

# ATYPON

Chess game system design

Prepared by:  
Mahmoud  
Haifawi

January-2023

# Table of Contents

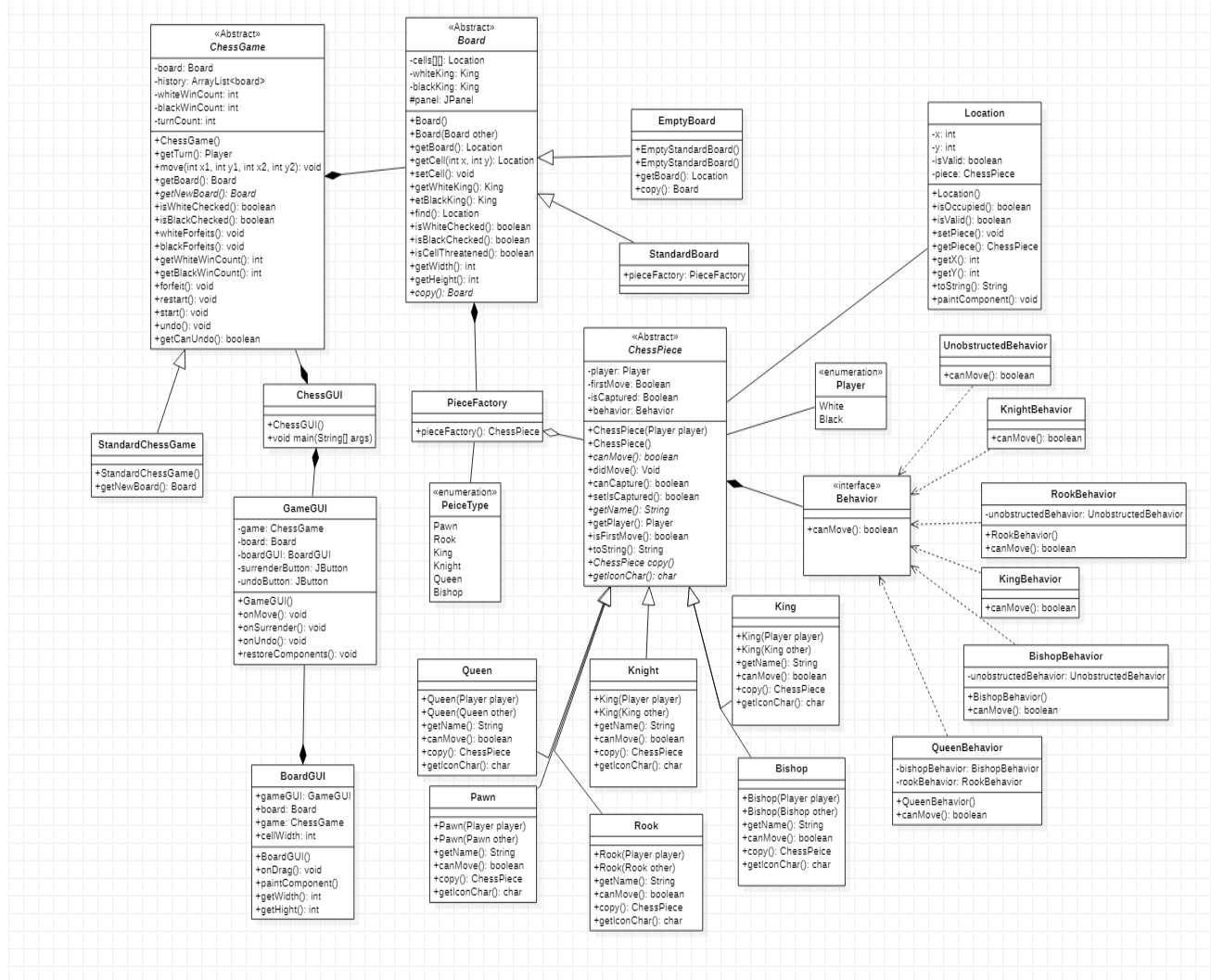
<b>1. Abstract.....</b>	<b>2</b>
<b>2. Class Diagram.....</b>	<b>3</b>
<b>3. Clean code.....</b>	<b>6</b>
<b>4. SOLID Design principles.....</b>	<b>10</b>
<b>5. Design patterns .....</b>	<b>12</b>
<b>6. Testing .....</b>	<b>15</b>

## 1. **Abstract**

In this task we were asked to design a system for Chess Game using principles of OODA, Design Patterns and SOLID Principles.

