Report

Assignment (UNO Game)

Name: Motaz Bataineh

Table of Contents:

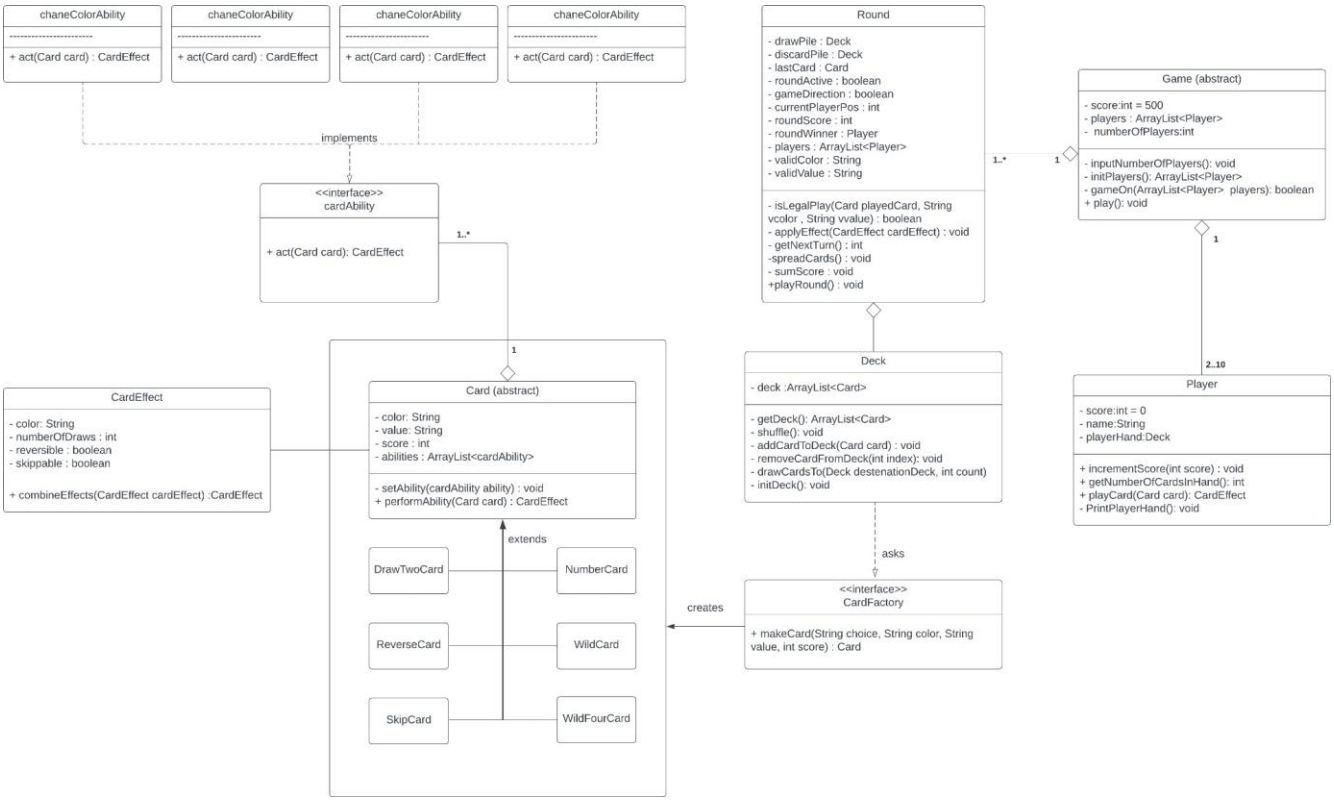
- # Introduction

  In this assignment we were required to build an Uno game engine using Java and OOP to build an Uno game engine to be used by other developers in which they can build their own variation of an Uno game.

- # Explanation before start

  The development of this engine required a deep understanding of OOP concepts, including abstraction, encapsulation, inheritance, and polymorphism. Additionally, the implementation leveraged well-known design patterns to ensure that the engine is both robust and easy to maintain. Throughout the project, emphasis was placed on adhering to best practices such as clean code principles, effective Java techniques, and the SOLID design principles.

