**SYST 17796 Deliverable 2:**

**Design Document Template**

# Project Background and Description

## Game Description:

Our chosen game is a simplified version of "Go Fish," a card game where each player attempts to collect as many sets of four matching cards (ranks) as possible. Players take turns asking other players for cards of a specific rank, and if the asked player has cards of that rank, they must hand them over. If not, the asking player draws a card from the deck. The game ends when all sets of cards are collected, and the player with the most sets wins.

## Project Scope:

* The game supports up to 4 players.
* Each player starts with a hand of 5 cards.
* Players take turns to ask for cards and draw from the deck if necessary.
* The game ends when all cards are drawn, and the player with the highest score (most sets of four) wins.

## Code Update with OO Principles:

* The game has been refactored to follow object-oriented principles.
* Classes created include `Game`, `Player`, `Deck`, and `Card`.

# Design Considerations

A screenshot of a computer program

Description automatically generated

*Class Diagram (Figure 1)*

Update and describe the Class Diagram, explaining the associations and multiplicities depicted.

### Associations and Multiplicities:

* **Game** has a 1...\* relationship with **Player**, meaning a game can have one or more players.
* **Player** has a 1...\* relationship with **Card**, meaning a player can have one or more cards in their hand.
* **Game** has a 1...1 relationship with **Deck**, meaning a game has one deck of cards.
* **Deck** has a 1...\* relationship with the Card class, meaning that a deck can have one or more cards.
* The multiplicities are shown next to the association lines in the diagram.
* 1..1 means a single instance of the class on one end of the association is related to a single instance of the class on the other end.
* 1... \* means a single instance of the class on one end of the association is related to one or more instances of the class on the other end.

### Encapsulation

* The classes are well-encapsulated, with private members and public methods to access and modify them. This protects the internal state of each class from external manipulation.
* Example: The Card class encapsulates the suit and rank of the card, and only provides methods to access or modify these values.

### Cohesion

* The classes have high cohesion, as they focus on a single responsibility.
* Example: The Deck class is responsible for managing the deck of cards, and does not handle any game logic or player interactions.

### Coupling

* The classes have low coupling, as they interact with each other through well-defined interfaces.
* Example: The Game class interacts with the Player class through the registerPlayer() method, and with the Deck class through the drawCard() method

### Inheritance

* There is no inheritance in the class diagram.

### Aggregation

* The Game class has an aggregation relationship with the Deck class, because the Game class can exist without the Deck class, but the Deck class cannot exist without the Game class.
* The Player class has an aggregation relationship with the Card class, because the Player class can exist without the Card class, but the Card class cannot exist without the Player class.

### Composition

* There is no composition in the class diagram

### Flexibility/Maintainability

* The class diagram is flexible and maintainable, as the classes are well-defined and have low coupling.
* This makes it easy to modify or extend the functionality of the game without affecting other parts of the code.
* Example: If a new card type is added to the game, only the Card class needs to be modified. The other classes will not be affected.
* Example: If a new game rule is added, only the Game class needs to be modified. The other classes will not be affected.