This report provides a full description and explanation for the submitted solution and a discussion for the object-oriented design, design patterns, and adherence to clean code and SOLID principles that is used in the solution

## 2. Class Diagram



## 2.1 Explanation of The Class Diagram

Lets start with the **Chess Piece** Class:

This class provide the main functionalities for the chess pieces, including :

1. **isFirstMove**: in this method we will check if it is the first move for the piece or not , especially some pieces have different behavior when it is the first move for it.
  2. **canMove**: its an abstract method which will allow the derived classes to provide different implementation , in the derived classes , the implementation for it will depends on the behavior class
  3. **didMove**:if we call it , the **firstMove** attribute's value will become false
  4. **canCapture**: it will check if the color of the attacked piece is different or not
  5. **getName**: it's an abstract method which will provide the name of the piece
- All chess pieces has an inheritance relationship with the chess Piece class

**PieceFactory** Class:

All piece will be created using the PieceFactory class which is smart enough to deice which piece will be created depending on the **pieceType**.

**Behavior** Interface:

It contains only one method which is **CanMove()**, this interface will be implemented by ALL the pieces, so that when any piece call the abstract method **CanMove()**, it will first check the implementation for the behavior interface and depending on the implemnation it will return the Boolean value.

**Location** class:

In this class it will be used mainly in board to trace the location of piece , it

has Important methods such as :

1. isOccupied : it will check if the provided location has a piece on it or not .
2. isValid: it will check if the location is valid value or not depending on the board .
3. setPeice : will put the piece on the location
4. paintComponent : will print the cell itself and if it is checked or not because if it is check the cell color will be changed

### Board Class:

This class has a 2D Array from Location Type , which will allow us to map the pieces with their allocations on the board.

There are some important methods such as:

1. getBoard: its an abstract method which will allow us for more flexibility in future if we added new board types, in general this method will create the board array
2. setCell: this method will allocate the piece in the specific location , I used this method to use the factory to create the piece type and color while setting it on the specific location
3. Find : this method is used to find a certain piece ,using nested loop
4. isWhiteChecked: checks if the white king checked or not
5. isCellThreatened : checks if the piece on the provided location is in danger or not, by checking if any pieces from different color can move to this cell or not.

- There are some classes in the source code, I haven't include them in the class diagram because they are just custom Exceptions I made to make the game more usable

## 3. Clean Code

In this section, I am going to discuss clean code principles and code smells defined by Robert C. Martin.

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."— Martin Fowler.

From the above quotation, we can know that writing clean, understandable, and maintainable code is a skill that is crucial for every developer to master.

## 3.1 Code smells

### 3.1.1 Comments

There are five roles for writing the comments in this section, including clarifying, and justifying the code. It is inappropriate to hold information in a comment rather than in a different system like your source code. This is what I did in my code by giving a meaningful name for the variable and implementing the algorithm in many steps instead of writing comments explaining the algorithm.

```
// Bishops can only move in perfect diagonals
if (Math.abs(xDist) != Math.abs(yDist))
    return false;

// They still have to move, though!
if (xDist == 0 && yDist == 0)
    return false;

// Is the destination is occupied, we must be able to capture it
if (destination.isOccupied()) {
    if (!origin.getPiece().canCapture(destination.getPiece()))
        return false;
}
```

### 3.1.2 Duplication

This principle is applied for all methods as there is no duplication in the code and the code that has been used multiple time it is refactor to method.

```

30 usages
public ChessPiece pieceFactory(PieceType pieceType, Player player) throws InvalidPlayerException {
    if (pieceType == PieceType.Bishop)
        return new Bishop(player);
    else if (pieceType == PieceType.King)
        return new King(player);
    else if (pieceType == PieceType.Knight)
        return new Knight(player);
    else if (pieceType == PieceType.Pawn)
        return new Pawn(player);
    else if (pieceType == PieceType.Queen)
        return new Queen(player);
    else if (pieceType == PieceType.Rook)
        return new Rook(player);
    return null;
}

