Software Design Pattern Applications in Chess

Wes Beard & Michael Leonard

CSI-340

12/7/2020

About

We were inspired to do this project due to our mutual interest in chess, as well as its current popularity of the game. We really wanted to make some kind of game to implement the functionality of the command pattern and its ability to replay and rewind actions. Chess seemed like the perfect subject for this since it's very tactical so undoing and replaying can be useful for mistakes and analysis. Additionally, there were many other patterns that we could use in the making of the program since there was a lot of data to be stored relating to the pieces, board, and gameplay. We also wanted to make something visual which is why we chose the graphics library Processing to allow us to display the chess game as well as implement interactivity through buttons and other user interface elements. Another goal we wanted for the project was to use the design patterns toward an actual product as opposed to having code singly devoted to showing off a specific pattern. This led us to adapting the patterns to what we needed them to do and making them work together to suit our specific needs. Overall it was great to use the strategies we've learned to create something that we were really passionate about.

Software Features

The project is a fully functional and featured version of chess, including:

- Basic gameplay: moving, capturing, putting in check, getting out of check, castling, etc.

- Display elements such as a chess clock that counts down for each player, pausing for the other and losing if out of time, as well as a display of all pieces currently held by either side.
- Interactive elements such as buttons for either side resigning or in the event of a draw as well as popup windows for information display.
- Move undo and redo functionality during gameplay which can skip forward or back between current and past board states as well as move from a past position and continue from there.
- Load an FEN file, or our custom FEN format, to replay any pre-existing game catalogued in that format, stepping through every move in the game.

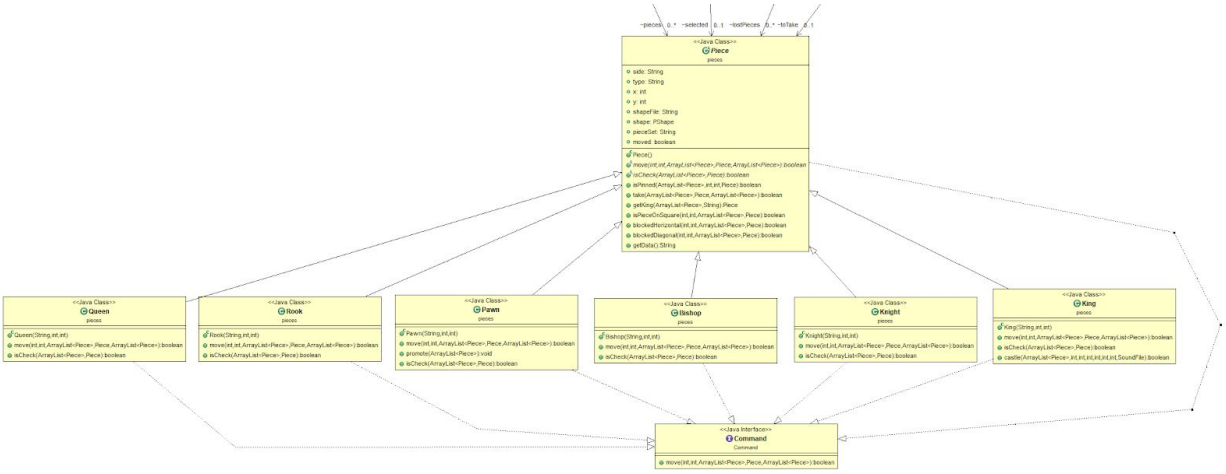
Design Patterns Used

Template

We chose this pattern specifically as the basis for our piece system. The pattern allowed us to easily keep common code in the abstract parent class which came in handy for actions that every piece could such as checking the king, taking a piece, or being pinned to the king. There was also a lot of useful utility code that all of the pieces needed such as a function to return the enemy king that was used to see if each individual piece's movement put the king in check for any given move. In addition to this the flexibility of the pattern was useful for each of the different piece types since, while they have many of the same functionality, they have some key differences as well. This includes how each piece moves and takes pieces, such as the pawn who can only move forward and can only take on a diagonal, as well as piece specific actions like castling

with the king which can't be done with any other piece. This pattern helped us cut down on duplicate code and streamline our chess piece classes.

