Group Project: Uno - Deliverable 2

Lian Asher Caraang, Mohamed Hamed, Cuong Luong, & Ryleigh Smith

Sheridan College

SYST 17796: Fundamentals of Software Design & Development

Section: 1255-1404

Dr. Syed Raza Bashir

July 28th, 2025

Table of Contents

Use Case Diagram

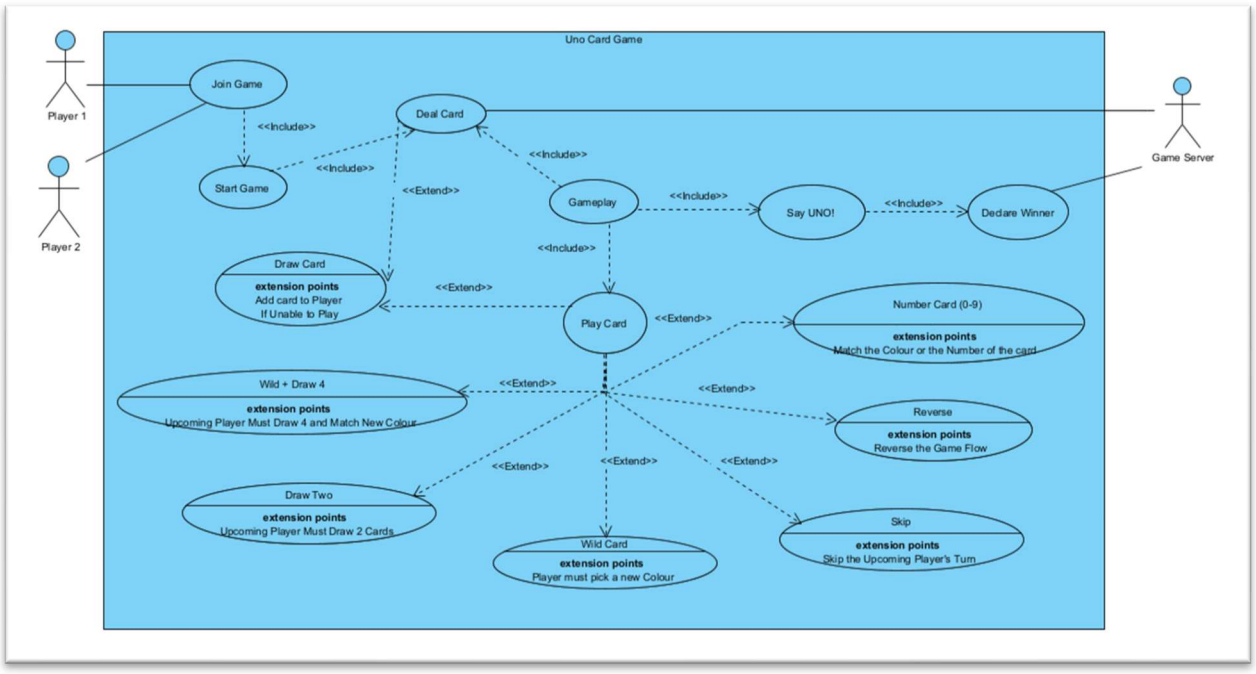

Figure 1: Use Case Diagram

Use Case Narratives

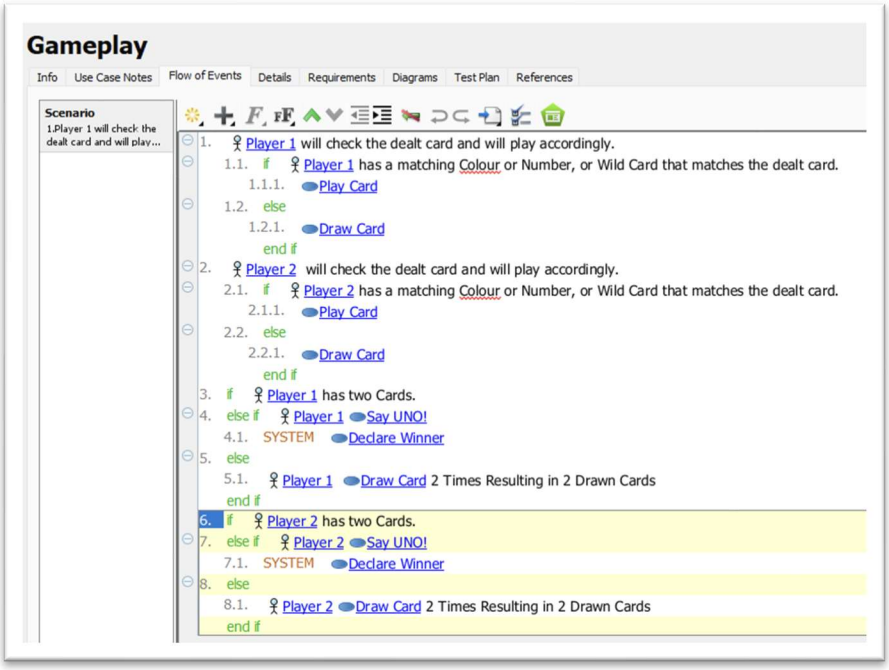

Figure 2: Gameplay Use Case Narrative

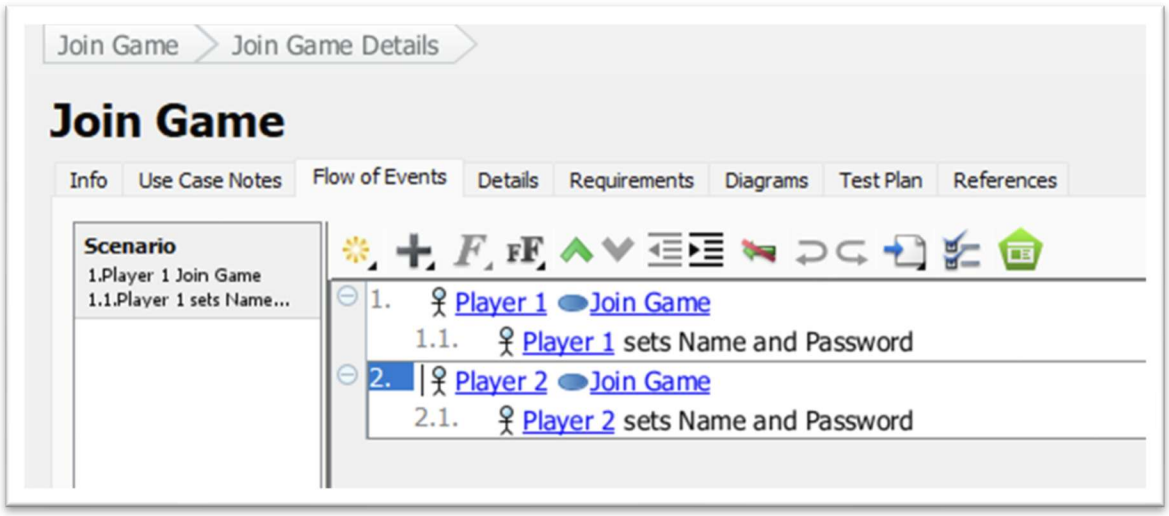

Figure 3: Join Game Use Case Narrative

Figure 4: Start Game Use Case Narrative

