Code for the Chess Game

Write the object-oriented code to implement the design of the chess game problem.

We'll cover the following

We've covered different aspects of the chess game and observed the attributes attached to the problem using various UML diagrams. Let's now explore the more practical side of things where we will work on implementing the chess game using multiple languages. This is usually the last step in an object-oriented design interview process.

We have chosen the following languages to write the skeleton code of the different classes present in the chess game:

- Java
- C#
- Python
- C++
- JavaScript

Chess game classes

In this section, we will provide the skeleton code of the classes designed in the class diagram lesson.

> **Note:** For simplicity, we are not defining getter and setter functions. The reader can assume that all class attributes are private and accessed through their respective public getter methods and modified only through their public method functions.

Enumerations and custom data type

The following code provides the definition of the enumeration and custom data type used in the chess game.

`GameStatus`: This enumeration keeps track of the active status of the player and the game, i.e, who wins and whether or not the game is a draw.

`AccountStatus`: We need to create an enumeration to keep track of the status of the account – whether it is active, canceled, closed, blocked, or none.

The `Person` class is used as a custom data type. The implementation of the `Person` class can be found below:

> **Note:** JavaScript does not support enumerations, so we will be using the `Object.freeze()` method as an alternative that freezes an object and prevents further modifications.

```
// Enumerations
const GameStatus = Object.freeze({
  Active,
  BlackWin,
  WhiteWin,
  Forfeit,
  Stalemate,
  Resignation
});

const AccountStatus = Object.freeze({
  ACTIVE,
  CLOSED,
  CANCELED,
  BLACKLISTED,
  BLOCKED
});

// Custom Person data type class
class Person {
    #name;
    #streetAddress;
    #city;
    #state;
    #zipCode
    #country;
    constructor(name, streetAddress, city, state,zipCode, country){
        this.#name = name;
        this.#streetAddress = streetAddress;
        this.#city = city;
```

Definition of enums and custom datatypes

Box and chessboard

The `Box` class holds the piece where the `Chessboard` contains the boxes and has the functionality of updating or resetting the board. The definitions of these classes are provided below:

```
public class Box {
  private Piece piece;
  private int x;
  private int y;
}

public class Chessboard {
  private Box[][] boxes;
  private Date creationDate;

  public List<Piece> getPieces()
  public void resetBoard()
  public void updateBoard()
}
```

The Box and Chessboard classes

Piece

`Piece` is an abstract class that is extended by `King`, `Queen`, `Knight`, `Bishop`, `Rook` and `Pawn`. These derived classes override the `canMove()` function of `Piece`. The definitions of these classes are provided below:

```
class Piece {
  #killed;
  #white;

  constructor(killed, white) {
    if (this.constructor == Vehicle) {
      throw new Error("Abstract classes can't be instantiated.");
    }
    else {
      this.#killed = killed;
      this.#white = white;
    }
  }
  isWhite();
  isKilled(hand);
  canMove();
}

class King extends Piece {
  #castlingDone;

    constructor(killed, white, castlingDone) {
      this.#killed = killed;
      this.#white = white;
      this.#castlingDone = castlingDone;
    }
  canMove() {
        // definition
    }
}
```

Piece and its derived classes

Move

The Move class represents the move that will be taken by the player. It can tell the source and destiation box of the active Piece and whether or not it was a castling move. It also identifies the captured piece. The definitions of these classes are provided below:

```
class Move {
  #start;
  #end;
  #pieceKilled;
  #pieceMoved;
  #player;
  #castlingMove;

  constructor(start, end, pieceKilled, pieceMoved, player, castlingMove) {
    this.#start = start;
    this.#end = end;
    this.#pieceKilled = pieceKilled;
    this.#pieceMoved = pieceMoved;
    this.#player = player;
    this.#castlingMove = false;
  }

  isCastlingMove();
}
```

The Move class

Account, player, and admin

The Account class is extended by the Player and Admin classes.

- The Player class records the player's information by storing the Person object, along with the chosen color, i.e., – whether or not the player is playing with white pieces.
- The Admin class decides whether or not the user is blocked.

The definitions of these classes are provided below:

```
class Account {
  #id;
  #password;
  #status;

  constructor(id, password, status) {
    this.#id = id;
    this.#password = password;
    this.#status = status;
  }
  resetPassword();
}

class Player extends Account {
  #person;
  #person;
  #whiteSide;
  #totalGamesPlayed;

  constructor(person, whiteSide, totalGamesPlayed) {
    this.#person = person;
    this.#whiteSide = false;
    this.#totalGamesPlayed = totalGamesPlayed;
  }
  isWhiteSide(bet);
  isChecked();
}

class Admin extends Account {
  blockUser();
}
```

Account and its derived classes

Chess move controller and the game view

The `ChessMoveController` class validates the moves and responds accordingly. The `ChessGameView` class represents the game view. The definitions of these classes are provided below:

```
class ChessMoveController {
  validateMove();
}

class ChessGameView {
  playMove();
}
```

The ChessMoveController and ChessGameView classes

Chess game

The `ChessGame` class represents the current situation of the game while keeping track of turns and moves, and also decides when the game ends. The definition of this class is provided below:

```
class ChessGame {
  #players;
  #board;
  #currentTurn;
  #status;
  #movesPlayed;

  constructor(players, board, currentTurn, status, movesPlayed) {
    this.#players = players;
    this.#board = board;
    this.#currentTurn = currentTurn;
    this.#status = status;
    this.#movesPlayed = movesPlayed;
  }

  isOver();
  playerMove(player, startX, startY, endX, endY) {
    /* 1. start box
       2. end box
       3. move
```

```
      4. call makeMove() method
    */
  }
  makeMove(move, player) {
    /* 1. Validation of source piece
       2. Check whether or not the color ofthe piece is white
       3. Check if it is a valid move or not
       4. Check whether it is a castling move or not
       5. Store the move
    */
```

The ChessGame class

Wrapping up

We've explored the complete design of the chess game in this chapter. We've looked at how a basic chess game can be visualized using various UML diagrams and designed using object-oriented principles and design patterns.

Back

Activity Diagram for th...

Next

Getting Ready: The H...

Mark as Completed