ATYPON

Uno Game Engine Assignment

Made by: Samer AlHamdan

Instructors:
Motasim Aldiab
Fahed Jubair

1. Introduction

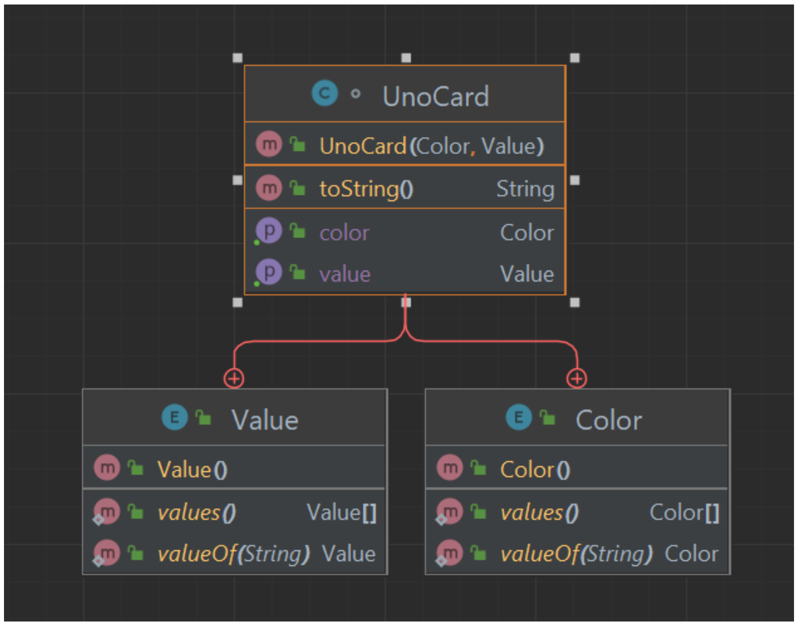
This report offers a comprehensive analysis of the Uno Game Engine Assignment during Atypon's Training in May 2023. It explores the object-oriented design, design patterns used, and ultimately the defense of my code against clean code principles , against “Effective Java” and against SOLID principles.

2. OOP Design and Classes

The design of the Uno Game has 3 parts , so i will describe its designs and purpose

1. The part that is responsible for identifying each individual card **UnoCard,** the class demonstrates good object-oriented design principles, such as encapsulation, abstraction, and information hiding.
    - **Encapsulation:** The class encapsulates the card's color and value using private instance variables. Access to these variables is provided through getter methods (`getColor()` and `getValue()`), allowing controlled access to the card's properties.
    - **Abstraction:** The class abstracts the concept of an Uno card, representing its properties (color and value) through the `Color` and `Value` enums. This abstraction simplifies card handling and allows for type-safe operations when working with cards.
    - **Information Hiding:** The internal state of a `UnoCard` object is hidden from the outside world, as the class declares its instance variables as private. Access to the card's properties is only possible through the public getter methods, ensuring proper encapsulation and preventing direct modification of the card's state.
    - **Immutability:** The `UnoCard` class follows immutability principles by declaring its color and value variables as `final`. Once created, a `UnoCard` instance cannot be modified, ensuring that the state of a card remains consistent throughout its lifetime.
    - **Proper ToString Implementation:** The `toString()` method is overridden to provide a meaningful representation of a `UnoCard` instance. It returns a string combining the color and value of the card, allowing easy printing or displaying of the card's information.

