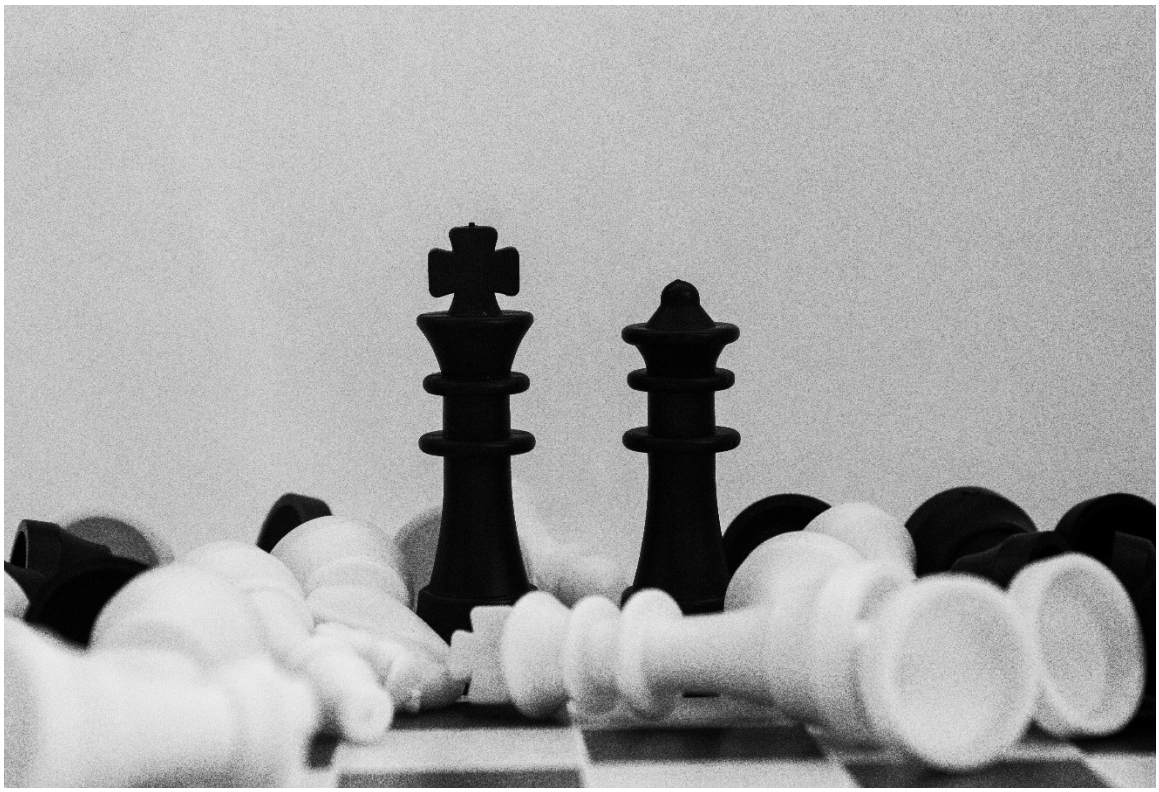
Chess Game Project

Atypon Software Engineering Internship

Jan - 2023

Discovering the inner workings of a Java console-based chess game, including a deep dive into OOD, and defending against design patterns, clean code, and SOLID principles.



Yazan Yousef

Table Of contents

1 Introduction

In this report, we will be discussing the development of a console-based chess game using object-oriented design principles. We will examine the various design patterns and techniques used to create a functional and efficient game, as well as how we addressed challenges and implemented solutions to ensure the game operates smoothly.

We will also delve into the use of SOLID principles and clean code practices to create maintainable and scalable code. Overall, our aim is to demonstrate the strengths and weaknesses of our chess game design and implementation.

First, I will introduce the work flow of the game in general, and then I will provide a detailed overview of each component in the project and how these components interact and work together to create a functioning chess game.

The full UML Diagram is in the link:
https://drive.google.com/file/d/1LyDoWUc2Ks2HuM3daTQnkZxx33Iacx2o/view
(Copy the link to your browser if it doesn't work)

2 Game Flow

The flow of the chess game is as follows: First, players are asked to enter their names and are assigned either the white or black pieces. Then, the game begins with the white player making a move. The player is prompted to enter their desired move.

The move is then passed through a series of validation steps to ensure its legality which in short is:

1) The given move is in a valid template, which contains three sections:
    a. “move” word in any format.
    b. Source position, one character [A-H] represents the column and one digit [1-8] represents the row.
    c. Destination has the same criteria of source.

