Uno Game Engine

Edrees Nabeel Ashab

Table of Contents

This is a report regarding the Uno Game Engine assignment. This report has five sections: **Object-oriented design**, this section explains how I implemented OO design in my project. **Design patterns**, here I explain what design patterns I've used and why I used them. **Clean code principles**. **Effective Java Items**. And lastly **SOLID principles** I've applied.

1. Object-oriented design:

1. **Abstraction:** In order to have a simplified design of the Uno Game Engine, I had to use abstraction focusing on the behavior and characteristics that are essential for interaction without exposing the internal detail. It provides a high-level view of the object. Abstraction is achieved through abstract classes and interfaces.
   Example of an **abstract class** in my code is the Game class which has the "template" or the "blueprint" of an Uno Game Engine that developers can extend the Game class and add the necessary implementations to make the game work.

```
package Game;

import Cards.CardsHandler;
import Player.PlayersHandler;

2 usages  2 inheritors  Edrees Nabeel Ashab
public abstract class Game {
    protected PlayersHandler playersHandler;
    protected CardsHandler cardsHandler;
    11 usages
    protected GameRules gameRules;

    1 usage  1 implementation  Edrees Nabeel Ashab
    public abstract void play();
    2 usages  1 implementation  Edrees Nabeel Ashab
    protected abstract void initializeGame();
    1 usage  1 implementation  Edrees Nabeel Ashab
    protected abstract void discarding();
    1 usage  1 implementation  Edrees Nabeel Ashab
    protected abstract void resetRound();
}
```

2. **Encapsulation:** encapsulation is simply hiding the implementation details of the features, methods, and classes. And we use access modifiers to control access to data fields and methods inside a class. **Deck** class uses encapsulation, the developers can only interact with the public classes to use its member.

```
package Cards;

import ...

3 usages   Edrees Nabeel Ashab *
public class Deck {
    16 usages
    private final Stack<Card> deck;
    3 usages
    private final List<Color> colors;
    4 usages
    private Color currentColor;

    1 usage   Edrees Nabeel Ashab *
    public Deck() {...}

    1 usage   Edrees Nabeel Ashab
    private void initializeDeck() {...}

    6 usages   Edrees Nabeel Ashab
    public Card drawCard() {...}

    3 usages   Edrees Nabeel Ashab
    public void setCurrentColor(Color currentColor) {...}

    1 usage   Edrees Nabeel Ashab
    public Color getCurrentColor() {...}

    2 usages   new *
    public void printCurrentColor() {...}
}
```

3. **Generalization:** generalization can be achieved by using **Inheritance** and **Interfaces**, to reduce code redundancy and increase reusability.
   An example of an **interface** is **GameRules** that has the essential rules that every Uno game has, the developers can implement the desirable rules for his own variation of the Uno game.

```java
package Game;

import Cards.Card;
import Cards.CardsHandler;
import Player.PlayersHandler;

9 usages  2 implementations  Edrees Nabeel Ashab
public interface GameRules {
    1 usage  1 implementation  Edrees Nabeel Ashab
    void dealCards(PlayersHandler playersHandler, CardsHandler cardsHandler, int numberOfCards);
    1 usage  2 implementations  Edrees Nabeel Ashab
    void discardCard(PlayersHandler playersHandler, CardsHandler cardsHandler, int cardIndex);
    1 usage  1 implementation  Edrees Nabeel Ashab
    boolean canDiscard(CardsHandler cardsHandler, Card card);
    2 usages  1 implementation  Edrees Nabeel Ashab
    boolean haveDiscardableCards(PlayersHandler playersHandler, CardsHandler cardsHandler);
    1 usage  1 implementation  Edrees Nabeel Ashab
    void noValidCardPenalty(PlayersHandler playersHandler, CardsHandler cardsHandler);
    2 usages  1 implementation  Edrees Nabeel Ashab
    int checkRoundWinner(PlayersHandler playersHandler);
    2 usages  1 implementation  Edrees Nabeel Ashab
    int checkGameWinner(PlayersHandler playersHandler);
}
```