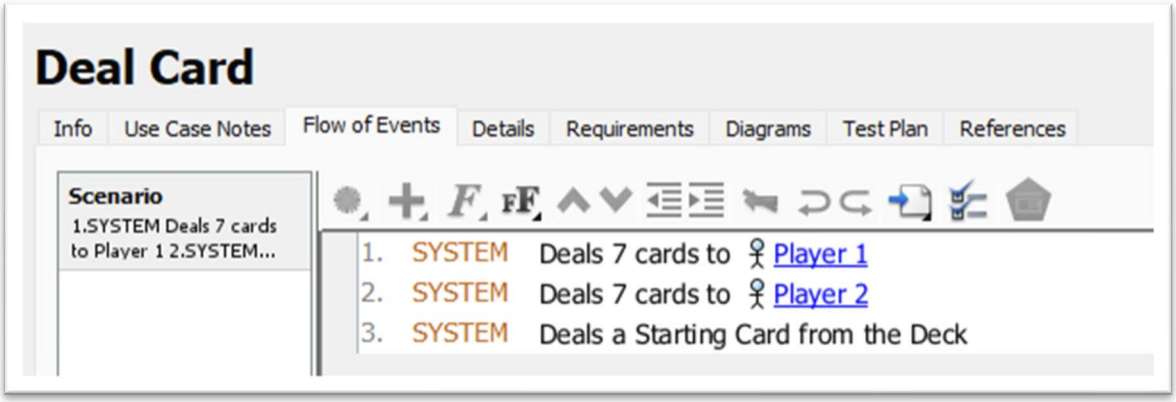

Figure 5: Deal Card Use Case Narrative

Class Diagram

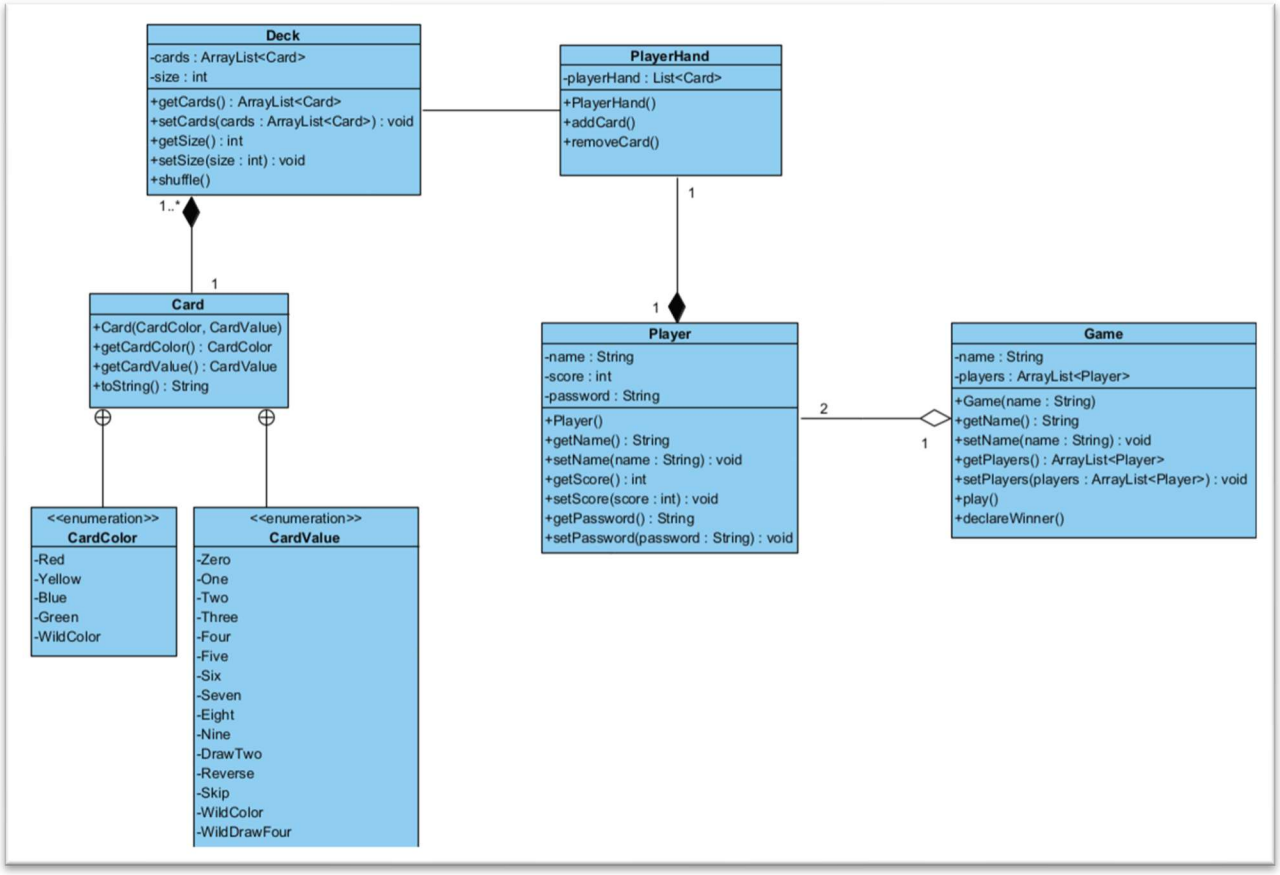

Figure 6: Class Diagram

Design Document Template

Project Background & Description

Game Chosen: Uno

How to Play: 2 players aim to be the first person to clear their hand (get rid of all their cards). Players take turns adding / removing cards from their hand based on the card played before them.

Number of Players: 2 players.

Score: Calculated by how many cards a player has in their hand. Each player has a starting score of 7 indicating the number of cards left in their hand.

Game Length: Game will terminate after one of the players score hits zero.

Gameplay: The game will have up to 2 players registered and playing at the same time. They will be able to play a full game of UNO with the players' score starting at 7 and lowering based on their hand. The game will then display a congratulations message for the winner and stop all operations.

Gameplay Interface: Text Based, players must type which card would they like to place from their hand. Error message will display if the card entered doesn't exist or in cases of invalid input.