```

### 3.1.3 Code at wrong level of abstraction

This principle is applied in a good manner as I implement abstraction when needed, Like the behavior interface is the super type for any class that implement it

```

6 implementations
public interface Behavior {

    6 implementations
    boolean canMove(Board board, Location origin, Location destination);
}
|

```

### 3.1.4 Too much information

This principle is applied for all classes as they implement only the necessary methods and depend on the necessary classes, like chess Piece depending on the piece factory to create the pieces and the ChessGUI class only have 2 needed method

```

▶ public class ChessGUI {

    1 usage
    public ChessGUI() {...}

    ▶ public static void main(String[] args) {...}
}

```



### 3.1.5 Dead code

This principle is applied for the whole project as there is no code that never used or in a block that never reached.

### 3.1.6 Names

#### 1. Choose descriptive name

I applied this principle for all methods and variables so that they describe their jobs.

```
private Board board;  
7 usages  
private ArrayList<Board> history;  
  
2 usages  
private int whiteWinCount;  
2 usages  
private int blackWinCount;  
6 usages  
private int turnCount;
```

#### 2. Choose name at appropriate level of abstraction

I applied this principle for all classes, methods, and variables so that they describe their jobs, with taking care about Java naming conventions.

```
2 usages 1 implementation  
abstract Board getNewBoard() throws Exception;  
  
public Board getBoard() { return this.board; }  
  
5 usages  
public boolean isWhiteChecked() throws ChessBoardInvalidException {  
    return this.board.isWhiteChecked();  
}  
  
5 usages  
public boolean isBlackChecked() throws ChessBoardInvalidException {  
    return this.board.isBlackChecked();  
}
```

### 3.1.7 Functions

#### 1. Too many arguments

This principle is applied for all methods in the project, as no method took more than two arguments.

```
public boolean canMove(Board board, Location origin, Location destination) {  
    return this.behavior.canMove(board, origin, destination);  
}
```

#### 2. Output argument

This principle is applied for all methods in the project, as I did not use any output argument

#### 3. Flag argument

This principle is applied for all methods in the project, as I implement all methods in single responsibility principle, so every method does only one job.

#### 4. Dead function

This principle is applied for all methods in the project, as all methods implemented are used in the code.

### 3.1.8 Enums and Annotations

This chapter discusses the using of Enum and annotation and when they should be used. An enumerated type is a type whose legal values consist of a fixed set of constants, such as the seasons of the year, the advantages of Enum types over int constants are compelling. Enums are more readable, safer, and more powerful.

As I did use Enum type, I used this principle to create PieceType and Player

```
public enum PieceType {  
    1 usage  
    King, Queen, Pawn, Knight, Bishop, Rook  
}  
  
public enum Player {WHITE, BLACK}
```

