UNO Game Engine

1. Game package:

Includes the Game class, which is the main class of my design. This class has two objects that are used to handle the game, Deck and Player. It is responsible for starting the game and initializing the main objects to start the game, also it is responsible for ending the game.

The constructor of Game class needs two attributes, list of players who will play the game, and the number of cards that the player will have in the beginning of the game, the developer who will extends this class will be responsible for passing these parameters to the constructor.

The developer who will extend this class will be responsible for override 3 methods:

- The first method is play method, this method is the only method that will invoked in the main, and this method is the method that will start the game. The developer can implement his specific role here. like how many cards can player draw from pile stock.

- The second method is getTurnInformation, this method is responsible for showing the information to the user.

- Third method is handleBehavior, this method will handle the behavior of each card and the special card if added.

And there is a different method that can be very useful here to make it easier for the developer to make a game.

2. Player package:

Includes the Player class, which is responsible for storing the name and the card of the player and some getter and setter method.

3. Deck package:

Includes the Deck class and DeckFactory class.
The DeckFactory class is responsible for creating the deck and initializing every card with its color, value, and behavior.

The Deck class which contains a DeckFactory object, has an object from Card class and has the main two properties pileStock and discardCard, which handles the cards in the game and refresh the pileStock when it is empty. Also, it is responsible for adding a special card if the developer ask for.

The constructor of the Deck class will initialize the pileStock and discardCard and fill the pileStock with 108 cards and then shuffle the card and remove a card from pileStock and add it to the discardCard (the top card in the deck).

In the Deck class there is a method called drawCardFromPileStock, this method is responsible for give a player a card from deck if he asks to, and if the pileStock is empty it will call reserPileStock method that will refresh the pileStock from the discardCard.

4. Card package:

Includes Card class, CardValue enum and CardColor enum.
The Card class has a reference from CardBehaviour interface. The main purpose of Card class is to store the color, value, and behavior of each card.

CardColor stores the colors of the card, also we have a special color to handle the case when developer add a special color.

The constructor of this class will take three parameters: the color of the card, the value of the card and the behavior of the card.

5. Behavior package:

Includes a CardBehavior interface and six classes that implement this interface DrawTwoBehavior, ReverseBehavior, SkipBehavior, NumberedBehaviour, WildBehaviour and WildDrawFourBehaviour.

