Uno Game Engine Report

Fares Qawasmi

Dr. Fahed Jubair

Abstract:

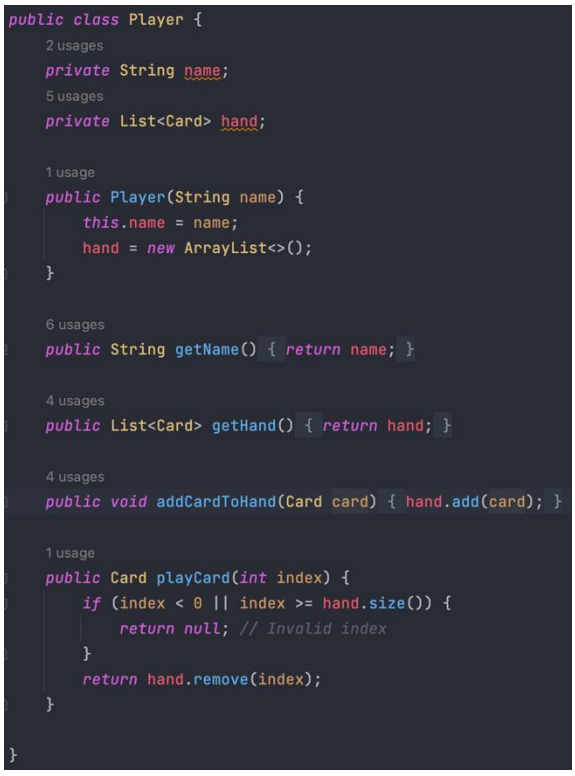
This report presents the development of an Uno game engine using Java and object-oriented programming principles. The engine incorporates an abstract Game class that serves as the foundation for creating custom Uno game variations. By extending the abstract class and implementing necessary methods, developers can easily define their own rules, card types, and game mechanics. The code follows clean code principles and "Effective Java" recommendations to ensure readability, maintainability, and efficiency. Additionally, the implementation adheres to SOLID principles, promoting modularity and extensibility. The report highlights the object-oriented design concepts employed, such as encapsulation, inheritance, and polymorphism, which contribute to a modular and adaptable code structure. Various design patterns are utilized to address common design challenges, enhancing code reusability, and promoting flexible development. The report concludes with a real example of an Uno game variation to demonstrate the engine's functionality and versatility, also as shown in the video documentation.

Object Oriented Concepts:

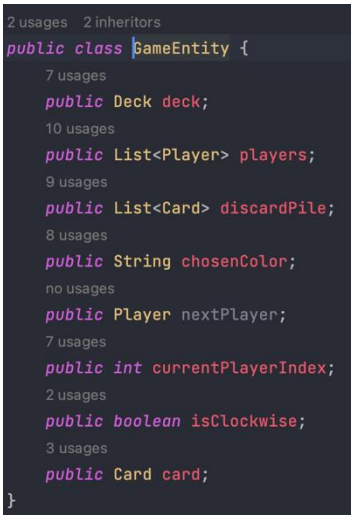
Object-oriented concepts promote code modularity, reusability, and maintainability by organizing code into meaningful entities and establishing relationships between them. They enabled me to create a flexible and extensible Uno game engine that can accommodate different game variations while providing a unified interface for interacting with the game.

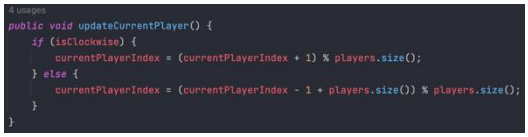
Let's go over these concepts one by one to show how they are implemented in my solution:

Encapsulation was easy to implement as it was a no-brainer this assignment. For an Uno Game Engine, sensitive data must be hidden from the user. In this case, cards, whether value or color, player's hand, and game rules are some examples where getters and setters were heavily needed. The code's encapsulation ensures that these sensitive data elements are encapsulated within the appropriate classes, providing controlled access through public methods. Moreover, the design of the code allows for flexibility, enabling the programmer to change one part of the code without affecting other parts. This flexibility is crucial when developers want to introduce new game variations or modify existing rules while keeping the core functionality intact.



