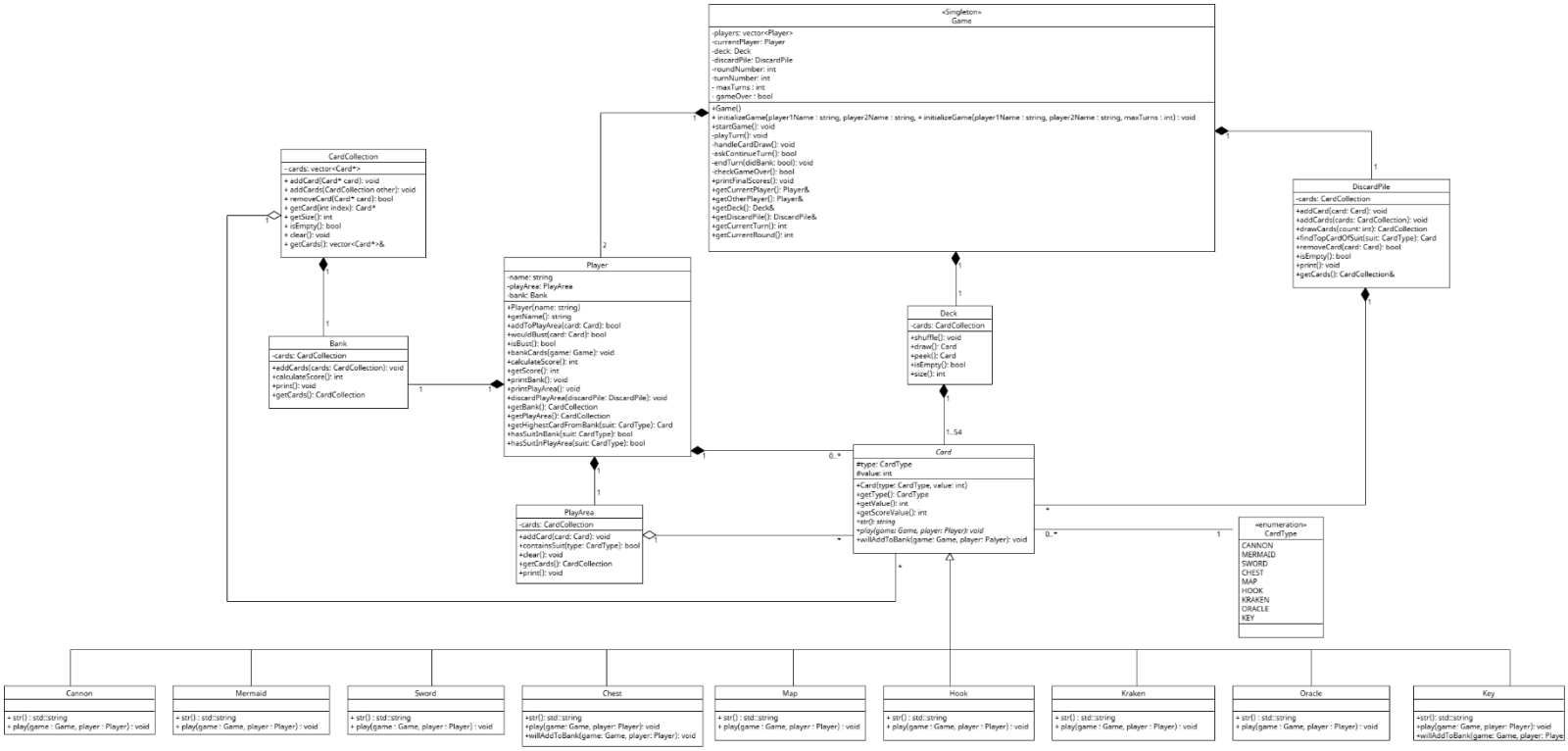
Task 1 – System design

Student Name: Shalini Ashika Algama

Student ID: 110395137

Email: 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 adding cards to play area, checking for busts, banking cards, discarding from play, and calculating score.
- Provides access to player state (name, score, bank, play area) for the game manager and card effects.

Card Class

- Abstract representing a general card in the game.
- Stores shared attributes such as type (CardType enum) and value (score value).
- Provides virtual methods: play(), willAddToBank(), and core methods like getType(), getValue(), getScoreValue().
- Includes utility methods like str() for string representation.
- Base class structure avoids duplication and enables polymorphism, allowing dynamic behavior for different card types via subclasses.

Card Subclasses (Cannon, Chest, Hook, Key, Kraken, Map, Mermaid, Oracle, Sword)

- Each subclass represents a unique card with a special ability.
- Inherits from the Card base class.
- Overrides the virtual play() method to implement its specific effect when played during a player's turn.
- Some subclasses (like Chest and Key) override willAddToBank() to define custom rules or interactions when banked.
- Overrides str() for specific string representations if needed (though base str might suffice if based on type/value).
- Allows clear, reusable, and extendable implementation of diverse card mechanics using inheritance and polymorphism.

PlayArea Class

- Represents the temporary area where cards are placed during a single player's turn.
- Uses a CardCollection to store cards drawn in the current turn.
- Used to check for busts (e.g., containsSuit to detect duplicate suits).
- Supports adding cards (addCard), clearing cards at the end of a turn (clear), and checking contents (getCards, print).
- Keeps turn-specific card management separated from the player's permanent bank, improving clarity.