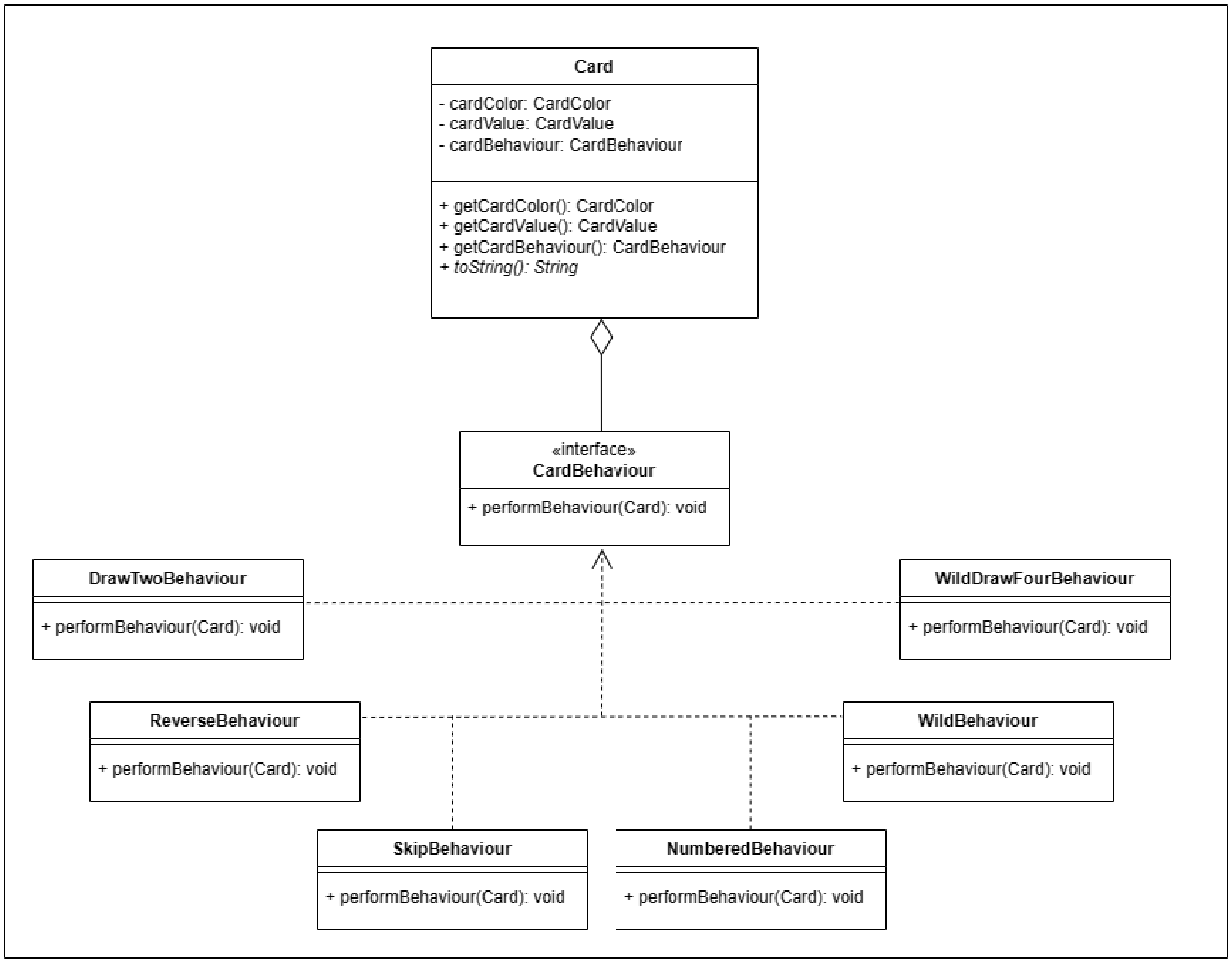
Every class in this package handle a specific behavior, and if the developer wants to add a new behavior all he need is just to add a new class that implements the CardBehavior interface and override the perform method and do his implementation.

The Design Patterns used in this assignment:

1. Strategy Pattern:

"The Strategy Pattern defines a family of algorithms, encapsulates each one and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it."

I use this pattern to encapsulate the behavior of cards, this will allow the developer to add new behaviors without modifying existing code. This provides a structured and maintainable way to handle variations in behavior. It promotes code reuse, flexibility, and extensibility.

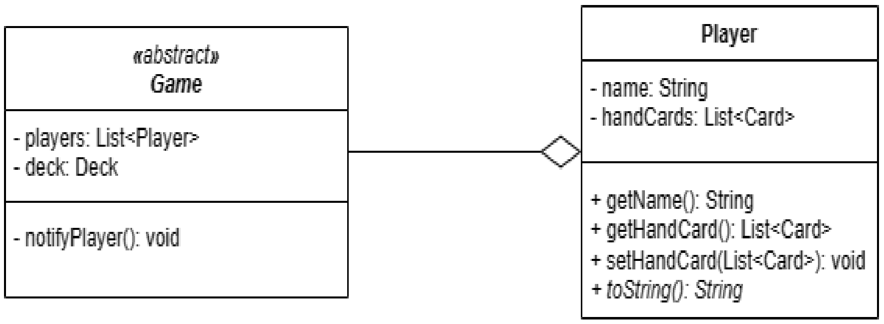

Card
- cardColor: CardColor
- cardValue: CardValue
- cardBehaviour: CardBehaviour
+ getCardColor(): CardColor
+ getCardValue(): CardValue
+ getCardBehaviour(): CardBehaviour
+ toString(): String
«interface»
CardBehaviour
+ performBehaviour(Card): void
DrawTwoBehaviour
+ performBehaviour(Card): void
WildDrawFourBehaviour
+ performBehaviour(Card): void
ReverseBehaviour
+ performBehaviour(Card): void
WildBehaviour
+ performBehaviour(Card): void
SkipBehaviour
+ performBehaviour(Card): void
NumberedBehaviour
+ performBehaviour(Card): void

2. Observer Pattern:

“The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically”.