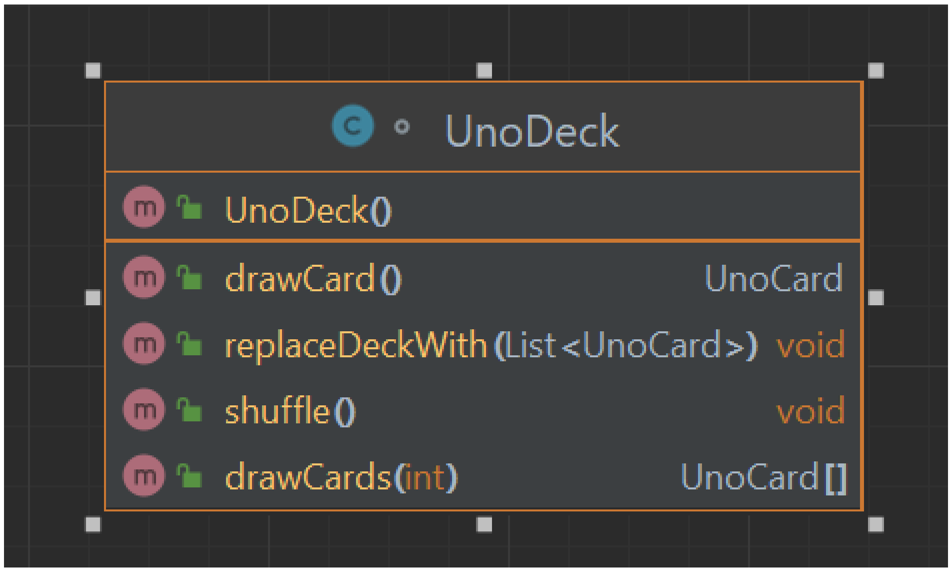
UML for the Class:



2. The part that is responsible for the deck **UnoDeck**, it adheres to good OOP design principles by encapsulating the deck, abstracting its concept, and ensuring information hiding and composition. It provides methods for shuffling the deck, drawing cards, and replacing the deck, offering the necessary functionality to manage an Uno deck effectively.
    - **Encapsulation:** The class encapsulates the deck of Uno cards using a private instance variable `cards`. Access to the deck is provided through public methods, such as `shuffle()`, `drawCard()`, `drawCards()`, and `replaceDeckWith()`. This encapsulation ensures that the internal state of the deck is protected and can only be modified through defined operations.
    - **Abstraction:** The class abstracts the concept of an Uno deck, representing it as a collection of `UnoCard` objects. The deck is populated with cards during construction using loops and conditionals based on the `Color` and `Value` enums from the `UnoDeck` class. This abstraction allows for a simplified representation of a deck of Uno cards.
    - **Information Hiding:** The internal state of the `UnoDeck` object is hidden from external access, as the `cards` variable is declared as private. Access to the deck's contents is provided through specific methods, ensuring that the deck's integrity and consistency are maintained.

- **Composition:** The `UnoDeck` class is composed of `UnoCard` objects. It maintains a list (`ArrayList`) of `UnoCard` instances to represent the deck. This composition relationship allows the `UnoDeck` class to manage and manipulate the collection of cards effectively.
- **Randomization:** The `shuffle()` method uses the `Collections.shuffle()` method to randomize the order of cards in the deck. This randomness adds a crucial element to the Uno game, ensuring unpredictability and fairness in the card distribution.
- **Return Values:** The `drawCard()` and `drawCards()` methods return `UnoCard` objects, representing cards drawn from the deck. This allows the calling code to receive and utilize the drawn cards for further processing or gameplay.

UML for the Class:



