COMP3023 A1 DESIGN DOCUMENT

[Kaldt001 – Dennis Kalongonda]

Kaldt001@mymail.unisa.edu

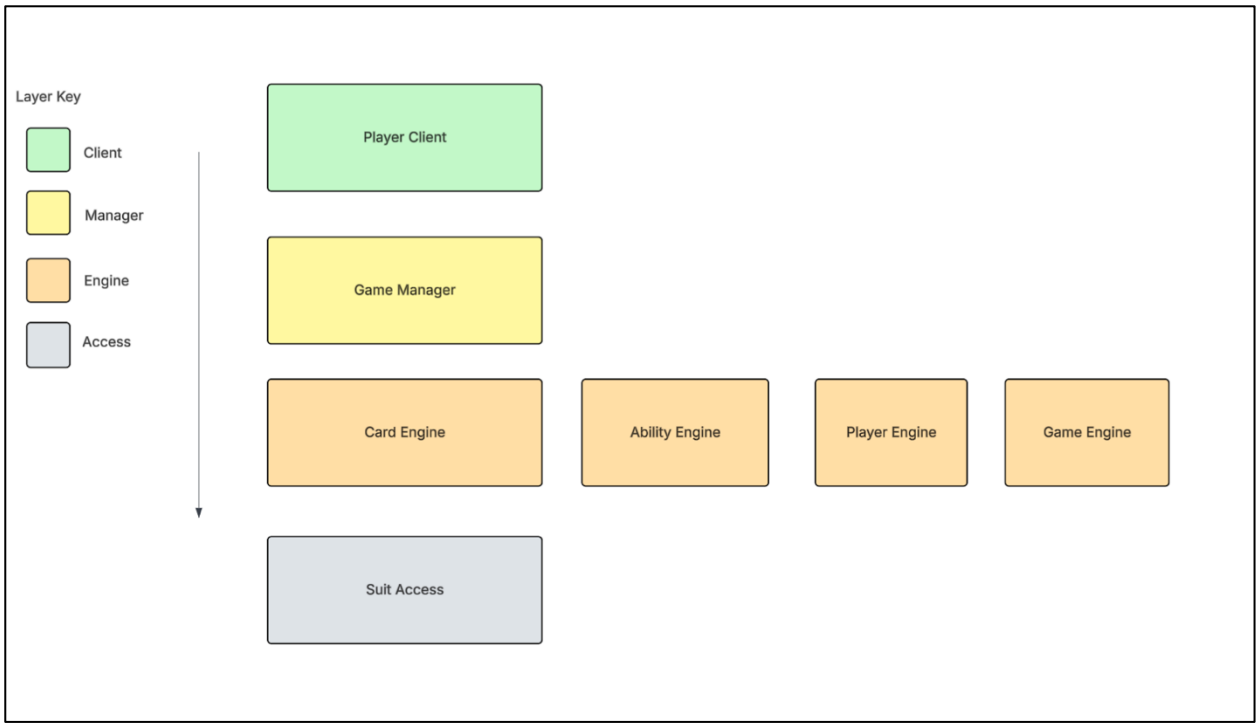
--

Abstract

An in depth zoom into the thought process behind the design of my associated code solution of Dead Man's Draw ++.

Brief Document Overview:
This document is centred on the design of a text-based code implementation of “DeadMan’s Draw”. The contents of this specific document will primarily consist of design artefacts, as well as justifications for any major design decisions, and omissions (should they exist).

Task 1 – Class Level Design
Artefact #1 – System Decomposition



Starting out, the above artefact is a simple layered system decomposition of the topical system design. It naturally follows the logical downwards flow below:

The Client Layer: Responsible for handling the actual player’s decision-making input.

The Manager Layer: As to not be confused with the client, this layer facilitates the gameplay flow and commands. The player isn’t handling actually running the game on the console from the client layer, that’s what this layer is for. This layer is not the decision maker, that’s of course, relegated to the client.

Engine Layer: This is where we handle core domain logic (e.g., card behaviour, player data management and game state to name a few. Effectively, defining what the moving pieces like, card’s and abilities, are capable of at a given point in time).

Access Layer: Finally, as things are, this is the final piece, here to provide essential constants (the Suit enum, for all of the different suits).

The core of this layered architecture is that higher level components do not depend on the implementation of lower levels of abstraction. For example, the Player Client component (client side) can operate by merely requesting player decisions, without needing to know about the inner workings of how the Game Engine resolves state changes like card drawing or bust detection. Similarly, the Game Manager component coordinates player actions without needing intimate knowledge of the individual mechanics of a Card or Ability.