UML



Code Example

** Specific code has been removed from functions for length but showing what code would be common and what was implemented by the child classes*

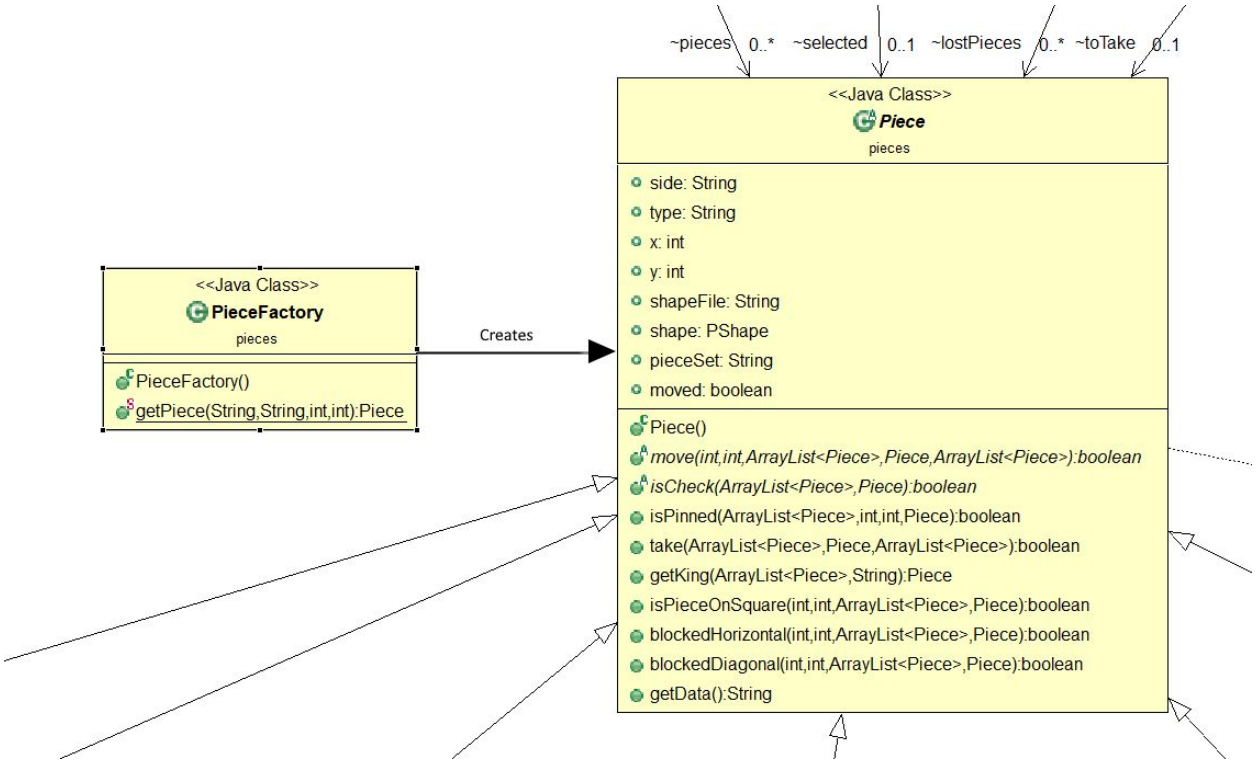
```
public abstract class Piece extends PApplet implements Command {

     public String side;
     public String type;
     public int x;
     public int y;
     public String shapeFile;
     public PShape shape;
     public String pieceSet = "tatiana";
     public boolean moved = false;

public abstract boolean move(int targetX, int targetY,
ArrayList<Piece> pieces, Piece toTake, ArrayList<Piece>
lostPieces);
```

```
public abstract boolean isCheck(ArrayList<Piece> pieces, Piece
toTake);


public boolean isPinned(ArrayList<Piece> pieces, int targetX,
int targetY, Piece toTake) {
     // function code here
}

public boolean take(ArrayList<Piece> pieces, Piece toTake,
ArrayList<Piece> lostPieces) {
     // function code here
}

public Piece getKing(ArrayList<Piece> pieces, String side){
     // function code here
}

public boolean isPieceOnSquare(int pieceX, int pieceY,
ArrayList<Piece> pieces, Piece toTake) {
     // function code here
}

public boolean blockedHorizontal(int targetX, int targetY,
ArrayList<Piece> pieces, Piece toTake) {
     // function code here
}

public boolean blockedDiagonal(int targetX, int targetY,
ArrayList<Piece> pieces, Piece toTake) {
     // function code here
}

public String getData() {
    // function code here
}
}
```