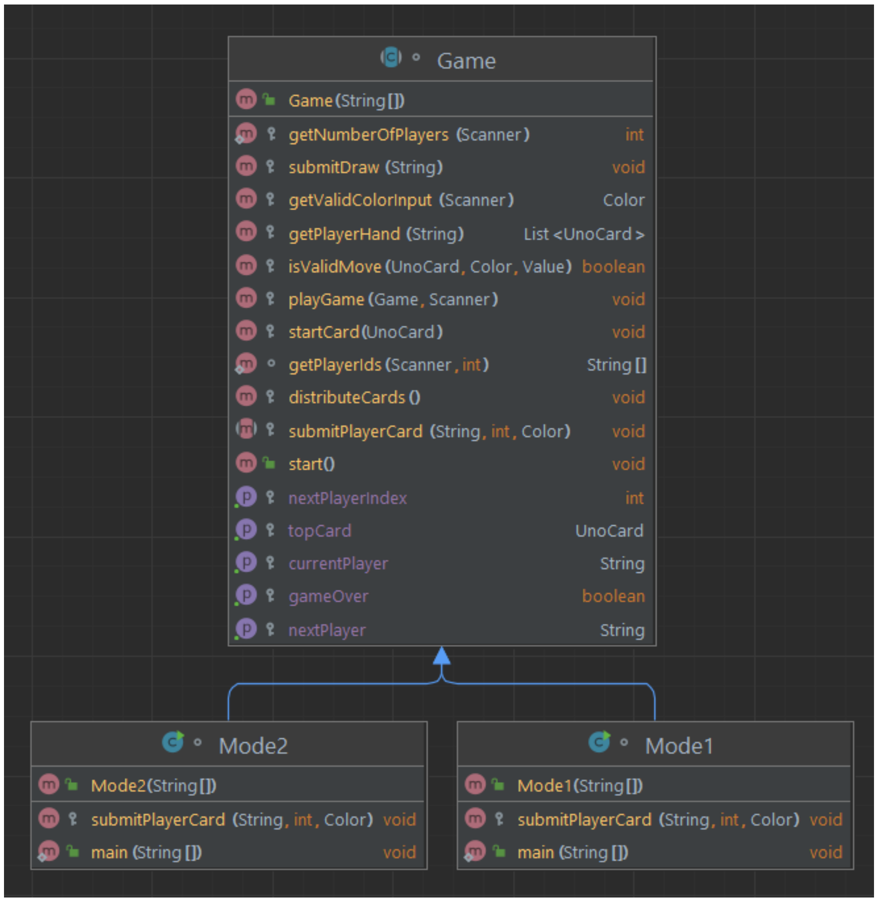
3. The last part which contains the main part of the project which is the abstract class Game which will extend from it other classes to create different type of game modes i have made 2 of them to demonstrate how it works simply. The Game class demonstrates good OOP design principles by encapsulating game-related data and behavior, utilizing inheritance and polymorphism, and following the principles of abstraction, composition, SRP, and DIP. It provides a solid foundation for implementing specific game variations by extending the class and implementing the necessary methods.

- **Inheritance:** The `Game` class is an abstract class, which suggests the intention of being extended by concrete game implementations. It provides a common structure and behavior for different types of games.
- **Encapsulation:** The class encapsulates the game-related data and functionality. The instance variables such as `playerIds`, `deck`, `pile`, `playerHands`, `currentPlayer`, `gameDirection`, `gameOver`, and `scanner` are all declared as protected or private, ensuring that they are accessed only within the class or its subclasses.
- **Abstraction:** The `Game` class abstracts the concept of a generic game. It provides common methods and attributes that can be utilized by specific game implementations. It contains abstract methods, such as `submitPlayerCard()`, that must be implemented by the concrete game classes.
- **Composition:** The `Game` class consists of various components, such as `UnoDeck`, `List<UnoCard>`, and `Map<String, List<UnoCard>>`, to represent the deck, pile, and player hands, respectively. This composition relationship allows the class to manage and manipulate these components effectively.
- **Polymorphism:** The class utilizes polymorphism by defining abstract methods, such as `submitPlayerCard()`, that must be implemented by its subclasses. This allows for different game implementations to have their own specific logic for handling player moves.
- **Single Responsibility Principle (SRP):** The `Game` class focuses on managing the overall game flow and state. It handles operations such as distributing cards, starting the game, determining valid moves, and checking for game over conditions. This adherence to the SRP helps in maintaining a clear and concise class design.
- **Dependency Inversion Principle (DIP):** The class depends on abstractions, such as the `UnoDeck` class, rather than concrete implementations. This promotes loose coupling and allows for flexibility in using different deck implementations.
- **Input/Output Separation:** The code separates the input/output operations from the core game logic. The `scanner` class is used for user input, and messages are printed to the console. This separation allows for easier testing and potential future adaptation to different input/output sources.

