CHESS GAME

Hashem Altabbaa

AUGUST 25, 2022

ATYPON TRAINING

Supervisors:
Dr. Motasem Aldiab
Dr. Fahed Jubair

Table of Contents

Introduction & Document Overview 2

Game Description 2

Object-Oriented Design & Design Patterns (Class Diagram) 3

- Chess Pieces 3
- Pieces movement behavior (Strategy Design Pattern) 4
- The Chess Board 5
- Chess Game & Game Controller 6
- Movement Validation 7
- Full Class Diagram 8

Critical evaluation of the software against clean code principles (Uncle Bob) 9

- Design Rules: 9
- Functions Rules 10
- Code Smells 10

SOLID Principles 11

- S: Single Responsibility Principle 11
- O: Open Closed Principle 11
- L: Liskov Substitution Principle 11
- I: Interface Segregation Principle 12
- D: Dependency Inversion Principle 12

Introduction & Document Overview

I was asked as a task to design and implement a fully functional chess game, following the Object-Oriented Programming (OOP) principles to build a software that complies with the clean code and SOLID principles while taking into consideration using the design patterns to produce more professional code.

This document discuses 4 things:

1. The object-oriented design (high level design) of the software.
2. What design patterns I have used.
3. Critical evaluation of the code against clean code principles (Uncle Bob).
4. Critical evaluation of the code against SOLID principles.

Game Description

The chess game consists of a chess board, tow players, and pieces on the chess board. The classic chess game has 6 different pieces (King, Queen, Rook, Pawn, Knight, and Bishop), each different piece has its **own behavior of moving**. A chess board consists of (8x8) Squares, where each square has its own (x, y) co-ordinates and might has a piece.

The most valuable piece in a chess game is the King. So, if a player's king is under attack by an opponent's piece, the player cannot neglect this threat and must protect the king, and if the player cannot protect the king, he/she will lose, and the game will be stopped. So, a player cannot make a move that would make his king under attack.

Important Rules:

- Each piece has its own way of moving.
- If the King is checked and cannot escape from the attack, the game will be ended.
- The king can short castle or long castle if he has not moved any move and if there are no pieces between him and the rook.
- Any move that causes your king to be under attack is invalid.
- Pawns can be promoted (replaced with any piece) if they reached the other side of the board.
- The game will result in forced draw if the number of moves exceeded 50 moves.

Object-Oriented Design & Design Patterns (Class Diagram)

While building the software design I was always taking into consideration the **Single Responsibility Principle (SRP)** for each class I have built. The fact that every entity in the software is actually responsible for one thing only, will make the design flexible, scalable, and understandable for anyone.

In this section I will discuss the software design going through each component in the software using the class diagram.

