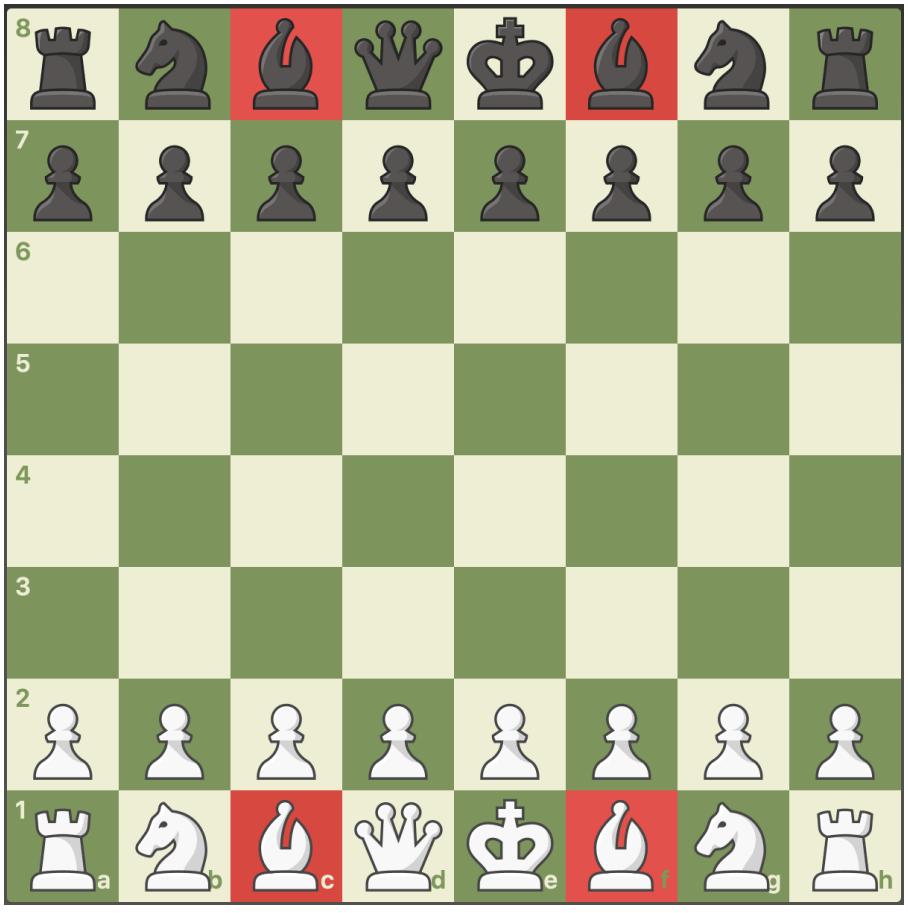
Chess Game Assignment / Atypon Java and Devops Bootcamp / Aug - 2022

Report to explain the OOP design and design patterns used in the project, in addition to defending the code against clean code and SOLID principles.

Yazan Istatiyeh



Contents

1 Introduction

The task was to write a fully functional Console-based chess game, and to show the implemented design. In this report, I will show my design and discuss it, discuss the used design patterns, defend my code against SOLID and clean code principles, and point out what I think are the weaknesses in my code that I did not know how to handle optimally.

2 Classes Description and Relationships

2.1 Enumerations

In my project, I had to implement multiple enumerations primarily to increase the readability of the code, these enumerations are:

