Uno Game Engine (Report)

Name: Akram Jaghoub

Table of Contents

1.0 Introduction

The following report discusses how I built Uno Game Engine to be used by other developers. The engine provides a solid foundation and framework for developers to build their own Uno game variations. By utilizing OOP principles and incorporating design patterns, and effective java items, the engine offers flexibility, extensibility, and ease of use, enabling developers to create engaging and customized Uno experiences.
To simplify the development process, the engine includes a set of predefined game rules that developers can choose from. These rules can be easily customized to meet the specific requirements of the game being developed.

2.0 Object Oriented Design

2.1 OOP design concepts

1. Inheritance

In the code, various classes demonstrate adherence to object-oriented design principles. One such example is the CardWithColor class, which acts as an abstract base class for colored cards. By encapsulating common behavior and providing a consistent interface for accessing the colors, which allows for code reusability and facilitates polymorphism. There are also other classes in the code that follow a similar pattern, like the ActionCard abstract class which encapsulates shared functionalities among action cards like skip, reverse, or draw-two cards.

2. Encapsulation

The code demonstrates encapsulation by hiding internal implementation details of classes and exposing them through well-defined methods and properties. For example, the Card Factory class which follows the Factory method design pattern, encapsulates the creation of different types of cards, and the DiscardPile and DrawPile classes encapsulate the management of cards in the respective piles. Also, the use of the singleton design pattern implements the concept of encapsulation, by restricting the instantiation of a class to a single and only object. Private members are used appropriately to protect the internal state of objects and access them through public getters like the figure below.

```
public class CardWithNumber extends CardWithColor {
    4 usages
    private final int number;
    1 usage   Akram Jaghoub *
    public CardWithNumber(int number, Color color){
        super(color);
        this.number = number;
    }
```

3. Polymorphism

Polymorphism is demonstrated in the codebase through the Card interface and its concrete implementations like CardWithNumber, Action, and Wild. It allows different card types to be treated uniformly using the Card interface. The canPlayOn() method in the Action class showcases polymorphism by handling different card types and performing specific checks based on their characteristics. Also, the peformAction() method in the Wild and Action classes serves the same purpose as subclasses that override it can have different implementations of the action to performed. Similarly, the cardScore() in the Card interface and the createCards() in the cardFactory class which allows different types of cards to be created, and many parts of my program serve the same purpose and all of these result in flexible and extensible code that enables easier management and manipulation of cards in the game.

4. Abstraction

Abstraction is applied through the Card interface and abstract classes like Wild, Action, and CardWithColor. These abstractions simplify how cards are represented and define common behaviors and features. To ensure proper usage, the constructors in these classes are protected as shown in the figure below, which limits access to them. This restriction allows only subclasses to interact with the constructors and enhancing encapsulation and maintaining control over the creation of objects. Which helps in achieving a more controlled and organized approach to managing game cards.