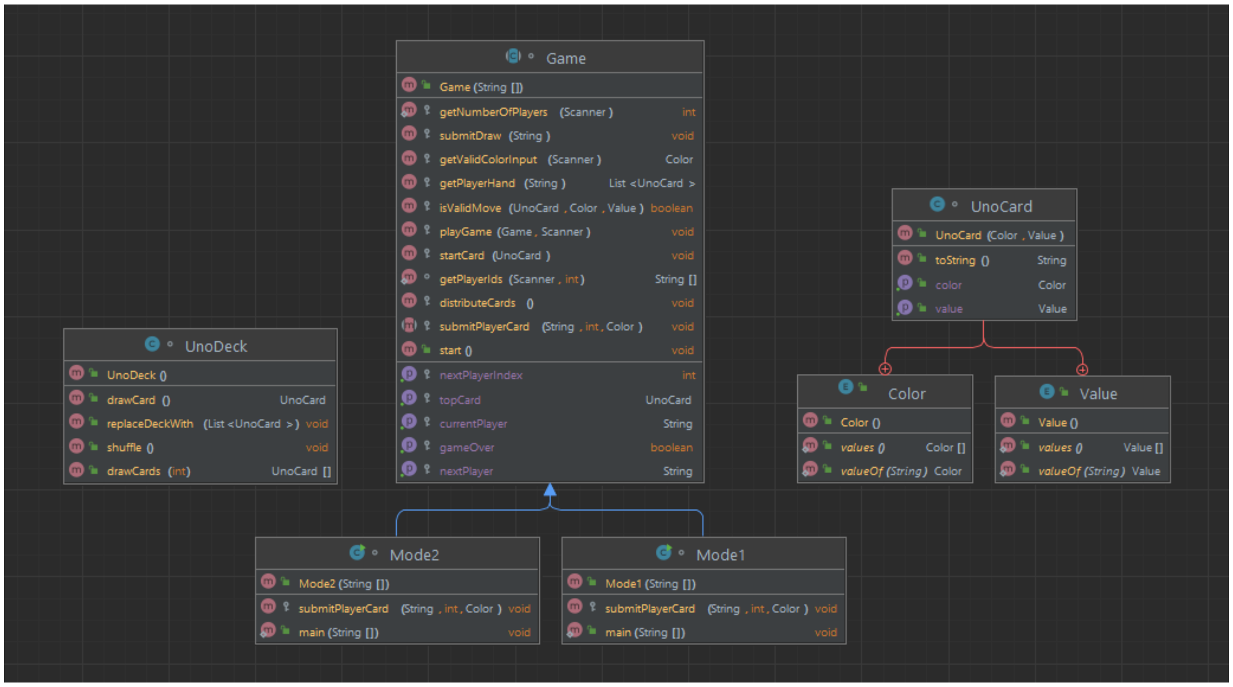
The modes that i have created class demonstrates good adherence to OOP design principles by extending the `Game` class, encapsulating specific behavior, implementing the SRP, and utilizing inheritance and polymorphism. It provides a concrete implementation of the "Mode1" game mode while reusing the generic game logic from the parent class.

- **Inheritance:** The modes classes extends the `Game` class, inheriting its attributes and methods. This allows the modes to reuse and override behavior from the parent class.
- **Polymorphism:** The modes classes override the `submitPlayerCard` method from the `Game` class, providing its specific implementation. This demonstrates polymorphism, where the method can be called based on the type of the modes objects, even when referred to as a `Game` object.
- **Encapsulation:** The class encapsulates its own data and behavior, as well as the inherited members from the `Game` class. It accesses and modifies game-related data using the protected methods and attributes inherited from the parent class.
- **Single Responsibility Principle (SRP):** The modes classes focus on implementing the specific rules and behavior for the modes game mode. It overrides the `submitPlayerCard()` method to handle card submissions according to the mode's rules. This adherence to the SRP ensures that the class has a clear and specific responsibility.
- **Code Organization:** The class includes a `main` method that sets up and starts the game. This separation of game setup and execution logic from the class's main structure improves code organization and readability.
- **Method Overriding:** The modes classes override the `submitPlayerCard()` method to tailor its behavior to the specific rules of "Mode1." This demonstrates the use of method overriding to provide a different implementation based on the specific game mode.