Factory

Only after we had implemented the template pattern and began to create pieces did we realize that it was very cumbersome to create each individual piece as an object to add to the piece array, from here we realized that something like the factory pattern would be perfect for the scenario. The factory fit in well with our existing template infrastructure so it was only a matter of having it create the pieces. Implementing the factory pattern greatly streamlined our piece creation process, with each piece only requiring one line and the pawns using loops instead of individual lines which previously wasn't possible with our old method. This cut the lines needed to create all of the pieces in their starting positions down from 36 lines, one line for every piece, to only 22. Our program uses the factory frequently to set piece positions for new and restarted games as well as create new pieces when needed so it's implementation no doubt increased efficiency as well as usability. In addition to these benefits it also allowed us to only need to access the factory itself instead of every single piece type. It would also be easy to add new classes to the structure if we needed to and construct them by simply adding them to the factory.

UML



Code Example

```
public class PieceFactory {


public static Piece getPiece(String pieceType, String sideColor,
int startX, int startY) {
     if (pieceType == null) {
          return null;
     }
     else if (pieceType.equals("P")) {
          return new Pawn(sideColor, startX, startY);
     }
     else if (pieceType.equals("K")) {
          return new King(sideColor, startX, startY);
     }
```

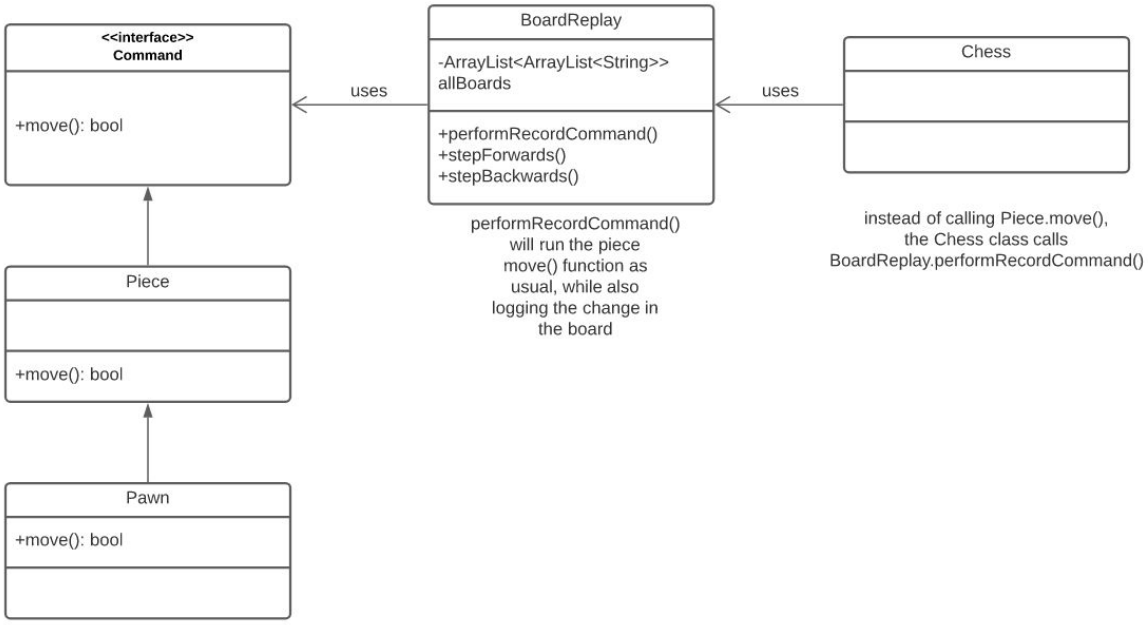
```
        else if (pieceType.equals("Q")) {
                return new Queen(sideColor, startX, startY);
        }
        else if (pieceType.equals("N")) {
                return new Knight(sideColor, startX, startY);
        }
        else if (pieceType.equals("B")) {
                return new Bishop(sideColor, startX, startY);
        }
        else if (pieceType.equals("R")) {
                return new Rook(sideColor, startX, startY);
        }
        return null;
        }
}
```

Command

Attempting to constantly save the board state to be used to step backwards and forwards in time seemed to be unreasonable and very difficult to implement, which is why we used the command pattern. The command pattern acts as an in-between when a piece's .move() function is being called, this way we can store the movement in a variable (in our case we used an ArrayList,) and then pass the normal move function onto whichever Piece is being called. Now, with each move saved in an ArrayList, it's as easy as stepping back and forward through the list when the user would like to stop back or forward in the game. Another unforeseen impact of this pattern is that when later importing games or exporting them, all that needs to be done is to process

whichever file is being imported, turn the format into one that our program can read, and then giving all of that data to the BoardReplay class (the controller of anything command-pattern-related.)