2) The piece at source belongs to the player.
3) The piece shouldn’t attack a piece from its same color.
4) The type of move is allowed to the piece at the source, for example diagonal move is allowed for a bishop.
5) After applying the move, the king of the player who did the move **must be safe** and not in check.

If the move is valid, it is executed on the board and the turn passes to the other player. This process continues until one player's king is in checkmate, at which point the game ends and the player with the safe king is declared the winner. Alternatively, if the maximum number of moves is reached without a checkmate occurring, the game is declared a draw. At any point during the game, players have the option to view the moves history and display the current state of the board.

3 Color Scheme

I have incorporated various colors in the game to make it more visually appealing and easy to understand. The different colors used are as follows:

- Red: This color is used to indicate an invalid move or when the king is in check.

- Green: This color is used to indicate the winning of a game.
- Blue: All moves are printed in blue.
- Black bold: This font style is used for the black pieces and the black player's name.
- White bold: This font style is used for the white pieces and the white player's name.
- Underline: The kings are underlined to make them stand out.

4 Used Packages

Since every class belongs for one package that organized my code.

- 4.1 GeneralUse package

This package will contain classes and one enumeration that is used in maybe every other class.

- **Position Class:** The Position class represents a position on the chess board. It consists of a row index and a column index. The Position class also has methods to move around with the position, and to check whether a position is within the bounds of the board. It makes the code more organized for example when getting the piece at some position in the board class.

- **Move Class:** The move class represents a move in a game, which consist of a source and destination **Positions**, as well as methods for accessing and comparing these positions.

  The class also provides a factory method for creating a **Move** object from a string input and for validating the input using regular expressions, this is to take the input more smoothly and make better user-experience, for example (Move D2 d4) will be

considered valid move template, regex also checks if the given coordinate is inside the board or not if everything is find it creates the Move object and returns it, otherwise returns null.

The **Move** class is used in the **ChessGame** class to manage the moves made during a game of chess and It is also used in the **ValidatorFacade** class to validate the legality of a move before it is executed on the board.

- **ANSI Class:** The ANSI class provides a collection of static strings that can be used to modify text decoration and color in the console, to make it visually appealing output in the console.

  Rather than putting the needed ANSI codes in each class
  I made this class to apply **single responsibility principle**.

- **Color Enumeration:** The Color enumeration is used to represent the color of players and pieces in the chess game. It contains three constants: WHITE, BLACK, and NONE. The NONE constant is used to represent the **NullPiece** class, which is a placeholder for empty positions on the board. The Color enumeration helps to distinguish between different players and pieces in the game, and makes it easier to implement game logic and rules.

- 4.2 MoveTypes Package

The MoveTypes package contains utility classes that provide validation for each move type. These classes contain a check method that determines whether a given move is valid for a specific piece type.
The idea of this package comes from the DRY principle (Don't repeat yourself) as there is common types of moves between pieces e.g. Queen and Rook, which makes each piece class concise and easier to maintain.