4. **Decomposition:** it has three types of relationships: **Association**, **Aggregation**, and **Composition**. Below I have an example of each relationship:
    a. **Association:** between **SkipActionCard** and **PlayersHandler** and also with **CardsHandler**, where **SkipActionCard** interacts temporarily with these two handlers to perform the intended action. And they're independent from each other.

```java
package Cards;

import Player.Player;
import Player.PlayersHandler;

1 usage  Edrees Nabeel Ashab
public class SkipActionCard extends Card {
    1 usage  Edrees Nabeel Ashab
    public SkipActionCard(Color color) { super(color, value: -1, score: 20, CardType.ACTION_SKIP); }

    2 usages  Edrees Nabeel Ashab
    @Override
    public void applyAction(PlayersHandler playersHandler, CardsHandler cardsHandler) {
        Player nextPlayer = playersHandler.getNextPlayer();
        System.out.println("Player[" + nextPlayer.getIndex() + "] got skipped!");
    }
}
```

    b. **Aggregation:** the **Game** superclass represents a "has-a" relationship with **GameRules**. This enables the developers to use different set of rules without having to change the class itself.

```java
package Game.Variations;

import ...

1 usage  1 inheritor  Edrees Nabeel Ashab *
public class StandardGame extends Game {
    1 usage  Edrees Nabeel Ashab
    public StandardGame(int numberOfPlayers, GameRules gameRules) {
        playersHandler = PlayersHandler.getInstance();
        cardsHandler = CardsHandler.getInstance();
        this.gameRules = gameRules;
        playersHandler.addPlayers(numberOfPlayers);
    }
```

c. **Composition:** where the **Player** class represents a strong “has-a” relationship with the **Card** class, the object called “hand” is used to store the cards the players has in each.

```java
package Player;

import ...

Edrees Nabeel Ashab *
public class Player {
    2 usages
    private final int index;
    4 usages
    private int roundScore;
    3 usages
    private int gameScore;
    14 usages
    private List<Card> hand;

    Edrees Nabeel Ashab
    public Player(int index) {
        this.index = index;
        roundScore = 0;
        gameScore = 0;
        hand = new ArrayList<>();
    }
```

2. Design patterns:

1. Creational patterns:

Singleton pattern:

This pattern is implemented in **PlayersHandler**, and **CardsHandler** classes, to ensure that these classes has only one instance and provides a global point of access to that instance, so in all the stages of the game, any class that has to access some data or modify it, it can use **PlayersHandler** and **CardsHandler**.

```
package Player;

import ...

Edrees Nabeel Ashab *
public class PlayersHandler {
    3 usages
    private static PlayersHandler instance;
    12 usages
    private final List<Player> players;
    6 usages
    private Player currentPlayer;
    12 usages
    private int turn;
    7 usages
    private int dir;

    1 usage  new *
    public static PlayersHandler getInstance() {
        if (instance == null) instance = new PlayersHandler();
        return instance;
    }
```

2. **Behavioral patterns:**

Template pattern:

Template pattern is used to define the skeleton of the Uno game variation using the **Game** class, this way developers can easily extend it and implement their own variations without modifying the **Game** class.

```
package Game;

import Cards.CardsHandler;
import Player.PlayersHandler;

2 usages  2 inheritors  Edrees Nabeel Ashab
public abstract class Game {
    protected PlayersHandler playersHandler;
    protected CardsHandler cardsHandler;
    11 usages
    protected GameRules gameRules;

    1 usage  1 implementation  Edrees Nabeel Ashab
    public abstract void play();
    2 usages  1 implementation  Edrees Nabeel Ashab
    protected abstract void initializeGame();
    1 usage  1 implementation  Edrees Nabeel Ashab
    protected abstract void discarding();
    1 usage  1 implementation  Edrees Nabeel Ashab
    protected abstract void resetRound();
}
```

