UNO Game

- Object Oriented Design (OOD):
  If you notice in my code that the Card and Deck class obey the **Encapsulation** mechanism, how? First if we look into the Card class it represents a single card and encapsulate the value and color of the card and so on for the Deck class.

  And there is the Game class which is an abstract class which is obey the **Abstraction** technique.

  Also, the **Inheritance** technique is applied in the code by MyGame class that extends the abstract class game and do the implementation of Game methods and implement it's own.

  Polymorphism also applied through methods overriding.

Relations Applied:

1. **Composition**: with My Game and Deck Classes.
2. **Aggregations**: with My Game and Card classes.

- Design Patterns:

A mix of Design Patterns used in my code and that's normal because each of the design patterns has it's own power, so if we say that combining them together make the design better it make sense.

Some of them:

1. **Singelton**: the Color and Value enums exhibit characteristics of the Singleton Pattern. Enum instances are implicitly singletons, providing a single point of access for each constant value.

2. **Factory Method**: the code structure looks like the Factory Method pattern in the sense that the MyGame class extends the abstract **Game** class, and it's responsible for creating and managing the game. The initializeGame method in MyGame acts like a factory method that initializes the game.

3. **Strategy**: Allowing different games to have different strategies for initialization such as MyGame class that extends Game class and implement it's own CustomRules.

- Defending against clean code principles:

As I see that my code is well structured, but there is some areas that need to be improved more, but because the short time I have I will mention some of them:

1. **Method Length**: some methods such as applyCustomRules in MyGame class is tend to be a little bit long than it must be, The separation is required to let the method to do one specific job and not to handle many cases.

2. **Comments:** for explaining complex logic of methods.

3. **Separation of Concerns:** MyGame class handles both game logic and user input (with the scanner).

- Defending against Effective Java:

1. **Item1:** Consider static factory methods instead of constructors.

2. **Item8:** Obey the general contract when overriding equals.

3. **Item10:** Always override hashCode when you override equals.

- Defending against SOLID principles:

1. **Single Responsibility Principle (SRP):** Each class in my code seems to have a single responsibility. For example, Card represents a card in the game, Deck manages the deck of cards.

2. **Open/Closed Principle (OCP):** My code seems open for extension and that's mean new games can be added by extending the Game class.

3. **Liskov Substitution Principle (LSP):** Ensure that any behavior expected from the base type Game is maintained in its subtypes without unexpected side effects.

4. **Interface Segregation Principle (ISP):** My code doesn't explicitly use interfaces, but the classes have well-defined methods.

5. **Dependency Inversion Principle (DIP):** My code seems to depend on abstractions rather than concrete implementations which good for flexibility.