Chess Pieces

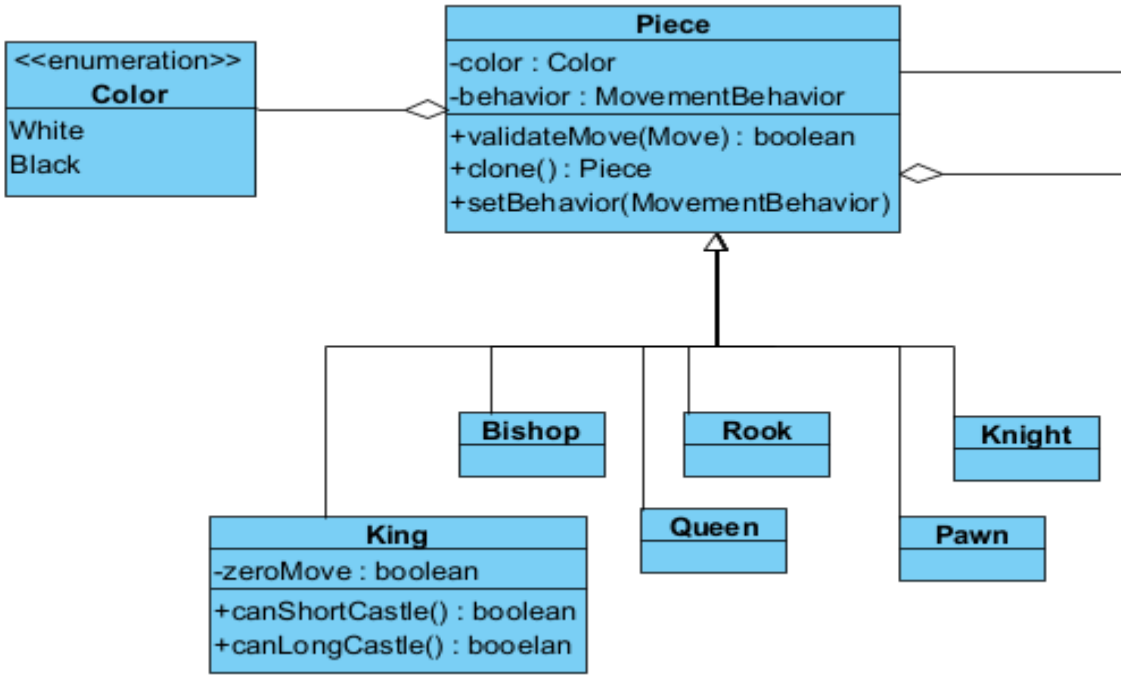


Figure 1 Pieces class diagram

The most important component of the game is the pieces. And I preferred to represent them as a **super abstract** class (Piece) that contains all the common attributes and functions, and each different piece extends the common things. Using polymorphism and inheritance I can build different types of pieces that behave in different ways.