3. Clean code principles:

1. **Meaningful name:** in my code I've used meaningful names for variables, functions, classes, and methods to improve code readability and understanding.
   Examples: **PlayersHandler** class that contains all the needed data and methods related to players.
   initializeGame method, its name clearly indicates the purpose it.
   And I also followed the naming conventions such **Pascal Case** for naming classes, **Camel Case** for naming methods and variables.
2. **DRY (Don't Repeat Yourself):** encapsulated repetitive logic into reusable functions and classes.
3. **Separation of Concerns (SoC):** each method and class has a specific purpose in order to manage complexity and improve code readability. Such as **ReverseActionCard**, it only does what its name suggests. Also organized classes in packages to indicate its purpose or functionality.

```
package Cards;

import Player.PlayersHandler;

1 usage  Edrees Nabeel Ashab
public class ReverseActionCard extends Card {
    1 usage  Edrees Nabeel Ashab
    public ReverseActionCard(Color color) { super(color, value: -1, score: 20, CardType.ACTION_REVERSE); }

    2 usages  Edrees Nabeel Ashab
    @Override
    public void applyAction(PlayersHandler playersHandler, CardsHandler cardsHandler) {
        playersHandler.revDir();
        System.out.println("Game direction reversed!!");
    }
}
```

4. Effective Java items:

1. **Item 3: Enforce the singleton property with a private constructor or an enum type.** To ensure that these classes has only one instance and provides a global point of access to that instance.

```java
package Cards;

import java.util.Stack;

Edrees Nabeel Ashab
public class CardsHandler {
    3 usages
    private static CardsHandler instance;
    2 usages
    private final Deck deck;
    3 usages
    private final Stack<Card> discardPile;

    1 usage  Edrees Nabeel Ashab
    private CardsHandler() {
        deck = new Deck();
        discardPile = new Stack<>();
    }

    1 usage  Edrees Nabeel Ashab
    public static CardsHandler getInstance() {
        if (instance == null) instance = new CardsHandler();
        return instance;
    }
```

2. **Item 15: Minimize the accessibility of classes and members.** By using **private** keyword to hide internal data and implementation details from other classes.
   Can be accessed only by using **public** setter and getter methods.

```
3 usages   Edrees Nabeel Ashab
public class Deck {
    16 usages
    private final Stack<Card> deck;
    3 usages
    private final List<Color> colors;
    4 usages
    private Color currentColor;
```

3. **Item 28: Prefer lists to arrays.** An example of it is in **Deck** class above with the list of colors.
4. **Item 58: Prefer for-each loops to traditional for loops.** I've used for-each loops whenever possible.

```
3 usages   Edrees Nabeel Ashab
public class DealRule1 {
    1 usage   Edrees Nabeel Ashab
    public void applyRule(PlayersHandler playersHandler, CardsHandler cardsHandler, int numberOfCards) {
        for (int i = 0; i < numberOfCards; i++) {
            for (Player player : playersHandler.getPlayers()) {
                Card card = cardsHandler.getDeck().drawCard();
                player.addCardToHand(card);
            }
        }
    }
}
```

5. **Item 59: Know and use the libraries.** One of the libraries I've used is the **Collections** library to do shuffling for the deck.

```
Collections.shuffle(deck, new Random(System.currentTimeMillis()));
Collections.shuffle(deck, new Random(System.currentTimeMillis()));
Collections.shuffle(deck, new Random(System.currentTimeMillis()));
```