My Approach:


chaneColorAbility
+ act(Card card) : CardEffect
chaneColorAbility
+ act(Card card) : CardEffect
chaneColorAbility
+ act(Card card) : CardEffect
chaneColorAbility
+ act(Card card) : CardEffect
implements
<<interface>>
cardAbility
+ act(Card card): CardEffect
1..*
1
Round
- drawPile : Deck
- discardPile : Deck
- lastCard : Card
- roundActive : boolean
- gameDirection : boolean
- currentPlayerPos : int
- roundScore : int
- roundWinner : Player
- players : ArrayList<Player>
- validColor : String
- validValue : String
- isLegalPlay(Card playedCard, String vcolor , String vvalue) : boolean
- applyEffect(CardEffect cardEffect) : void
- getNextTurn() : int
-spreadCards() : void
- sumScore : void
+playRound() : void
1..*
1
Game (abstract)
- score:int = 500
- players : ArrayList<Player>
- numberOfPlayers:int
- inputNumberOfPlayers(): void
- initPlayers(): ArrayList<Player>
- gameOn(ArrayList<Player> players): boolean
+ play(): void
1
2..10
Player
- score:int = 0
- name:String
- playerHand:Deck
+ incrementScore(int score) : void
+ getNumberOfCardsInHand(): int
+ playCard(Card card): CardEffect
- PrintPlayerHand(): void
CardEffect
- color: String
- numberOfDraws : int
- reversible : boolean
- skippable : boolean
+ combineEffects(CardEffect cardEffect) :CardEffect
1
Card (abstract)
- color: String
- value: String
- score : int
- abilities : ArrayList<cardAbility>
- setAbility(cardAbility ability) : void
+ performAbility(Card card) : CardEffect
extends
DrawTwoCard
NumberCard
ReverseCard
WildCard
SkipCard
WildFourCard
Deck
- deck :ArrayList<Card>
- getDeck(): ArrayList<Card>
- shuffle(): void
- addCardToDeck(Card card) : void
- removeCardFromDeck(int index): void
- drawCardsTo(Deck destenationDeck, int count)
- initDeck(): void
asks
creates
<<interface>>
CardFactory
+ makeCard(String choice, String color, String value, int score) : Card

