# **Go Fish Project Documentation**

***Git Repository URL:***

* [***https://github.com/HarrisonDsouza/Project-GoFish.git***](https://github.com/HarrisonDsouza/Project-GoFish.git)

***Team Members and Roles***

* ***Harrison Daniel Dsouza:*** *Project Manager and Lead Developer*
* ***Misha Punj****: Developer and Tester*
* ***Neil Patrick Saldanha*** *: Developer and Documentation Specialist*
* ***Sukhmanpreet Singh Aulakh****: Developer and Documentation Specialist (2)*

**Project Background and Description**

***Game Description:***

The Go Fish card game is a game designed for 2-6 players. The objective is to collect as many sets of four matching cards, called “books,” as possible. Here is an overview of the game:

1. **Dealing Cards:** Each player is dealt a number of cards:
   * 7 cards if there are 2-3 players.
   * 5 cards if there are 4-6 players.
2. **Draw Pile:** The remaining cards form the draw pile.
3. **Player Turns:** Players take turns asking opponents for specific ranks of cards.
   * If the opponent has the card(s), they must give them to the requesting player.
   * If the opponent does not have the card(s), they say “Go Fish,” and the requesting player draws a card from the draw pile.
4. **Forming Books:** Players collect books by forming sets of four cards of the same rank.
5. **Game End:** The game concludes when all cards from the draw pile are exhausted. The winner is the player with the most books.

***Project Scope:***

* **Game Termination:** The game ends when the draw pile is empty. The winner is determined by the player with the most books.
* **Rounds:** The game does not have a fixed number of rounds but continues until the deck is empty and all books are formed.

**Design Considerations**

A diagram of a game

Description automatically generated

*Figure 1 shows the class diagram for the Go Fish project. It includes the Card, Deck, Player, Game, Admin, and GameEngine classes, their attributes, methods, and the relationships between them.*

***Detailed Association Descriptions*:**

1. **Game and Player:**
   * **Association:** A Game contains between 2 and 6 Player objects.
   * **Description:** This means that the Game class manages multiple Player instances, reflecting the number of players participating in the game. The Game class is responsible for organizing and coordinating the actions of these players.
2. **Player and Card:**
   * **Association:** A Player holds zero or more Card objects.
   * **Description:** Each Player has a hand that can contain any number of cards, starting from none and potentially growing as cards are drawn from the deck. This association allows the Player to manage their own cards.
3. **Deck and Card:**
   * **Association:** A Deck contains exactly 52 Card objects.
   * **Description:** The Deck class represents a full deck of playing cards, consisting of 52 individual Card objects. The deck is responsible for shuffling and dealing these cards during the game.
4. **Game and Deck:**
   * **Association:** A Game uses a single Deck.
   * **Description:** The Game class relies on one Deck instance to deal cards and manage the draw pile. The deck is essential for the game’s operations, including dealing cards to players and drawing from the pile.
5. **GameEngine and Game:**
   * **Association:** The GameEngine manages a single Game.
   * **Description:** The GameEngine class oversees the operations of one Game instance, handling game initialization, progression, and conclusion. It interacts with the Game to control various aspects of gameplay.
6. **Admin and Game:**
   * **Association:** An Admin oversees one Game.
   * **Description:** The Admin class is responsible for managing a single game, including adding and removing players, monitoring gameplay, and handling any disputes. The admin's role is to ensure the smooth operation of the game.

***Design Principles:***

**Encapsulation**

**Encapsulation** is a core concept of object-oriented programming that involves bundling the data (attributes) and the methods (functions) that operate on the data into a single unit or class. It restricts direct access to some of an object's components, which is a means of preventing unintended interference and misuse.

* **Implementation in Go Fish Project:**
  + **Card Class:** The Card class has private attributes suit and rank, which are accessed through public methods getSuit() and getRank(). This hides the internal representation of the card’s suit and rank, allowing controlled access and modification.
  + **Deck Class:** The Deck class maintains a private list of Card objects. Methods such as shuffle() and deal() are provided to interact with this private list, ensuring that the deck’s state is only modified in a controlled manner.