The Main benefit of using this pattern is:
Notify players of changes, players receive updates about changes in the game state, like the played card, draw pile status, or current turn, without the need for constant polling or explicit checks. This keeps the game flow smooth, and everyone informed.

When the top card in the deck is changed, all players in the game will notify with this change and get a message that contains the new top card in the deck. There is a method in the Game class called notify, this method will iterate in each player and send a message to the player containing the change in the top card.

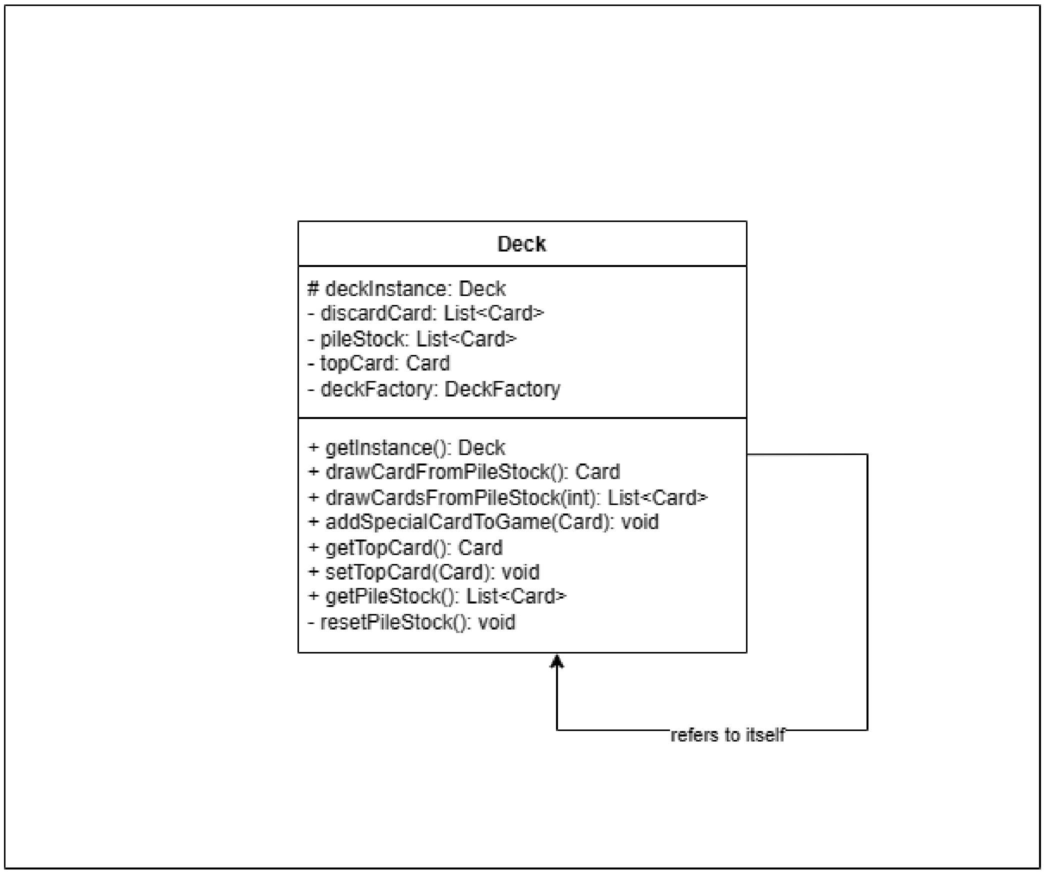


3. Singleton Pattern:

"The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it".