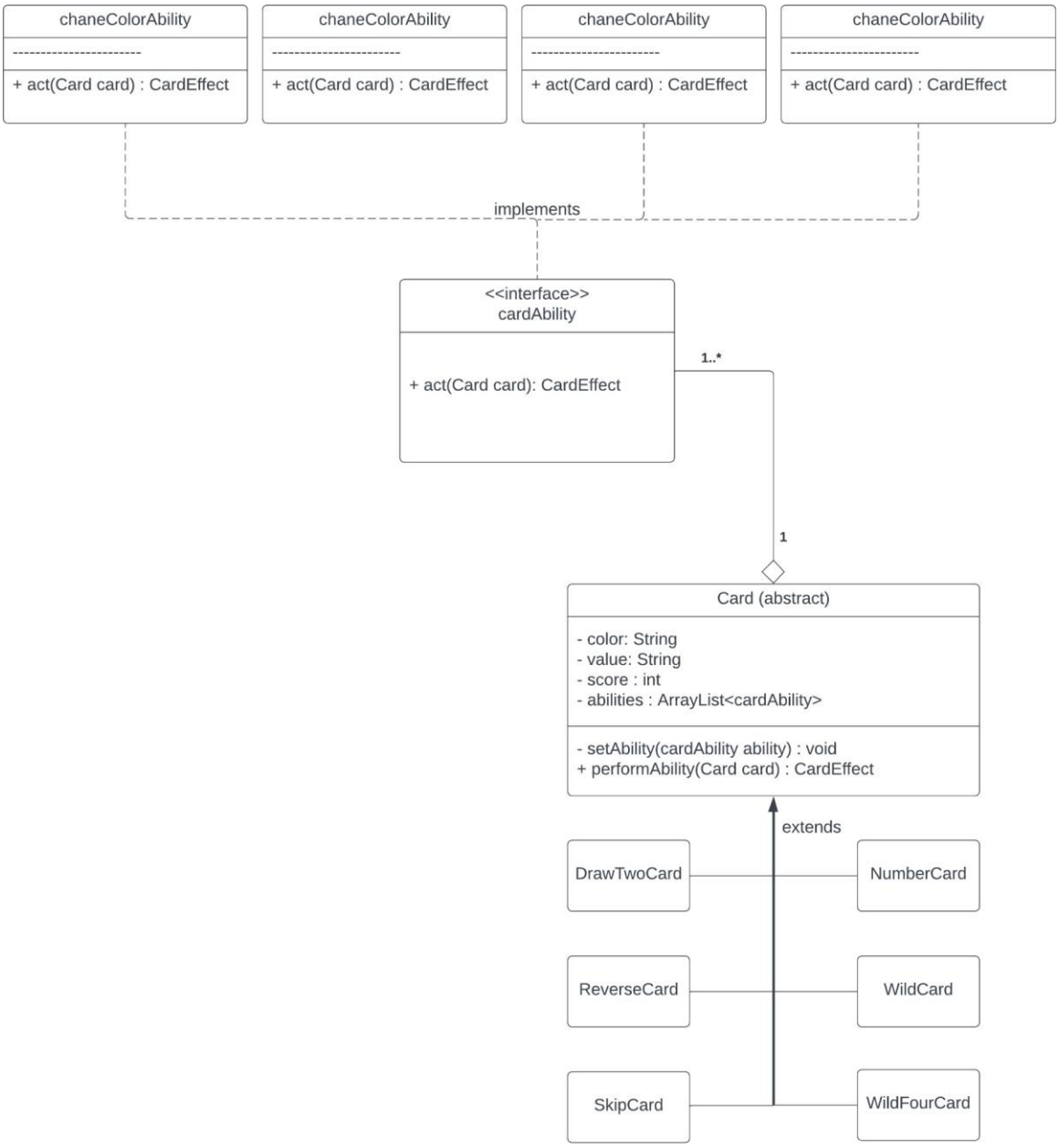
The Cards:

The cards serve as the foundation of the Uno game, representing the core gameplay mechanics. In my design, I encapsulated each card's functionality within an abstract class named "Card." This abstraction allows for a unified interface while accommodating the diverse abilities that different cards possess, such as altering the flow of the game or impacting specific players.

Each card in the game is designed to have a unique effect, whether it influences the entire game state or targets the subsequent player. Despite their distinct abilities, the cards share a set of common characteristics, such as color, number, or special action. To manage these shared traits while keeping the individual behaviors flexible, I implemented the Strategy Design Pattern.