UML for the Classes:


Game
Game(String[])
getNumberOfPlayers (Scanner) int
submitDraw (String) void
getValidColorInput (Scanner) Color
getPlayerHand (String) List<UnoCard>
isValidMove (UnoCard, Color, Value) boolean
playGame (Game, Scanner) void
startCard(UnoCard) void
getPlayerIds (Scanner, int) String[]
distributeCards() void
submitPlayerCard (String, int, Color) void
start() void
nextPlayerIndex int
topCard UnoCard
currentPlayer String
gameOver boolean
nextPlayer String
Mode2
Mode2(String[])
submitPlayerCard (String, int, Color) void
main (String[]) void
Mode1
Mode1(String[])
submitPlayerCard (String, int, Color) void
main (String[]) void

Uml for the whole project:



3. Design Patterns

In the `UnoCard` class, there are no explicit design patterns being used. While the `UnoCard` class does not exhibit any specific design patterns, it provides a foundation for other design patterns to be applied in a larger context. For example, the `UnoCard` class could be used in conjunction with the Factory Method pattern to create instances of `UnoCard` objects based on different color and value combinations. Additionally, the `UnoCard` class could be part of a larger system that applies patterns like Composite or Decorator to represent complex card combinations or special card effects. Overall, the `UnoCard` class serves as a basic building block for modeling the Uno game and can be used within the context of other design patterns to create more sophisticated and extensible systems.

The `UnoDeck` class does not exhibit any specific design patterns. While no explicit design patterns are present in the `UnoDeck` class itself, it can be used as part of a larger system that applies design patterns like the Singleton pattern to ensure there's only one instance of the deck, or the Iterator pattern to provide an iterator for iterating over the cards in the deck. Overall, the `UnoDeck` class serves as a fundamental component in modeling the Uno game and can be used in conjunction with other design patterns to create more advanced and flexible systems.

In the `Game` class i have used a few design patterns that helped in structuring and organizing the code, improving its maintainability, extensibility and reusability. Here are some of the design patterns i have used

- ❖ **Template Method Pattern:** The `Game` class follows the Template Method pattern. It provides a skeletal structure for the game by defining the overall flow and sequencing of the game. It contains abstract and concrete methods that subclasses (such as `Mode2`) can override to provide specific implementations.
- ❖ **Factory Method Pattern:** The `Game` class utilizes the Factory Method pattern indirectly. It provides a factory-like method called `getPlayerIds()` that creates an array of player names based on the input from the `scanner`. Although it's not a full-fledged factory class, it follows the concept of creating objects in a separate method.
- ❖ **Strategy Pattern:** The `Game` class uses the Strategy pattern for the `submitPlayerCard()` method. It declares the abstract method and allows subclasses to provide their own implementations. The modes classes override this method to implement the specific logic for each individual game mode.
- ❖ **Iterator Pattern:** The iteration over players in the `getNextPlayerIndex()` and `getNextPlayer()` methods follows the Iterator pattern. It provides a way to iterate through the list of players in both forward and backward directions, depending on the game direction.
- ❖ **Observer Pattern:** Although not explicitly shown in the given code, the game flow can involve the Observer pattern. For example, if there were a mechanism to notify players about game events or to update the UI based on game state changes, the Observer pattern could be used to establish the communication between the game and its observers.

4. Code Cleaness

The `UnoCard` class demonstrates good usage of enums, immutability, encapsulation, and adherence to naming conventions, which contribute to clean and maintainable code, here is a more detailed description

- ➔ **Enum Usage**: The class effectively utilizes enums to represent the colors and values of Uno cards. This approach ensures type safety and improves code readability by providing a predefined set of valid options.

- **Meaningful Enum Values**: The enum values for both `Color` and `Value` are named in a clear and self-explanatory manner, making the code more readable and understandable.
- **Immutable Object**: The `UnoCard` class implements immutability by making the `color` and `value` fields `final` and not providing any setters. This ensures that once a card is created, its state cannot be changed, promoting code reliability and reducing unexpected behavior.
- **Encapsulation**: The class encapsulates the card's color and value within private fields, providing access to them through public getter methods (`getColor()` and `getValue()`). This encapsulation helps maintain data integrity and facilitates future modifications to the internal representation if needed.
- **Override `toString()`**: The class overrides the `toString()` method to provide a meaningful string representation of the card, concatenating the color and value. This improves debugging and logging capabilities by providing a human-readable representation of the card's state.
- **Consistency**: The class maintains consistency in naming conventions, adhering to standard Java naming conventions (e.g., using camel case for method names, uppercase for enum values, etc.), enhancing code readability and maintainability.

