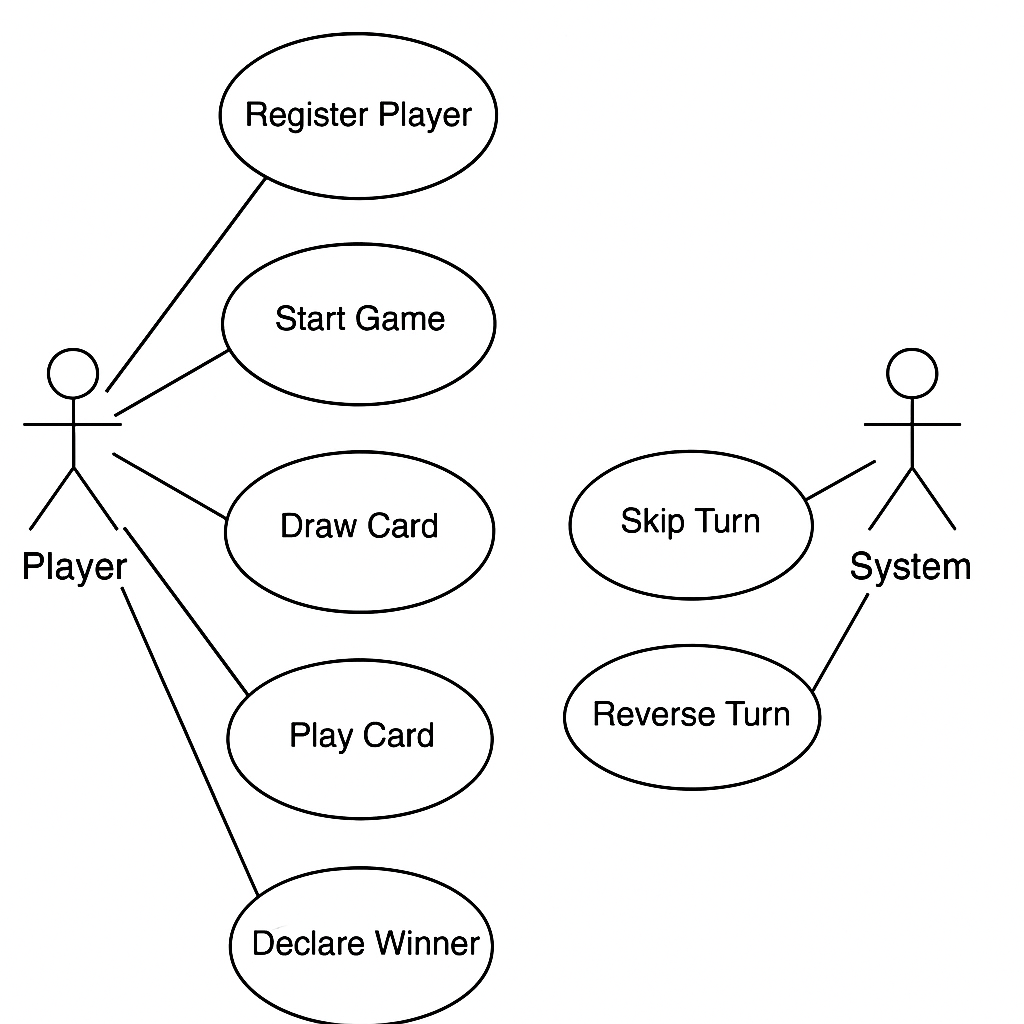
**Deliverable 2** for SYST17796 UNO Java OOP Game – Group 3

