# System Design Document

This document outlines the design of a card game system, and key decisions made during its development.

System is built around the **Card** class with specialised subclasses for each type. Each card has, point value, card type and ability to play. During the UML class diagram design process was made decision to use inheritance for card types rather than composition or other approach. This decision was made because each card type has own ability, and inheritance provides a way to override the *play()* method for each card (derived classes). Additionally, it is easy to add new card type by simply creating a new subclass and implement specific behaviour. It supports the "*open for extension, closed for modification*" principle by allowing new card types to add without changing existing code.

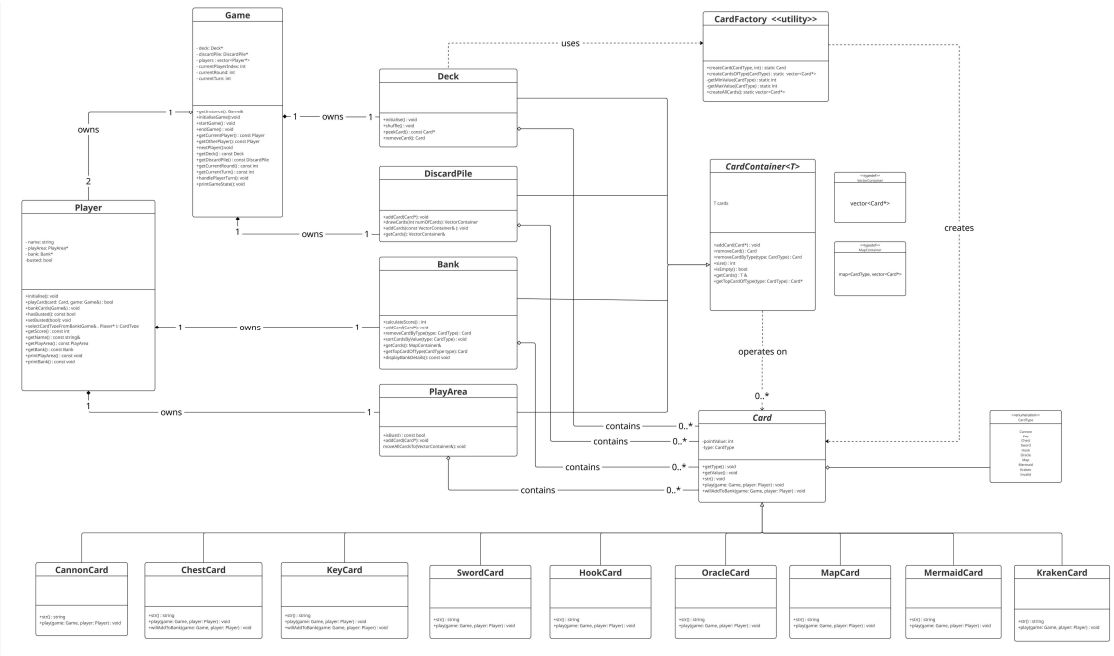
This design uses an abstract template class **CardContainer<T>** to define a common interface for different container implementations. CardContainer depends on Card for its method signatures but does not directly store Card objects. This is represented by a dependency relationship. The child classes (Deck, DiscardPile, PlayArea, Bank) use either VectorContainer or MapContainer to fill in the details of the template. This approach provides flexibility by using an abstract template class for containers, it provides a common interface while allowing for different implementation strategies. For instance, Bank uses a map to quickly access cards by type, which is essential for scoring and card retrieval. Other containers use vectors for simpler sequential access and modification. Each container type can implement methods suited to its needs while sharing common functionality. This approach

The **CardFactory** is implemented as a concrete utility class with **static factory methods** for creating cards. This centralises all card creation logic in one place and makes it easier to modify card properties or add new types. During making this decision was also emphasized that created card objects are immutable and does not change the state, otherwise more complex abstract factories or factory methods with inheritance would be implemented. Furthermore, static factory methods simply encapsulate the object creation logic and does not require create multiple constructors, what makes code more simplified. The following design shows the relationship between Deck and CardFactory class, Deck only needs to call a single method to get all cards. CardFactory is responsible for creating cards and Deck manages them. This encapsulates well of details how each card is created within CardFactory. For example, If the card creation logic changes, only need to update the factory or the value ranges changes for different card types, can only update the getMinValue and getMaxValue methods.

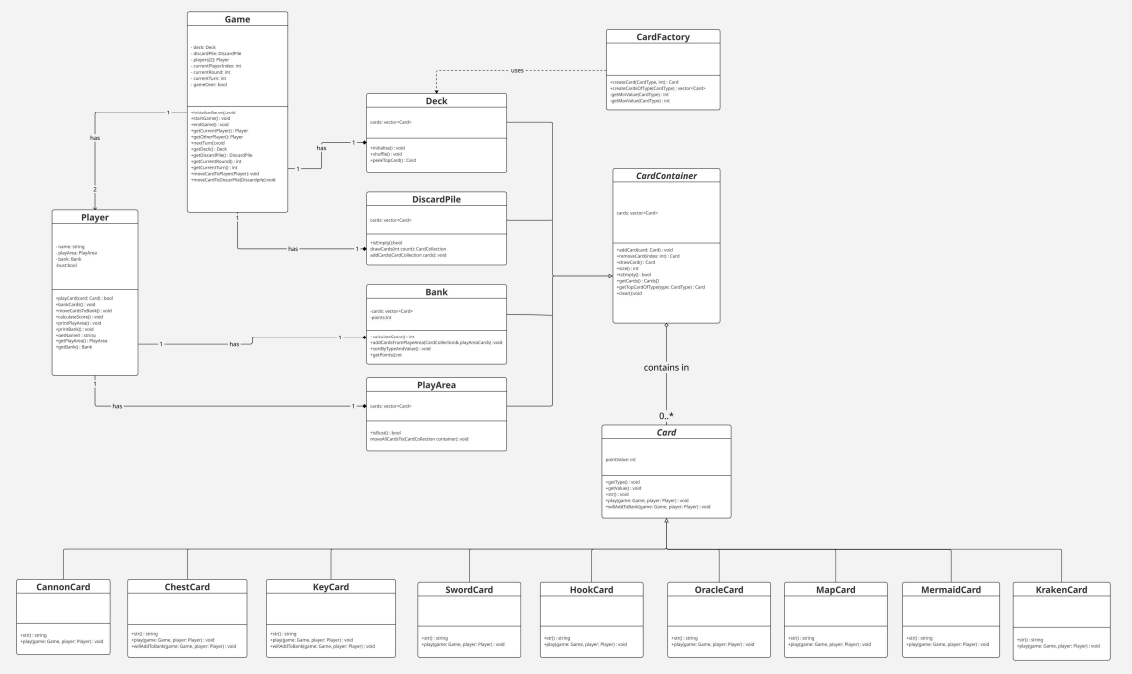
The **Game** class serves as a central controller and managing game state, game flow and game components. Game is implemented as a **Singleton** to ensure only one game

instance running at any time. This approach prevents creating multiple game instances that could lead inconsistent state. Another aspect is that it simplifies access to game components, makes game state globally accessible for players and cards.

## 1.2 Final Design




## 1.1 Modified



# Initial Design 1.0