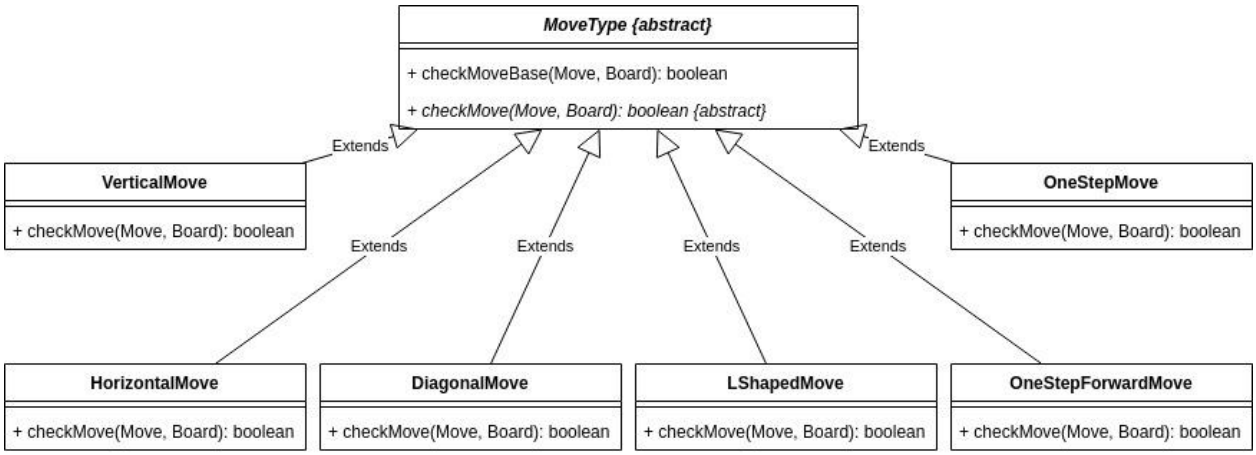
- **Color:** Which is basically the color of a chess piece, it can either be "*WHITE*" or "*BLACK*".
- **GameStatus:** An enumeration that indicates the current status of the game so the program knows when to stop the game and what is the result of it, it can either be "*ON_GOING*" which means that the game did not end, "*DRAW*" which means that the game ended with a draw, "*WHITE_WIN*" or "*BLACK_WIN*" which means that the white player or the black player won the game.
- **KingStatus:** The king is the main piece in the game of chess, the status of the game and the acceptly of many moves depends on the king piece, which means that we should keep up with the king status, this enumeration has four states, "*IN_CHECK*" which means that the king is currently in check but it can escape it with some move, "*CHECKMATE*" which means that the king is in checkmate and the game will end with the result of the player that has that king losses, "*STALEMATE*" which means that the king is not currently in check, but there are no other possible legal moves to do that will not put the king in check, this means that the game will end with a draw, finally "*SAFE*" is the state that indicates that the king is not in danger and the game will continue normally.

2.2 “chesspieces” Package

This package included an abstract class “ChessPiece” which included the properties that every chess piece has, and included a subclass for every chess piece (King / Queen / Bishop / Knight / Rook / Pawn).

- **ChessPiece:** Abstract class that included the color of the piece, the number of moves the piece took already (which is needed for some kind of moves like the pawn’s first move), and a list of type “MoveType” which will be discussed later on, but it simply contains the possible moves for a specific chess piece. It also contains the function *checkMove,* which takes a move and a board where the move starts from a specific piece, and checks if this move is applicable for the given piece on the given board. It also contains function *add* which is responsible for adding new applicable moves for the piece, it also contains 3 abstract methods, *clone, hashCode, addAllMove*, the clone and hashCode will be discussed later, and addAllMoves is for collecting the different applicable moves for a piece and add them all to the array *applicableMoves*.
- **King, Queen, Bishop, Knight, Rook, Pawn:** All of these are classes that extend the abstract class ChessPiece and fill the abstract methods each with the appropriate way looking at the piece capabilities.

  The method *toString* in the abstract class ***ChessPiece*** returns a string with the color of the piece (**“W” or “B”)**, and in the subclasses it calls the same method from the super class and concatenates to it a character that represents the piece type.



2.3 Position Class

This class simply represents a position on the chess board, so it contains two attributes which are ***row and column***.