**Benefits:**

* Protects the integrity of the data by preventing unauthorized access or modification.
* Makes the code more modular and easier to maintain by exposing only necessary details.

**Delegation**

**Delegation** refers to an object relying on another object to perform a specific task or action. It allows for the distribution of responsibilities among classes, promoting a cleaner design and adherence to the Single Responsibility Principle.

* **Implementation in Go Fish Project:**
  + **Game Class and Deck Class:** The Game class delegates the task of shuffling and dealing cards to the Deck class. This means that the Game class is not responsible for the intricacies of managing the deck but can focus on game logic.
  + **Player Class and Deck Class:** The Player class delegates the action of drawing cards from the deck to the Deck class’s deal() method.

**Benefits:**

* Reduces the complexity of a class by offloading specific tasks to other classes.
* Promotes code reuse and improves modularity.

**Cohesion**

**Cohesion** refers to how closely related and focused the responsibilities of a single class are. High cohesion within a class means that its responsibilities are closely related and serve a single purpose.

* **Implementation in Go Fish Project:**
  + **Card Class:** The Card class has a single responsibility—representing a playing card with a suit and rank. All its methods and attributes are related to this single purpose.
  + **Game Class:** The Game class is responsible for managing the overall state of the game, including handling player turns, managing the deck, and determining the winner. Its methods and attributes are aligned with managing game flow.

**Benefits:**

* Makes classes easier to understand and maintain by focusing on a single responsibility.
* Enhances code readability and reusability.

**Coupling**

**Coupling** refers to the degree of interdependence between classes. Low coupling means that classes are minimally dependent on each other, which makes them more modular and easier to maintain.

* **Implementation in Go Fish Project:**
  + **Game Class and Player Class:** The Game class interacts with the Player class through well-defined methods. For example, it uses methods to add or remove players but does not need to know the internal details of the Player class.
  + **Player Class and Deck Class:** The Player class uses methods from the Deck class to draw cards, but it does not directly manipulate the deck's internal state.

**Benefits:**

* Reduces the impact of changes in one class on others, making the codebase more robust and maintainable.
* Facilitates easier testing and debugging by isolating changes.

**Inheritance**

**Inheritance** allows a class to inherit attributes and methods from another class. It promotes code reuse and establishes a natural hierarchy among classes.

* **Implementation in Go Fish Project:**
  + **RegisteredPlayer and NewPlayer Classes:** Both RegisteredPlayer and NewPlayer inherit from the Player class. This allows them to reuse common functionality, such as managing a hand of cards, while adding specific features related to player registration and login for RegisteredPlayer and handling new player registration for NewPlayer.

**Benefits:**

* Promotes code reuse and reduces redundancy.
* Establishes a clear hierarchical relationship among classes.

**Aggregation**

**Aggregation** is a "has-a" relationship where one class contains references to objects of another class, but these objects can exist independently of the container class.

* **Implementation in Go Fish Project:**
  + **Game Class and Player Class:** The Game class aggregates Player objects. Players can exist independently of the game, but the game manages and organizes their participation.
  + **Game Class and Deck Class:** The Game class aggregates a Deck object. The deck is managed by the game but can be used in different contexts.

**Benefits:**

* Represents relationships where the contained objects have a life cycle independent of the container class.
* Enhances flexibility by allowing objects to be shared among multiple classes.

**Composition**

**Composition** is a stronger form of aggregation where the contained objects’ lifecycle is tied to the lifecycle of the container class. If the container class is destroyed, the contained objects are also destroyed.

* **Implementation in Go Fish Project:**
  + **Deck Class and Card Class:** The Deck class is composed of Card objects. The lifecycle of these Card objects is tied to the deck, meaning that when a deck is created or destroyed, the cards it contains are also created or destroyed.

