(Report)

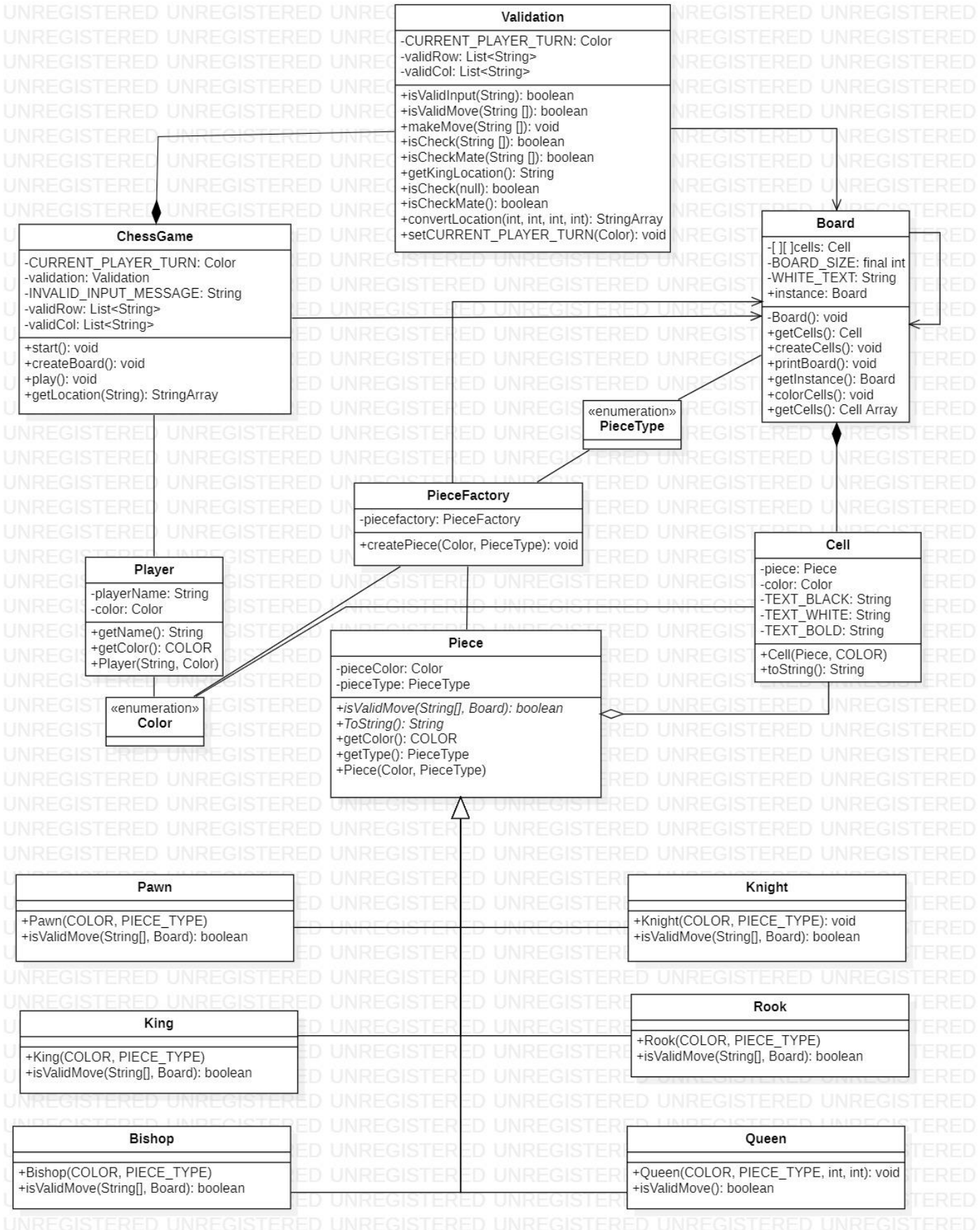
Assignment (Chess Game)

Name: Zain Alabdeen Adelelah Aburayya

Table of contents

Contents

Object-oriented design



To design a class diagram for a chess game, I found it necessary to create 13 classes and include two enumerations to represent the color and type of each piece selected. These classes and enumerations allowed me to effectively model the different components and rules of the game and enabled me to implement a functional chess game.

ChessGame Class

In the class diagram for the chess game, there is an object of the Validator type that is used to ensure the game is played with correct input and moves. This object, along with variables such as player turn and a list of allowed moves, and methods such as start(), createBoard(), and play(), are all included in a single class to achieve cohesion and keep all related game logic together. This helps to keep the code organized and maintainable. The Validator object is particularly important in ensuring the game is played correctly, as it handles input and move validation to prevent any invalid or illegal actions from being taken.

Validator Class

As previously mentioned, the Validator class is responsible for checking the input and verifying the correctness of moves in the chess game. It also checks for instances of a checked or checkmated king and ensures that the chosen piece is able to make the requested move. Keeping the Validator class separate from the Chess Game class helps to achieve cohesion, as each class has a single responsibility. There is also an association relationship between the Validator class and the Board class, which will be discussed further in the next paragraph. By following these design principles, the code for the chess game is easier to understand and maintain.