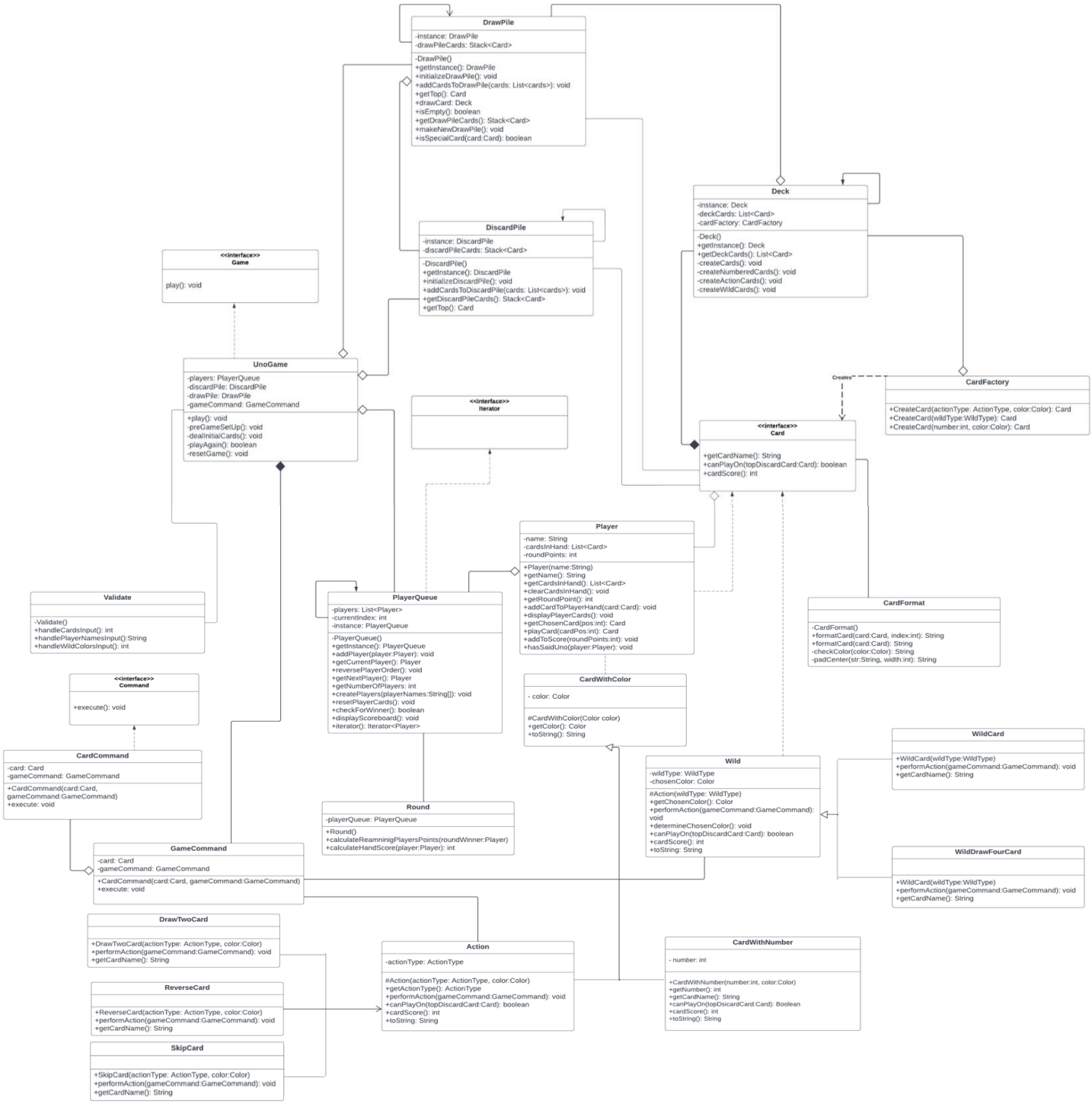
5. Aggregation

In the code, there is an aggregation relationship between the Deck and DrawPile. The Deck class aggregates the DrawPile. This aggregation represents a "has-a" relationship, where the Deck has a DrawPile. This allows for efficient card management and enables the game to effectively handle the cards in the draw and discard piles during gameplay.

6. Composition

In the code, composition is used between the Deck and Card classes. This means that the Deck class contains Card objects as part of its internal implementation. The composition relationship ensures that the Deck and its Cards are tightly coupled, with the Deck managing and manipulating the Cards directly.

2.2 Class Diagram



3.0 Design Patterns

3.1 Creational

3.1.1 Singelton

The Singleton pattern is implemented in the PlayerQueue, DrawPile, DiscardPile, and Deck classes to ensure that there is only a single instance of each class in the program. This is achieved by making the constructor of these classes private, preventing direct instantiation from outside the class. The classes provide a static method, named getInstance(), which returns a single instance of the class. This ensures that all components in the game operate on the same queue, deck, and piles, maintaining consistency and avoiding conflicts. The Singleton pattern is used in these classes to provide a global point of access and ensure that there is only one instance of each class throughout the game.