It has function *isValid* to check if the stored position is inside the board or not, also the hashCode function has been overridden, and will be used in the class **board**. The equals function is overridden too to compare the given position to other positions, also a function getDistance is used to calculate the distance between positions and will be used in the **PathChecker** class.

2.4 Move Class

This class represents a move given by a player from a position **(from)** to another **(to)**, it has two attributes which are the position the move starts from and the position where it is headed, it has method *isValid* which makes sure both positions are valid and that both positions are not the same, it has other methods like *getAbsRowDif* and *getAbsColDif* which calculates the absolute difference for rows and columns between the position where the move starts and the position where it ends, it has other function called *invertMove* which simply flips the **from** and **to** positions.

2.5 ModularArithmetic

This class has arithmetic operations applied to some modulus, this is frequently used with hashing, and for a reason will be explained later we will need to calculate a unique hash for each board (an integer number that represents a certain board with certain positions), and for that we will be using modular arithmetic operations. Also, operations in this class are relatively fast, where it can calculate the power for any number applied to any power in $O(\log_2(\text{Power}))$. All methods in this class are static methods accessible from any other class.

2.6 board Package

This package included three classes, ***Board, BoardPositions, BoardPrinter***; they are all related to the board itself.

- **Board (2.6.1):** This class represents the board of a chess game, it has a 2D array of chess pieces to represent the game, the board will be initialized with pieces and the empty cells will be represented by nulls, and it has an attribute from the same **Board** type called *previousBoard* which is used to restore the previous state of the board, and it has many functions like:
    - *getPiece:* which returns the chess piece that is in the given position.
    - *getKingPosition:* which returns the position of the king of the given color
    - *remove*: which removes a piece from the given position (turns it to null)
    - *applyMove*: which applies the given move to the board, and removes the taken pieces if there is any, also it contains some implementations for situations that include the moving of two pieces in one move like castling.
    - *replacePiece:* which replaces the piece at a given position with a new given piece; this can be used in the case of pawn promotion.
    - *hashCode:* it calculates a unique hash for every board with different positions and pieces, the hash calculation is easy, every king of piece has its own hash implemented in *hashCode* function of every piece, it sums this hash raised to the power of the *hashCode* of the position the piece is in. the hashCode of any position is (Row - 1) * 8 + Column. So for example, if the hashCode for a pawn is 19, and the hashCode for a king is 13, and we had two pawns at positions (2, 3) and (4, 1) and a king at position (5, 2), then the hash = (19 ^ (1 * 8 + 3)) + (19 ^ (3 * 8 + 1)) + (13 ^ (4 * 8 + 2)). (The ***ModularArithmetic*** class is used here)
    - *clone:* which implements making a copy from the board.
    - *revertLastMove:* which restores the board to the previous state (undoing the last move) using the attribute previousBoard (implemented like a linked list where at first the previousBoard will be null)
- **BoardPositions (2.6.2):** This class contains all the 64 positions of the board, to avoid the nested iterating over the board which will look ugly when repeated, so this class implements the Iterable interface and has an iterator.
- **BoardPrinter (2.6.3):** This class is implemented based on the visitor design pattern, it simply takes a board and does the implementation responsible for printing this to the console.

2.7 PathChecker Class

This class can also be considered as a visitor design pattern variation. This class is responsible for taking a move and a board, and checking if the path that this move takes is empty and valid to pass or not.

The implementation of the checking function in the class is so close to the implementation of the DFS algorithm in graph theory, even though our function takes only one path which makes the distance to the final destination shortest (which is the path that the chess piece takes), and it has two directional arrays to check the 8 adjacent cells to the one that the function is currently processing, this class can be used in validating the moves of pieces that takes a horizontal, vertical or diagonal paths.