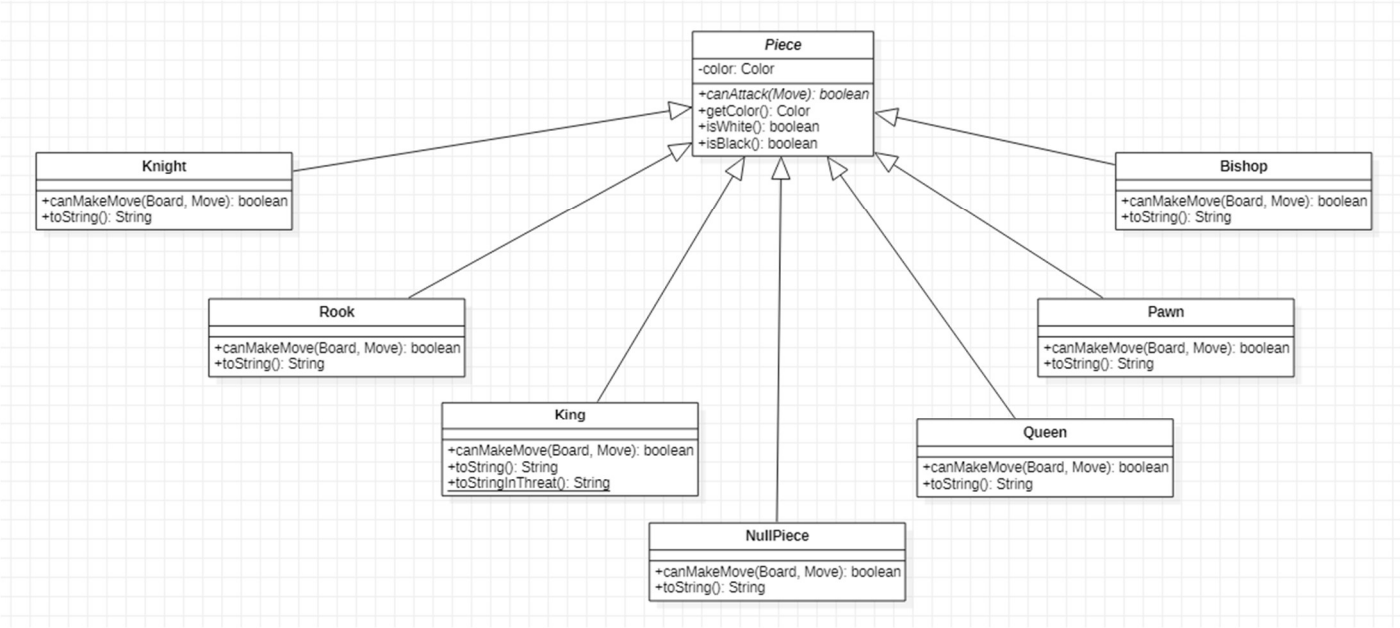
- **LShapeMove Class:** Helps the Knight piece by checking that the absolute difference between the source and destination positions in the rows is 2 and in the columns is 1, or vice versa. This represents a valid L-shaped move and there is no need to check the path, as the Knight can jump over other pieces.
- **DiagonalMove Class:** It helps the Bishop and Queen, it starts from the highest position (the one with the lowest row index) of the source and destination, provided by the **Move** class. Then either moves down and left in a loop until it reaches the lowest position, or moves down and right. The path should not contain any other pieces (except for NullPiece objects) and the class checks that the current position is still within the board as it moves.
- **VerticalMove Class:** The VerticalMove class is a utility class that is used to validate vertical moves made by the Rook and Queen pieces. To determine if a move is valid, the VerticalMove class checks that the source and destination positions have the same column index, and that the path between them does not contain any pieces (with the exception of NullPiece objects). It does this by iterating through the rows between the source and destination positions, starting from the highest position and moving towards the lowest.

- **HorizontalMove Class:** Here we apply the same logic as

vertical but from the right most toward the left most between source and destination, the path should be empty and I will keep double checking that the current position is inside the board.

- 4.3 ChessPiece Package

- The ChessPiece package contains all the chess pieces as separate classes, including **King**, **Queen**, **Rook**, **Knight**, **Bishop**, **Pawn**, and **NullPiece**.



- These classes extends the Piece abstract class, which holds the piece color and defines the abstract **canMakeMove** method, the implementation of it is based on each piece allowed moves, for example a Rook checking if a move is vertical or horizontal using the **moveType** package.

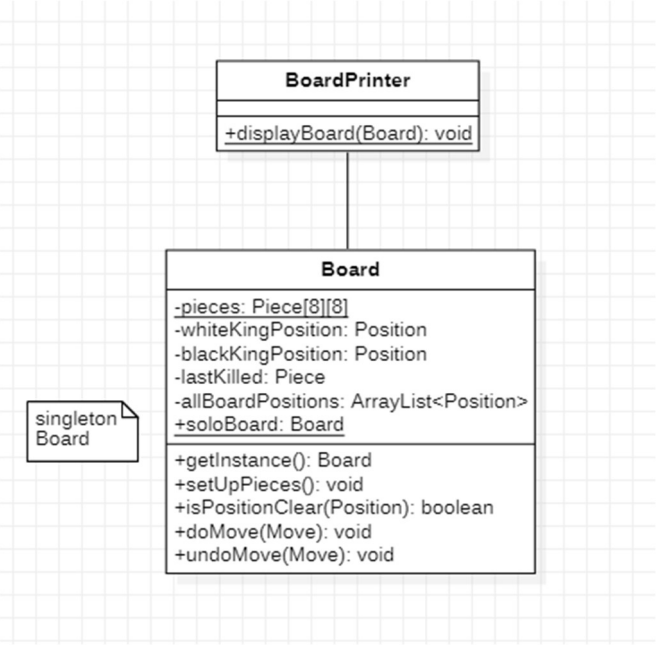
- The use of the canMakeMove method by each piece class, demonstrates the use of the **Strategy design pattern**. Each piece also implements the toString method, using the ANSI class to properly color the piece based on its color. This helps the **BoardPrinter** class in displaying the board.