As I mentioned earlier in the game rules, each piece has its own way of moving, and to implement this requirement I have separated the movement behavior from the piece itself with the (Movement Behavior) Interface.

Pieces movement behavior (Strategy Design Pattern)

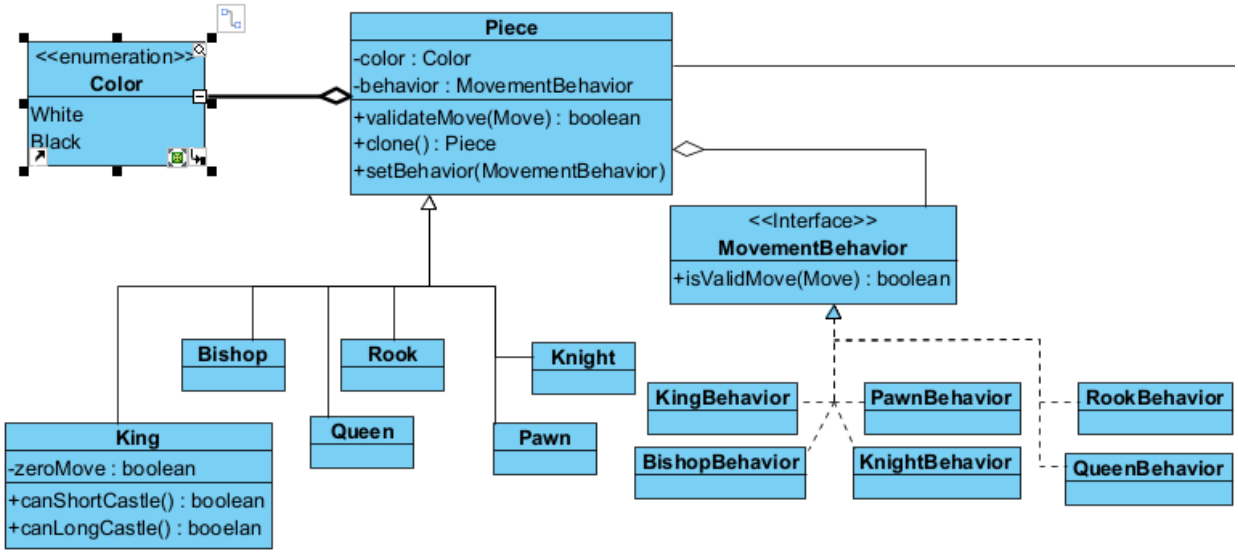
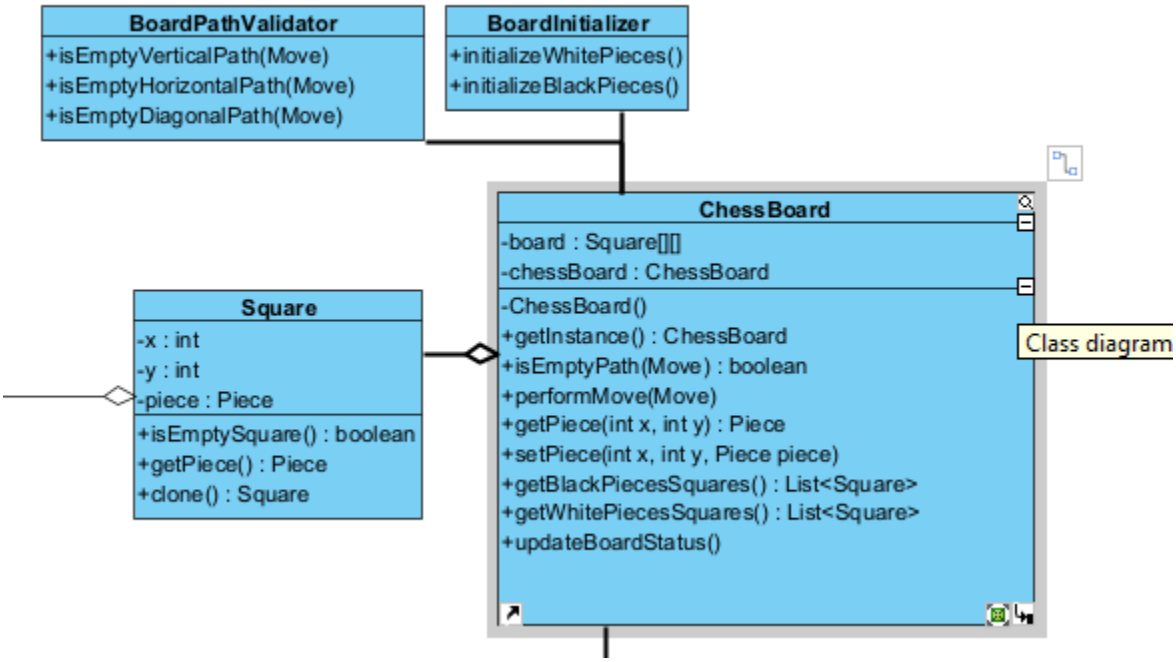


