Atypon Java & DevOps | 2024 Summer Cohort

Uno Game Assignment

Mahmoud J. Qudah

Introduction:

This report explores the creation of an Uno game engine developed using Java, emphasizing strong object-oriented principles such as modularity, cohesion, and adherence to SOLID principles. The core of the engine is an abstract Game class, which provides a flexible framework for building various Uno game variants. Developers can easily customize the game by extending this class and implementing specific rules, card behaviors, and game mechanics.

The design prioritizes best practices in software engineering, incorporating advanced Java techniques from "Effective Java" to enhance code quality, readability, and maintainability. By focusing on reducing coupling and enhancing cohesion, the codebase is both modular and adaptable. SOLID principles are rigorously applied to ensure that the system is both extensible and easy to modify or scale.

To address common design challenges, the implementation leverages several design patterns, which contribute to greater code reusability and flexibility. Key object-oriented concepts like encapsulation, inheritance, and polymorphism are integral to the design, ensuring a robust and adaptable structure. The report concludes with a demonstration of the engine's capabilities through a specific Uno game variation, showcasing its versatility and effectiveness.

Key concepts that have covered on this assignment:

1) Object oriented programming & design

2) SOLID principles

3) Design patterns

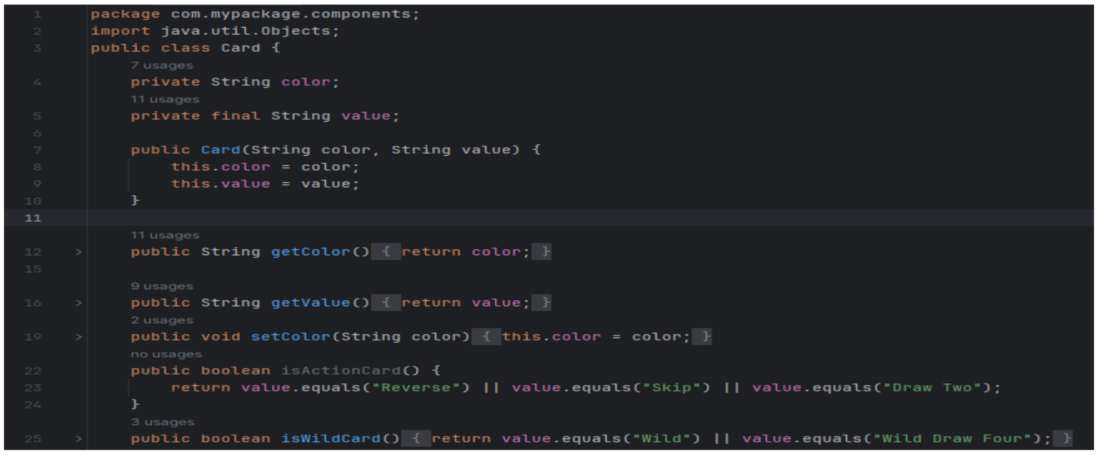
4) Java items from *Effective Java*

5) Clean Code snippets

1. Object oriented programming & design:

Object-Oriented Programming (OOP) is a fundamental paradigm in software development, and its application is crucial to the design and implementation of the Uno game engine. In this assignment, OOP principles like encapsulation, abstraction, and inheritance play a central role in creating a modular, maintainable, and flexible codebase.