3.1.2 Factory method

The CardFactory class uses the Factory Method design pattern to simplify the creation of different types of cards. By encapsulating the card creation logic in one place, it provides a convenient and flexible way to instantiate cards without exposing the details to the client code. This promotes code reuse, maintainability, and scalability by allowing the addition of new card types in the future.

3.2 Behavioral

3.2.1 Command

The Command design pattern is used in the code to encapsulate card-related actions as commands. CardCommand implements the Command interface and represents a command for playing a card. GameCommand acts as the invoker, managing game-related objects and enabling various game actions. This pattern achieves decoupling, promotes flexibility, code reuse, and extensibility. It organizes the code, separates concerns, and allows for independent execution of card actions as command objects.

4.0 Clean Code

4.1 Meaningful and consistent names

Used meaningful and descriptive names in all parts of the code that represents the functionality, and it follows consistent naming convention and formatting style of it such as:

1. Variables

```
public class Player {
    2 usages
    private final String name;
    8 usages
    private final List<Card> cardsInHand;
    2 usages
    private int roundPoints;
```

2. Methods

```
    1 usage
    public int calculateHandScore(Player player) {
        int handScore = 0;
        for (Card card : player.getCardsInHand()) {
            handScore += card.cardScore();
        }
        return handScore;
    }
}
```

3. Exceptions

```
    colorIndex = input.nextInt();
    if(colorIndex < 1 || colorIndex > 4) {
        throw new InvalidColorIndexException();
    }
    isValidInput = true;
```

4.2 Comments

Used comments only when needed for the parts that need to be understood.

```
        nextPlayer.addCardToPlayerHand(drawnCard);
    }
    players.getNextPlayer(); //blocks the next player who had drawn the 2 cards and moves to the person after them
}
```

```
        colorIndex = input.nextInt();
        if(colorIndex < 1 || colorIndex > 4) {
            throw new InvalidColorIndexException();  //if player enters the wrong index
        }
```

4.3 Don't Repeat Yourself

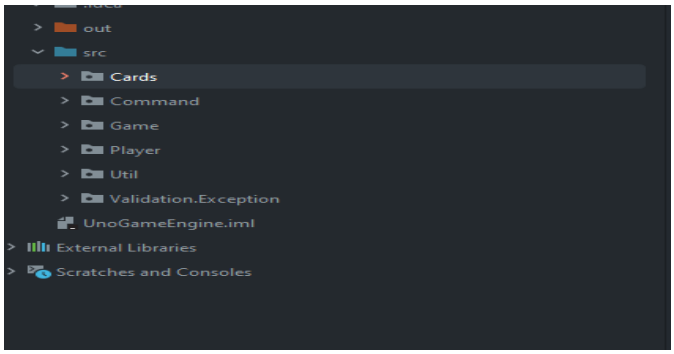
I tried to avoid duplicated code as much as I could and here is an example of it:

```
if (card instanceof Wild wildCard) {
    colorCode = wildCard.getChosenColor() ≠ null ? checkColor(wildCard.getChosenColor()) : "\u001B[37m";
    cardValue = " " + wildCard.getCardName() + " ";
} else {
    CardWithColor coloredCard = (CardWithColor) card;
    colorCode = checkColor(coloredCard.getColor());
    cardValue = card instanceof CardWithNumber ? String.valueOf(((CardWithNumber) card).getNumber()) : card.getCardName();
}
```

```
2 usages   Akram Jaghoub *
private static String checkColor(Color color) {
    return switch (color) {
        case RED → "\u001B[31m";
        case GREEN → "\u001B[32m";
        case YELLOW → "\u001B[33m";
        case BLUE → "\u001B[34m";
    };
}
```

4.4 Separation of concerns

The code demonstrates separation of concerns by organizing classes into separate packages based on their functionality or groupings. The Player package handles player-related concerns, the Game package focuses on game-related logic and management, and the Cards package deals with card-related functionality. This approach improves code organization and modularity.