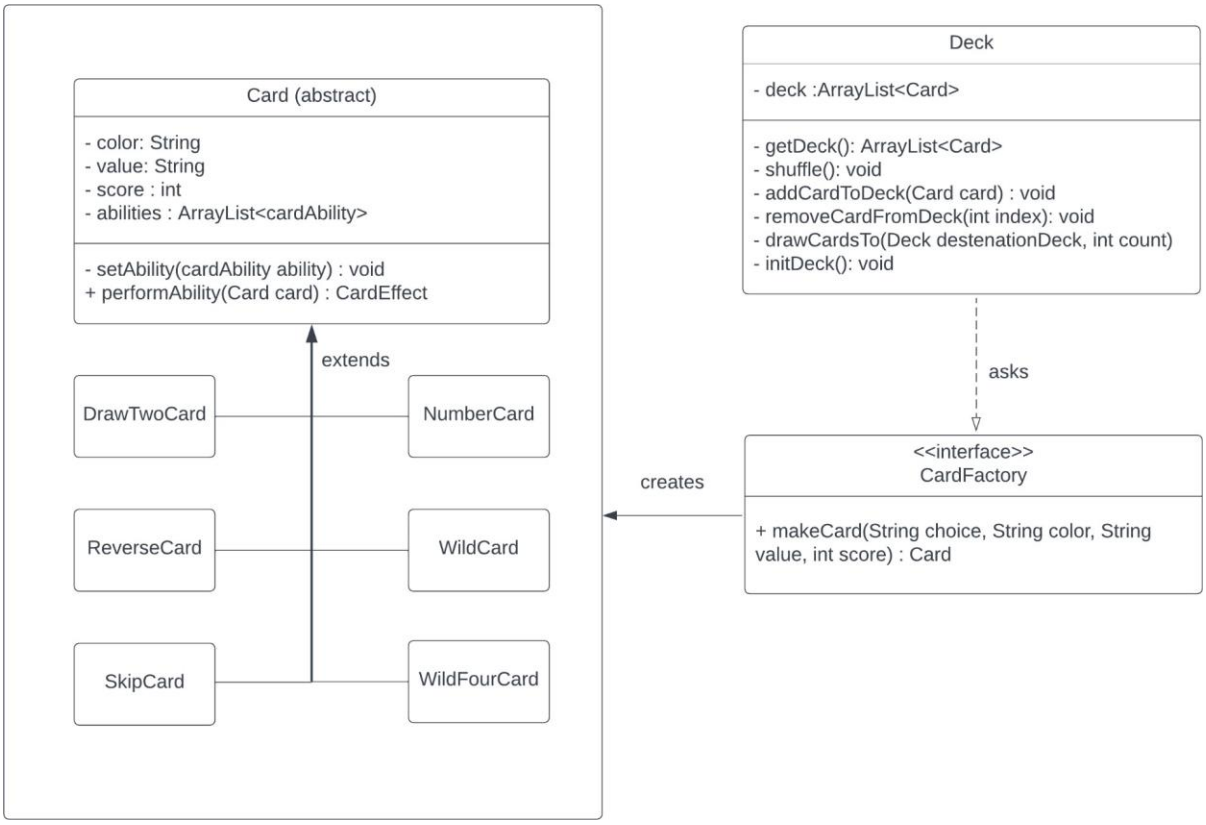
The Strategy Design Pattern was particularly effective in this context because it enables the dynamic assignment of behaviors to each card. By decoupling the card abilities from the card objects themselves, the pattern allows for easier modifications and extensions in the future. For instance, adding new types of cards or altering existing behaviors can be achieved without altering the core card structure, ensuring that the code remains clean, maintainable, and adaptable to new game variations.

Cards Strategy:



Having high Encapsulation where the logic and details are hidden from the client and given low profile methods to use, leading to minimize the accessibility like what effective java recommends.

Cards Factory:



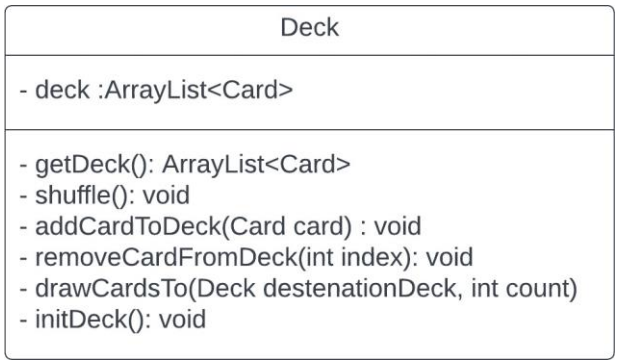
As discussed earlier, each card in the game is defined by its unique specifications, which may include attributes like color, number, and special effects. To streamline the process of creating and integrating new cards into the game, I employed the Factory Design Pattern, a type of Creational Design Pattern.

The Factory Pattern is particularly useful in this context because it abstracts the instantiation process, allowing for the creation of card objects without exposing the exact class being instantiated. This design simplifies the addition of new card types by centralizing the creation

logic in one place. As a result, introducing new cards or modifying existing ones becomes a straightforward task, requiring minimal changes to the overall codebase.

By leveraging the Factory Pattern, the game engine gains significant flexibility, enabling developers to easily expand the game with new cards that can have unique effects or interactions. This approach not only reduces redundancy but also ensures consistency across the game, as all card objects are created through a standardized process.