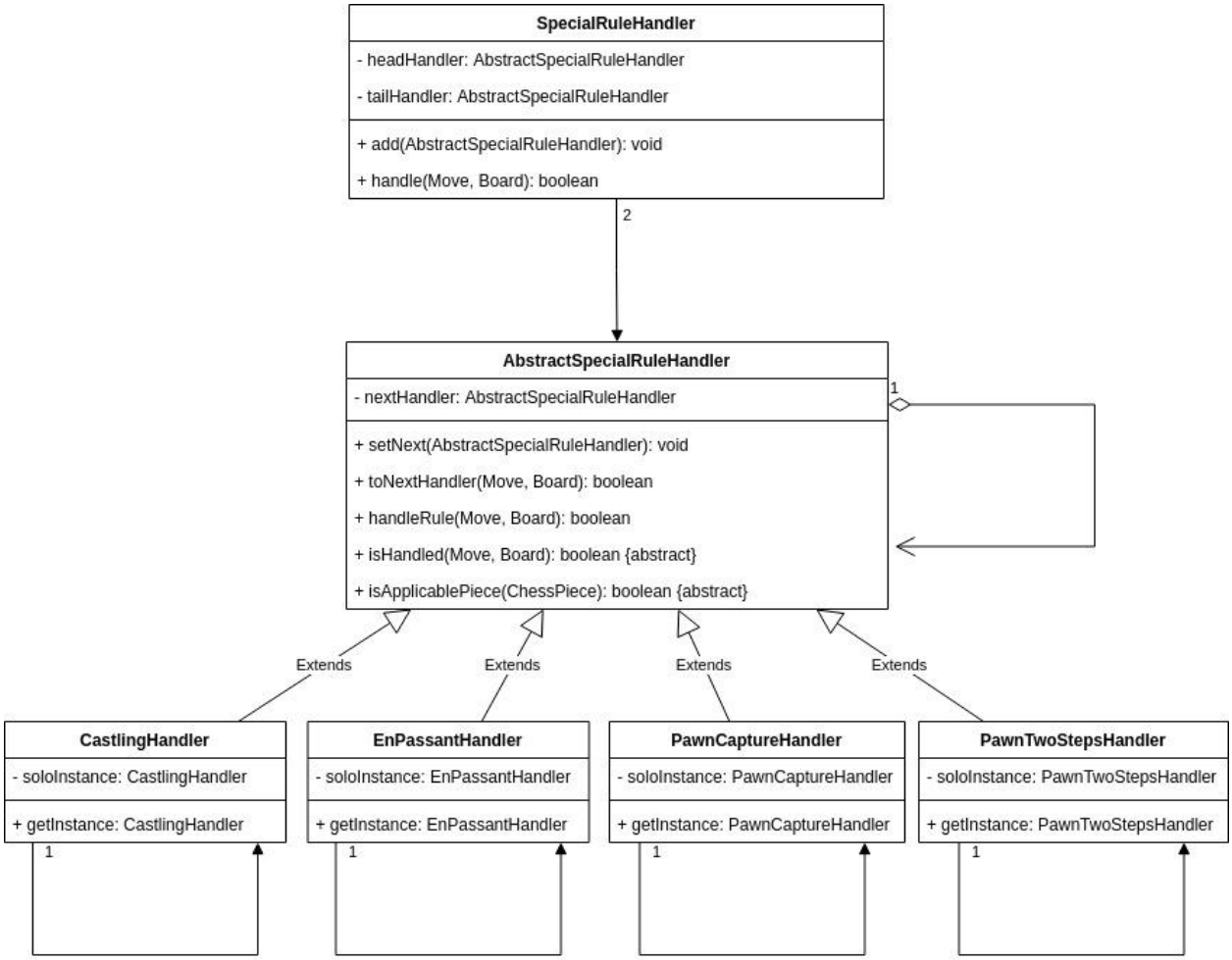
2.8 moves Package

This class contains all different types of moves that the chess pieces can make, and it is based on two design patterns, and it has another subdirectory called ***specialmoves***.

(2.8.1) First, we have the basic moves of chess pieces, and since many pieces can share the same moves and to avoid reimplementing these moves, we can use a simple strategy pattern here.

