**Build a Uno game engine to be used by other developers**

**Name**: Zaid saadedin

**Position**: Trainer

**Organization**: Atypon

**Date**: 2023/6/23

Overview:

Dear DR Fahed, I am pleased to submit the following detailed report for highlighting the problem solving steps and optimizations by using object-oriented design, design patterns, clean code principles, Effective Java, SOLID principles and the game logic that I used to create a Uno game that will be use by other developers, as requested.

Introduction:

This project aims to create a Java implementation of Uno to be played in the console by other Developers and can implement their own logic to the game in an easy way, providing players with an immersive experience of this popular game.

Uno is played with a deck of 108 cards, consisting of four suits (red, blue, green, and yellow), each containing cards numbered from 0 to 9. Additionally, the deck includes special action cards such as Skip, Reverse, and Draw Two, as well as wild cards like Wild and Wild Draw Four. These cards add excitement and unpredictability to the game, allowing players to strategically hinder their opponents or change the game's direction.

The rules of Uno are straightforward but offer ample room for strategic decision-making. Players take turns, trying to match the top card of the discard pile either by number, color, or action. If a player cannot play a matching card, they must draw from the deck until they draw a playable card. The game introduces twists through action cards: Skip skips the next player's turn, Reverse changes the direction of play, and Draw Two forces the next player to draw two cards and skip their turn. Wild cards provide opportunities for players to change the color of play, while Wild Draw Four cards allow them to both change the color and make the next player draw four cards.

The goal of Uno is to be the first player to discard all of their cards.

Understanding the Problem:

The initial step in problem-solving is to thoroughly understand the problem statement and

Requirements, at first I read the Uno pdf that was given in the assignment and understood what the

Uno functionalities and how the game works, and after that I make sure to understand the principles provided surely.

1. 1.Analyzing the code:

I start to think of the classes I should implement and the relationship between them, then I started by implementing the UML diagram for the classes, after that I started to do the logical side of the game by following the object-oriented design, design patterns, clean code principles, Effective Java and SOLID principles.

1. Defining the Problem:

Based on the assignment given, I can define the problem as follows: The problem is to create a Uno game that follow these principles object-oriented design, design patterns, clean code principles, Effective Java and SOLID principles, And to played by Developers not players.

**Problem-Solving Approach:**

1. **Object-Oriented Design:**
2. the concept of object-oriented design:

Object-oriented design is a software development approach that focuses on organizing code around objects. Objects are instances of classes that represent real-world entities or concepts. The design principles of encapsulation, inheritance, polymorphism, and abstraction are used to structure the code, promote code reusability, modularity, and maintainability. Object-oriented design enables the creation of modular and scalable systems by modeling relationships between objects and emphasizing the interaction and behavior of these objects.

1. how object-oriented principles are applied in my code:
2. Encapsulation: Encapsulation is achieved through the use of classes and access modifiers (e.g., private, protected, public) to control access to class members (variables and methods). For example, the variables and methods in the Uno class are encapsulated using access modifiers to define their visibility and restrict direct access.
3. Inheritance: Inheritance is used when defining classes in your code. For example, the Uno class extends the Game class, inheriting its methods and properties. Similarly, the NumberUnoCards, ActionUnoCard, and WildUnoCards classes inherit from the Card class.
4. Polymorphism: Polymorphism is demonstrated through method overriding and method overloading. For example, the play() and dealCards() methods in the Uno class override the abstract methods defined in the Game class. Polymorphism allows these methods to be called based on the actual object type at runtime.
5. Abstraction: Abstraction is achieved through the use of abstract classes and interfaces. The Game class is declared as abstract, and it contains abstract methods (play() and dealCards()) that must be implemented by concrete subclasses. The Deck and DiscardPile interfaces define the contract for their implementing classes.
6. Composition is used to create complex objects by combining simpler objects. In my code, composition is seen in the Uno class, which has composed objects such as List<UnoPlayer>, Deck<Card>.
7. Dependency represents a relationship where one class depends on another class. In my code, there are dependencies between classes. For example, the Uno class depends on the UnoPlayer, Deck, and DiscardPile classes to function properly.Dependency injection:
8. Association represents a relationship between two classes, where one class is related to another class. In my code, there is an association between the Uno class and the UnoPlayer, Deck, and DiscardPile classes. The Uno class has references to instances of these classes.
9. **Design Patterns:**
10. the concept of Design Patterns:

Design patterns are reusable solutions to common software design problems. They provide proven approaches for structuring code, managing relationships between objects, and solving design complexities. By following these established patterns, developers can improve code organization, maintainability, and scalability. Design patterns promote code reuse, modularity, and communication among developers, making it easier to understand and collaborate on software projects.

