Report

Object-Oriented Design in the Code:

The code exhibits object-oriented design principles by effectively utilizing classes and objects to represent various entities and their interactions. It employs encapsulation to encapsulate data and behavior within classes and uses inheritance and interfaces for code reuse and flexibility. The Card, SpecialCard, Deck, Player, and Game classes each have well-defined responsibilities, ensuring a clear separation of concerns and promoting modularity.

Design Patterns Used in the Code:

The code incorporates the following design patterns:

1-Factory Method: The CardFactory interface and its implementations, BasicCardFactory and SpecialCardFactory, implement the Factory Method pattern. This pattern provides a way to create cards without exposing the creation logic to the client (Deck class), promoting loose coupling and extensibility.

2-Observer: The GameObserver interface and its implementation in the Player class follow the Observer pattern. It allows players to observe game events and be notified accordingly. This decouples the game logic from the players and enables better maintainability and extensibility.

3-Template Method: The Game class, along with its subclass UnoGame, employs the Template Method pattern. The Game class defines the overall game algorithm in the play() method, while subclasses can override specific steps to customize the game behavior. This pattern promotes code reuse and modularity.3. Algorithm Description

Clean Code Principles (Uncle Bob)

a defense of the code against some of the Clean Code principles proposed by Uncle Bob:

1- Single Responsibility Principle (SRP): The code demonstrates a reasonable level of adherence to SRP by assigning clear responsibilities to classes such as Card, Deck, Player, and Game. However, there might be areas for improvement. For example, the Game class could be further split into separate classes responsible for game flow and game rules.

2- Open-Closed Principle (OCP): The code follows OCP to some extent, as new card types can be added by implementing the CardFactory interface without modifying existing code. However, there might be room for improvement by applying the OCP more rigorously to other aspects of the game, such as introducing new game modes or extending player behavior.

3- Liskov Substitution Principle (LSP): The code exhibits LSP as the UnoGame subclass can be substituted for the base Game class without breaking the code's correctness. Subclasses adhere to the contract defined by the base class.

4- Interface Segregation Principle (ISP): The code adheres to ISP by providing specific interfaces (CardFactory and GameObserver) that define focused contracts tailored to their respective responsibilities. This allows clients to implement only the necessary methods, avoiding the burden of implementing unnecessary ones.

5- Dependency Inversion Principle (DIP): The code partially follows DIP by depending on abstractions (CardFactory and GameObserver) rather than concrete implementations. However, there might be opportunities to further decouple dependencies by introducing dependency injection frameworks or applying inversion of control principles.

Effective Java Items (Joshua Bloch)

A defense of the code against some of the "Effective Java" items proposed by Joshua Bloch:

1. **Item 1:** Consider static factory methods instead of constructors: The code does not explicitly utilize static factory methods. However, it employs the CardFactory interface to provide a flexible mechanism for creating different types of cards, which is similar to the concept of static factory methods. This allows for improved encapsulation and future extensibility.

2. **Item 2:** Consider a builder when faced with many constructor parameters: The code does not have classes with excessive constructor parameters. However, if the Card class were to have many optional parameters, implementing a builder pattern could provide a more readable and flexible way to construct Card objects.

3. **Item 9:** Prefer try-with-resources to try-finally: The code does not explicitly deal with resource management, so there are no opportunities to apply try-with-resources. However, if the code were to include file or network operations, adopting try-with-resources for proper resource handling would be advisable.

4. **Item 15:** Minimize mutability: The Card and Player classes are effectively immutable, as they do not provide any public setters and the state of the objects cannot be changed after creation. This promotes immutability, which has benefits such as simplifying reasoning about the code and thread safety.

5. **Item 16:** Favor composition over inheritance: The code demonstrates composition over inheritance by using the Card and Player classes as components within the Deck and Game classes, respectively. This approach allows for greater flexibility, promotes code reuse, and avoids the pitfalls associated with inheritance.

While the code may not explicitly implement all the items mentioned in "Effective Java," it aligns with several principles and exhibits good practices. It emphasizes encapsulation, promotes immutability where applicable, utilizes composition over inheritance, and provides flexibility through the use of interfaces and abstract classes.

SOLID principles

A defense of the code against SOLID principles:

1- Single Responsibility Principle (SRP): The classes in the code have clear responsibilities. For example, the Card class represents a card with color and value, the Deck class manages the deck of cards, the Player class represents a player in the game, and the Game class handles the game logic. Each class focuses on a single aspect of the overall functionality, adhering to the SRP.

2- Open/Closed Principle (OCP): The code exhibits a level of extensibility through the use of interfaces (CardFactory, GameObserver) and abstract classes (Game). The CardFactory interface allows for different implementations to create cards, enabling the introduction of new card types without modifying existing code. Similarly, the GameObserver interface allows for easy addition of new observers to track game events. The Game class, being abstract, can be extended to create different game implementations while reusing the common game logic.

3- Liskov Substitution Principle (LSP): The code does not explicitly demonstrate inheritance hierarchies that could violate the LSP. However, it utilizes the Card class as a base class and extends it with the SpecialCard class, which is a valid and substitutable specialization of a card. The SpecialCard class can be used wherever a Card is expected, preserving the behavior and adhering to the LSP.

4- Interface Segregation Principle (ISP): The code employs interfaces (CardFactory, GameObserver) that define specific contracts for implementing classes. This allows clients to depend on specific interfaces

instead of broad ones, ensuring they are only exposed to the methods they require. By separating the interfaces, the code adheres to the ISP and avoids forcing clients to depend on methods they don't need.

5- Dependency Inversion Principle (DIP): The code demonstrates the DIP by relying on abstractions (CardFactory, GameObserver) rather than concrete implementations. The Game class depends on the abstractions instead of concrete classes, promoting loose coupling and facilitating the introduction of different implementations without modifying the high-level code.

Overall, the code shows adherence to the SOLID principles by promoting modularity, extensibility, flexibility, and reusability through interfaces, abstract classes, and clear separation of responsibilities. It enables easier maintenance, modification, and extension of the codebase.