The main abstract class here is ***MoveType***, which another class can extend to represent a type of move so the subclass would be able to check the validity of a move when it is given to it, the ***MoveType*** class has a *checkBaseMove* method which checks for basic things that any move should have, like starting from a position that is not a NULL or empty position on the board, and ending on an empty cell or a cell with a piece of the opposite color. Also it has the abstract method *checkMove* which has a different implementation between every subclass.

Then the moves package has classes for all basic moves that pieces take in chess, *DiagonalMove (Bishop and Queen), HorizontalMove (Rook and Queen), LShapedMove (Knight), OneStepForwardMove (Pawn), OneStepMove (King), VerticalMove (Queen and Rook),* so now we can easily add a move type for every piece based on the way it move



(2.8.2) Now, we should talk about the sub-package, ***specialmoves***.

This package is responsible for handling special moves and special cases in the game, like the EnPassent move, the two-step pawn move, the pawn capturing, and castling.

To make the adding of special rules easy, and to distribute the implementation details, I decided to go with the chain of responsibility design pattern here. Where every node in the chain is a special rule, so whenever we find a move that does not apply to the basic moves of pieces, we pass this move to the first handler.

We have the abstract class ***AbstractSpecialRuleHandler***, which has the basic implementation for a handler, and the method that passes the handling to the next handler and the method that sets the next handler.

We have special rules handlers that extend the abstract class ***AbstractSpecialRuleHandler,*** *CastlingHandler, EnPassentHandler, PawnCaptureHandler*, and *PawnTwoStepsHandler*. All of these classes have the singleton design pattern applied to them, since it makes sense to have one instance of every handler and have them all connected to each other in the chain.

Some of the handler class has methods that can be used from other classes like the board class to know the type of the move and change the positions of the pieces based on that, like the castling handler has a function that can be used by the board class so it moves the Rook piece in addition to moving the king.

Also, So we connect the handlers to each other and have a clean interface for the user to handle moves, we made a ***SpecialRuleHandler*** *as a wrapper for the chain,* which is the class actually used for handling special moves, it connects the handlers to each other in a LinkedList style, so it has the head and the tail handlers stored in it, which makes adding handlers really easy, clean, and easy to understand only using the *add* method.

Most of the special rules handlers had some complicated implementation in their classes, for example, the castling handlers had functions to get the Rook position and the position the Rook will move to, and the En Passant handler had a function that checks the pawn's position in the previous board to validate the move. Also, the Pawn's two steps handler checks the number of moves the Pawn took before this move.

2.9 KingStatusChecker Class

This class is based on the visitor design pattern too, because its implementation is a little messy and cannot be placed in the ***ChessGame*** class.

This class is responsible for taking the current board and a color, so it checks the current state of the king of the given color (in check, in stalemate, in checkmate, or safe), so the ***GameStatusChecker*** knows what state the game is in, because the king is the main piece in the game and the game status is based on that piece status.

The class has the method *isInCheck* which takes the color of the king to see if it is currently in check, through trying to move all opposite color pieces to the king position, and if any of these moves is legal, then the king is in check.

It also has the methods *isInCheckmate* and *isInStalemate*, the difference between a checkmate and a stalemate is the original status of the king, if the king is currently in check and all other moves (if any) still lets him be in check, then this is a checkmate. If the king currently is not in check, but all the moves to do make him in check, or that there are no moves at all, then this is a stalemate (the game ends in a draw). To check if there are moves that cancels or keeps a check, we have another method *tryAllMoves* which finds all possible moves, applies it to the board, checks for check using the *isInCheck* method, and then reverts the applied move using the method *revertLastMove* in class ***Board***.

Note: the implementation here has been made so much simpler because of using the ***BoardPositions*** class to iterate over all positions, without it we would have had to do 4 nested loops and instantiate two new positions.

Finally, if the king has not been found to be in check, checkmate, or stalemate, then it returns that the king is safe.

