DeckDevs: Derivable 2

Divyadeep Maan

Anshul Alpesh Patel

Sane Sunny

Mohemmed Faaiz Shaikh

Sheridan College

Sivagama Srinivasan

SYST17796: Fundamentals of Software Design and Development

July 26th, 2023

Table of Content

Use Case Diagram

Main Path:

1) The program starts by asking the user to input the number of players for the UNO game (between 2 and 10 players).
2) After receiving the valid number of players, the program proceeds to create the specified number of UNOPlayer objects and adds them to the list of players.
3) The UNOGame object is created with the name “UNO Game” and the list of players passed to it.
4) The deck of cards is then shuffled.
5) Initial cards are dealt to each player in the game. Each player receives 7 cards from the deck.
6) The game loop starts, and the players take turns playing the UNO game.
7) During each player’s turn, the top card on the discard pile is displayed, and the player’s hand is shown.
8) The player is prompted to select a card index from their hand to play.
9) If the selected card is valid (matching color, number, or type of card), it is added to the discard pile, and any special effects are handled (e.g., Skip, Reverse, Draw Two, Wild, Wild Draw Four).
10) If the player has only one card left, they are prompted to call UNO. If they fail to do so and are caught, they draw two penalty cards.
11) The game continues in this loop until one player has no cards left.
12) The winner is declared, and the game ends.

Alternate Path:

If the user enters an invalid number of players (less than 2 or more than 10), the program displays an error message indicating “Invalid number of players” and terminates.

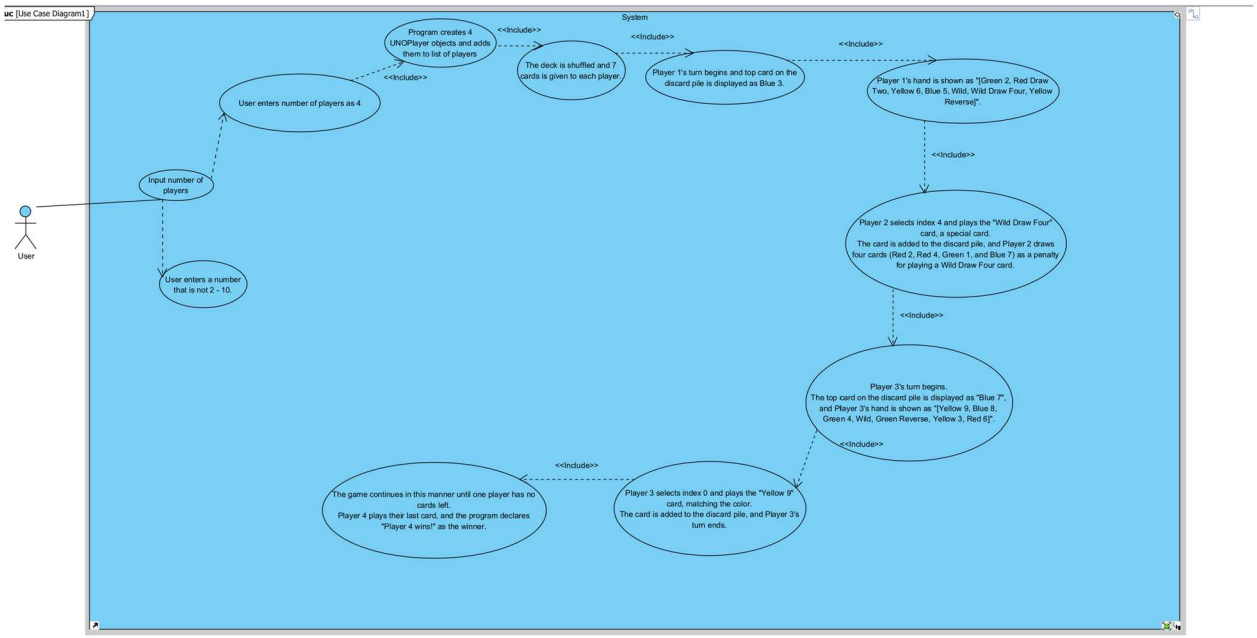

Fig.1 : Use Case Diagram

Class Diagram

The rules for making the class diagram are as per the following:

UNO Game Rules:

1) The game is played with a standard deck of 108 UNO cards, consisting of four colors: red, yellow, green, and blue.
2) Each color has cards numbered from 0 to 9, along with "Skip," "Reverse," and "Draw Two" cards.
3) Additionally, there are special "Wild" and "Wild Draw Four" cards, which can be played on any color and allow the player to change the color being played.
4) The objective is to be the first player to get rid of all your cards.
5) Players take turns to match the top card on the discard pile by either color, number, or type of card. If a player cannot play a card, they must draw from the draw pile until they get a playable card.
6) The "Skip" card skips the next player's turn, the "Reverse" card changes the direction of play, and the "Draw Two" card forces the next player to draw two cards.
7) When a player has only one card left, they must announce "UNO." Failure to do so and getting caught will result in drawing two penalty cards.
8) The game continues until one player has no cards left, and they are declared the winner.