The base of almost any solution for any problem in coding is **Inheritance**, and it wasn't any different for the Uno Game Engine assignment. To create any complex game engine, you will need a significant number of attributes and methods and distributing them between classes with the right relationships would be the right practice. In my solution, main Uno entities like Cards, Players, and Deck are all instantiated in the *GameEntitiy* class, then the *GameAction* class inherits that class. This allows the *GameAction* class to implement crucial functions that relate to the cards and players, basically the building blocks of any Uno game. Separating these classes makes it much more readable and easier to understand. This is just one example of how I used inheritance in my code to allow myself to reuse attributes and methods of an existing class when I create a new class.





The *Rule* abstract class in my code serves as a base for defining specific rules within the Uno game, demonstrating the principle of **Abstraction**. By using the *Rule* abstract class, with the DrawTwoRule, SkipRule, and ReverseRule classes, you achieve a modular and extensible design that promotes code reuse and separates the logic of different rules. This abstraction simplifies the management and organization of rules within the game engine, enhancing the code's maintainability and flexibility while adhering to the principle of Single Responsibility. A developer

could simply create a new rule by creating a new class in the Rule package, then do the necessary implementation for it.

```java
9 usages  5 inheritors
public abstract class Rule {
    2 usages
    private String ruleName;

    5 usages
    public Rule(String ruleName) { this.ruleName = ruleName; }

    1 usage
    public String getRuleName() { return ruleName; }

    1 usage  5 implementations
    public abstract GameRules apply(GameRules action);

}
```

```java
2 usages
public class WildDrawFourRule extends Rule {

    1 usage
    public WildDrawFourRule() { super( ruleName: "Wild Draw Four"); }

    1 usage
    @Override
    public GameRules apply(GameRules action) {
        action.wildDrawFourRule();
        return action;
    }
}
```

The *Game* abstract class provides a blueprint for creating different variations of the Uno game. It defines common attributes and methods for initializing the game, managing players, handling turns, and determining the winner. By extending this class and implementing its abstract methods, developers can easily create their own Uno game variations with custom rules and behavior. This abstraction promotes code reuse, encapsulation, and separation of concerns, making the code flexible, extensible, and maintainable.

```java
1 inheritor
public abstract class Game implements IGame {
    2 usages
    private List<Rule> rules = new ArrayList<>();
    20 usages
    private GameRules action;

    public Game(GameRules gameRules) { action = gameRules; }

    5 usages
    public void addRule(Rule _rules) { rules.add(_rules); }

    1 usage
    public void handleCardEffect(Card card) {
        action.card = card;
        for (Rule rule : rules) {
            if(Objects.equals(card.getValue(), rule.getRuleName()))
                action= rule.apply(action);
        }
    }
```

```java
public class UnoGame extends Game {
    1 usage
    public UnoGame() {
        super(new GameRules());

        addRule(new ReverseRule());
        addRule(new DrawTwoRule());
        addRule(new SkipRule());
        addRule(new WildRule());
        addRule(new WildDrawFourRule());
    }
    2 usages
    @Override
    public void play() {
        super.initializeGame();
        super.play();
    }
```

SOLID Principles:

Prior to explaining the adherence of my code to the SOLID principles, it is worth acknowledging that while certain aspects of these principles were employed in programming prior to knowing them, gaining a comprehensive understanding of them invariably alters one's approach to solving coding problems.

- Single Responsibility Principle (SRP): Each class in my Uno game engine is designed to have a single responsibility. For example, the *Deck* class is responsible for managing the deck of cards, the *Player* class handles player-specific actions, and the *Game* class orchestrates the overall game flow. This division of responsibilities promotes better organization, maintainability, and extensibility which is shown in my code.

- Open/Closed Principle (OCP): I allow developers to extend the functionality of the game engine without modifying the existing code. Developers can create new classes that extend the *Game* class to implement their own variations of the Uno game. The *Game* class provides abstract methods that need to be implemented by the subclasses, enabling the addition of new game rules or features without modifying the core engine.