2.10 GameStatusChecker Class

This class is based on the visitor design pattern too; it is supposed to take a board and return the current state of the game (On going, white wins, black wins, or draw).

It has a hashmap which stores the frequency of each board using its hashcode, because there is a rule that says if a position is repeated three times, then it is a draw.

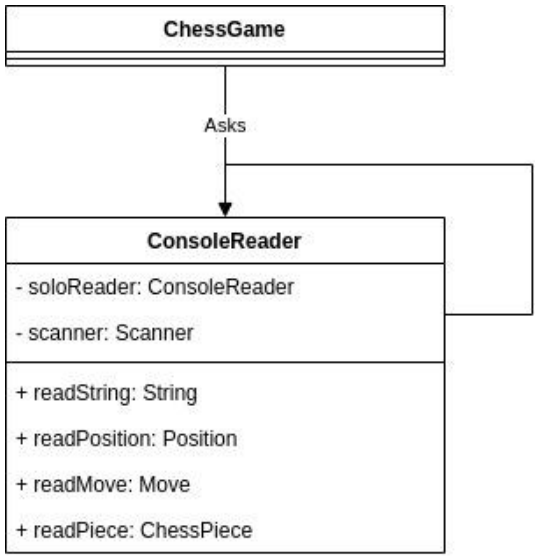
To check if the white player wins, it uses the KingStatusChecker to check the black king status, if it was in checkmate, then the white player wins, and the same for black player.

Then it checks for a draw, if a position has been repeated for 3 times, or the number of moves exceeds some number, or the current player has a stalemate, then the game ends with a draw. Otherwise, the game is on going.

2.11 ConsoleReader Class

Because it would be smelly and ugly to define scanners in the ***ChessGame*** class, and to not have instantiation all over it, we had to make class that is responsible for reading the input from the console, and instantiate the new requested object, this class is based on the singleton design pattern, and the idea of instantiation in it is derived from the factory design pattern, also every read method in it can receive a message to be printed to the user (on the console) before entering the input.

The singleton design pattern was used here because it makes sense to not have many scanners in the program running time which would cause problems, and because this project is not made for multithreading purposes.



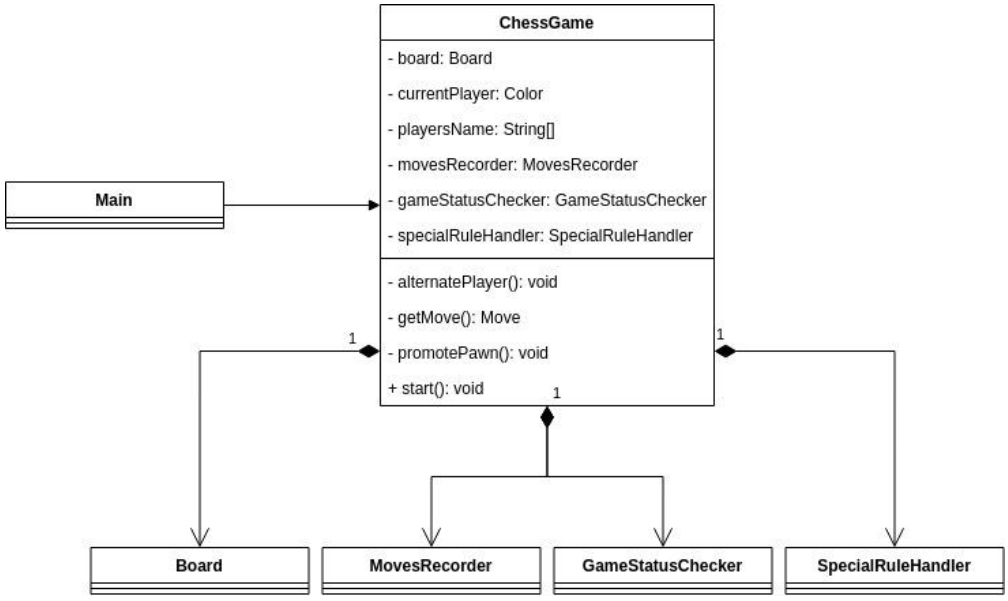
2.12 MovesRecorder Class

This class simply does the role of a paper in the chess games, where each player writes the moves he/she does, so every applied move gets added to an array list of moves, and the moves of the game can be displayed.