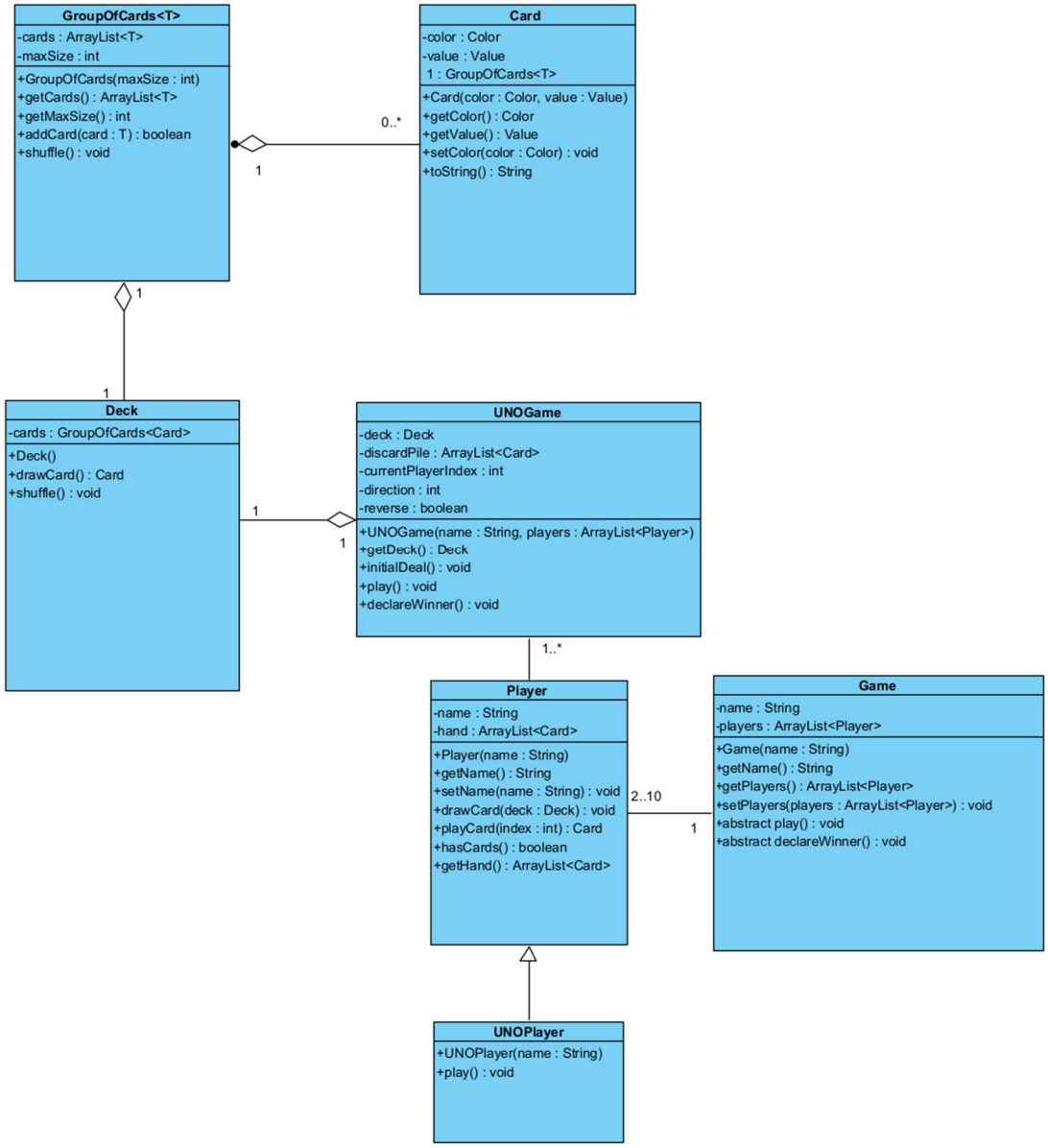

Fig. 2: Class Diagram

The explanation of the above class diagram is given as:

1) **GroupOfCards<T> and Card:**

Relationship: Association (Aggregation)

Explanation: GroupOfCards<T> contains instances of Card, representing a collection of cards. Since the cards can exist independently of the GroupOfCards<T>, it is an aggregation.

2) **Deck and GroupOfCards<Card>:**

Relationship: Association (Aggregation)

Explanation: Deck contains instances of GroupOfCards<Card>, representing the deck's collection of cards. The cards in the deck can exist independently of the deck itself, making it an aggregation.

3) **UNOGame and Deck:**

Relationship: Association (Aggregation)

Explanation: UNOGame uses a specific Deck for playing the game. The Deck can exist independently of the UNOGame, making it an aggregation.

4) **UNOGame and Player:**

Relationship: Association

Explanation: UNOGame has a list of Player objects representing the players participating in the game. There is no clear indication of aggregation or composition since the players can exist independently of the UNOGame, but they are an essential part of the game.