- Liskov Substitution Principle (LSP): LSP states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. In my code, the LSP is adhered to by ensuring that subclasses of the Game class can be used interchangeably with the Game class itself. Developers can create new classes that extend the Game class to define their own Uno game variations, and these subclasses can be used in place of the Game class without introducing any unexpected behavior or breaking the functionality of the game engine.

- Interface Segregation Principle (ISP): The *IGame* interface defines the common methods that any Uno game implementation should have, such as starting the game, playing a turn, and determining the winner all with play() which would be there for any game variation. By using an interface, the code ensures that clients or developers implementing their own game variations only need to implement the necessary methods for their specific game without being burdened by unused methods. Same thing goes for the rules, I allow developers to selectively choose and implement the desired rules for their game variations, adhering to the ISP by only depending on the relevant rule interfaces.

- Dependency Inversion Principle (DIP): The *Game* class has a private field named action of type GameRules, which represents the game rules associated with the instance of the *Game* class. The GameRules class seems to be an abstraction or interface representing the various game rules. The constructor of the *Game* class accepts a parameter of type GameRules, which is used to initialize the action field. By injecting the GameRules object through the constructor, the *Game* class depends on the abstraction (GameRules) rather than concrete implementations. This design allows for flexibility and customization of game rules by creating different implementations of the GameRules class that extend the GameAction class. Each implementation can define its specific game rules and mechanics while leveraging the shared functionality provided by the GameAction class.It adheres to the DIP by depending on the abstraction, GameRules, rather than specific implementations, promoting loose coupling and facilitating easier maintenance and future enhancements.

Clean Code Principles:

- **Clean code naming** conventions were strictly followed in the code, enhancing readability and clarity. Meaningful variable names such as deck, players, and chosenColor effectively convey their purpose. Consistent use of camelCase for class and method names, such as currentPlayerIndex, handCardEffect(), and isValidPlay(), promotes consistency and distinguishes words within names. Methods are appropriately named as actions, reflecting their functionality, such as playCard(), drawCard(), and nextPlayer(). Abbreviations are avoided in favor of descriptive names, ensuring code comprehension. Class names like Card, Deck, and Player are intuitive and aligned with established conventions. By following these clean code-naming guidelines, the code becomes more readable.

- The code structure exhibits a focus on **single responsibility**, with classes designed to handle specific tasks such as managing the game state, implementing game rules, or controlling the flow of the game. This promotes better code organization, modularity, and ease of understanding, modification, and testing of individual components.

- The **separation of concerns** is evident in the code, with different responsibilities allocated to separate classes and methods. For instance, the *Game* class manages the game state, while the *Rule* classes handle the implementation of specific game rules. This separation enhances code clarity and maintainability.

- The use of abstract classes, interfaces, and design patterns, particularly the Strategy pattern, facilitates **code reusability**. The abstract *Game* class serves as a foundation for creating custom game variations, reducing code duplication, and enabling easy addition or modification of game rules or variations.

- The code structure promotes effective **unit testing**. By adhering to the SOLID principles and separating concerns, individual components can be tested in isolation, ensuring their correctness, and aiding in the identification of bugs or issues. This improves the overall quality and reliability of the codebase.

Design Patterns:

- **Template Method** Pattern: The abstract *Game* class serves as a template for creating Uno game variations. By defining the overall structure and common methods in my abstract class, developers can now easily extend it and provide their own specific implementations for abstract methods to create their own Uno games with little to no effort.

- **Strategy** Pattern: The *Rule* classes, such as DrawTwoRule and SkipRule, implement specific game rules by encapsulating them as separate strategies. This allows for flexibility in adding or modifying game rules without affecting the core game engine. Developers can easily select and combine different rule strategies to create unique game variations.

- **Observer** Pattern: The *Game* class implements the observer pattern by maintaining a list of observers (players) and notifying them of game events such as card plays or game completion. This promotes loose coupling between the game engine and the players, allowing for easy addition or removal of observers and supporting real-time updates for players.