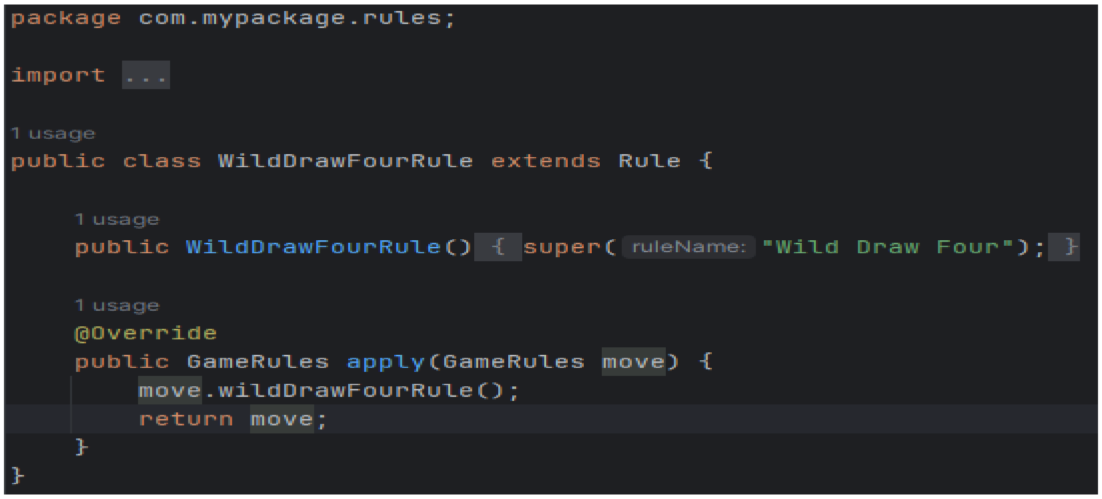
Encapsulation is the practice of bundling attributes and methods that operate on the data into a single unit, typically a class. In the context of the Uno game engine, encapsulation ensures that each class manages its own state and behavior, reducing the risk of unintended interference from other parts of the code. For instance, the Card class encapsulates the properties of a card, such as its color and value, along with the methods that operate on these properties. By keeping the card's attributes private and providing public getter methods, we protect the internal state of the card from being altered directly by other classes, which maintains the integrity of the game logic.



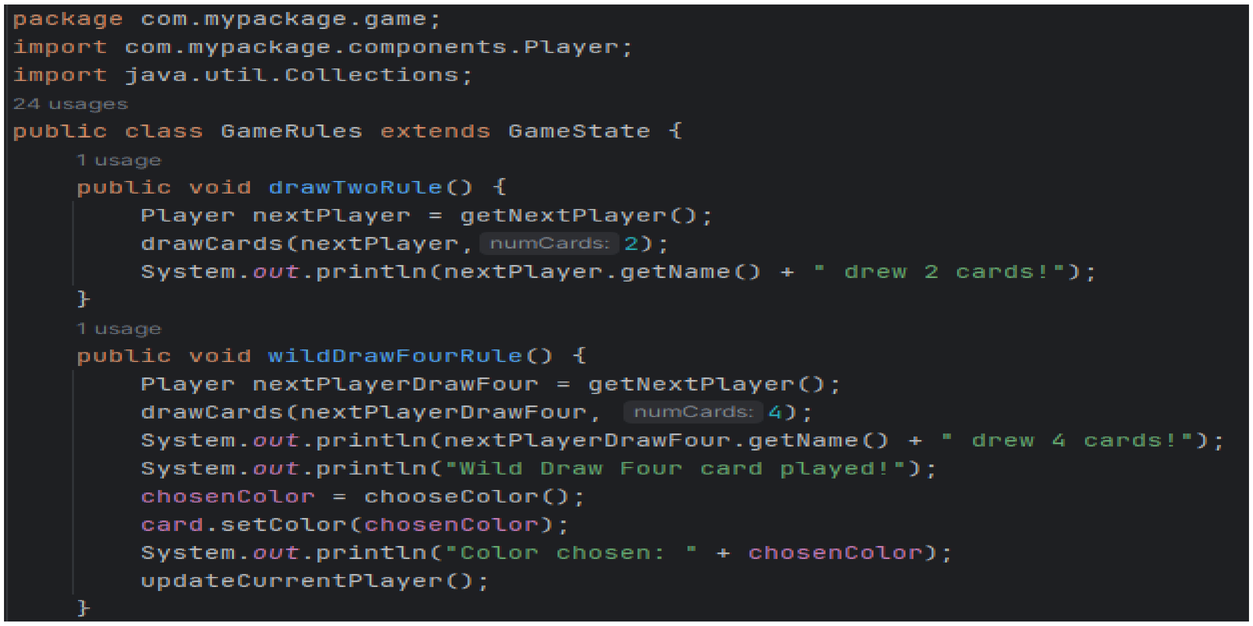
```java
package com.mypackage.components;
import java.util.Objects;
public class Card {
    7 usages
    private String color;
    11 usages
    private final String value;

    public Card(String color, String value) {
        this.color = color;
        this.value = value;
    }

    11 usages
    public String getColor() { return color; }

    9 usages
    public String getValue() { return value; }
    2 usages
    public void setColor(String color) { this.color = color; }
    no usages
    public boolean isActionCard() {
        return value.equals("Reverse") || value.equals("Skip") || value.equals("Draw Two");
    }
    3 usages
    public boolean isWildCard() { return value.equals("Wild") || value.equals("Wild Draw Four"); }
```