The `UnoDeck` class demonstrates good encapsulation, separation of concerns, optimization, consistent naming, and proper handling of edge cases, which contribute to clean and maintainable code, here is a more detailed description

- **Separation of Concerns**: The `UnoDeck` class focuses on managing the Uno deck's functionality, such as card creation, shuffling, drawing, and replacing the deck. This separation ensures that each class has a single responsibility, improving code readability, maintainability, and testability.
- **Encapsulation**: The class encapsulates the deck of Uno cards within a private `List<UnoCard>` field, preventing direct access and manipulation from external classes. This encapsulation promotes data integrity and allows controlled access through public methods.
- **Initialization in Constructor**: The class initializes the Uno deck in the constructor. By doing so, it ensures that a new deck is created when an instance of `UnoDeck` is instantiated, promoting object consistency and preventing uninitialized state errors.
- **Loop Optimization**: The nested `for` loops used to populate the deck with cards are optimized to avoid adding wild cards (`Color.WILD` and `Color.WILD_FOUR`) to the deck when iterating over `Color` and `Value` enums. This optimization reduces unnecessary iterations and improves performance.

➔ **Consistent Naming and Formatting**: The class follows consistent naming conventions and formatting throughout the code, using meaningful variable names, proper indentation, and appropriate spacing. This enhances code readability and maintainability.

➔ **Method Clarity**: The methods `Shuffle()`, `drawCard()`, `drawCards()`, and `replaceDeckWith()` have clear and concise names, accurately describing their functionality. This improves the code's self-documenting nature and makes it easier for other developers to understand and use these methods.

➔ **Graceful Handling of Empty Deck**: The `drawCard()` method gracefully handles the case when the deck is empty by returning `null` instead of throwing an exception. This prevents potential runtime errors and allows the calling code to handle the empty deck scenario appropriately.

➔ **Reusability**: The `UnoDeck` class can be reused in different Uno game implementations, providing a consistent and reliable deck management mechanism. This reusability promotes code modularity and reduces code duplication.

The abstract `Game` class demonstrates good encapsulation, separation of concerns, input validation, clear method names, and exception handling. These practices contribute to clean and maintainable code for implementing Uno variations, here is a more detailed description

➔ **Encapsulation**: The class encapsulates the game-related data and logic within its scope. It keeps track of player IDs, the Uno deck, the pile of played cards, player hands, the current player index, game direction, and game over status. Encapsulation helps maintain data integrity and prevents direct access to internal state from external classes.

➔ **Consistent Naming and Formatting**: The class follows consistent naming conventions and formatting throughout the code. It uses meaningful variable names, proper indentation, and appropriate spacing. This enhances code readability and maintainability.

➔ **Separation of Concerns**: The `Game` class focuses on managing the game's functionality, such as initializing players, starting the game, playing turns, and determining the winner. It separates the game-specific logic from the Uno card and deck management. This separation promotes code modularity and makes the class easier to understand, maintain, and test.

- ➔ **Initialization in Constructor**: The class initializes essential game components, such as the player IDs, deck, pile, player hands, current player index, game direction, and game over status, in the constructor. This ensures that the game starts in a consistent state when an instance of `Game` is created.
- ➔ **Method Clarity**: The methods in the `Game` class have clear and concise names, accurately describing their functionality. For example, `isGameOver()`, `getCurrentPlayer()`, `getPlayerHand`, `getTopCard()`, `getNextPlayer()`, `getNextPlayerIndex()`, `isValidMove()`, `distributeCards()`, `startCard()`, `submitDraw()`, etc. This enhances the code's self-documenting nature and makes it easier for other developers to understand and use these methods.
- ➔ **Input Validation**: The class includes input validation mechanisms to ensure that user input is validated before proceeding. For example, the `getNumberOfPlayers()` method validates the input for the number of players, and the `getValidColorInput()` method ensures the user selects a valid color. This improves the robustness of the code by preventing invalid inputs from causing runtime errors or unexpected behavior.

- ➔ **Code Reusability**: The `Game` class serves as an abstract base class that can be extended to implement specific Uno game variations. By providing a generic game framework, it allows for code reusability and promotes the implementation of different game rules without modifying the core game structure.
- ➔ **Exception Handling**: The code includes exception handling to catch and handle potential exceptions, such as `NumberFormatException`. This ensures that invalid input doesn't cause crashes and provides user-friendly error messages to guide correct input.