5.0 Effective Java Items

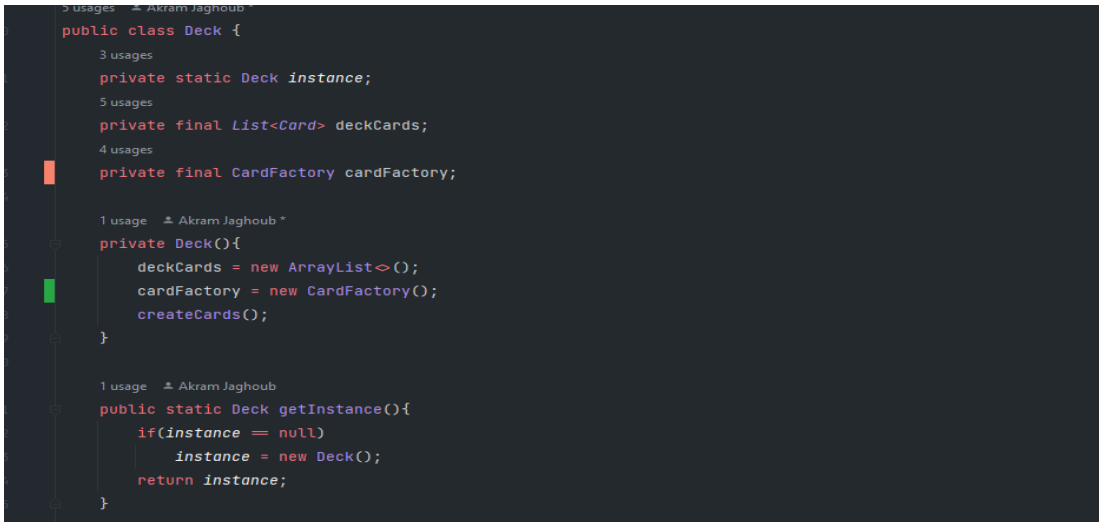
5.1 item1: Consider static factory methods instead of constructors

```
2 usages  Akram Jaghoub *
public class CardFactory {
    1 usage  Akram Jaghoub *
    public Card createCard(ActionType actionType, Color color){
        return switch(actionType){
            case DRAW_TWO → new DrawTwoCard(actionType, color);
            case SKIP → new SkipCard(actionType, color);
            case REVERSE → new ReverseCard(actionType, color);
            case default → throw new CardNotFoundException();
        };
    }

    1 usage  Akram Jaghoub *
    public Card createCard(WildType wildType){
        return switch(wildType) {
            case WILD_CARD → new WildCard(wildType);
            case WILD_DRAW_FOUR → new WildDrawFourCard(wildType);
            case default → throw new CardNotFoundException();
        };
    }
```

5.2 item3: Enforce the singleton property with a private constructor or an enum type

Which I explained in section 3.1.1.



```
5 usages  Akram Jaghoub *
public class Deck {
    3 usages
    private static Deck instance;
    5 usages
    private final List<Card> deckCards;
    4 usages
    private final CardFactory cardFactory;

    1 usage  Akram Jaghoub *
    private Deck(){
        deckCards = new ArrayList<>();
        cardFactory = new CardFactory();
        createCards();
    }

    1 usage  Akram Jaghoub
    public static Deck getInstance(){
        if(instance == null)
            instance = new Deck();
        return instance;
    }
```

5.3 item4: Enforce noninstantiability with a private constructor

Used in utility classes like the CardFormat and Validate classes.

```
8 usages  Akram Jaghoub *
public final class CardFormat {

    no usages  new *
    private CardFormat() {}
```