5) **Player and UNOPlayer:**

Relationship: Generalization (Inheritance)

Explanation: UNOPlayer is a specific type of Player representing a player in the UNO game. UNOPlayer inherits from the general Player class, which allows it to inherit common attributes and methods.

6) **UNOGame and Card:**

Relationship: Association

Explanation: UNOGame uses instances of Card when playing the game. There is a relationship between the game and the cards being played.

Project Background and Description

The chosen game for this project is the classic UNO card game. The goal of the project is to create a user-friendly and functional application that allows players to play UNO according to the official rules. Players will take turns matching cards from their hands with the top card on the discard pile until one player has no cards left, at which point the game will be won.

The technical scope of the project includes several key features:

1) **Player Registration:** The application will have a player registration feature that allows each player to sign up for the game. Players will enter their names, and the system will create player instances for them to participate in the game.
2) **Game Logic Implementation:** The core game logic will be implemented, allowing players to match cards, draw cards, and follow the UNO rules throughout the game. This will involve handling the interactions between players and the deck of cards, determining the playability of cards, and enforcing UNO call rules.
3) **User-Friendly Interface:** The application will have a user-friendly interface that displays relevant information such as player scores, the cards currently in play, and the progress of the game. The interface will allow players to interact with the game easily and understand the current state of the game.
4) **Game Outcome Alerts:** The application will provide alerts when the game is won or lost. When a player wins, the system will declare the winner, and when the game is over, the system will notify all players of the outcome.
5) **Compliance with UNO Rules:** The project will ensure that the game follows the official UNO rules to provide an authentic gaming experience for the participants. This includes handling special cards (Skip, Reverse, Draw Two, Wild, Wild Draw Four), managing the discard pile, and handling UNO calls.

Design Considerations

Enumeration:

1) **Color Enum:** The Color enum represents the color of the UNO cards, including RED, YELLOW, GREEN, BLUE, and WILD. By using an enum, the code ensures that only valid colors can be used, preventing any accidental misuse or incorrect values. The Color enum is used in the Card class to specify the color of a card, and in the UNOGame class, it allows the player to choose a new color when a Wild card is played.
2) **Value Enum:** The Value enum represents the value or type of the UNO cards, including numeric values (ZERO to NINE), action cards (SKIP, REVERSE, DRAW_TWO), and special cards (WILD, WILD_DRAW_FOUR). Using an enum ensures that only valid card values can be used, avoiding any mistakes in assigning card values. The Value enum is used in the Card class to specify the value of a card and in the UNOGame class to handle the effects of special cards when played.

Encapsulation: Encapsulation refers to the bundling of data and methods within a class, hiding the internal implementation details and exposing only necessary interfaces to the outside world. In this design, encapsulation is well-implemented in all classes. Private fields are used to store data, and access to these fields is controlled through getter and setter methods. For example, in the Player class, the hand field is private, and its data is accessed through methods like drawCard(), playCard(), and getHand().

Delegation: Delegation involves one object passing a task to another object to perform it on its behalf. In this design, delegation is seen in the UNOGame class, where it delegates tasks to other classes. For example, the play() method of UNOGame delegates tasks to helper methods such as isPlayable(), handleSpecialCards(), handleUNOCall(), updateCurrentPlayerIndex(), and updateDirection().

Cohesion: Cohesion refers to how closely the members within a class are related to each other and how well a class focuses on a single responsibility. In this design, each class exhibits high cohesion as they are designed to have specific and well-defined responsibilities. For example, the GroupOfCards class is responsible for managing a group of cards, the Deck class handles operations related to the deck of cards, the Card class models the properties and behavior of a single card, and so on.

Coupling: Coupling represents the level of dependency between classes. Low coupling is desirable as it promotes modularity and reduces the impact of changes in one class on other classes. This design shows relatively low coupling, as classes interact with each other through well-defined interfaces. For example, UNOGame interacts with the Deck, Player, and Card classes through their public methods and not their internal implementations.

Inheritance: Inheritance is a mechanism where one class inherits properties and behaviors from another class. In this design, we have used inheritance in the form of extending the Game class for the UNOGame class and the Player class for the UNOPlayer class. This allows us to reuse the common functionality and structure defined in the parent classes.

Aggregation: Aggregation represents a "has-a" relationship between classes, where one class contains instances of another class. In this design, we can observe aggregation in the UNOGame class, where it contains instances of Deck and ArrayList<Player>, creating a relationship between them. Similarly, in the Deck class, we see aggregation with the GroupOfCards<Card> instance.

Composition: Composition represents a more tightly coupled form of aggregation, where the contained class cannot exist independently without the container class. In this design, we don't see explicit composition relationships, as the contained classes (Deck, GroupOfCards, Card, Player) can exist independently without their parent classes (UNOGame, UNOPlayer).

Flexibility/Maintainability: The design exhibits good flexibility and maintainability. The use of abstraction through abstract classes (Game, Player) allows for future extensibility and customization

without modifying the core functionality. The code uses well-defined methods and access modifiers to control the behavior and visibility of class members, making it easier to maintain and modify the code.