- The **NullPiece** is a special piece that demonstrates the Null Object Design pattern. It is used as a placeholder for empty positions on the board (so it returns spaces when implementing toString), rather than using null.

- The NullPiece always returns false in the canMakeMove method and is implemented as a single object, following the Flyweight Pattern. This allows for more efficient memory usage and helps to avoid null reference errors.

- 4.4 Board Package (Part of the MainComponent package)

- **Board Class:** The Board class is a **singleton** class meaning there is only one instance of the Board class throughout the entire project from this class.



- It has a two-dimensional array of Piece objects that stores the chess pieces on the board, and an Array to store all board positions which makes it easier to iterate over all positions.

- The Board class also has variables to store the positions of the white and black kings. It has a setUpPieces method that

sets up the board with the correct pieces in their starting positions, and also a useful method isPositionClear.

- **doMove & undoLastMove**: The doMove and undoLastMove methods are responsible for updating the board by moving pieces from one position to another.

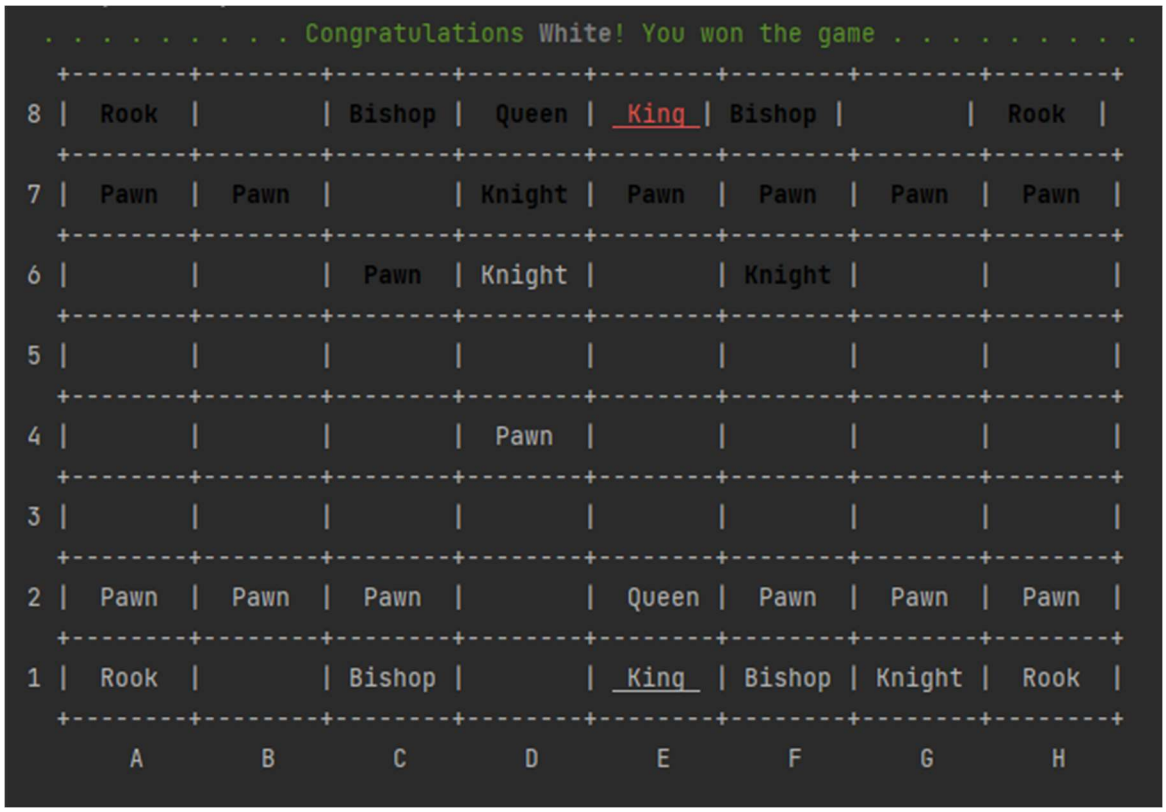
  The doMove method comes when all validation steps is passed, it takes the piece at source and puts it on the destination (simple swap in the 2d array) and put NullPiece on the source.