The Deck:

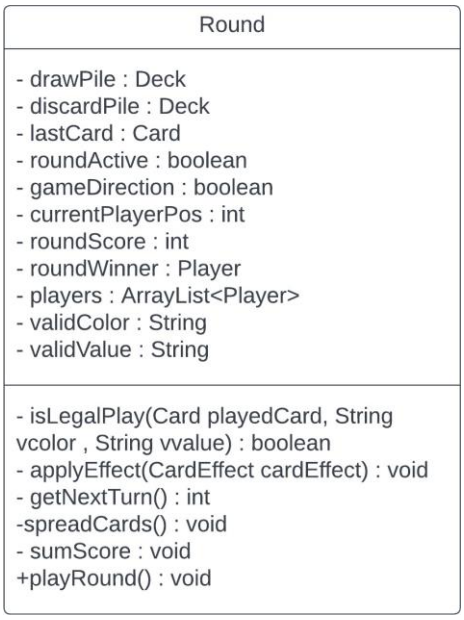


The Deck is basically a collection of cards, where we can shuffle cards, take cards from and/or add cards to, what that exactly means is that the draw pile, the discard pile and the player hand are all considered as decks.

The design ensures that the system has a high Abstraction by

breaking the design down into simplified classes with easy and straightforward usage, using Inheritance and Polymorphism to create and consume various types of cards.

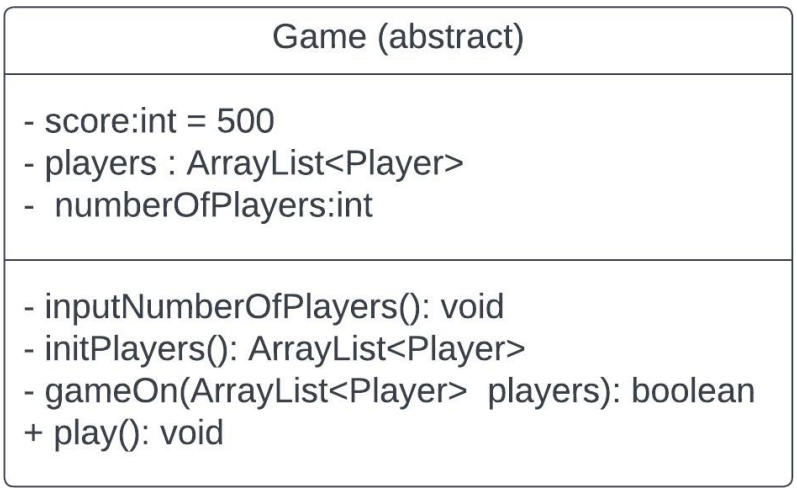
The Round:



At the start of each round we initiate the deck and spread the cards , and at the end of the round we sum the score and add it to the winner’s score.

The round is where the game is really played, and invoked only by using the (playRound) method.

The Game:



It’s an abstract class where In order to create a new

game variation, the developer will only need to create a new class

that extends the Game class, in order to create an Uno Game.

Conclusion:

The design of the Uno game engine meticulously adheres to the fundamental principles of Object-Oriented Programming (OOP), ensuring that every component is highly abstracted and encapsulated. Classes and methods are crafted with a focus on abstraction, allowing for clear and manageable code structures. The design leverages generalization and polymorphism extensively, enabling seamless handling of various card types and their unique abilities.

To achieve a modular and extendable codebase, appropriate design patterns were followed throughout the development process. These patterns facilitate future enhancements, making it easier to add new features or modify existing ones without disrupting the overall architecture.

A key consideration during development was maintaining clean code, guided by the principles outlined in Uncle Bob's Clean Code philosophy. I made a concerted effort to keep the code DRY (Don't Repeat Yourself), addressing potential issues like data clumps and the presence of long parameter lists, particularly when implementing the Strategy Design Pattern. Ensuring high cohesion within classes and loose coupling between them was a priority, leading to a design where each class and method serves a well-defined purpose with minimal interdependencies. Additionally, I focused on reducing the number of method arguments wherever possible to enhance readability and maintainability.

The principles from "Effective Java" also played a significant role in shaping the design decisions. Key practices included minimizing the accessibility of classes and members to protect the integrity of the code and using accessor methods to control access to internal state. Furthermore, I aimed to reduce mutability across most classes, enhancing the robustness and reliability of the engine.