And the mode classes demonstrates good use of inheritance, polymorphism, adherence to the SRP, clear method implementations, input validation, and code reusability. These practices contribute to clean and maintainable code for implementing different variations in Uno, here is a more detailed description about `Mode2` class

- ➔ **Method Clarity**: The overridden method `submitPlayerCard()` in the `Mode2` class is well-structured and easy to understand. It clearly handles the logic specific to the "Seven" card, including displaying hand sizes, validating player numbers, swapping hands with the chosen player, and updating the current player. The method promotes code readability and maintainability.

- ➔ **Code Reusability**: By extending the `Game` class, the `Mode2` class leverages the existing game framework and implements the specific rules of the "Seven" card variation. This design allows for code reusability, as other game modes can be implemented as separate classes, utilizing the same base game structure and modifying specific rules as needed.
- ➔ **Input Validation**: The code includes input validation for the player number when using the "Seven" card. It ensures that the player number is within the valid range and doesn't allow the current player to switch cards with themselves. This validation enhances the robustness of the code and prevents unexpected behavior.
- ➔ **Clear Output Messages**: The code provides clear and descriptive output messages to guide players throughout the game. Messages such as displaying hand sizes, informing players about skipped turns or reversed game direction, and indicating successful card switching enhance the user experience and make the game more enjoyable.
- ➔ **Main Method Separation**: The `main` method in the `Mode2` class separates the game setup and execution logic from the class implementation. This separation promotes code modularity, readability, and testability by isolating the game-specific code within the `Mode2` class and keeping the `main` method clean and concise.

5. SOLID Principles

The SOLID principles are a set of design principles that promote software design that is modular, maintainable, and flexible.

While not all SOLID principles are directly applicable to the `UnoCard` class, it adheres to the Single Responsibility Principle and Open-Closed Principle. It provides a clear and focused representation of an Uno card and allows for easy extension by modifying the enumerations

A. **Single Responsibility Principle (SRP)**: The `UnoCard` class has a single responsibility, which is to represent an individual card in the Uno game. It encapsulates the color and value of the card and provides access to these attributes. It does not have any additional responsibilities, making it compliant with the SRP.

B. **Open-Closed Principle (OCP)**: The `UnoCard` class is closed for modification but open for extension. The card colors and values are defined as enumerations, providing a fixed set of options. If new colors or values need to be added, they can be easily extended by modifying the enumerations. This adherence to the OCP allows for easy extension without modifying the existing class.

C. **Liskov Substitution Principle (LSP)**: The `UnoCard` class does not have any subclasses, so the Liskov Substitution Principle does not directly apply. However, it provides a foundation for potential future subclasses that could inherit from it while maintaining the contract of representing an Uno card.
D. **Interface Segregation Principle (ISP)**: The `UnoCard` class does not implement any interfaces, so the ISP is not applicable in this context.
E. **Dependency Inversion Principle (DIP)**: The `UnoCard` class does not have any explicit dependencies on other classes or abstractions. It only depends on the `Color` and `Value` enumerations, which are self-contained within the class. Therefore, the DIP is not explicitly applicable here.

The `UnoDeck` class adheres to the Single Responsibility Principle, Open-Closed Principle, and Dependency Inversion Principle. It focuses on managing the deck of Uno cards, provides extensibility through subclassing, and depends on abstractions. While the Liskov Substitution Principle and Interface Segregation Principle are not directly applicable in this context, the class provides a solid foundation for potential future enhancements or variations.

A. **Single Responsibility Principle (SRP)**: The `UnoDeck` class is responsible for managing the Uno cards in the deck, including shuffling, drawing cards, and replacing the deck. It encapsulates these functionalities within its methods. Thus, it adheres to the SRP by having a single responsibility related to managing the deck of Uno cards.
B. **Open-Closed Principle (OCP)**: The `UnoDeck` class is closed for modification but open for extension. It provides a concrete implementation of a deck of Uno cards and defines the initial deck composition and behavior. If there is a need to modify the deck or add new functionalities, it can be achieved by extending the class. Therefore, the OCP is maintained.
C. **Liskov Substitution Principle (LSP)**: The `UnoDeck` class does not have any subclasses, so the Liskov Substitution Principle does not directly apply. However, it provides a foundation for potential future subclasses that could inherit from it, such as customized decks or variations of the Uno game.
D. **Interface Segregation Principle (ISP)**: The `UnoDeck` class does not implement any interfaces, so the ISP is not directly applicable in this context.