**Benefits:**

* Ensures that the contained objects are managed as part of the container class’s lifecycle.
* Provides a more tightly coupled relationship that guarantees the integrity of the contained objects.

**Flexibility/Maintainability**

**Flexibility/Maintainability** refers to how easily a system can adapt to changes and how easily it can be maintained.

* **Implementation in Go Fish Project:**
  + **Use of Interfaces/Abstract Classes:** Future enhancements, such as adding AI players or new game rules, can be supported by using interfaces or abstract classes. This allows for easy extension of the system without modifying existing code.
  + **Separation of Concerns:** The project’s design separates different functionalities into distinct classes. For instance, the GameEngine handles game operations, while the Game class manages game state. This separation allows for easier modifications and enhancements.

**Benefits:**

* Facilitates easy updates and additions to the system.
* Promotes a modular design that enhances the system’s overall maintainability.

**3. Use Case Diagram**

A diagram of a diagram

Description automatically generated

**Figure 2:** Use Case Diagram for Go Fish Project

*Figure 2 illustrates the use cases for the Go Fish project. It includes interactions between actors (New Player, Registered Player, Admin, Game Engine) and use cases (Register, Login, Request Card, Draw Card, Form Book, View Score, Add Player, Remove Player, Monitor Players, Track Scores, Initialize Game, Check Winner, Reset Game, Start Game, Stop Game).*

**Actors and Use Cases:**

1. **New Player:**
   * **Use Cases:**
     + Register
   * **Description:** A new player can register an account to join the game.
2. **Registered Player:**
   * **Use Cases:**
     + Login
     + Request Card
     + Draw Card
     + Form Book
     + View Score
   * **Description:** A registered player can log in, interact with other players by requesting and drawing cards, form books, and view their score.
3. **Admin:**
   * **Use Cases:**
     + Add Player
     + Remove Player
     + Monitor Players
     + Track Scores
   * **Description:** The admin manages the game by adding or removing players, monitoring player performance, tracking scores, and handling disputes.
4. **Game Engine:**
   * **Use Cases:**
     + Initialize Game
     + Check Winner
     + Reset Game
     + Start Game
     + Stop Game
     + Monitor Players
     + Track Scores
   * **Description:** The Game Engine oversees game operations, including initialization, managing game progress, checking for a winner, and tracking scores.