2.13 ChessGame Class

This class is a variation of the Facade design pattern, where it collects the classes in the program, adds some logic to them and combines their functionality to have a valid running chess game, it alternates between players and keeps asking for input in case it was not valid, also it indicates pawn promotions and asks the user for the new piece type he/she wants to replace the pawn with.

The chess game simply has a board (***Board Class)***, a current player color (***Color enum)***, two players names (***Strings)***, a moves recorder (***MovesRecorder Class)***, a game status checker (***GameStatusChecker Class)*** and a special rules handler (***SpecialRulesHandler Class)***.



3 Used Design Patterns

3.1 Strategy Pattern

The strategy pattern was used in moves types, since many pieces share the same types of moves, some of them have multiple types, it made sense to use the strategy pattern here to avoid reimplementing the move for every piece.

The diagram for this pattern can be found above in the moves package section.

3.2 Chain of Responsibility Pattern

This pattern was used in the special moves package, it made the implementation for each special move separated from other moves, and made the classes much simpler, also it made us avoid reimplementing the basic move checking.

The UML Diagram for this pattern is above in the specialmoves package section (with Singleton Pattern).

3.3 Singleton Pattern

The Singleton design pattern was used multiple times, first in the special rules handlers, because it would not make sense to have an instance of some handler connected to another handler, and at the same time have another instance of it connected to a different handler, also it made connecting handlers and instantiating them much easier.

Also, it was used in the ***ConsoleReader*** class, because it makes sense to not have more than one console reader in the program, and to avoid the problems that might happen from having multiple active scanners.

The UML Diagram is in the ***ConsoleReader*** class and specialmoves Package (with Chain Of Responsibility Pattern) sections.

3.4 Visitor Pattern

The visitor pattern was used multiple times in the program to separate the implementations, to keep the classes simple, and to apply the single responsibility principle, so it decoupled some operations from the main classes like the ***Board*** and ***ChessGame*** classes.

Some of the classes that used the visitor design pattern or was a variation of it: ***BoardPrinter, GameStatusChecker, KingStatusChecker, PathChecker.***

3.5 Facade Pattern

The ***ChessGame*** class used the Facade design pattern, where it collected the program classes hiding their complexity, and added some logic to them to make the game run mainly from it. So the ***ChessGame*** class basically wrapped the whole project in it.

The UML diagram for the pattern is mentioned above in the ***ChessGame*** class section.

3.6 Inspiration From The Decorator Pattern

In the strategy pattern used before with chess pieces, the way the *toString* method was implemented was inspired by the decorator pattern, where it calls the super class's *toString* method and adds another string to it to make it represent the subclass based on the super class too.

3.7 Null Object Pattern

I did not have the time to implement this, but I just have recently read about the Null Object Pattern, and if I had more time I would have used it with the chess pieces, to represent an empty cell on the board, instead of having so many lines of handling nulls and throwing exceptions which looks a bit ugly.

4 Defending Against Clean Code Principles

Naming Matters / Naming Guidelines: All names in the project were self-descriptive, you can guess the functionality of a class or a function or a variable just by its name. Also, all naming guidelines for classes, methods, and variables were followed.

Constructor Chaining: To follow the *DRY* principle, in subclasses like those for chess pieces, constructors used the super type constructor to avoid repeating the same implementation.

Constructor Telescoping: It was used in the code in the Position class, but it made sense to not use the whole builder pattern for a simple problem like that, this might be wrong, but it was my opinion to not use the builder pattern here.