## 4. SOLID design principles

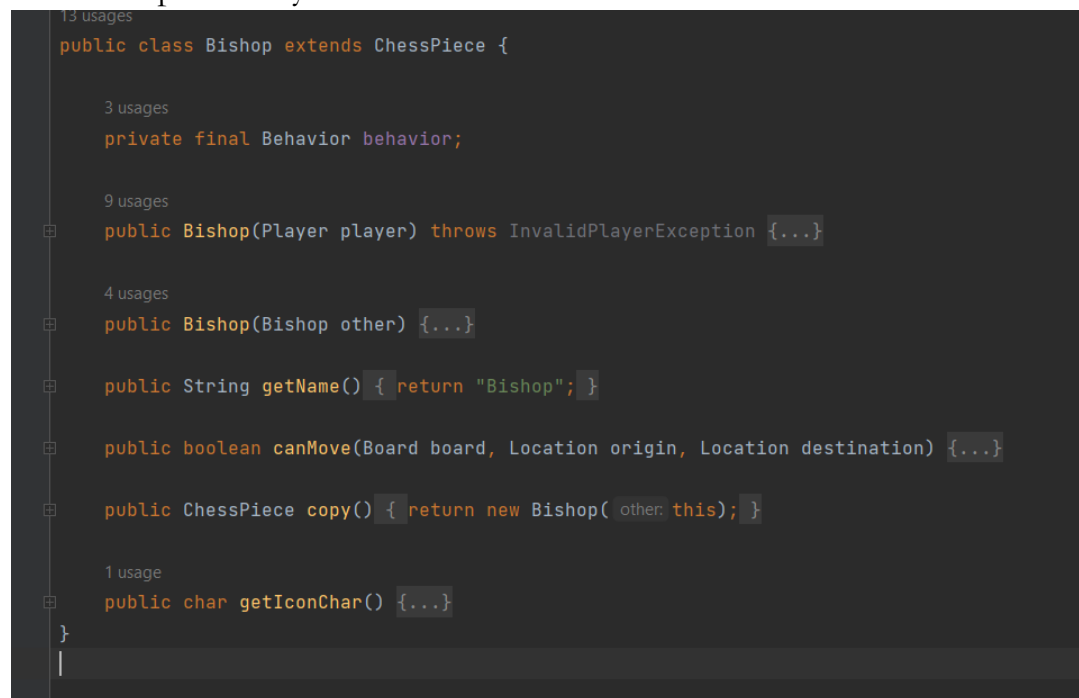
### 4.1 Introduction

Martin and Feathers' design principles encourage us to create software that is more maintainable, understandable, and flexible. As our applications grow, we can reduce their complexity and save ourselves headaches in the future.

### 4.2 Single responsibility

“A class should only have one responsibility. Furthermore, it should only have one reason to change.”

This principle is applied for all classes and methods as they have only one job to do. Classes are built carefully tacking care about each one and its responsibility.



```
13 usages
public class Bishop extends ChessPiece {

    3 usages
    private final Behavior behavior;

    9 usages
    public Bishop(Player player) throws InvalidPlayerException {...}

    4 usages
    public Bishop(Bishop other) {...}

    public String getName() { return "Bishop"; }

    public boolean canMove(Board board, Location origin, Location destination) {...}

    public ChessPiece copy() { return new Bishop( other: this); }

    1 usage
    public char getIconChar() {...}
}
```

### 4.3 Open/close principle

“Classes should be open for extension but closed for modification”

An example for this principle is KnightBehavior class as it depends on a Block interface not on its subtype so if we want to create a new move strategy it just need to implement Block interface and KnightBehavior class code will not modify.

```
5 usages
public class KnightBehavior implements Behavior {
    public boolean canMove(Board board, @NotNull Location origin, @NotNull Location destination) {...}
}
```

### 4.4 Liskov Substitution

“Super types can be substituted by subtypes without affecting correctness. In other words, a subclass should not change the expected behavior inherited from a superclass”

An example from the project is Behavior interface and KnightBehavior class, as KnightBehavior class implements Behavior interface, and they are substituted in a place of each other, and they do not affect the correctness.

```
this.behavior = new KingBehavior();
```

### 4.5 Interface segregation

“Larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them”

There is no larger interface in the project, all interface contains only the methods that should be implemented by their subtypes

```
6 implementations
public interface Behavior {

    6 implementations
    boolean canMove(Board board, Location origin, Location destination);
}
```

## 4.6 Dependency injection

“A class should not depend on low-level concrete classes; instead, it should depend on abstractions. In other words, client classes should depend on an interface or abstract class, rather than a concrete resource”

An example is the classes depends on the Behavior, actually they are depend on Behavior interface instead of that, so add new type of Behavior or modifying QueenBehavior class will not affect them as they are depend on abstraction.

## 5. Design patterns

### 5.1 Introduction

Design patterns represent the best practices used by experienced object-oriented software developers.

Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period.

There is three type of design patterns:

1. Creational Patterns: These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
2. Structural Patterns: These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3. Behavioral Patterns: These design patterns are specifically concerned with communication between objects.

In this section, I will discuss the design pattern I used in implementation of the project.