Abstraction allows us to focus on the essential features of an object while hiding the unnecessary details. In this assignment, abstraction is achieved using abstract classes and interfaces. The abstract Game class, for example, defines the core structure and common behavior that all game variants should have, without dictating the specific details of each variant. This abstraction enables developers to create different versions of the Uno game by simply extending the Game class and implementing the required methods, without worrying about the underlying implementation of other game mechanics. Interfaces further promote abstraction by defining a contract that implementing classes must adhere to, ensuring consistency across different components of the game.

Example of Abstraction:



```java
package com.mypackage.rules;

import ...

1 usage
public class WildDrawFourRule extends Rule {

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

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


```
package com.mypackage.game;
import com.mypackage.components.Player;
import java.util.Collections;
24 usages
public class GameRules extends GameState {
    1 usage
    public void drawTwoRule() {
        Player nextPlayer = getNextPlayer();
        drawCards(nextPlayer, numCards: 2);
        System.out.println(nextPlayer.getName() + " drew 2 cards!");
    }
    1 usage
    public void wildDrawFourRule() {
        Player nextPlayerDrawFour = getNextPlayer();
        drawCards(nextPlayerDrawFour, numCards: 4);
        System.out.println(nextPlayerDrawFour.getName() + " drew 4 cards!");
        System.out.println("Wild Draw Four card played!");
        chosenColor = chooseColor();
        card.setColor(chosenColor);
        System.out.println("Color chosen: " + chosenColor);
        updateCurrentPlayer();
    }
```

Inheritance is a powerful OOP feature that allows a new class to inherit properties and behaviors from an existing class. This concept is utilized extensively in the Uno game engine to promote code reuse and reduce redundancy. The abstract Game class is a prime example of inheritance in action. By creating a base class that encapsulates the common logic shared by all game variants, we can easily extend this class to create new game types without duplicating code.

2. SOLID Principles:

2.1 Single Responsibility Principle (SRP):

The Single Responsibility Principle emphasizes that a class should have only one reason to change, meaning it should have a single responsibility. In the provided code, each class focuses on a specific role:

Card Class: Manages the properties and behaviors of a card, such as its color and value, and provides methods to check if it's an action or wild card.

Deck Class: Handles the initialization and shuffling of the deck, as well as drawing cards. It does not concern itself with the rules of the game or how cards are played.

Player Class: Manages the player's hand and provides methods to play a card or add a card to the hand.

Rule Class and its Subclasses: Each rule (e.g., SkipRule, ReverseRule) has its own class, responsible for implementing a specific game rule.

This separation of responsibilities ensures that changes in one part of the code do not unnecessarily affect others, promoting cohesion and reducing coupling.

2.2 Open/Closed Principle (OCP):

The Open/Closed Principle states that classes should be open for extension but closed for modification. This principle is evident in the design of the game rules:

Rule Class and Subclasses: The abstract Rule class allows for the creation of new rules by extending it without modifying existing code. For example, the SkipRule, WildRule, and DrawTwoRule classes extend the Rule class to implement specific behaviors. This design allows new rules to be added easily without altering the existing codebase, adhering to the OCP.

Game Class: The Game class is designed to accommodate new rules by simply adding instances of new Rule subclasses, further illustrating the application of the OCP.

2.3 Liskov Substitution Principle (LSP):

The Liskov Substitution Principle ensures that subclasses should be substitutable for their base classes without altering the correctness of the program. In this code:

Rule Subclasses: Each rule class (e.g., SkipRule, DrawTwoRule) inherits from the Rule base class. These subclasses can be used interchangeably in the Game class without affecting the behavior of the game, demonstrating adherence to the LSP

2.4 Interface Segregation Principle (ISP):

The Interface Segregation Principle suggests that clients should not be forced to implement interfaces they do not use. In this code, the principle is applied using specific interfaces:

IGame Interface: The **IGame** interface provides only the necessary methods (play, initializeGame) for a game implementation. This ensures that any class implementing the **IGame** interface, like **UnoGame**, is not burdened with unnecessary methods, adhering to the ISP.

2.5 Dependency Inversion Principle (DIP):

The Dependency Inversion Principle advocates for the dependency on abstractions rather than concrete classes. This principle is evident in:

Rule and Game Relationship: The Game class depends on the **abstract Rule** class rather than concrete rule implementations. This design allows the Game class to work with any rule that extends the Rule class, promoting flexibility and reducing tight coupling between game logic and specific rule implementations.

Game Initialization: The Game class is constructed using **GameRules**, an abstraction, which allows for different game rules to be injected, further emphasizing the DIP.

3. Design patterns:

3.1 Factory Method Pattern:

Class Involved: **PlayableStrategyFactory**

Description: The PlayableStrategyFactory class is a clear example of the Factory Method pattern. It is used to create different strategy objects (ColorMatchStrategy, ValueMatchStrategy, and WildCardStrategy) based on the properties of a Card object.

3.2 Strategy Pattern:

Classes Involved: **PlayableStrategy**, **ColorMatchStrategy**, **ValueMatchStrategy**, **WildCardStrategy.**

Description: The Strategy pattern is used here to encapsulate the different algorithms for determining whether a card can be played. Each strategy (ColorMatchStrategy, ValueMatchStrategy, etc.) implements the PlayableStrategy interface and defines the canPlay method.

3.3 Template Method Pattern

Classes Involved: **Rule and its subclasses (SkipRule, DrawTwoRule, etc.).**

Description**:** The Rule class can be considered a template method pattern where the core structure of applying a rule is defined, and the actual application of specific rules is left to the subclasses.

3.4 Chain of Responsibility Pattern:

Classes Involved: **Game and Rule**

Description: In the **handleCardEffect** method of the **Game** class, a card is processed through a chain of rules. Each rule checks if it applies to the current card, and if so, it modifies the game state.

3.5 Observer

The Game class utilizes the Observer pattern by keeping a list of observers (players) and notifying them of game events like card plays or **game** completion. This design fosters loose coupling between the game engine and the players, enabling the easy addition or removal of observers and ensuring real-time updates for players

4. Java items Effective Java" Items (Jushua Bloch) *with examples :*

Item 1: Consider Static Factory Methods Instead of Constructors
The PlayableStrategyFactory uses static factory methods (createStrategies) to create instances of PlayableStrategy. This follows the advice of preferring static factory methods over constructors for creating objects, which offers better control over the instance creation process and can return instances of any subtype of the return type.

Item 5: Prefer dependency injection to hardwiring resource acquisition
Flexibility: It allows for easier testing and modification. For example, you can inject a mock Deck for testing without changing the Game class.
Decoupling: The class becomes less dependent on specific implementations and more focused on its core functionality.

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

Item 23: Prefer Interfaces to Abstract Classes
The PlayableStrategy is an interface, which is preferred over abstract classes because it allows for greater flexibility. Multiple interfaces can be implemented by a class, whereas it can only extend one class.

Item 59: Know and use the libraries
In the Deck class, I used the Collections.shuffle() method to shuffle the deck of cards

Item 17: Minimize Mutability
The Rule and PlayableStrategy objects appear to be stateless, which aligns with the principle of minimizing mutability. Immutable objects are inherently thread-safe and easier to reason about.

Item 34: Use Interfaces Only to Define Types
The PlayableStrategy interface is used purely to define a type and does not contain any implementation details. This is a good practice that keeps the separation of concerns clear.

Item 55: Return Empty Collections or Arrays, Not Nulls
In PlayableCards.getPlayableCards, an empty list is returned if no cards are playable. This is a good practice that avoids potential NullPointerExceptions and simplifies the handling of collections

5. Clean Code by uncle Bob:

In designing the solution, we adhere to the clean code principles outlined by Uncle Bob, focusing on readability, maintainability, and simplicity. The code is structured to ensure clarity and ease of understanding, with meaningful variable and method names that convey their purpose clearly. For instance, the Card and Deck classes encapsulate their respective responsibilities with straightforward methods like drawCard and shuffle, adhering to the Single Responsibility Principle. Encapsulation is rigorously applied to hide internal states and expose only necessary functionalities through public methods, thus protecting the integrity of the data and reducing the risk of unintended interactions. Additionally, the use of dependency injection and interface-based programming promotes flexibility and modularity, allowing for easier testing and future enhancements. This adherence to clean code practices ensures that the solution remains robust, scalable, and maintainable, facilitating effective collaboration and ongoing improvements.

Thank You.