These design patterns enhance code flexibility, modularity, and extensibility. They facilitate the addition of new features, rules, or variations with minimal code changes, as well as promote code reuse and maintainability. I'm sure more design patterns could have been used, or may have been unnoticeably used, however after learning all these topics, I feel that the more your practice them, the more efficient their use will be in future code.

As a quick example of how this would benefit the code, lets imagine that the Strategy pattern was not used. All the rules would be implemented in one class and would result in a long if-else statement to know which rule is going to be used. Additionally, this violates the Open-Closed Principle, where adding a new rule will need a developer to open the method and add a new rule inside it instead of just creating a new class for their own rule and do the necessary implementation for it.

Effective Java Items:

- Item 5: Prefer dependency injection to hardwiring resources: This item is demonstrated in the code through the usage of dependency injection to provide necessary resources and dependencies to classes. A different example than the one mentioned in the section of SOLID principles, is the GameAction class, for example, the Deck instance is injected into the class through the constructor. By relying on dependency injection instead of hardwiring the Deck instance within the class itself, the code becomes more flexible and reusable.

```
public GameAction(){
    deck = new Deck();
    discardPile = new ArrayList<>();
    players =  new ArrayList<>();
    chosenColor = "";
    currentPlayerIndex  = 0;
    isClockwise = true;
}
```

- Item 10: Observe the general contract when overriding equals(): The *Card* class overrides the equals() method to provide custom equality comparison based on the card's value and color. By following the general contract of equality, I ensured consistent and reliable behavior when comparing card objects.

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }
    Card otherCard = (Card) obj;
    return Objects.equals(value, otherCard.value) && color.equals(otherCard.color);
}
```

- Item 15: Minimize the accessibility of classes and members: An example of minimizing accessibility can be seen in the *Deck* class. The initializeDeck() method is marked as private, ensuring that it can only be accessed within the Deck class itself. By encapsulating the deck initialization logic, the class maintains control over its internal state and prevents other classes from directly modifying or manipulating the deck. This was followed throughout the whole code.

```
1 usage
private void initializeDeck() {
    String[] colors = {"Red", "Green", "Blue", "Yellow"};
    String[] values = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
            "Skip", "Reverse", "Draw Two"};

    for (String color : colors) {
        for (String value : values) {
            cards.add(new Card(color, value));      // Add number cards
        }

        for (int i = 0; i < 2; i++) {
            cards.add(new Card(color,  value: "Skip"));
            cards.add(new Card(color,  value: "Reverse"));      // Add action cards
            cards.add(new Card(color,  value: "Draw Two"));
        }
    }

    // Add wild cards
    for (int i = 0; i < 4; i++) {
        cards.add(new Card( color: "Wild",  value: "Wild"));
        cards.add(new Card( color: "Wild",  value: "Wild Draw Four"));
    }
}
```

- Item 17: Minimize mutability: Throughout my code, I prioritize immutability for objects like cards and players. Once created, these objects cannot be modified, promoting simplicity, thread safety, and reducing the risk of unintended changes.

- Item 18: Favor composition over inheritance: This is shown in the relationship between the *Game* class and the *GameRules* class. Instead of relying on inheritance, the *Game* class uses composition to interact with different game rules implementations through the GameRules interface, providing flexibility and extensibility.

- Item 12**:** Override toString(): The *Card* class overrides the toString() method to provide a meaningful string representation of a card, displaying its value and color.

```
@Override
public String toString() {
    return color + " " + value;
}
```

- Item 59: Know and use the libraries: In the *Deck* class, I used the Collections.shuffle() method to shuffle the deck of cards, leveraging the built-in functionality provided by the Java Collections framework.

```
1 usage
public void shuffle() {
    Collections.shuffle(cards);
}
```

- • Item 49: Check parameters for validity: In the Player class, the playCard(int index) method includes a check to ensure that the provided index is within the valid range of the player's hand. This practice promotes code stability and helps catch programming mistakes early on.

```
1 usage
public Card playCard(int index) {
    if (index < 0 || index >= hand.size()) {
        return null;        // Invalid index
    }
    return hand.remove(index);
}
```