Player Class

The Player class in the class diagram has two attributes: name and color. The name attribute represents the name of the player and the color attribute represents the color of the pieces that the player is using in the game. The color attribute is an instance of the Color enumeration, which is used to represent the two possible colors that a player can have (white or black). The Player class also has getter methods for both of these attributes, which allow other classes to retrieve the values of the name and color attributes. This can be useful for displaying the player's name and the color of their pieces during the game, as well as for keeping track of which player's turn it is.

Board Class

In the Board class, there is an attribute of the same class, which is used to implement the Singleton design pattern. The Board class also contains several methods that are related to the board and its functions, such as createCells(), printBoard(), colorCells(), getCells(), and getInstance(). These methods all contribute to the cohesion of the class, as they are all related to the board and its properties. The getInstance() method is used to implement the Singleton pattern, which ensures that there is only one instance of the Board class at any given time. This can be useful in ensuring that the game is played on a single, consistent board and can help to prevent any inconsistencies or errors .

There is an attribute that is a 2D array of Cell objects. This attribute represents the cells on the chess board and is used to store information about each cell, such as its color, and the piece (if any) that is occupying it.

Cell Class

The Cell class is another class in the class diagram that is used to represent each cell on the board and contains its own set of attributes and methods. The Board class uses the Cell class to store and retrieve information about the cells on the board and to manipulate them as needed during the course of the game.

Piece Class (abstract class)

In the class diagram for the chess game, the Piece class serves as the abstract base class for all chess pieces. It defines the properties and methods that are common to all pieces, including the color (white or black) and type of Piece, as well as a method to verify if a move is valid. By using an abstract base class, I was able to provide a foundation for the specific implementations of each type of chess piece, while also keeping the code organized and efficient.

The Piece class is further broken down into several subclasses, each representing a specific type of chess piece present in the game. These subclasses include the Knight , Bishop , Pawn, Queen, King, and Rook classes. Each of these classes contains a unique set of methods and conditions for movement, as the movements and rules for each type of chess piece vary. By separating the different types of pieces into their own subclasses, it is easier to manage and maintain the code for the chess game.

PieceFactory Class

The PieceFactory's createPiece() method takes in a PieceType enumeration rather than a string. The PieceType enumeration is used to represent the different types of pieces that can be created by the PieceFactory, such as Knight, Pawn, Bishop, Queen, King, and Rook. The createPiece() method uses a switch statement to determine the type of piece to create based on the value of the PieceType enumeration that is passed to it.

Using an enumeration as a parameter for the createPiece() method has several benefits. First, it allows you to clearly and explicitly specify the type of piece you want to create, rather than relying on a string representation that could be prone to errors. Second, it makes the code easier to read and understand

isCheck() and isCheckMate()

In the isCheck() and isCheckMate() methods, I attempted to create efficient and reusable methods to determine the check and checkmate status of the king in the chess game. The isCheck() method checks if any enemy pieces can attack the king, and if so, I then check if the king is in a checkmate situation by verifying that there are no available moves that would break the check. I also used the isCheck() method to determine if a move to a specific location would result in the king being placed in check.

The isCheckMate() method checks if any moves can be made by any piece on the board to break the check on the king. If no such moves exist, the method returns true, indicating that the king is in checkmate. If any moves are available that would break the check, the method returns false. This method is important in enforcing the rules of the chess game, as a player is not allowed to make any moves that would result in their king being placed in checkmate.

Design Patterns

In my project, I implemented both the Singleton and Factory Method design patterns in the PieceFactory and Board classes. In my project, I implemented both the Singleton and Factory Method design patterns , Both the Singleton and Factory Method design patterns can help to reduce coupling between classes in a project. The Singleton pattern ensures that there is only one instance of a class, which can reduce dependencies between classes that rely on that instance. The Factory Method pattern centralizes the creation of objects in a single location, which can make it easier to manage dependencies and make the code more flexible and easier to modify.

Clean code principles

In order to write clean and maintainable code, I took steps to follow good coding practices and principles. This included using descriptive and conventional names for my classes, variables, and packages, and keeping the coupling between classes low by using the Singleton design pattern. I also made an effort to create reusable code, such as the isCheck() and isCheckMate() methods, which helped to reduce redundancy and make the code easier to understand and modify. Additionally, I tried to avoid adding unnecessary complexity to the code, as this can make it more difficult to understand and maintain. Overall, my goal was to create code that is well-structured, easy to understand, and easy to modify.

SOLID principles

I applied the Single Responsibility Principle (SRP) in all my classes and methods, including the validator class. The SRP states that a class should have only one reason to change, which means that it should have a single, well-defined responsibility. By following this principle, I aimed to create code that is more modular, easier to understand, and easier to maintain. In the context of the validator class, I limited its responsibilities to tasks such as checking input for correctness and verifying that a move is legal in the chess game. This helped to make the class easier to understand and maintain, as it was not responsible for any other unrelated tasks.

THE END