UML



Code

When called from Chess' main class, a move looks like the snippet titled "Chess class."

This attempts to make a move (through BoardReplay), the BoardReplay object will log that move if successful, otherwise it will return the success of the move call (which would be false if the move could not be performed by the Piece.)

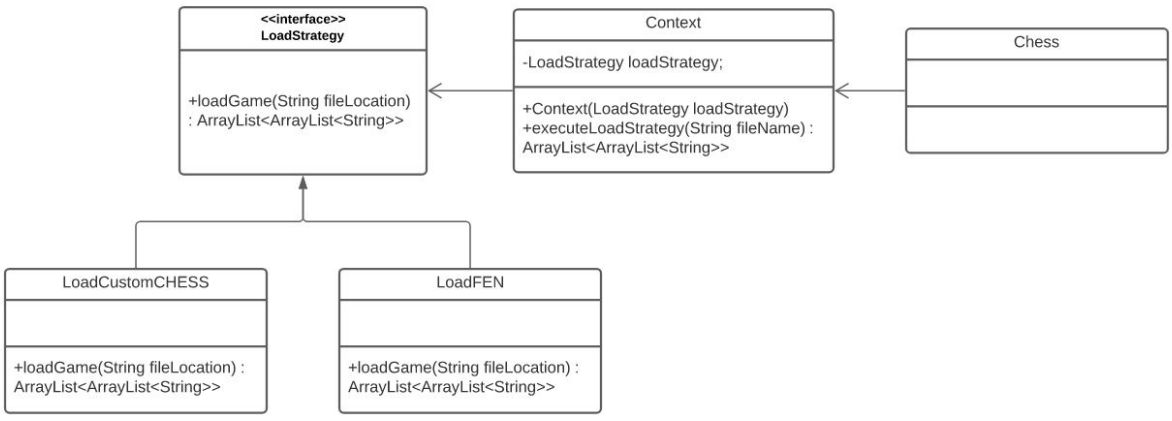
```
// Chess class:


replay = new BoardReplay(pieces, lostPieces);

...

// here, 'selected' is a Piece object

if (replay.performRecordCommand(selected, targetX, targetY,
pieces, toTake, lostPieces))




// BoardReplay class


public boolean performRecordCommand(Piece pieceToMove,
                                    int targetX,
                                    int targetY,
                                    ArrayList<Piece>entireBoard,
                                    Piece toTake) {


        // pass the move call onto the piece like normal

        boolean successful = pieceToMove.move(targetX, targetY,
entireBoard, toTake);


        if(successful) {

          ... // long piece of code to log moves

        }


        return successful;
```

Strategy

The strategy pattern is meant for algorithms that are chosen at runtime. This is perfect for a file loader, since the format is unknown up until the user selects their desired file (whether that be .chess or .fen.) Once the user picks a file to load, the 'Context' class will pass the call onto a specific strategy (LoadFEN or LoadCustomCHESS) depending on the file type.

This pattern was able to be implemented perfectly, since loading a file is independent from any other code we've written thus far. This strategy was implemented last, but wasn't an issue since it works independent from the main code. When the user selects a file, all of the logic gets put onto either LoadCustomCHESS or LoadFEN classes.

UML



CODE

```
public interface LoadStrategy {
    public ArrayList<ArrayList<String>> loadGame(String
fileLocation);
}
```

```
public class LoadCustomCHESS implements LoadStrategy {
     ... // code to load a file
     ... // code to convert that file's text to a format that
our game can read
     // entireGame is returned so that the caller can send this
data to the BoardReplay class
     return entireGame;
}
```

```
public class LoadCustomFEN implements LoadStrategy {
     ... // code to load a file
     ... // code to convert that file's text to a format that
our game can read
     // entireGame is returned so that the caller can send this
data to the BoardReplay class
     return entireGame;
}
```

```
// uses LoadStrategy
public class Context {
    private LoadStrategy loadStrategy;

    public Context(LoadStrategy loadStrategy) {
        this.loadStrategy = loadStrategy;
    }

    public ArrayList<ArrayList<String>>
executeLoadStrategy(String fileName) {
        return loadStrategy.loadGame(fileName);
    }
}
```