In Uno, there should always be exactly one deck of cards in play. The singleton pattern ensures that only one instance of the Deck class is ever created, preventing accidental creation of multiple decks, which would disrupt gameplay.

Easy Access from Anywhere: Any part of the game code can easily access the same Deck instance through a single, well-known reference point. This simplifies interactions with the deck and eliminates the need to pass deck references around different game components.


Deck
# deckInstance: Deck
- discardCard: List<Card>
- pileStock: List<Card>
- topCard: Card
- deckFactory: DeckFactory
+ getInstance(): Deck
+ drawCardFromPileStock(): Card
+ drawCardsFromPileStock(int): List<Card>
+ addSpecialCardToGame(Card): void
+ getTopCard(): Card
+ setTopCard(Card): void
+ getPileStock(): List<Card>
- resetPileStock(): void
refers to itself

Explain A Demo of Game:

A developer will create a new package called developer, that contain a class called MyGame that extends Game class and override the main three methods to implement the game with its roles.

The constructor of MyGame class don't need to do anything, just call the super constructor, and pass to it the players and the number of cards in hands of each player in the beginning of the game, and all work will done in the constructor of the Game class.

In the handleBehaviour method, the developer can use the behavior of each card by just calling the performBehaviour method. The developer can add its own roles here by using the defined behavior and add its behavior.

If the developer wants to add new behavior, all he needs to do is to create a new class that extends the CardBehavior interface and override the perform method and put the implementation inside the perform method.

As we see in the demo, the developer adds the dance behavior and adds a new method in the MyGame class (addDanceCard) that adds the special card to deck and shuffle the deck.

And that is, now we have a new Card in our Uno Game.

SOLID Principles:

1. Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should have only one responsibility or job.
   As we see in the design every class has a single responsibility. (Card, Player, Deck…).

2. Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
   The CardBehavior interface is open for extension (new behavior can be added) but closed for modification (existing code does not need to be changed).

3. Liskov Substitution Principle (LSP): Objects of a superclass should be able to be replaced with objects of a subclass without affecting the correctness of the program.
   When we call the perform method of specific behavior, we can use the super type (CardBehavior) instead of using its children (DrawTwoCard).

4. Interface Segregation Principle (ISP): A class should not be forced to implement interfaces it does not use, and clients should not be forced to depend on interfaces they do not use.
   As we see in the CardBehavior interface and all its children, every class implements this interface will use its method.

5. Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.
   As we see in the Card class, Card class has instance of CardBehavior interface which can be WildBehavior, SkipBehavior etc...

KISS (Keep It Simple, Stupid):

The KISS principle advocates simplicity in design and implementation. Simple and straightforward solutions are usually easier to understand, maintain, and troubleshoot.

As we develop a Game Engine, we will make it simpler so the other developers can easily extend the defined code and add their specific roles.

DRY (Don't Repeat Yourself):

This principle emphasizes that duplication in code (such as repeating the same logic) should be avoided. Instead, a single, authoritative source should exist for a particular piece of knowledge.

In the Uno Game Engine there is no code duplication, as every steps that used more than one time is put in method and called by method name.

YAGNI (You Ain't Gonna Need It):

This principle suggests that you should not add functionality until it is necessary. Avoid speculative or premature features that are not currently required.

In this code all functionalities written is used by the developer.

The Boy Scout Rule:

Leave the code cleaner than you found it. Whenever you make changes to the code, take the opportunity to improve it by refactoring or cleaning up any mess you encounter.

As I am making this engine, I follow up this role, and do it more than once.

Composition Over Inheritance:

Favoring composition (building complex objects by combining simpler ones) over inheritance can lead to more flexible and maintainable code.

I gave priority to composition over inheritance in my design, as this will make it simpler.

Tell, Don't Ask:

Encourage the use of telling objects what to do instead of asking them about their state and making decisions based on that information. This can lead to more encapsulated and modular code.

As we see when we call perform method, we call this method without asking the object about its type.