1. explaining the design patterns used in my code:
2. Singleton: There usage of the Singleton pattern in my code, a single instance of the Uno game throughout the application.
3. Factory Method Pattern: Although not explicitly mentioned in my code, the UnoDeck class can be seen as an implementation of the Factory Method pattern. It encapsulates the creation of Card objects by providing a method (getTop()) to get a card from the deck. The deck itself is populated with different types of cards based on predefined rules.
4. Strategy Pattern: The Game class and its subclass Uno demonstrate the Strategy pattern. The play() method in the Uno class represents the main gameplay algorithm, but the specific steps and actions within the gameplay can vary based on the type of game being played. By defining the play() method as abstract in the Game class, it allows each specific game implementation (such as Uno) to define its own unique gameplay logic while adhering to the common structure provided by the superclass.
5. Observer Pattern: The checkWinner() method in the Uno class can be considered an implementation of the Observer pattern. It iterates over the list of players and checks their hand sizes to determine if there is a winner. The UnoPlayer instances are observed by the Uno class to identify the winning player when a certain condition is met (i.e., when a player's hand size becomes 0).
6. Iterator Pattern: The UnoPlayer class utilizes the Iterator pattern implicitly. The getCardByIndex() method allows accessing a specific card in the player's hand by providing its index. Although not shown in the code, it implies that the hand list in UnoPlayer can be iterated over to access individual cards using the index-based iterator

.

**3) Clean Code Principles:**

A) The concept of Clean Code Principles:

Clean Code Principles, popularized by Robert C. Martin, emphasize writing code that is clear, readable, and maintainable. By using descriptive names, keeping functions small and focused, avoiding duplication, following proper formatting, and writing meaningful comments, developers can create code that is easier to understand and work with. Adhering to Clean Code Principles leads to improved code quality, better collaboration among developers, and reduced chances of introducing bugs or technical debt.

1. Discussing the principles presented in "Clean Code" by Uncle Bob (Robert C. Martin) in my code:
2. Descriptive Naming: my code demonstrates good descriptive naming practices. All of the classes, methods, and variables have names that convey their purpose and functionality. For example, class names like Uno, Uno Player, and Uno Deck clearly indicate their role in the game. Similarly, method names like dealCards(), checkWinner(), and displayGameStatus() provide a clear understanding of their functionality.
3. Single Responsibility Principle (SRP): In my code shows adherence to the SRP to all extent. Each class has a specific responsibility, such as Uno handling the game logic, UnoPlayer representing a player, UnoDeck managing the deck of cards, and UnoDiscardPile managing the discard pile.
4. Avoiding Code Smells: My code seems relatively clean and avoids major code smells:
   1. Dead code: In my code there is no dead code all the code is needed.
   2. Long methods: In my code there is no long methods.
   3. Avoid switch cases: I avoided switch cases and use polymorphism instead.
   4. Comments: In my code there is no unnecessary comments.
   5. unnecessary primitive variables: In my code there is no unused variables
   6. duplicate code: In my code there is no duplicated code
   7. Don’t repeat yourself: I avoided repeating myself.
   8. Methods have at most three parameters.
   9. Method names are verbs.
5. Meaningful Comments: The code includes some comments that provide explanations for certain parts of the code. However, the code is very clean and understandable meaningful comments, especially to describe the purpose and behavior of classes.
6. Code Formatting and Style: The code provided follows a consistent formatting and style, which enhances readability. The indentation, spacing, and overall structure of the code make it easy to understand and navigate.

**4) Defending Against "Effective Java" Items:**

A) The concept of "Effective Java" Items Principles:

"Effective Java" is a highly regarded book by Joshua Bloch that provides practical guidance and best practices for writing efficient and reliable Java code. It covers various aspects of Java development, offering valuable insights and techniques to improve code quality, performance, and maintainability. By following the recommendations in "Effective Java," developers can enhance their understanding of Java's features and use them effectively to build robust and high-performing applications.

1. Discussing the principles presented in "Effective Java" Items in my code:
2. Consider static factory methods instead of constructors: Uno. Class
3. Consider a builder when faced with many constructor parameters: Deck. Class
4. Enforce the singleton property with a private constructor or an enum type: Uno. Class, DiscardPile.class and Deck. Class
5. Avoid creating unnecessary objects : yes
6. Eliminate obsolete object references: yes
7. Avoid finalizers and cleaners : yes
8. Prefer try-with-resources to try-finally 3 Methods Common to All Objects :yes
9. Always override toString : yes
10. Minimize the accessibility of classes and members : yes
11. In public classes, use accessor methods, not public fields: yes
12. Favor composition over inheritance: yes
13. Prefer interfaces to abstract classes : yes
14. Use interfaces only to define types: yes
15. Don’t use raw types : yes
16. Prefer lists to arrays :yes
17. Use enums instead of int constants: yes

**5) Defending Against SOLID Principles:**

A) The concept of “SOLID" Principles:

The SOLID principles are a set of five principles that guide software design. They promote clean and maintainable code by emphasizing single responsibility, open for extension and closed for modification, the substitution of subtypes, interface segregation, and dependency inversion. By following these principles, developers can create code that is modular, flexible, and easier to understand and maintain.