  When I want to undo a move I return the piece at destination to the source, but how to know what were in the destination originally? Simply it stores it in the **lastKilled** reference. to handle this reference, each time you do a move store the piece at the destination before you override it, even if it's a NullPiece.

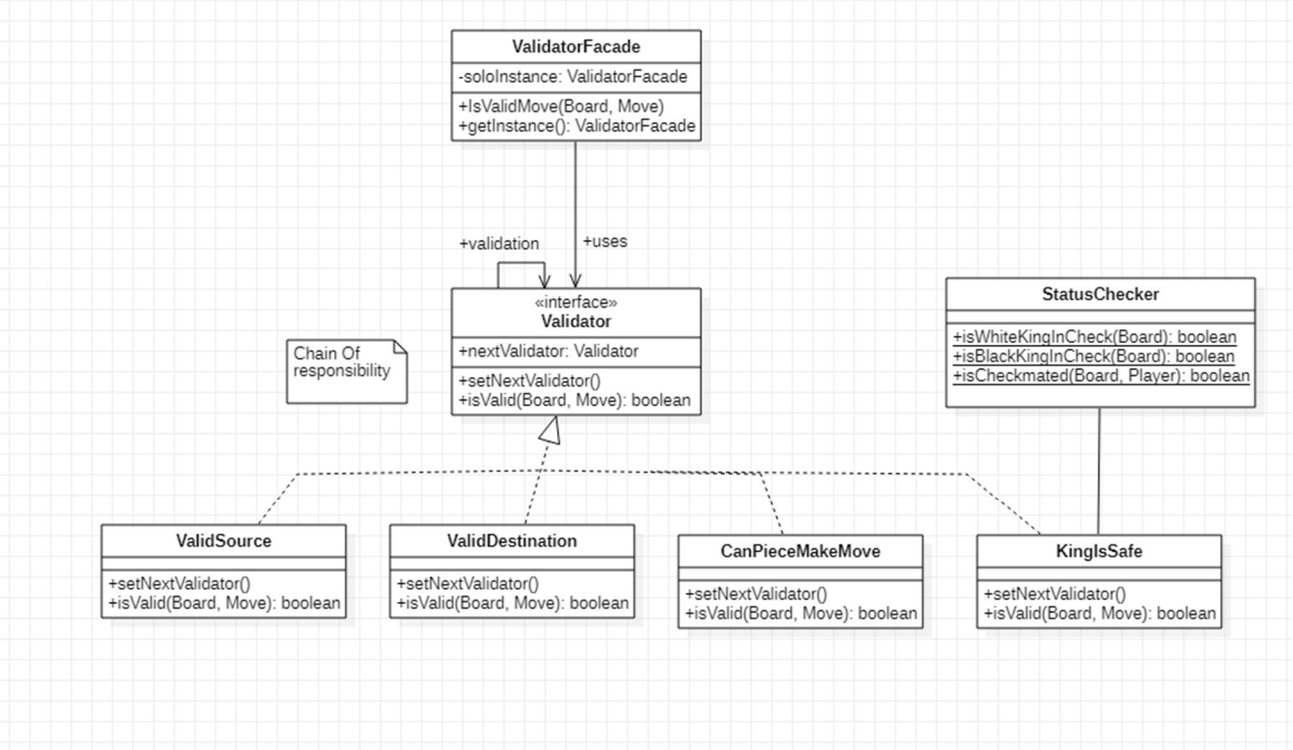
  In addition to updating the board, the doMove and undoLastMove methods also track the positions of the kings by updating the whiteKingPosition and blackKingPosition variables.

The BoardPrinter: This class is responsible for printing the current state of the board to the console. It achieves this by iterating through the 2D array of pieces and using the toString method of each piece to print its representation to the console and it checks if the king is in check to print it in red.

By separating the responsibility of printing the board from the Board class, we are able to maintain a clear separation of concerns and keep the Board class focused on its core responsibilities of storing and manipulating the pieces on the board. This helps to make the code more maintainable and easier to understand.



- 4.5 Validation Package



- **Status checker** : The StatusChecker class is a utility class that provides methods to check whether the king of a particular player is in check or checkmate.

The check happens when there is an opponent piece that can attack the king of a player.

The **isWhiteKingInCheck** and **isBlackKingInCheck** methods iterate through all positions on the board and check if any of the pieces of the opposite color can attack the king, this is done by using the canMakeMove method for opposite color pieces and see if they can attack our king.