Figure 2 pieces movement behavior

The movement behavior interface is implemented by 6 different classes, each of them represent a piece behavior, by validating the move of that piece according to its behavior. The fact that the movement behavior is separated from the piece itself, will make the design so flexible and able to accept any updates. So, if I wanted to change the movement behavior for the rook for example, I will only change the movement behavior object for the rook instance.

The design ensures that the system has a high Abstraction by breaking the design down into simplified classes with easy and straightforward usage. Also having a high Encapsulation where the logic and details are hidden from the client and given low profile methods to use, leads to minimizing the accessibility and accessorizing it like what effective java recommended.

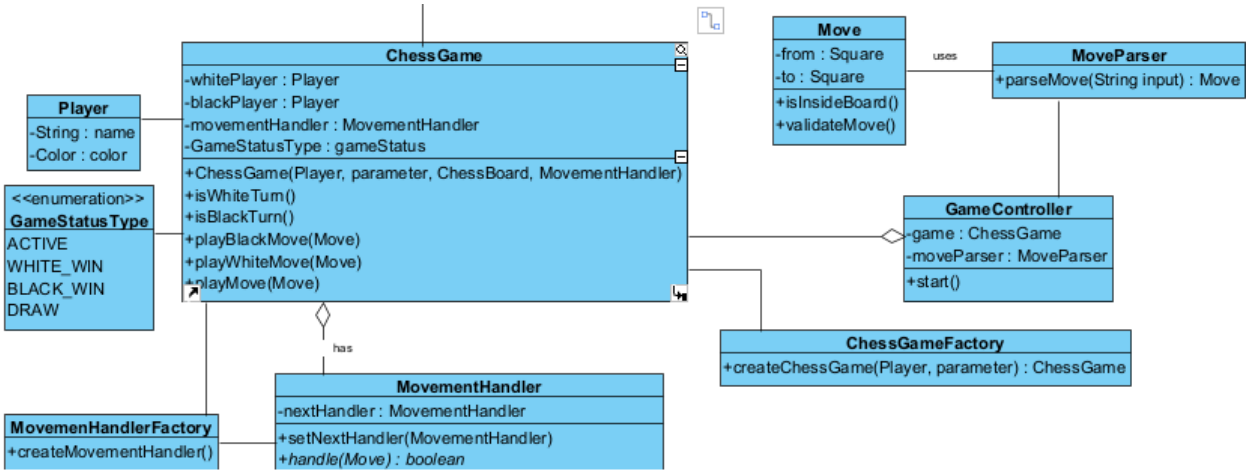
The Chess Board



As I mentioned previously the chess board consists of Squares that has its own co-ordinates. The fact that a chess game cannot have more than one chess board, made me build the chess board class using the **Singleton design pattern.**

Following the SRP, I have separated the initialization of the board and the empty path validator from the board itself. Because the board is not responsible for initializing itself and validating the board paths.

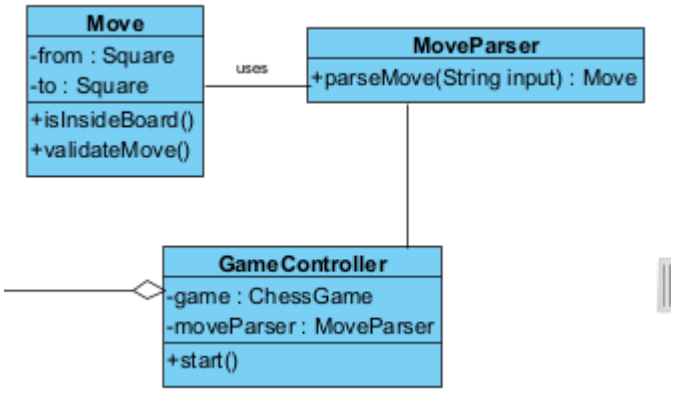
Chess Game & Game Controller



A chess game consists of white and black player, movement handler, and uses a chess board. The **chess game** has its own movement handler to validate, and handle moves. And it also has its own status which could be (Active, white win, black win, or draw). And the game will continue running while the game's status is Active. Because constructing a chess game object might be complex and requires building another object (Movement Handler) I have used **factory design pattern** using the Chess game factory class

The **game controller** is responsible for controlling the game's flow, by checking the players' turns and the game's status. It also uses the Move parser class to validate the string input using regular expressions and parse the input into a Move object. The Move class consists of a starting Square and ending square of the move.

The game controller class uses the **Façade design pattern** by adding layer of abstraction and provides a simple interface that hides the complexity of the system.



Movement Validation

In a chess game there is 3 different types of moves that will be handled differently, and they are:

1. King castling moves
2. Promotion moves
3. Regular pieces move.
4. In order to avoid building huge classes with complex implementation, I have used the **Chain of Responsibility design pattern.** To handle and validate different types of moves according to their types.