B) Discussing the principles presented in "SOLID” in my code:

1) Single Responsibility Principle (SRP): In my code all classes have one responsibility and do one thing for example UNO class do the UNO game, The card class its provide the cards features.

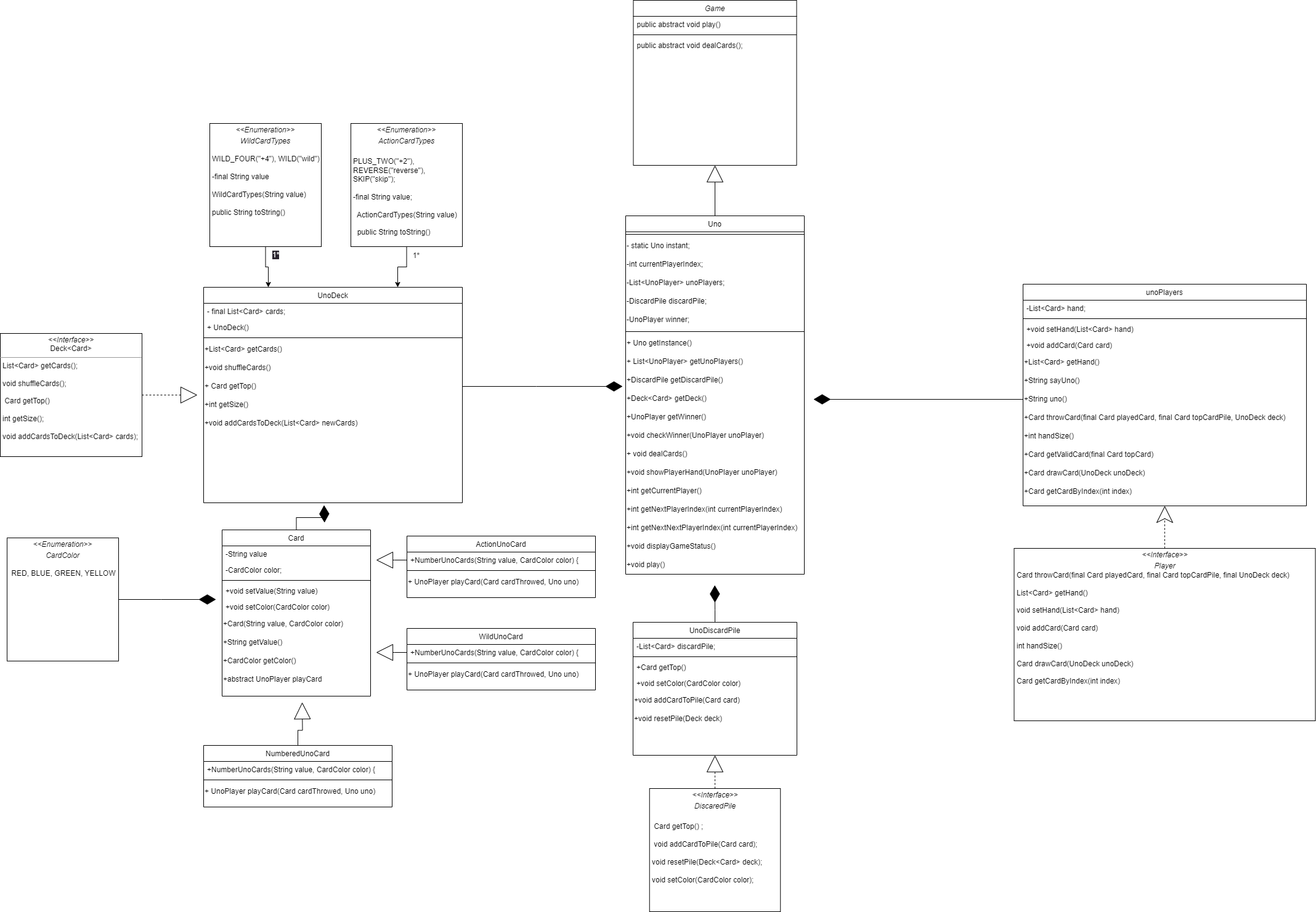
2) Open/Closed Principle (OCP): In my code all classes are made to be closed for modification and open for extension for example you can’t modify the card class or the Deck class or the Player class but you can extend them (Implement).

3) Liskov Substitution Principle (LSP): In my code if there one thing is uncommon between my classes I do for it a new class to avoid code smell and redundant.

4) Interface Segregation Principle (ISP): In my code I have many specific interfaces rather than one general interface.

5) Dependency Inversion Principle (DIP): In my code all sub classes are dependent on super classes for example: NumberUnoCards, ActionUnoCard, and WildUnoCards depend on the Card class.

Design:



UML DIAGRAM

Conclusion:

The "UNO" code effectively created a Uno game. It follows a systematic approach, considering various conditions, to achieve the desired outcome. By analyzing and understanding the problem-solving steps outlined in this report, you can gain a comprehensive understanding of the code's functionality and its solution to the problem at hand.