Avoiding Returning Nulls: In some functions in the ***Board*** class we return null to represent an empty cell which makes sense, although using the Null Object Pattern would have made sense too, probably I would have used it if I had more time.

Using Special codes: Instead of using some hard-to-understand codes, multiple enumerations have been used in the program to keep the code simple, readable, and clean.

Method Parameters: Almost no methods accepts more than 2 parameters in the whole project, the only method that accepts 3 parameters is *isPathEmpty* in the ***PathCheker*** class, which also makes sense because it keeps track of the path it should take, the board, and the current position, so there is no way to decrease the number of parameters here in a clean way, also, the method is private and will not be used outside the scope of this class.

Flag Arguments / Magic Numbers: No flag arguments or magic numbers were used in the code.

Fail Fast & Return Early: In almost every class, you can notice that we check for validity first thing in almost every method, which can easily simplify the implementation and decrease complexity.

Handling Exceptions: Exceptions have been handled in all classes.

Cohesion: Every class only includes attributes that are strongly connected together. Also, cohesion was taken care of at every package, where every class in every package has some relation to other classes in the same package.

Coupling: I tried to reduce coupling between classes as possible, but sometimes it can be so hard to reduce it because of the nature of the project, where every class has some relation to other classes.

Principle of Proximity: I made sure to organize the code in a way that every related or similar method is close to another to increase the readability and make the code understanding easier, this can be clearly seen in the ***Board*** and ***PathChecker*** classes.

Comments: No useless comments were used, most of the comments were to indicate the use of a design pattern or to talk about the class functionality in short. Also, TODOs were used during writing the code to not forget to implement anything.

5 Defending Against SOLID Principles

Single Responsibility Principle: It was made sure that every class and method has only one responsibility, which made the code well organized and made the method's implementations simpler. Although, the ***Board*** class has a function responsible for reverting the board to the previous state, which is a little out of the ***Board*** class responsibility, I had no idea how to fix this problem.

Open/Closed Principle: All classes are stable in the system which implies the closing part of the principle. And it would be easy to add new features to the classes using generalization.

Liskov's Substitution Principle: All subtypes only add functionality to super types in the program, although there are no supertypes in the project that are not abstract, but it would cause no problems to replace a supertype with a subtype.

Interface Segregation Principle: No interfaces were used in the project, and there were no additional methods that any subtype might not need to implement when it comes to abstract classes.

Dependency Inversion Principle: In the ***ChessPiece*** class, the class depends on the abstract class ***MoveType*** not on some of its subclasses. Also, in the ***Board*** class, the board depends on the ***ChessPiece*** class instead of making 32 pieces and assigning each of them with a position.

6 Testing Process

Testing the project was a bit tricky due to having so many cases, so it took some time.

When implementing a new move or a new special case, I edit the ***Board*** class constructor to have less pieces on the board, which makes the testing for some moves easier and more direct, like the En Passant Rule and the Castling Move.

When I finished the whole project, for testing, I searched for the fastest ways to end a game with a draw (Stalemate), or a win for either types, I will leave the followed moves for these. I applied these games to my project and all of them passed perfectly.

Finally, for one last testing, I played a long game with a friend and applied multiple special moves in it, and the program worked without any errors.

Fastest Stalemate: c2 c4 / h7 h5 / h2 h4 / a7 a5 / d1 a4 / a8 a6 / a4 a5 / a6 h6 / a5 c7 / f7 f6 / c7 d7 / e8 f7 / d7 b7 / d8 d3 / b7 b8 / d3 h7 / b8 c8 / f7 g6 / c8 e6 (The game should end here with draw). Resource

Fast Check: a2 a4 / b7 b5 / a4 b5 / c7 c6 / b5 c6 / d8 a5 / c6 d7 (Should indicate check here).

Fast Win: f2 f3 / e7 e5 / g2 g4 / d8 h4 (Black should win here). Fool's Mate