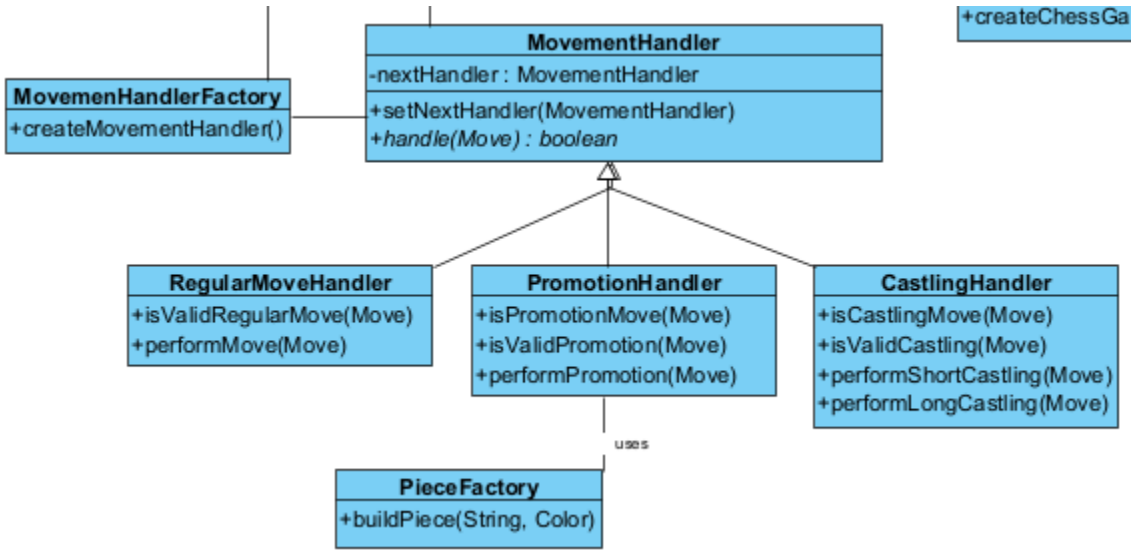


Figure 3 Movement handlers

These three different types of moves validation are represented as a chain of handlers, where each handler will check if that move is under its responsibility and if true, the handler will handle (validate and perform) the move, otherwise it will pass it to the next handler.

The piece factory class uses the **factory design pattern**, to create a piece object according to the request that comes from the promotion handler class. As well as the movement handler factory.

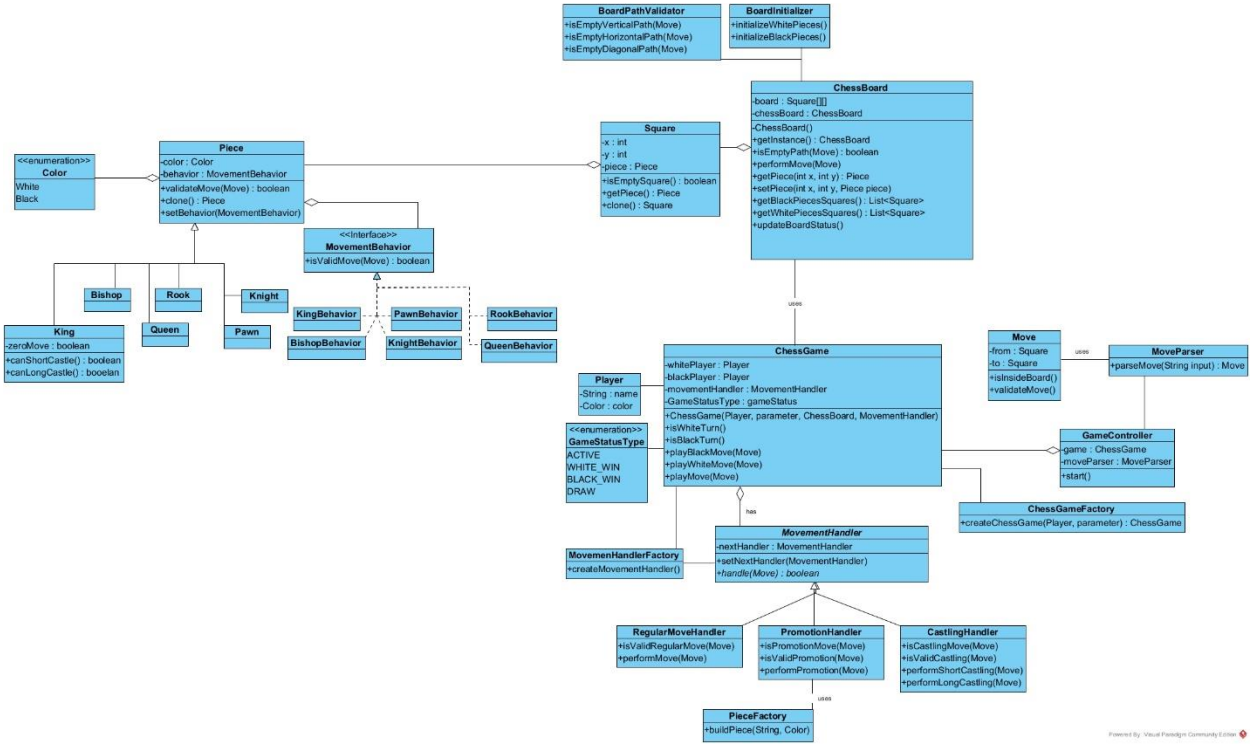
```
MovementHandler regularMovementHandler = new RegularMovementHandler();
MovementHandler castlingHandler = new CastlingHandler();
MovementHandler promotionHandler = new PromotionHandler();

promotionHandler.setNextHandler(castlingHandler);
castlingHandler.setNextHandler(regularMovementHandler);

movementHandler = promotionHandler;
```

Figure 4 Initializing Movement Handler in Chess Game class

Full Class Diagram



To conclude, the design follows the OOP principles, with a highly abstracted and encapsulated entities while taking the advantage of polymorphism and generalization to produce a modular software that is loosely coupled and highly cohesive.

