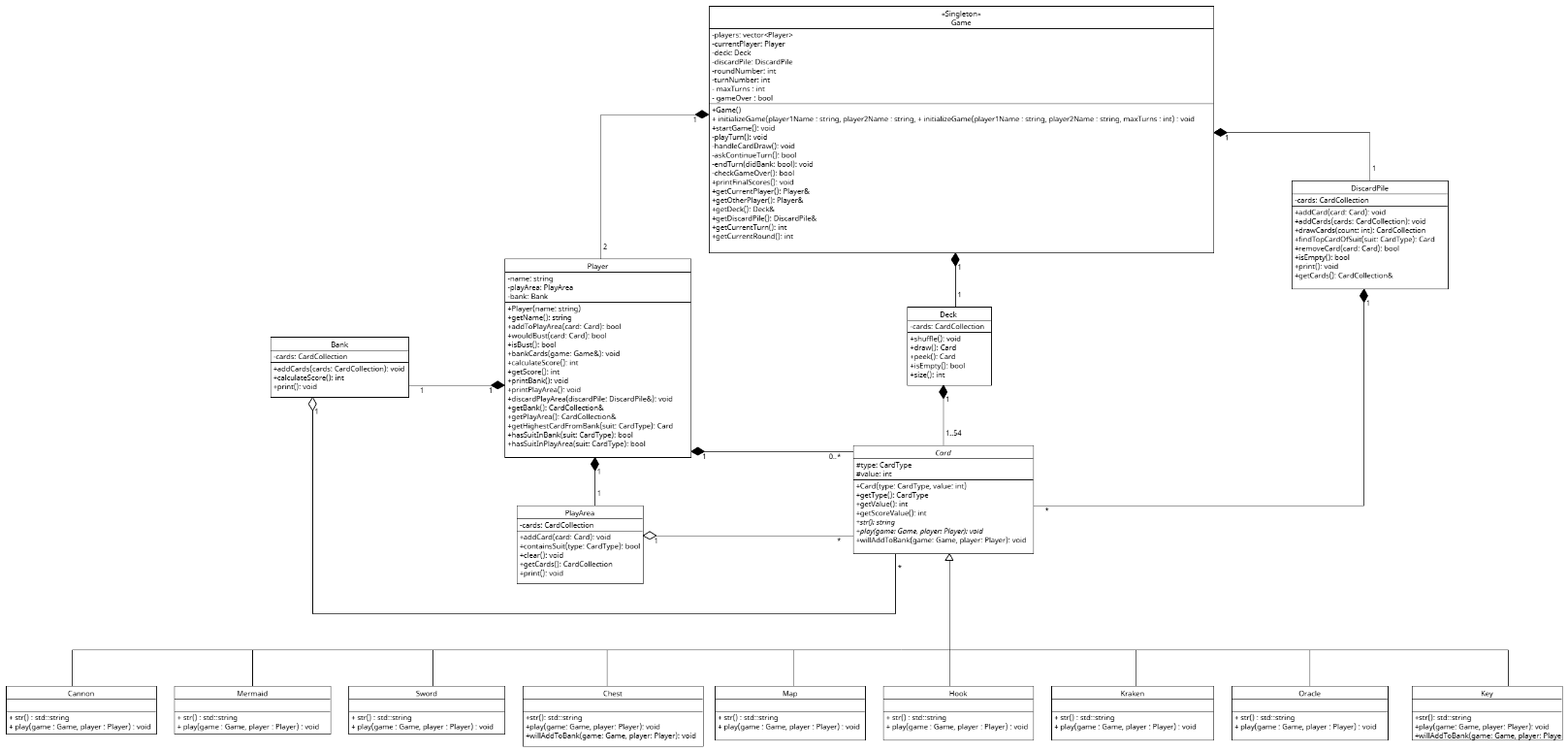
# Task 1 – System design

Student Name: Shalini Ashika Algama

Student ID: 110395137

Email: [shaay208@mymail.unisa.edu.au](mailto:shaay208@mymail.unisa.edu.au)

## UML class diagram



## Justification

#### Class Justifications

**Game Class**

* Represents the overall controller of the card game.
* Designed as a singleton to ensure only one game instance controls the session.
* Stores key components: list of players, deck, discard pile, round and turn information.
* Responsible for starting the game, managing turns, checking busts, ending rounds, and declaring winners.
* Coordinates interaction between players, deck, and card effects.

**Player Class**

* Represents each individual player in the game.
* Stores the player's name, current play area (cards drawn in a turn), and bank (cards safely collected).
* Handles actions like drawing cards, banking them, discarding from play, and calculating score.
* Provides access to player state for the game manager and card effects.