## 5.2 Factory Method

The factory design pattern is a creational design pattern that provides an interface for creating objects in a super class but allows subclasses to alter the type of objects that will be created. I used this pattern to create various types of chess pieces, such as pawns, knights, and bishops, without specifying the exact class of object that will be created.

```
3 usages
public class PieceFactory {
    30 usages
    public ChessPiece pieceFactory(PieceType pieceType, Player player) throws InvalidPlayerException {
        if (pieceType == PieceType.Bishop)
            return new Bishop(player);
        else if (pieceType == PieceType.King)
            return new King(player);
        else if (pieceType == PieceType.Knight)
            return new Knight(player);
        else if (pieceType == PieceType.Pawn)
            return new Pawn(player);
        else if (pieceType == PieceType.Queen)
            return new Queen(player);
        else if (pieceType == PieceType.Rook)
            return new Rook(player);
        return null;
    }
}
```

So when we created the board object all the pieces will be created and allocated on the board

```
4 usages
public StandardBoard() throws Exception {
    super();

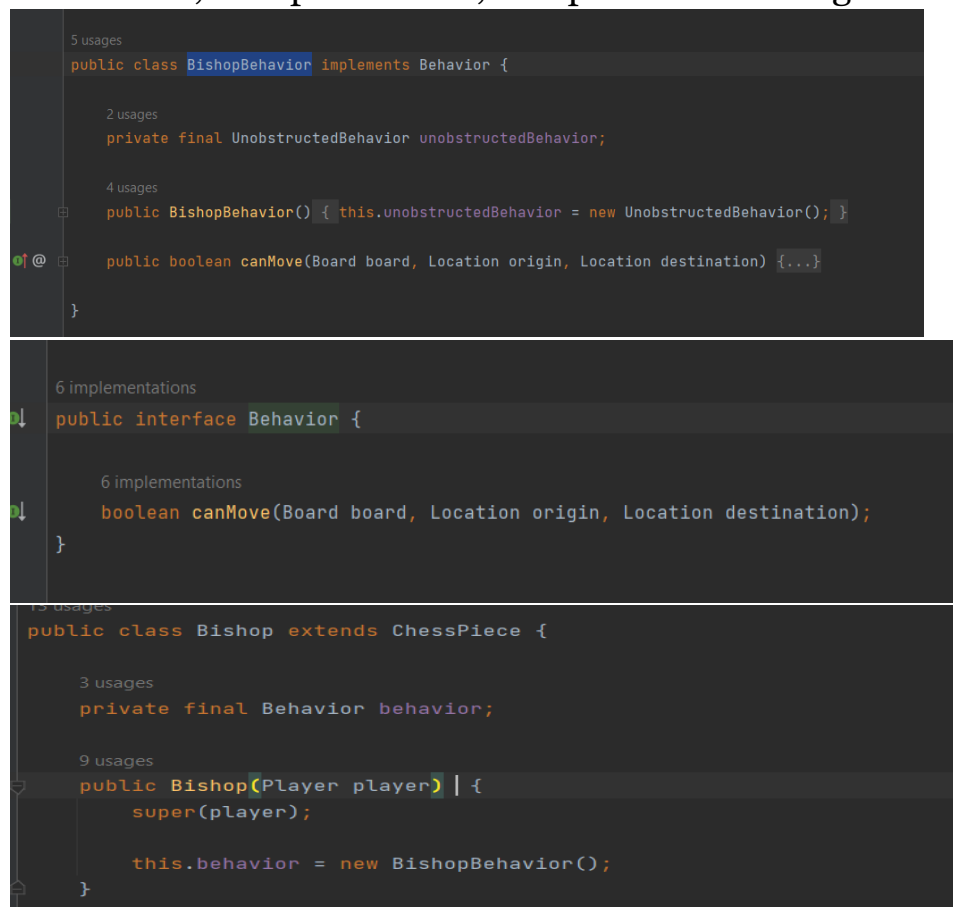
    PieceFactory pieceFactory = new PieceFactory();
    this.setCell( x: 0, y: 0, pieceFactory.pieceFactory(PieceType.Rook, Player.WHITE));
    this.setCell( x: 1, y: 0, pieceFactory.pieceFactory(PieceType.Knight, Player.WHITE));
    this.setCell( x: 2, y: 0, pieceFactory.pieceFactory(PieceType.Bishop, Player.WHITE));
    this.setCell( x: 4, y: 0, pieceFactory.pieceFactory(PieceType.Queen, Player.WHITE));
    this.setCell( x: 5, y: 0, pieceFactory.pieceFactory(PieceType.Bishop, Player.WHITE));
    this.setCell( x: 6, y: 0, pieceFactory.pieceFactory(PieceType.Knight, Player.WHITE));
    this.setCell( x: 7, y: 0, pieceFactory.pieceFactory(PieceType.Rook, Player.WHITE));
    this.setCell( x: 0, y: 1, pieceFactory.pieceFactory(PieceType.Pawn, Player.WHITE));
    this.setCell( x: 1, y: 1, pieceFactory.pieceFactory(PieceType.Pawn, Player.WHITE));
    this.setCell( x: 2, y: 1, pieceFactory.pieceFactory(PieceType.Pawn, Player.WHITE));
    this.setCell( x: 3, y: 1, pieceFactory.pieceFactory(PieceType.Pawn, Player.WHITE));
    this.setCell( x: 4, y: 1, pieceFactory.pieceFactory(PieceType.Pawn, Player.WHITE));
    this.setCell( x: 5, y: 1, pieceFactory.pieceFactory(PieceType.Pawn, Player.WHITE));
    this.setCell( x: 6, y: 1, pieceFactory.pieceFactory(PieceType.Pawn, Player.WHITE));
    this.setCell( x: 7, y: 1, pieceFactory.pieceFactory(PieceType.Pawn, Player.WHITE));
}
```