Hashem Altabbaa

Critical evaluation of the software against clean code principles (Uncle Bob)

The clean code book that was written by Robert Cecil Martin (Uncle Bob) deeply discusses how a clean code looks like from different aspects. The code I have written follows the clean code principles. I will go few of the clean code principles and code smells and reflect them on my code.

Design Rules:

- **Prefer polymorphism over if/else or switch/case.**
  That is what I have done in different situations, such as the Movement Validation , where I used Chain of Responsibility to avoid having multiple if/else. Furthermore, separating thePieces movement behavior (Strategy Design Pattern) from the chess pieces applied polymorphism and helped me avoid using if/else.
- **Use dependency injection.**
  Instead of constructing the dependencies objects inside the constructor of a dependent. It is preferred to inject these dependencies through the constructor to reduce coupling between classes. And that what I did in the Chess Game & Game Controller class and theThe Chess Board class. Instead of constructing the dependencies inside their constructor, I injected them through the constructor parameters and with the help of the factory design pattern, I made it simpler to build an object of a chess game and chess board without knowing the instantiating process that requires building another object.
- **Avoid logical dependencies.**
  Which mean briefly, do not write a function that depends on another function to work correctly. In my code, each function works independently, and has its own error handling to make sure that it works independently.

Functions Rules

- **Small functions.**

  Each function does only one thing. Therefore, the functions are considered small

- **Do not use flag arguments.**

  I was always careful not to do this mistake. For instance, the (get white pieces) and (get black pieces) functions in the chess board class could be merged into one function with flag argument, but this is a mistake.

Code Smells

- **Rigidity (Shotgun surgery).**

  This is one of the worst code smells, which mean that any change in a class would affect many other classes. But as long as the software design is modular and follows the **SRP** this mistake would not apply to the code. This was also done with the huge advantage of generalization, polymorphism, and the use of interfaces. Using interfaces and super classes in the right way will ensure that making any changes will not affect other classes, because other classes are using the abstract classes and interfaces that will not be changed.

- **Inappropriate Intimacy**

  This happens when a class depends on the implementation details of another class, which could be caused by two methods of different classes calling each other causing a cycle. There is no code that contains no dependencies, but it must be handled properly, and that what I did in my code.

- **Primitive Obsession**

  Meaning that the code uses a lot of primitive variables instead of classes. While designing the code I always tried hardly to think in Object-Oriented way by representing any entity in the chess game as a separate class, even the color property was represented as enumeration to avoid primitive obsession.

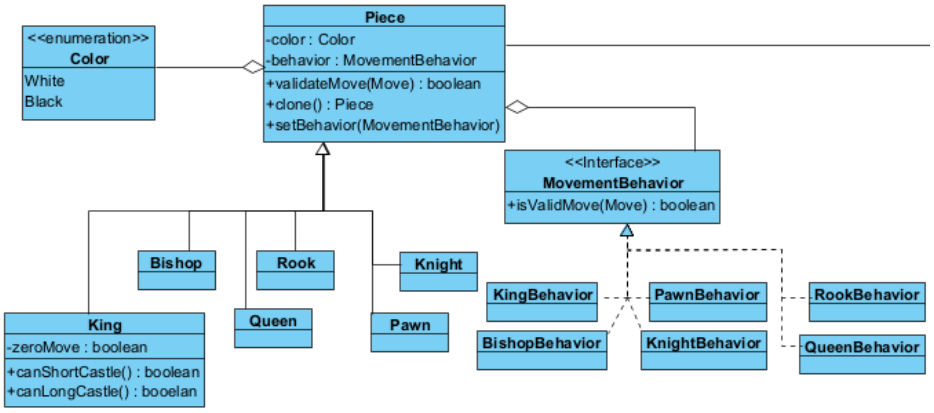
SOLID Principles

S: Single Responsibility Principle

Since I have started designing the software, I always was asking myself ("Is this class/function responsible for doing this thing?"). The code contains many classes, and each class represent only one thing in a chess game starting from Color, piece, chessboard, pieces-movement behavior, movement handler, Square, Move. As you can see each class represents only one thing. About functions, the longest function I have written was less than 50 lines. I am not saying that the small number of lines mean Single Responsibility, but this is an indication that each function does only thing. This can be proven by reading any class or function.