Card Details:

- 108 cards total, 25 of each colour; red, yellow, green, blue & 8 wildcards.
- Each colour consists of; 1 zero card, 2 cards of each number (1-9), 2 skip cards, 2 reverse cards, & 2 draw cards.
- As for wild cards, there are 4 colour change cards, and 4 draw four colour change cards.
    - **Draw Two Card:** When a draw two card is placed, the next player draws 2 cards.
    - **Reverse Card:** Reverses the direction of the flow of the players placing cards. Since there are only two players, this acts as a "skip" card.
    - **Skip Card:** Skips the turn of the next player.
    - **Wild Card (Colour Change):** Changes the colour of the deck.
    - **Wild Card (Draw 4 + Colour Change):** Changes the colour of the deck and forces the next player to draw 4 cards.

Game Rules Selected:

- **No Available Card:** Player must draw one card if they cannot match the number or the colour of the placed card.
- **No Stacking:** When a draw card is placed, opponent cannot place any other draw card and must draw the number of cards required.
- **Uno:** If a player does not announce that they have uno, they will automatically draw two cards.

Design Considerations

We have updated our class diagram (*figure 6*) to reflect our final code layout. The following is an explanation of all relationships and multiplicities as shown in the diagram.

Game & Player:

- **Multiplicity:** Within our game system, a single game can be played by no players or 2 players. A single player can play one and only one game at a time.
- **Relationship:** The relationship between game and player is aggregation. Aggregation is referred to as a "HAS-A" relationship, implying that the child class exists independently of the parent, and does not get destroyed when the parent is destroyed. In the context of our game, this means that our game **HAS** multiple players, but when the game ends, the players still exist.

PlayerHand & Player:

- **Multiplicity:** Within our game system, a single player can have one and only one "hand" of cards, and each card hand can belong to one and only one player.
- **Relationship:** The relationship between PlayerHand and Player is composition. The hand of cards cannot exist independently of the player, as the hand of cards is generated for each player and does not exist without the player.

Deck & Card:

- **Relationship:** The relationship between Deck and Card is composition. A group of cards cannot exist without the cards existing. If cards do not exist, you cannot have a group of them.

Coding Concepts

Encapsulation

Encapsulation is the concept of "encapsulating" data and related methods within a class. This provides controlled access rather than direct access to interact with class variables. In our game design, this is shown in multiple places, for example, our Player class. In the Player class, name, score, and password are not directly accessible, rather accessible and changeable through controlled methods.

Delegation

Delegation is when one class relies on another to perform certain tasks. Within our design, we used the concept of delegation between our PlayerHand and Player classes. Instead of using the Player

class to manage each player's hand of cards, we created a PlayerHand class to manage the hands instead. This way, the Player class does not directly deal with or manage the cards, it only deals with the players.

Cohesion

Our design is highly cohesive, meaning all classes contain methods and attributes that are closely related to their roles.

Examples include:

- Card class ONLY focuses on creating / representing cards.
- PlayerHand class only manages cards within a player's "hand".
- Player class only manages players' information.

Coupling

Our design is loosely coupled, meaning that most if not all classes are independent of each other. The classes do not rely on each other for details they simply interact with each other. Most classes within our system can be changed with minimal to no effect on other classes.

Inheritance

Inheritance is referred to as a "IS-A" relationship. This relationship is not used in our design.

Aggregation

Aggregation is referred to as a "HAS-A" relationship. This relationship is modelled between the Game and Player classes. Our game can have multiple players but when the game ends the players do not cease to exist.

Composition

Composition is also referred to as a "HAS-A" relationship, but it implies that the child class cannot exist independently of the parent. In our game design, you see this relationship modelled in two relationships, PlayerHand & Player, and Deck & Card. In the first relationship, the hand of cards cannot exist independently of the player, as the hand of cards is generated for each player and does not exist without the player. In the latter, a deck cannot exist without the cards existing. If cards do not exist, there cannot be a deck.

Flexibility/Maintainability

Flexibility and Maintainability refers to how easily our system can accept changes or additions. A system should allow for easy updates that won't immediately destroy the whole system.

Implementations:

- Within our system, we used enumerations for CardColor and CardValue, which allows for easy updates or extensions. We (if needed) could change or add colours or values without needing to rework an entire class or part of our code. Modifying enumerations much simpler (flexible) than needing to modify classes.
- Each class has its own responsibility and handles its own tasks. This makes it easy for classes to be updated without system damages. For example, Player class only holds player related attributes and methods, much like PlayerHand only handles "hand" related attributes and methods. If we needed to update these or add other methods, they would only need to be changed within their related classes, not across multiple.