The checkmate happens when the king of the player is checked and **No** move can take the king out of the check,
isWhiteKingCheckmated method first checks if the white king is in **check**, and if not, it returns false.

If the king in check, the method iterates through all possible moves for the white player and uses the **ValidatorFacade class**
- Will be explained later - to validate each move. If a move passes all layers of validation, it means that there exists a move that can take the white king out of check, and therefore the game is not a checkmate. If no such move is found, the method returns true, indicating that the white king is in a checkmate, and the same for the **isBlackKingCheckmate** method.

- The Validation package serves to ensure that only valid moves are played in the game of chess. It does this through a series of validation layers, each of which performs a specific check. The first layer, the **ValidSource** class, verifies that the piece being moved belongs to the player making the move.

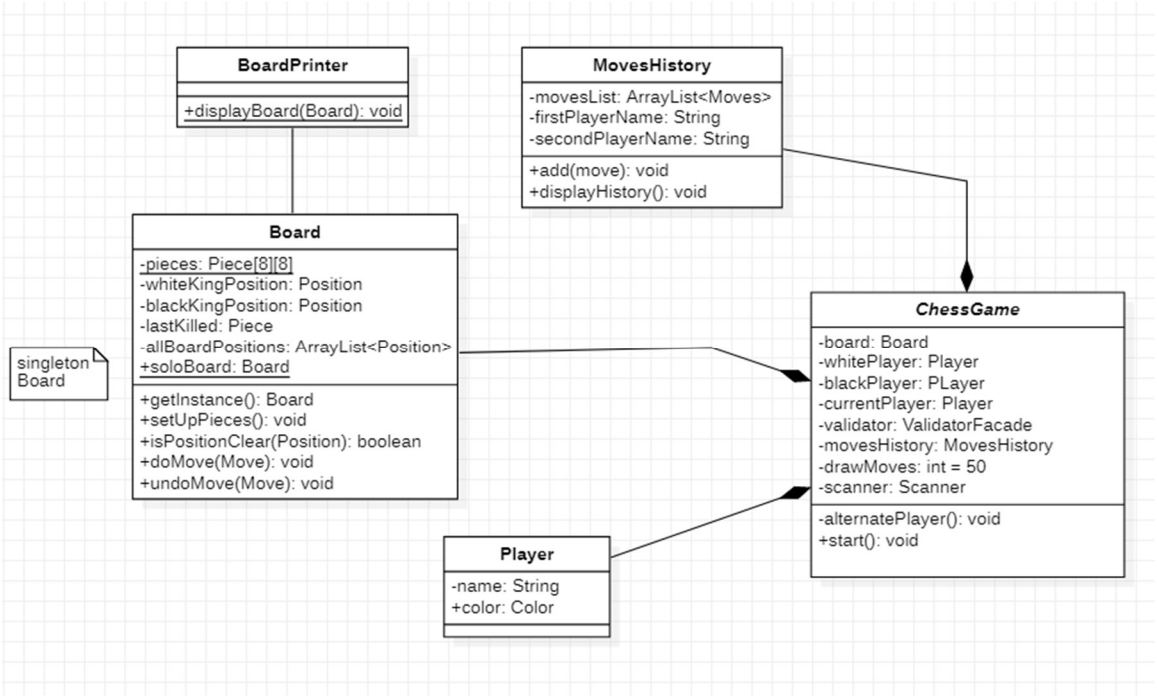
The second layer, the **ValidDestination** class, checks that the destination of the move does not contain a piece belonging to the current player.
The third layer, the **CanPieceMakeMove** class, calls the canMakeMove method of the piece at the source position to ensure that it is able to move to the destination.

The final layer, the **KingIsSafe** class, uses the StatusChecker to check that the move does not leave the player's king in check.

The **ValidatorFacade** serves as a facade for this validation process, setting up the chain of responsibility and providing a single point of access for performing validation checks on a given move. This design allows for flexibility, as new validation checks can be easily added by

creating a new class and inserting it into the chain in the ValidatorFacade.
It also follows the open-close principle, as the validation process can be easily extended without modifying existing code.

- 4.6 MainComponent Package



This package contains the main componenents of the game, The first component is the **Board Packge** mentioned before.

- **Player Class:** The Player class represents a player in the game of chess. It includes the player's name and their associated color (either white or black). It has a getName() method which returns a string representation of the player's name that is colored according to their associated color, using the ANSI class. This allows the player's name to be displayed in the correct color when printed to the console.