**Card Class**

* Abstract representing a general card in the game.
* Stores shared attributes such as CardType (e.g., Cannon, Sword) and value (score value).
* Provides virtual methods: play(), winAddToBank(), and getScoreValue() to be overridden by child classes.
* Includes utility methods like str() for string output.
* Base class avoids duplication and enables polymorphism, allowing dynamic behavior for different card types.

**Card Subclasses (suits)**

* Each subclass represents a unique card with a special ability.
* Inherits from Card and overrides the play() method to implement its own effect during a player’s turn.
* Some override winAddToBank() to define custom rules when banked (e.g., Key and Chest interaction).
* Allows clear, reusable, and extendable implementation of diverse card mechanics using inheritance and method overriding.

**PlayArea Class**

* Represents the area where cards are placed temporarily during a player’s turn.
* Used to store cards as they are drawn, and to check for busts.
* Supports clearing cards and checking contents.
* Keeps turn-specific logic separated from long-term game state improving clarity and logic flow.

**Bank Class**

* Stores cards that a player has safely banked at the end of a successful turn.
* Provides functionality to add cards and calculate total score from banked cards.
* Interacts with scoring logic to determine who wins the game.
* Allows players to accumulate points over multiple rounds and separates temporary cards from permanent ones.

**Deck Class**

* Represents the deck of cards from which players draw during the game.
* Contains a collection (such as a list or vector) of 54 Card objects.
* Provides methods to shuffle, draw the top card, peek at cards, and check if the deck is empty.
* Isolating deck behavior improves modularity, and methods like shuffle() and draw() reflect real card game behavior.

**DiscardPile Class**

* Manages all cards discarded during the game.
* Stores discarded cards and supports actions like adding cards, removing specific types, drawing a number of cards, or searching for cards.
* Enables game logic such as Oracle and Map effects to interact with discarded cards.
* Handling discarded cards separately prevents confusion and enables effects that rely on card history.

#### Relationship Justifications

**Composition Relationships**  
Composition represents strong ownership. When the parent is destroyed, its parts are also destroyed. It is used where the contained object cannot logically exist without its owner.

* Game → Deck [1]  
  The Game class owns the Deck. A Deck is created during game initialization and destroyed when the Game ends. It cannot exist independently.
* Game → DiscardPile [1]  
  The DiscardPile is created and managed solely by the Game. It exists only within the context of a Game instance.
* Game → Players [2]  
  Exactly two Player objects are created and managed by the Game. These Players exist only within the lifecycle of the Game session.
* Deck → Cards [1..54]  
  The Deck is initialized with exactly 54 cards (9 suits × 6 cards each). As cards are drawn, the number of cards in the Deck decreases, but during the Game, it always contains between 1 and 54 cards until it is empty.
* DiscardPile, Bank, PlayArea → Cards [0..\*]  
  These classes own collections of Cards during the game. For example, a PlayArea contains drawn cards, which are either banked or discarded. Once moved, those Cards no longer belong to the PlayArea. This signifies strong ownership during containment.
* Player → Bank [1] and PlayArea [1]  
  Each Player is initialized with one Bank and one PlayArea. These components are exclusive to the Player and are destroyed alongside the Player. They are not shared or reused.

**Aggregation Relationships**  
Aggregation represents a weaker ownership or association, where the child can exist independently of the parent. It reflects logical connections but not lifecycle dependency.

* Player ↔ Game  
  While the Game manages the Players, it does not strictly own them in terms of lifecycle. This connection is logical — the Game uses Players to operate, but they can be passed externally or reused.
* Card ↔ Game and Card ↔ Player  
  Cards interact with the Game and Players (e.g. when played or when abilities are triggered), but they are not owned by them. A Card may reference the Game state or a Player during execution without being permanently tied to either. This allows Cards to move freely between containers (Deck, DiscardPile, PlayArea, Bank) without tight coupling.

#### Multiplicity Justifications

* Game → Players [2]  
  The game is designed for two players only. This fixed multiplicity enforces the game rules.
* Deck → Cards [1..54]  
  A Deck is initialized with 54 cards and, throughout gameplay, contains between 1 and 54 cards until empty. This range reflects both the starting state and the dynamic reduction during play.
* DiscardPile → Cards [0..\*]  
  The DiscardPile starts empty and accumulates discarded cards over time. It has no upper bound.
* Player → Bank and PlayArea [1]  
  Each Player has exactly one Bank and one PlayArea throughout the game session. These are permanent and exclusive components.
* PlayArea → Cards [0..\*]  
  Cards are added to the PlayArea as the Player draws them. It can be empty or hold multiple cards depending on the player's decisions in a turn.