5.4 item12: Always override toString

```
Akram Jaghoub
@Override
public String toString() {
    String color = getChosenColor() == null ? "" : ", Color = " + getChosenColor();
    return "Card{" +
            "wildType = " + wildType +
             color +
            '}';
```

```
Akram Jaghoub
@Override
public String toString() {
    return "Card{" +
            "number = " + number +
            ", color = " + getColor() +
            '}';
}
```

5.5 Item 15: Minimize the accessibility of classes and members

I wrote class members as private, and they can only be accessed outside the class through getters

```
public class CardWithNumber extends CardWithColor {
    4 usages
    private final int number;
    1 usage  Akram Jaghoub *
    public CardWithNumber(int number, Color color){
        super(color);
        this.number = number;
    }
    3 usages  Akram Jaghoub *
    public int getNumber() { return number; }
```

5.6 item 28: Prefer lists to arrays

In my code I used lists instead of arrays

```
3 usages
private static Deck instance;
5 usages
private final List<Card> deckCards;
4 usages
private final CardFactory cardFactory;
```

5.8 Item 40: Consistently use the Override annotation

```
1 usage  Akram Jaghoub
@Override
public void performAction(GameCommand gameCommand) {
    determineChosenColor();
    gameCommand.getPlayerQueue().getNextPlayer();
}

4 usages  Akram Jaghoub
@Override
public String getCardName() { return "Wild"; }
}
```

5.8 Item 58: Prefer for-each loop to traditional for loops

Used the for each in many parts of my code

```
1 usage  new *
public void resetPlayerCards(){
    for(Player player : players)
        player.clearCardsInHand();
    currentIndex = 0;
}
```

6.0 Solid Principles

6.1 Single Responsibility Principle

In my code, each class has a specific role and is responsible for performing distinct actions. For example, the Round class calculates the points for each round, while the Deck class handles the creation and management of cards. The Command class, along with the Command interface, focuses on executing specific commands when a card is played. This approach ensures that each class has a clear and well-defined responsibility, promoting modularity and maintainability. By separating different aspects of the system's functionality into individual classes, the code becomes easier to understand and modify.

6.2 Open Closed Principle

OCP is demonstrated in my code through the use of inheritance, interfaces, and abstract classes. Each set of rules is defined as a separate class, such as DrawTwo which extends the Action class, or WildCard which extends the Wild class. If a new rule is introduced, like a colored DrawSix card, it can be easily accommodated by creating a new class that extends the existing Action class, without needing to modify the existing codebase. The same applies to new wild cards or entirely new types of cards, which can extend either the CardWithColor or Card classes. Additionally, the GameCommand class is designed to be open for extension, allowing for the support of new commands in the future, without requiring changes to its existing implementation.

6.3 Liskov Substitution Principle

In my code, the LSP is exemplified by the ability of the cardScore() method to determine the specific instance it is called from automatically. For example, in the calculateHandScore(Player player) method, the cardScore() method is invoked on each card in the player's hand, regardless of their specific type (e.g., Wild, Action, Numbered). Similarly, when executing the performAction() method within the CardCommand, it recognizes the instance it belongs to (e.g., Skip, DrawTwo, WildCard) and executes the corresponding action accordingly. This adherence to LSP ensures that substituting different card instances does not disrupt the intended behavior of the program.

6.4 Interface Segregation Principle

My code follows ISP through the use of Card and Command interfaces as each defines a specific set of methods related to their respective responsibilities. The Card interface defines methods related to card attributes and behavior, while the Command interface defines methods for executing commands. This separation allows clients to depend on specific interfaces, preventing them from being forced to implement unnecessary methods.

6.5 Dependency Inversion Principle

The provided code effectively applies the DIP by utilizing abstractions and managing dependencies. By using interfaces like Card and Command, the code promotes loose coupling and flexibility in substituting different implementations. The PlayerQueue class demonstrates dependency injection by depending on the Player class through its methods, allowing for easy integration of various player implementations. This approach enhances modularity, extensibility, and maintainability, making the code adaptable to future requirements.