- **MovesHistory Class:** The **MovesHistory** class is responsible for storing and displaying a list of moves made in a chess game. It has a list of **Move** objects and two strings representing the names of the players. This class can be useful for keeping track of the progress of a game and reviewing past moves.

- **ChessGame Class:** is a facade that provides a simple interface for users to play a game of chess. It manages the board, players, and moves made during the game. The **start()** method initializes the game by setting up the players and creating an instance of **MovesHistory** to track the moves made during the game.

- The loop in the **start()** method iterates a fixed number of times, representing the number of moves allowed before a draw is declared. We have multiple types of constructors mainly to initialize **drawMoves** and the **scanner** (read from file/System). Within the loop, the board is displayed, the current player is prompted to enter a move.

- If the **Move** object is invalid, the loop continues without incrementing the counter of valid moves. If the move is valid, it is added to the **MovesHistory** and the board is updated with the move. The **StatusChecker** is then used to check if there is a checkmate. If the game has ended, the loop is broken and the game is declared over. Otherwise, the turn is passed to the other player and the loop continues. When the game ends with draw or checkmate it asks the user if we want to show the history or exit the game.

5 Defending Against Clean Code

- **Naming Conventions And Guidelines:** In this project, the names of classes, methods, and variables were chosen to be self-explanatory and easy to understand. It helped to ensure that the code was organized and easy to read. Also, all naming guidelines for classes, methods, and variables were followed.

- **Comments:** In general the code is self descrteptive, however, every class, method, and important code block has a short description to explain its purpose, there weren't any useless comments, also almost every class is documented using JavaDoc.

- **Constructor Chaining:** Subclasses used the super constructor to avoid repeating the same code, and to follow the DRY principle, for example all chess piece took advantage of Piece super class constructor.

- **Avoiding Returning Null**: To avoid returning null and using it I implemented the Null object Pattern which is mentioned in the design patterns section.

- **Avoiding Magic Numbers:** instead of having hard coded numbers for example number of moves before the draw I have it in a variable instead.

- **Method parameters & Flag Arguments:** all methods at most 3 parameters, and that's in the validation chain where I have to take the board, move, and current player, also **No** flag arguments

were used in the code as it makes that code harder to understand and maintain.

- **conditions:** Instead of having very big conditions that is hard to understand, I divided that conditions into smaller parts by using helper functions.

- **Fail Fast & Return Early**: In every method, I prioritize the validation of input and conditions before executing the main logic. This helps to catch potential issues early on and prevent unexpected behavior later in the code.

- **Exception handling**: In order to make the code more reliable and efficient, every exception is handled in the code.

- **Coupling:** To make the code more maintainable and modular, I attempted to minimize coupling between classes as much as possible. Design patterns were used to decouple certain components, but in some cases it was difficult to reduce coupling due to the high level of interdependence between certain classes. When a dependency between classes was necessary, I tried to use data coupling rather than content coupling to promote loose coupling and allow for easier modification and maintenance of the code.

- **Cohesion:** I ensured that each class only included attributes and methods related to its own functionality, which made the code easier to understand, debug, and maintain. Additionally, I organized the code into cohesive packages, further improving its readability and maintainability.

- **Principle Of Proximity:** I followed the principle of proximity by organizing the attributes and functions in a way that related components are placed close to each other. This helps to reduce the cognitive load on the developer, as they don't have to search through

different parts of the code to understand how a specific component works

6 Defending Against SOLID Principles

• **Single Responsibility Principle**: I made sure that each class in the project has a single responsibility, which makes the code more organized. This helps to make the code high cohesive, loosely coupled and easier to maintain and improve the readability of the code.

• **Open/Closed Principle**: I designed the classes in the project to be open for extension, but closed for modification. This means that new functionality can be added without changing the existing code, for example adding new validation step will be as simple as creating new validation class and adding it to the chain without effecting already existing code, which helps to reduce the risk of introducing bugs and makes the code more maintainable.

• **Liskov Substitution Principle**: I made sure that the subclasses of the classes in the project can be used in place of the parent class without affecting the correctness of the program.

- **Interface Segregation Principle**: There is no interface or abstract class that forces it's subclasses to implement methods they can't hold.
- **Dependency Inversion**: The board for example depends on the abstraction (**Abstract Piece Class**) not on the concrete classes.

7 Used Design Patterns

- **Singleton Pattern:** In this project, the **Board** and the **ValidatorFacade** classes are implemented as singletons. This means that there is only one instance of these classes at any time. Overall, the singleton pattern helps to improve memory and time consuming as creating a new board and 32 pieces in each move is too much, instead we could have only one board.

