This encapsulates creating and assembling complex parts of an object into a separate Builder object. It also allows a Builder to delegate object creation instead of directly building the objects.

UML Diagram For Builder Pattern



Iterator Pattern

Description

The iterator pattern adds moves to the MoveRepository to allow a user to look at previous moves that were done in the game. It uses a move iterator which iterates through the moves list and allows the user to view their own or the computer's previous move in the console. The moveIterator.java is used to traverse through the moves list accessing the moves in a reverse fashion.

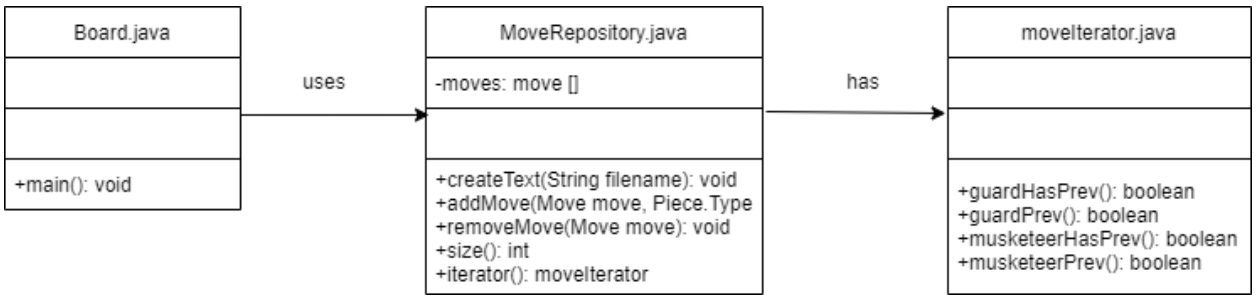
Implementation

Add a move to the list from the moves stack from ThreeMuskeeters.java. Looks at previous moves and prints the previous move that occurred in the game. Stops when there are no more previous moves to view.

What It Solves

It allows users to look at both humans and computers moves and shows which move was last occurred. You can do this consecutively to see how the game is played out in reverse.

UML Diagram For Iterator Pattern



Mediator Pattern

Description

The Mediator Pattern uses a repository as a means of communicating with the move log. The move repository will transfer move messages to the MoveLog.java file and keeps it updated whenever a valid move is conducted by a Musketeer or Guard agent. The concrete mediator is the movesRepository.Java which keeps reference to the moves stack and communicates it to the MoveLog.java.

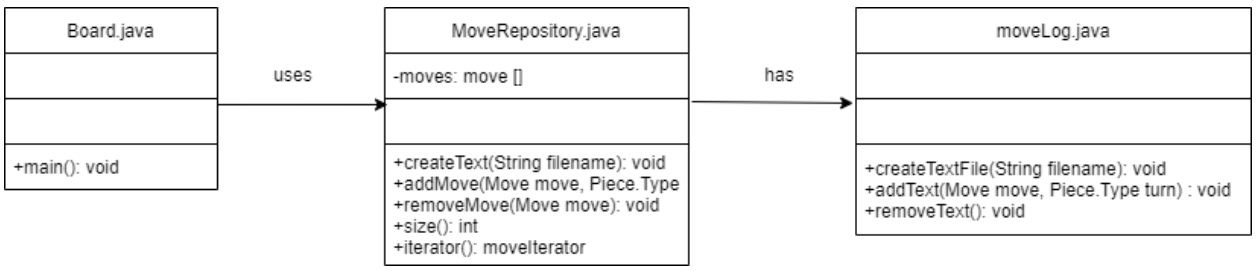
Implementation

The Mediator Pattern uses the board.java and uses its move function to create a repository class to store moves in a list which is updated from the moes stack. Then each time a move is added to the stack, the move list is updated and the list of moves appends each move to a file in MoveLog.java. The file needs to be created and then updated after each move representation.

What It Solves

This allows the use of a move log history of the game which will allow players to view the moves that were done throughout the game and look at their opponents moves.

UML Diagram For Mediator Pattern



Singleton Pattern

Description

The Singleton pattern defines an instance operation which can be accessed globally by clients. For the ThreeMusketeers board game it will use Board.java's board function to access an instance of the board in the ThreeMusketeers class. This will return an instance of the board to ThreeMusketeers which is now accessible through getBoard().

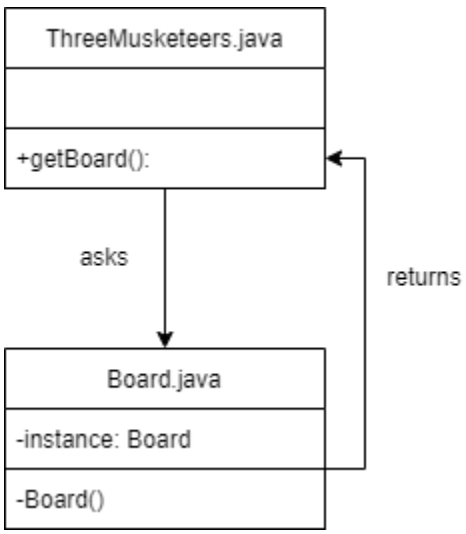
Implementation

There will be a method which calls upon the Board.java Board() function.

What It Solves

Allows accessing the board in ThreeMusketeers which can be used for changing the appearance of the board and updating its pieces (create custom pieces for example).

UML Diagram For Singleton Pattern



Factory Pattern

Description

The factory Pattern is used to create objects without exposing the instantiation process to the client. The client class calls the factory class which returns a product object. For the Three Musketeers game, the factory pattern will be used to create pieces for the cells during the creation process of the board.

Implementation

The abstract product class will be the Piece class which will be implemented by the Guard class, Musketeer class, and EmptyCell class. The factory class will be called PieceFactory. It will have one method called createPiece() which takes in a String object that can be equal to “Guard”, “Musketeer”, or “EmptyCell”. The createPiece() method will then return a concrete Piece object according to the specified input String.

What It Solves

The implementation of the factory pattern will allow the Board class to fill the board cells with different Piece objects without having to instantiate each piece. It will allow the Board class to receive Piece objects without having to interact with the Guard class, Musketeer class, or EmptyCell class separately. Utilizing the factory pattern in this way will reduce tight coupling between the Board class and the Piece subclasses and alleviate the responsibility of creating Piece objects for the Board class.

UML Diagram For Factory Pattern