O: Open Closed Principle

Open for extension but closed for modification. As an actual proof for applying this principle, changing or adding a new movement behavior does not require any modification on the code itself, you will have only to create a new movement behavior class that implements the Movement Behavior interface and then you can give this behavior to any piece. Even adding a new piece type to the chess game does not require any modification, you only need to create the new piece class and you are good to go.



L: Liskov Substitution Principle

Meaning that each subclass should be substitutable with their parent class. In other words, all entities in the parent class must be related and common in the child class. For instance, it is a mistake to leave an inherited function unimplemented because it does not relate to the child class. I avoided this mistake by making sure that all functions in the parent class are implemented and related in the child class. For instance, all children of the Piece superclass are related and use the attributes and functions in the superclass, as well as the movement behavior super class.

Hashem Altabbaa

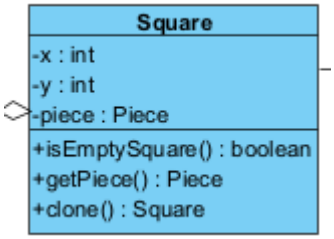
I: Interface Segregation Principle

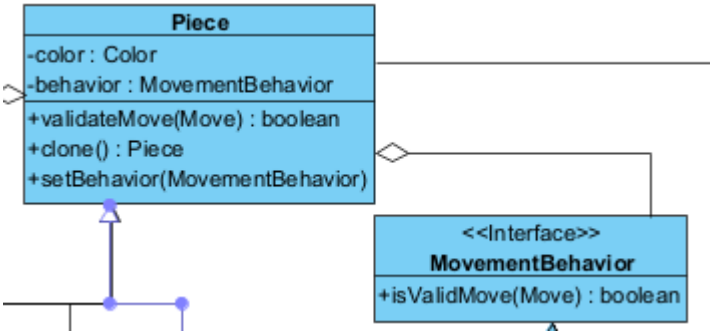
Meaning that a class should not be forced to implement an interface or depend on methods that are not used. In my code there is only one interface that is used to represent the pieces movement behavior, and it is properly used in the subclasses. In addition, there is two abstract classes, Piece and Movement Handler, which are also extended, overridden, and used in the child classes

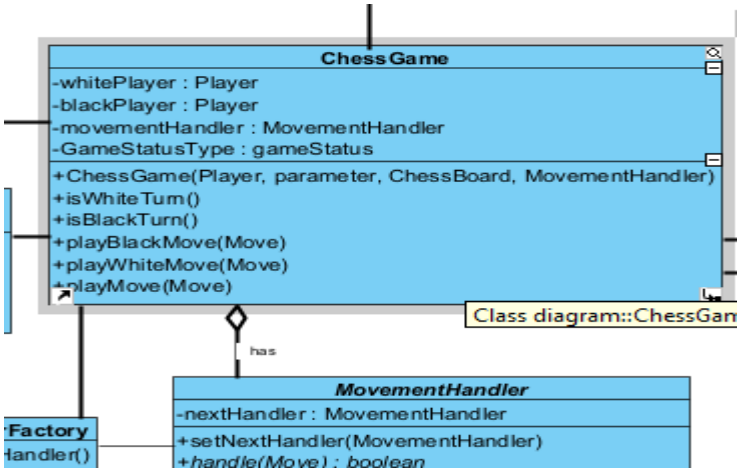
D: Dependency Inversion Principle

Which mean classes and functions should depend on abstract classes and interfaces, instead of concrete classes. This principle is so obvious in my code, and I have multiple examples.

- The square class depends on the Piece class which is an abstract class that is extended by 6 different piece types.
- The Piece class depends on the Movement Behavior interface, which contains the piece move behavior logic.
- The Chess game class depends on the Movement Handler abstract class.







Hashem Altabbaa