It also makes the code more organized by enforcing a clear and consistent way to access and manipulate the game state.

- **Chain of Responsibility Pattern:** The chain of responsibility design pattern is used in the validation process of the game to ensure that moves made by players are valid. It consists of a series of classes, each one responsible for a specific validation step. **ValidSource**, **ValidDestination**, **CanPieceMakeMove**, **IsKingSafe**, are the validation layers currently. This design allows for flexibility in the validation process, as new validation steps can be easily added to the chain by implementing the Validator interface and inserting the new class into the chain. It also follows the **open-close principle**, as the validation process can be easily extended without modifying existing code.

- **Factory Method Pattern**: In the **Move** class, the **createMove** method uses the Factory Method Pattern to create a new **Move** object. This method takes in a string representation of the move, and based on the input, it creates and returns the appropriate **Move** object. This is useful because it allows the **Move** class to handle the creation of **Move** objects in a central location, rather than allowing other parts of the code to create **Move** objects directly and using some static method to do that.
It creates only valid moves, so if the move is invalid it returns null.

This helps to keep the code organized. Additionally, using the Factory Method Pattern allows for flexibility in the creation of **Move** objects.

- **Null Object Pattern**: In order to maintain clean code and avoid null reference errors, the **Null Object Pattern** was implemented in the form of the **NullPiece class**. This class serves as a placeholder for empty positions on the board, rather than returning null, it has a color of **NONE** from the Color enumeration, and returns false always in canMakeMove. Method.

- **Strategy Pattern**: I used the Strategy Pattern to handle the different ways of moving for each chess piece. Each piece has a specific way of moving, and instead of implementing these behaviors (LshapeMove, DiagonalMove, etc) in the individual piece classes, I created a separate class for each behavior and used composition to apply the desired behavior to each piece. This allows for easy customization and extension of the game, as new behaviors can be added simply by creating a new strategy class and applying it to the desired piece. Additionally, the use of the Strategy Pattern allows for better code organization and separation of concerns, as the behaviors are encapsulated in their own classes and do not clutter the piece classes.

- **Flyweight Pattern**: The Flyweight pattern is applied to optimize the use of NullPiece objects in the chess board. As most of the board is usually occupied by null pieces and they are replaced when a piece is captured, creating a single NullPiece object can save memory and improve performance by reusing the same object instead of creating a new one each time it is needed.

Note: I guess it could be used in all pieces as well, I do that later.

- **Façade Pattern**: the Facade design pattern was used in the ChessGame class to provide a simple interface for users to play the game. The ValidatorFacade class serves as the facade, hiding the

details of the validation process and allowing users to simply provide a board and a move to check if it is a valid move. This simplifies the process for users.

8 Testing And Debugging Process

White I am creating the project I had tested each class alone using unit testing to make sure each component within the project is working correctly.

I had created multiple test cases which is fast checkmates instead of doing very big test case which makes it hard to find where the problem is, and the case were:

- **Fools Mate**
- **Schoolr's Mate**
- **Caro-kann Defense Smothered Mate**

I stored each one of them in a separate file, so I keep running them when I am reviewing my code and adding more features, this is to make sure the modification is working properly.

One example was, after some modifications I tested the fools mate, the queen couldn’t move in a diagonal way but the path was empty, right there I used **Intellij debugger** to figure out what’s wrong, and I found that the queen reached the destination but it didn’t state that, so I knew that the problem is with **equals** method, I removed it from the **Position** class by mistake thinking it not useful and the code was using the equals from Object class, so I fixed it.

Lastly, I played a full game with a friend and tried to do checks and looking for edge cases, and everything was functioning correctly.

Summary

- In summary, the chess project is a program that allows users to play a game of chess against each other.

- It includes classes for managing the board, pieces, and moves made during the game.

- To improve the code's reliability and maintainability, I implemented various design patterns, such as the Flyweight pattern for optimizing the creation and usage of chess pieces, and the Strategy pattern for implementing the different chess piece movements.
- and defended my code against SOLID principles, like open-close principle.

- Additionally, I followed good coding practices and clean code principles such as using self-descriptive names, reducing coupling, and increasing cohesion.

- I also thoroughly tested the project using unit testing and debugged any issues that arose during development. Overall, the chess project is a well-designed and efficient program that allows users to enjoy a game of chess.