Bank Class

- Stores cards that a player has successfully banked at the end of a turn.
- Uses a CardCollection to manage the banked cards.
- Provides functionality to add cards (addCards - likely takes a CardCollection from PlayArea) and calculate the total score from banked cards (calculateScore).
- Interacts with scoring logic (via Player and Game) to determine the player's overall score.
- Allows players to accumulate points safely over multiple turns/rounds.

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.
- Contains a CardCollection holding all the cards available to be drawn.
- Provides methods to shuffle the deck (shuffle), draw the top card (draw), look at the top card without drawing (peek), and check its state (isEmpty, size).
- Isolates deck behavior, ensuring proper card management and randomization.

DiscardPile Class

- Manages cards that have been discarded during the game (e.g., after busting, specific card effects, or clearing PlayArea without banking).
- Uses a CardCollection to store discarded cards.
- Supports actions like adding single cards (addCard), adding multiple cards (addCards), potentially drawing cards back under certain rules (drawCards), finding specific cards (findOneCardOfSuit), removing cards (removeCard), and checking state (isEmpty, print).
- Enables game logic such as Oracle and Map effects that might interact with discarded cards.
- Keeps discarded cards separate from the deck and player hands/banks.

CardType Enumeration

- Defines a fixed set of possible card types (suits) in the game (CANNON, CHEST, etc.).
- Used by the Card class to specify its type.
- Ensures type safety and consistency when referring to or checking card suits.
- Improves code readability and maintainability by using meaningful names instead of magic numbers or strings.

CardCollection

- A dedicated container class responsible for managing a collection of Card pointers (vector<Card*>).

- Used by Deck, DiscardPile, Bank, and PlayArea to hold their respective cards.
- Provides common card collection operations: adding (addCard, addCards), removing (removeCard), accessing (getCard, getCards), checking size/emptiness (getSize, isEmpty), and clearing (clear).
- Promotes code reuse and encapsulates the logic for handling groups of cards.

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 to Deck: The** Game owns and manages a single Deck. The Deck is created when the Game starts and typically destroyed when the Game ends. It doesn't exist independently.
- **Game to DiscardPile:** The Game owns and manages a single DiscardPile. Its lifecycle is tied to the Game.
- **Game to Player:** The Game owns and manages exactly two Player objects. These players are integral parts of the game session and their lifecycle is controlled by the Game.
- **Player to Bank:** Each Player owns exactly one Bank. The Bank is specific to the Player and is created and destroyed with the Player.
- **Player to PlayArea:** Each Player owns exactly one PlayArea. The PlayArea is specific to the Player and managed within the Player's lifecycle.
- **Bank to CardCollection:** Each Bank owns exactly one CardCollection to store its cards. The CardCollection is an internal component of the Bank and its lifetime is managed by the Bank.
- **Deck to Card [1..54]:** The Deck is initially composed of 54 Card objects. While cards are drawn (pointers moved), the Deck conceptually owns the cards it currently holds. The initial composition reflects the setup.

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.

- **DiscardPile to Card:** The DiscardPile contains zero or more Card objects (likely via pointers stored in its CardCollection). The Card objects themselves might have originated from the Deck or PlayArea. Aggregation here signifies the DiscardPile holds references to cards, but doesn't manage their fundamental creation/destruction in the same way composition does.
- **PlayArea to Card:** The PlayArea temporarily holds zero or more Card objects (via its CardCollection). These cards are drawn from the Deck and will later move to the Bank or DiscardPile. Aggregation reflects this temporary holding of references.
- **CardCollection to Card:** The CardCollection itself holds pointers (vector<Card*>) to zero or more Card objects. This is the core mechanism for Bank, Deck, DiscardPile, and PlayArea to manage their cards. Aggregation accurately reflects that the CardCollection manages *pointers* to Cards, not necessarily the Card objects' entire lifecycle intrinsically (though the owning class like Deck might manage that).

Inheritance Relationships

- **Card to Cannon, Chest, etc.:** The specific card types (Cannon, Chest, ...) are specializations of the general Card class. They inherit common properties and methods (type, value, getType, etc.) and provide specific implementations for virtual methods (play, willAddToBank). This uses polymorphism.

Association relationships

It shows one class uses or interacts with another, but they don't rely on each other for their lifecycle.Objects can exist independently of each other.

- **Card to CardType:** Each Card object has an associated CardType (via its type attribute). This is a fundamental property defining the card's suit/identity. The CardType enum exists independently of any single Card.
- **Card to Game/Player:** The play method of a Card takes references to the Game and the current Player. This represents a temporary association during the execution of the play method, allowing the card's effect to interact with the game state and the player, without implying permanent ownership or aggregation line in the static structure.

Multiplicity Justifications

- **Game to Player [2]:** The game requires exactly two players.
- **Game to Deck [1]:** There is only one Deck per game instance.
- **Game to DiscardPile [1]:** There is only one DiscardPile per game instance.
- **Player to Bank [1]:** Each Player has exactly one Bank.
- **Player to PlayArea [1]:** Each Player has exactly one PlayArea.
- **Bank to CardCollection [1]:** Each Bank uses exactly one CardCollection instance to hold its cards.
- **Deck to Card [1..54]:** The Deck starts with 54 cards and can hold between 1 and 54 during play (until empty, where it might technically be 0 before reshuffling, though 1..54 covers active play).
- **DiscardPile toCard [*]:** The DiscardPile can contain any number of cards, from zero upwards. (* means 0 or more).
- **PlayArea toCard [*]:** The PlayArea can temporarily hold zero or more cards during a player's turn.
- **CardCollection toCard [*]:** A CardCollection instance can hold zero or more card pointers.
- **Card to CardType [1]:** Each Card must have exactly one CardType.