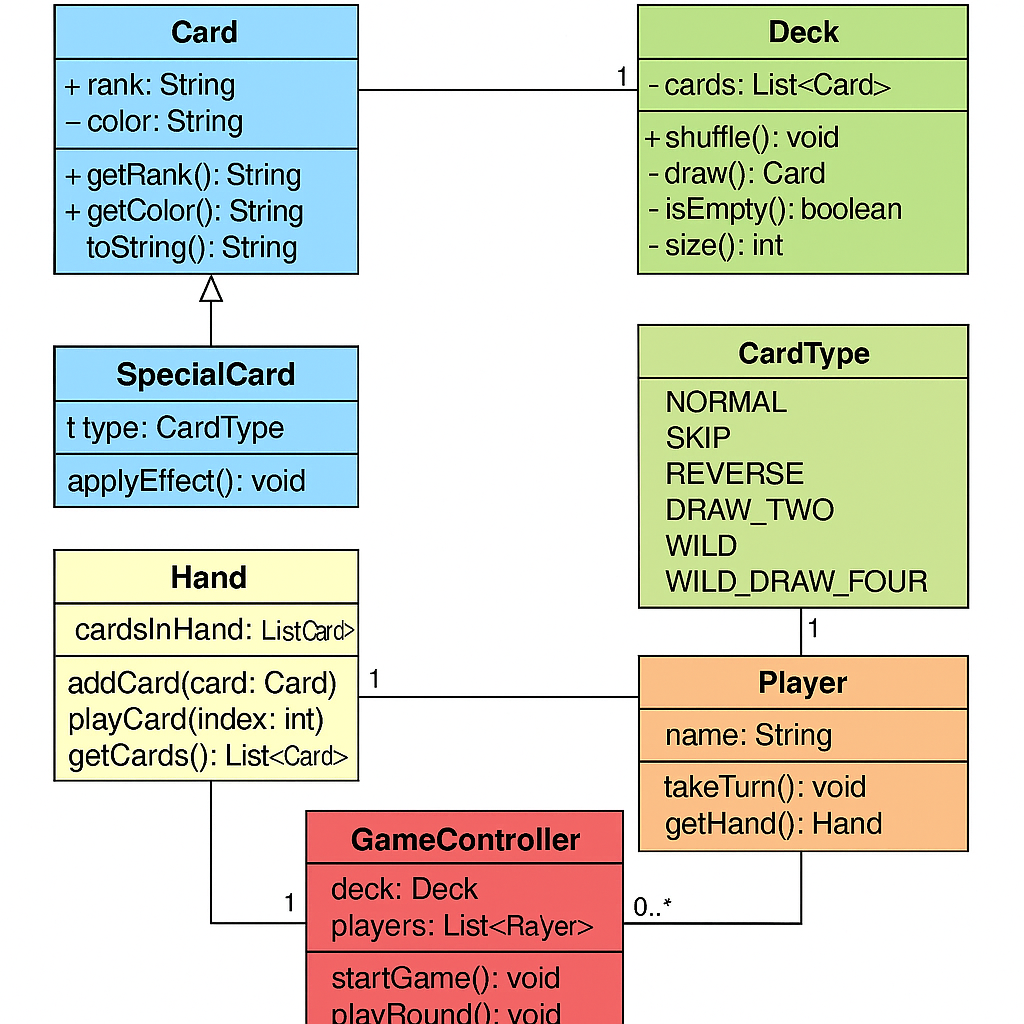
# Table of contents

* Use Case Diagram
* Class Diagram
* Use Case Narratives
* Design Document Template

# Use Case Diagram



Class Diagram



# 

# Use Case Narratives

* **Use Case: Register Player**
* **Actor**: Player
* **Precondition**: Game has not started
* **Main Flow**:
  1. Player enters name
  2. System creates Player instance
* **Alternate Flow**: Name empty → prompt again
* **Postcondition**: Player is registered
* **Use Case: Start Game**
* **Actor**: System
* **Precondition**: 2+ players registered
* **Main Flow**:
  1. Shuffle deck
  2. Deal 7 cards
  3. Flip top card
* **Postcondition**: Game in progress
* **Use Case: Draw Card**
* **Actor**: Player
* **Precondition**: No playable cards
* **Main Flow**:
  1. Draw top card
* **Alternate Flow**: Deck empty → reshuffle discard
* **Postcondition**: Card added to hand
* **Use Case: Play Card**
* **Actor**: Player
* **Precondition**: Game is active
* **Main Flow**:
  1. Select card
  2. Validate move
  3. Add to discard pile
* **Alternate Flow**: Invalid → error message
* **Postcondition**: Card played or action failed
* **Use Case: Skip Turn**
* **Actor**: System
* **Precondition**: Skip card played
* **Main Flow**:
  1. Next player's turn skipped
* **Postcondition**: Turn moved to next-next player
* **Use Case: Reverse Turn**
* **Actor**: System
* **Precondition**: Reverse card played
* **Main Flow**:
  1. Flip direction of play
* **Postcondition**: Reversed turn order
* **Use Case: Declare Winner**
* **Actor**: System
* **Precondition**: Hand size = 0
* **Main Flow**:
  1. Check hand
  2. Display winner
* **Postcondition**: Game ends

# Design Document Template

**Project Overview**

The well-known card game UNO is simulated textually in the UNO Java OOP project. It concludes when the player runs out of cards and features special card logic (Skip, Reverse, Draw Two, and Wild). Being the first to empty your hand while utilizing unique rules that impact other players and turn order is the aim.

**In our execution:**

* Until one player's hand runs out, the game goes on.
* Players use special actions, draw cards, and take turns.
* If the deck is empty, it is reshuffled.
* The remaining hand values might be included to the scoring (optional extension).

**System Flexibility**

* Permits adding additional card kinds (like future power-ups).
* Changing the number of players is simple.
* Modular and adjustable game rules

. **Design Considerations**

* **Associations & Multiplicities**
* The deck includes a list of
* The player only has one hand.
* A list of players is managed by GameController.
* Card is the source of SpecialCard.
* Types (SKIP, REVERSE, etc.) are defined by the CardType enum.
* **Principles Applied:**
* Encapsulation: Every class field, including hand, color, and rank, is private and made public by getters and setters.
* GameController assigns the Deck the task of drawing cards, the Player the task of playing cards, and the SpecialCard the task of applying effects.
* Cohesion: Every class is in charge of a specific idea:
* Card = card logic
* Player = actions
* Deck = sketching/shuffle.
* Coupling: Classes solely rely on the information they require (for example, Hand returns a List<Card> without changing the game state).
* Inheritance: In order to add logic for Draw Two, Wild, and other operations, SpecialCard overrides applyEffect() and derives from Card.
* Aggregation: Without transferring ownership, Deck combines several Card objects.
* Composition: A player's lifespan is tied, meaning that if the player is removed, their hand is also removed.
* Flexibility/Maintainability:
* By overriding applyEffect() and extending CardType, the system allows the addition of additional card types.
* Without changing base classes, game logic can be modified for scoring or team-based play.