**4. Use Case Narrative**

**1. Register**

1. New Player wishes to register for the game

1.1 Player decides to cancel registration

1.2 System verifies input fields

1.2.1 Player provides a unique username

1.2.1.1 Username is available

* Create a new player account

1.2.1.2 Username is not available

* Display “Username already exists” message

1.2.2 Player provides an existing username

* Display “Username already exists” message

1.3 System fails to validate input

* Display “Invalid input” message

1. Player completes registration

**2. Login**

1. Registered Player wishes to log in

1.1 Player decides to cancel login

1.2 System verifies credentials

1.2.1 Credentials are correct

* Log player in and display main game interface

1.2.2 Credentials are incorrect

* Display “Invalid credentials” message

1.3 System fails to verify credentials

* Display “System error” message

1. Player logs in successfully

**3. Request Card**

1. Registered Player wishes to request a card

1.1 Player decides to cancel card request

1.2 Player requests a card from an opponent

1.2.1 Opponent has the requested card(s)

* Transfer the card(s) to the requesting player

1.2.2 Opponent does not have the requested card(s)

* Player is prompted to “Go Fish” and draws from the draw pile

1.3 System fails to process card request

* Display “Error processing request” message

1. Card request is processed

**4. Draw Card**

1. Registered Player wishes to draw a card

1.1 Player decides to cancel the draw

1.2 Player draws a card from the draw pile

1.2.1 Draw pile has cards

* Draw a card and add it to the player’s hand

1.2.2 Draw pile is empty

* Display “Draw pile is empty” message

1.3 System fails to process draw

* Display “Error drawing card” message

1. Card is drawn successfully

**5. Form Book**

1. Registered Player wishes to form a book

1.1 Player decides to cancel forming a book

1.2 Player forms a book with four cards of the same rank

1.2.1 Player has four cards of the same rank

* Form the book and update player’s score

1.2.2 Player does not have four cards of the same rank

* Display “Not enough cards to form a book” message

1.3 System fails to form a book

* Display “Error forming book” message

1. Book is formed and player’s score is updated

**6. View Score**

1. Registered Player wishes to view their score

1.1 Player decides to cancel viewing score

1.2 Player requests score view

1.2.1 Score is available

* Display player’s current score and books

1.2.2 Score is not available

* Display “Error retrieving score” message

1.3 System fails to retrieve score

* Display “Error retrieving score” message

1. Player’s score is displayed

**7. Add Player**

1. Admin wishes to add a new player

1.1 Admin decides to cancel adding a player

1.2 System verifies player details

1.2.1 Details are valid

* Add the new player to the game

1.2.2 Details are invalid

* Display “Invalid player details” message

1.3 System fails to add player

* Display “Error adding player” message

1. New player is added

**8. Remove Player**

1. Admin wishes to remove a player

1.1 Admin decides to cancel removing a player

1.2 System verifies player details

1.2.1 Player exists and can be removed

* Remove the player from the game

1.2.2 Player does not exist or cannot be removed

* Display “Player not found or cannot be removed” message

1.3 System fails to remove player

* Display “Error removing player” message

1. Player is removed

**9. Monitor Players**

1. Admin wishes to monitor players

1.1 Admin decides to cancel monitoring

1.2 System provides player activity updates

1.2.1 Updates are available

* Display player activities and game status

1.2.2 Updates are not available

* Display “Error retrieving updates” message

1.3 System fails to provide updates

* Display “Error monitoring players” message

1. Player activities and game status are displayed

**10. Track Scores**

1. Admin wishes to track scores

1.1 Admin decides to cancel score tracking

1.2 System provides score updates

1.2.1 Scores are available

* Display player scores

1.2.2 Scores are not available

* Display “Error retrieving scores” message

1.3 System fails to track scores

* Display “Error tracking scores” message

1. Player scores are displayed

**11. Initialize Game**

1. Game Engine wishes to initialize a new game

1.1 Initialization is cancelled

1.2 System sets up the game

1.2.1 Players and deck are prepared

* Initialize players and shuffle deck

1.2.2 System fails to initialize

* Display “Error initializing game” message

1.3 System fails to set up game

* Display “Error setting up game” message

1. Game is initialized

**12. Check Winner**

1. Game Engine wishes to check the winner

1.1 Checking is cancelled

1.2 System checks for the winner

1.2.1 Winner is identified

* Display winner’s name

1.2.2 Winner cannot be determined

* Display “Error determining winner” message

1.3 System fails to check winner

* Display “Error checking winner” message

1. Winner is announced

**13. Reset Game**

1. Game Engine wishes to reset the game

1.1 Reset is cancelled

1.2 System resets the game

1.2.1 Game is reset successfully

* Clear current data and prepare for a new game

1.2.2 Game fails to reset

* Display “Error resetting game” message

1.3 System fails to reset game

* Display “Error resetting game” message

1. Game is reset

**14. Start Game**

1. Game Engine wishes to start the game

1.1 Start is cancelled

1.2 System starts the game

1.2.1 Game starts successfully

* Initiate player turns and gameplay

1.2.2 Game fails to start

* Display “Error starting game” message

1.3 System fails to start game

* Display “Error starting game” message

1. Game is started

**15. Stop Game**

1. Game Engine wishes to stop the game

1.1 Stop is cancelled

1.2 System stops the game

1.2.1 Game stops successfully

* Process game results and conclude the game

1.2.2 Game fails to stop

* Display “Error stopping game” message

1.3 System fails to stop game

* Display “Error stopping game” message

Game is stopped