E. **Dependency Inversion Principle (DIP)**: The `UnoDeck` class depends on the `UnoCard` class, but it relies on the abstraction provided by the `UnoCard` class rather than concrete implementations. It does not have explicit dependencies on other classes or abstractions. Therefore, it adheres to the DIP by depending on abstractions rather than concrete implementations.

The `Game` class adheres to the Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle (for potential subclasses), and Dependency Inversion Principle. Although the Interface Segregation Principle is not directly applicable in this context, the class provides a solid foundation for defining different variations of the Uno game by extending it.

A. **Single Responsibility Principle (SRP)**: The `Game` class is responsible for managing the gameplay logic and flow of the Uno game. It encapsulates the game-related functionalities within its methods, such as distributing cards, starting the game, validating moves, and handling player actions. Thus, it adheres to the SRP by having a single responsibility related to managing the game.
B. **Open-Closed Principle (OCP)**: The `Game` class provides an abstract framework for defining different variations of the Uno game by allowing subclasses to implement the abstract method `submitPlayerCard()`. This allows the class to be extended and modified without directly modifying the existing code. Therefore, the OCP is maintained.
C. **Liskov Substitution Principle (LSP)**: The `Game` class is designed to be extended by subclasses to define different variations of the Uno game. It does not violate the LSP as it provides a well-defined base for subclasses and does not introduce any new behaviors or restrictions that would break the expected behavior of its subclasses.
D. **Interface Segregation Principle (ISP)**: The `Game` class does not implement any interfaces, so the ISP is not directly applicable in this context.
E. **Dependency Inversion Principle (DIP)**: The `Game` class has dependencies on the `UnoDeck`, `UnoCard`, `List`, `ArrayList`, `Map`, and `Scanner` classes. However, it relies on abstractions rather than concrete implementations for most of these dependencies. For example, it uses the `List`, `ArrayList`, and `Map`interfaces, which allows for flexibility and easier substitution of different implementations. The `Scanner` class is used directly, but it is a low-level dependency and can be considered acceptable in this context. Therefore, the class adheres to the DIP by depending on abstractions and having lower-level dependencies.

The modes classes adhere to the Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, and Dependency Inversion Principle. Although the Interface Segregation Principle is not directly applicable in this context, the class provides a specific implementation for Mode 1 and Mode 2 of the Uno game while leveraging the existing framework of the base class.

A. **Single Responsibility Principle (SRP)**: The modes classes are responsible for defining and implementing the gameplay logic specific to the mode of the Uno game. It extends the `Game` class and overrides the `submitPlayerCard()` method to provide the specific behavior for the modes. The class encapsulates the mode-specific rules and interactions within its methods. Thus, it adheres to the SRP by having a single responsibility related to managing the modes gameplay.
B. **Open-Closed Principle (OCP)**: The modes classes extend the `Game` class, utilizing its abstract methods and providing an implementation specific to each mode. It does not modify the existing code of the `Game` class but builds upon it to define a mode of gameplay. Therefore, the class adheres to the OCP by being open for extension and closed for modification.
C. **Liskov Substitution Principle (LSP)**: The modes classes extend the `Game` class and do not violate the LSP. It maintains the expected behavior of the base class and provides additional functionality specific to each mode without altering the behavior of the base class.
D. **Interface Segregation Principle (ISP)**: The modes classes do not implement any additional interfaces apart from the ones inherited from the `Game` class. Therefore, the ISP is not directly applicable in this context.
E. **Dependency Inversion Principle (DIP)**: The modes classes rely on the dependencies inherited from the `Game` class, such as `UnoDeck`, `UnoCard`, `List`, and `Scanner`. It follows the same principles of dependency inversion as the base class, depending on abstractions rather than concrete implementations for most dependencies. The `Scanner` dependency is a lower-level dependency and can be considered acceptable in this context. Therefore, the class adheres to the DIP.