Using the factory design pattern in this way allows you to easily add new types of chess pieces to the game without changing the main game code. It also makes it easy to switch between different implementations of the same type of chess piece (for example, using different AI algorithms for the pieces).

### 5.3 Strategy Pattern

The strategy design pattern is a behavioral design pattern that allows an object to change its behavior at runtime by switching between different algorithms.

since the each type of pieces has different movements algorithms and some pieces have more than one move algorithms, so it will be so easier to use the strategy pattern to solve this problem, such as the pawn has 3 different movement , 2 steps forward , 1 step forward or diagonal .



The image displays three screenshots of Java code, likely from an IDE, illustrating the Strategy Pattern for chess pieces.

The first screenshot shows the `BishopBehavior` class, which implements the `Behavior` interface. It contains a private final `UnobstructedBehavior` instance and a `canMove` method that delegates the move logic to this instance.

```
5 usages
public class BishopBehavior implements Behavior {

    2 usages
    private final UnobstructedBehavior unobstructedBehavior;

    4 usages
    public BishopBehavior() { this.unobstructedBehavior = new UnobstructedBehavior(); }

    1 @
    public boolean canMove(Board board, Location origin, Location destination) {...}

}
```

The second screenshot shows the `Behavior` interface, which defines the `canMove` method that all behavior implementations must adhere to.

```
6 implementations
public interface Behavior {

    6 implementations
    boolean canMove(Board board, Location origin, Location destination);

}
```

The third screenshot shows the `Bishop` class, which extends `ChessPiece`. It holds a `Behavior` instance and initializes it to `BishopBehavior` in its constructor.

```
13 usages
public class Bishop extends ChessPiece {

    3 usages
    private final Behavior behavior;

    9 usages
    public Bishop(Player player) {
        super(player);

        this.behavior = new BishopBehavior();
    }

}
```

Strategy pattern will help us for any future maintenance and deployment or changing the requirements for the piece's behaviors, For example, if we wanted to add a new movement for the king , we only need to define a new class that implements the behavior interface ,which this new class will contain the new implementation for the king .

## 6. Unit Testing

“You should be able to run all the unit tests with just one command. In the best case you can run all the tests by clicking on one button in your IDE”

So I did this and I wrote some test case and used JUNIT testing also I tested the project manually.