6. **Item 69: Use exceptions only for exceptional conditions.** Used exceptions only when necessary.

```
6 usages   Edrees Nabeel Ashab *
public Card drawCard() {
    if (!deck.isEmpty()) {
        Card card = deck.peek();
        deck.pop();
        return card;
    } else {
        throw new EmptyDeckException();
    }
}
```

```
2 usages   Edrees Nabeel Ashab
private void chooseDiscard() {
    Scanner sc = new Scanner(System.in);
    System.out.println("Choose a card to discard:");
    int cardIndex = sc.nextInt();
    if (cardIndex >= 0 && cardIndex < playersHandler.getCurrentPlayer().getHandSize())
        if (gameRules.canDiscard(cardsHandler, playersHandler.getCurrentPlayer().getCard(cardIndex)))
            discardCard(cardIndex);
        else
            throw new InvalidDiscardIndex();
    else
        throw new InvalidCardIndexException();
}
```

5. SOLID principles:

1. **Single Responsibility principle:** Each class and function in my code and a single responsibility and a single functionality, increasing cohesion, reducing dependences with other classes, and making code more organized and easy to read, reusable, and modify if needed. **CardsHandler** class is an example, where this class handles all the operations regarding cards. PlayersHandler class is another example, it contains the required data about players, and also how to get the next player's turn, or adding players.
2. **Open/Close principle:** With the design I've implemented the code can be easily extended and built upon, by using some design patterns, generalization, and decomposition principles. For example, in order to add rules, all the developers have to do is create a class and implement the needed functionality, then instantiate the class and call the method, without having to change the existing code.
3. **Liskov's Substitution principle:** To ensure that inheritance is not misused, whenever I extend a class, I keep the expected behavior of that class unchanged. For example, the **DrawTwoActionCard**, **SkipActionCard**, **ReverseActionCard** classes inherits the **Card** class which has a method called **applyAction** which does what its name implies.
4. **Interface Segregation principle:** As in the **GameRules** interface, all classes that implements **GameRules** interface, they all need to have these methods.

```
9 usages  2 implementations  Edrees Nabeel Ashab
public interface GameRules {
    1 usage  1 implementation  Edrees Nabeel Ashab
    void dealCards(PlayersHandler playersHandler, CardsHandler cardsHandler, int numberOfCards);
    1 usage  2 implementations  Edrees Nabeel Ashab
    void discardCard(PlayersHandler playersHandler, CardsHandler cardsHandler, int cardIndex);
    1 usage  1 implementation  Edrees Nabeel Ashab
    boolean canDiscard(CardsHandler cardsHandler, Card card);
    2 usages  1 implementation  Edrees Nabeel Ashab
    boolean haveDiscardableCards(PlayersHandler playersHandler, CardsHandler cardsHandler);
    1 usage  1 implementation  Edrees Nabeel Ashab
    void noValidCardPenalty(PlayersHandler playersHandler, CardsHandler cardsHandler);
    2 usages  1 implementation  Edrees Nabeel Ashab
    int checkRoundWinner(PlayersHandler playersHandler);
    2 usages  1 implementation  Edrees Nabeel Ashab
    int checkGameWinner(PlayersHandler playersHandler);
}
```

5. **Dependency Inversion principle:** In my code, classes doesn't depend on concrete classes, developers can add rules and features without having to depend on the low-level implementation details by using abstractions and interfaces. For example, when implementing the **GameRules** interface, developers can add rules that they created somewhere else or even use predefined rules just by instantiating them and calling their methods.

```java
package Rules;

import ...

1 usage  1 inheritor  Edrees Nabeel Ashab
public class StandardRules implements GameRules {
    1 usage  Edrees Nabeel Ashab
    @Override
    public void dealCards(PlayersHandler playersHandler, CardsHandler cardsHandler, int numberOfCards) {
        DealRule1 dealRule = new DealRule1();
        dealRule.applyRule(playersHandler, cardsHandler, numberOfCards);
    }

    1 usage  1 override  Edrees Nabeel Ashab
    @Override
    public void discardCard(PlayersHandler playersHandler, CardsHandler cardsHandler, int cardIndex) {
        DiscardRule1 discardRule1 = new DiscardRule1();
        discardRule1.applyRule(playersHandler, cardsHandler, cardIndex);
    }
```