This separation means that if we were to modify internal logic (for example, introduce new card abilities or change player storage structures), no changes would be required at the Client or Manager layers. This structure enforces loose coupling, single responsibility, and keeps the system highly extensible for future adaptations — such as automated CPU players (as noted in artefact #2), additional card types, or even expanded rulesets.

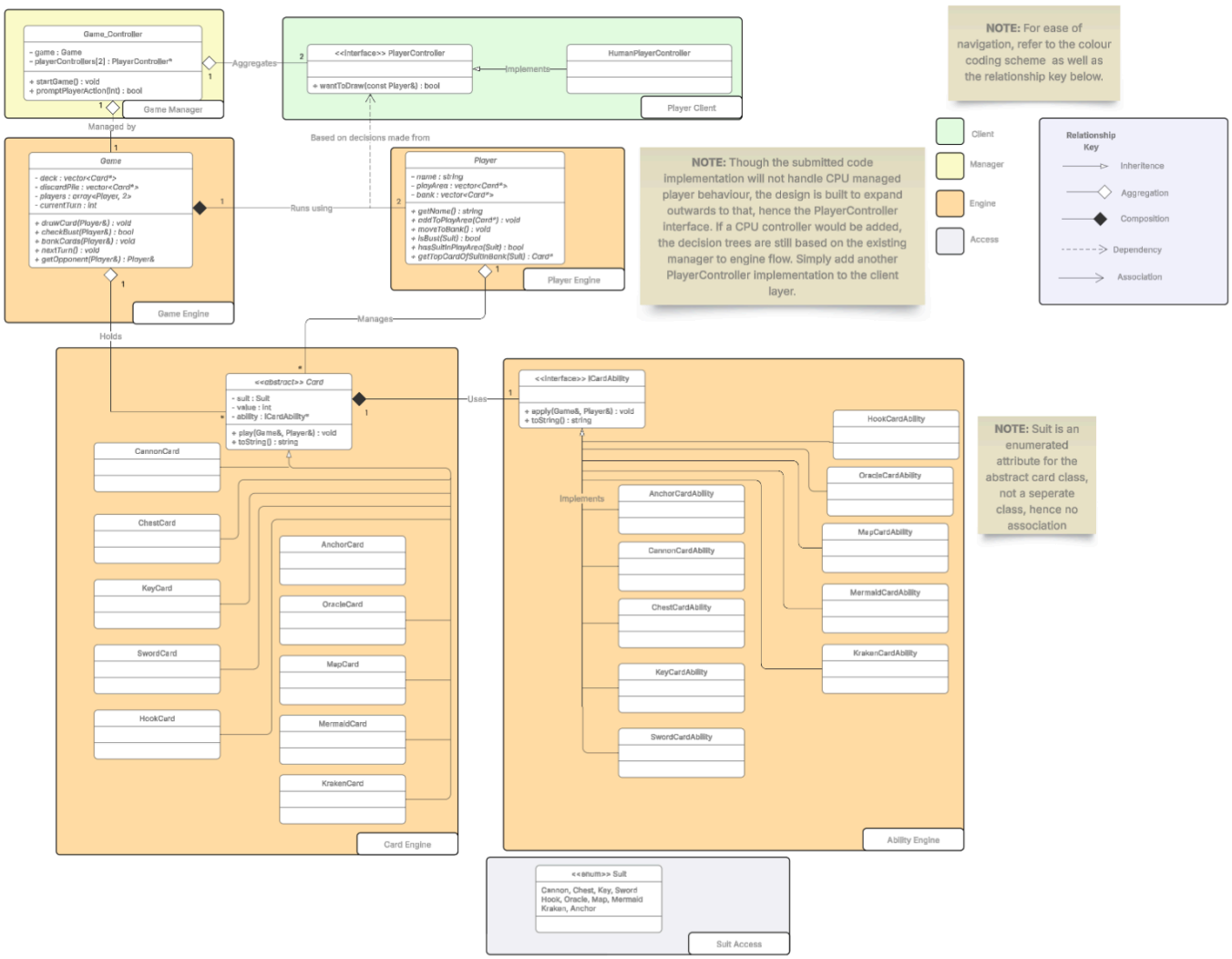
Decomposition & Responsibility Map
Following along from the overview of the architecture on the page prior, is the given decomposition and responsibility map below:

Component	Layer	Responsibility
Player Client	Client	The Player Client's primary role is to provide player decision input (e.g., draw another card or bank cards) through PlayerController implementations (See Artefact #2). It does not handle any core gameplay logic itself.
Game Manager	Manager	The Game Manager coordinates the game flow, prompting players for decisions, and invoking engine methods to modify the game state. Note that it does not directly manipulate game data.
Game Engine	Engine	The Game Engine manages core game mechanics, including the deck, discard pile, drawing cards, progressing turns, and bust checking. It ensures that all player actions have the expected game-wide consequences.
Player Engine	Engine	The Player Engine manages per-player state, such as a player's bank, play area, and any game-specific conditions (e.g., suit bust detection). It acts under the direction of the Game Manager but maintains its own internal logic.
Card Engine	Engine	The Card Engine governs card-specific behaviour through Card implementations. Note that when a card is played, it may activate an ability. How that functions, however, is delegated to the Ability Engine to avoid bloating the card management responsibilities.
Ability Engine	Engine	The Ability Engine handles different card effects via ICardAbility implementations.
Suit Access	Access	Suit Access provides an enumerated list of suits usable across the engines.

NOTE: See Artefact #2 for references to specific classes

Key Dependency Decisions:
A major goal within the system design, beyond what has already been discussed was minimising coupling between engines. For this, do note that engines such as the Card Engine and the Ability Engine interact only via simple method calls and interfaces, with the intentional choice of deciding against hard composition on the concrete implementations. Controllers, such as the Game Manager, are responsible for orchestrating player actions but deliberately avoid owning engine behaviours. This together, naturally allows for flexibility whilst also preserving a clean separation of concerns across the system. On the client side, input is entirely isolated from business logic; through the Player Client merely requesting actions to be taken, without directly manipulating game state or underlying data structures. Though already implied, the design also strongly favours the use of interfaces and abstractions. For instance, we can take the PlayerController interface. This exists as an interface over a sole implementation, to ensure that future extensions can be made without requiring modification of existing core classes. (As, functionality should have automatic compatibility with concrete implementations based on the interface) Finally, the overall architecture enforces the strict layered structure identified above: higher layers such as the Client and Manager layers are prevented from depending on the internal details of lower layers like the Engine or Access layers. Having this layered structure in place, preserves a clean top-down control flow, poising the system to be in alignment with SOLID principles, with the byproduct of ensuring it remains modular, extensible, and maintainable.

Artefact #2 – Class Diagram



Class Level Design Assumptions & Justifications:

The above class diagram was built on the assumption that current specifications only require supporting player-driven gameplay and core card abilities. Therefore, I have decided to omit any unneeded complexity in the form of say CPU player logic. I still decided, however, that it was worthwhile designing for extension throughout the system as a whole, in the event additional ideas or perhaps a change in requirements crops up during development. Then, trailing back to the point on player-driven gameplay, we actually have a PlayerController interface, which exists even though only a HumanPlayerController is initially implemented. This of course, allows the seamless integration of the potential CPU-controlled players discussed before. In a similar vein, the Card class is abstracted with a CardAbility interface relationship, so that new cards or special abilities can be introduced later on without altering existing code. Between many major components (e.g., GameController to PlayerController array) I've also opted for an aggregation over hard composition to maintain low coupling, and to as well, preserve modularity while simplifying dependency management. (Less of these hard implementation relationships nets more leeway with what can be done which each individual relation, as they aren't directly impacting other relations that depend on them whenever manipulation occurs.)

Design Pattern Discussion
Moving along, a look at how player decision-making and card abilities have been handled, reveals a portion of my approach on design patterns throughout the architecture. In essence, the strategy pattern is present within both the player interaction flow and card behaviour systems. Each card ability is effectively a pluggable strategy for modifying game state, depending on the card played. Another thing to keep in mind is that, while factory patterns could be introduced later for dynamically constructing players or cards, the current scope did not warrant that extra abstraction where simplicity within an already robust and modular design (driven by requirements) was prioritised.

SOLID Principles
As alluded to throughout this document, the current working architecture also aligns closely with SOLID principles. Single Responsibility is adhered to by carefully restricting each class to a narrow domain. For instance, the GameController manages game flow but not player data storage, while the PlayerEngine manages player state without coordinating the game loop. Open/Closed is respected by the use of abstract interfaces like PlayerController and ICardAbility, allowing easy extensions without modifying base code. Liskov Substitution is naturally maintained, as all subclasses correctly fulfil the contracts defined by their base types or interfaces, avoiding side effects when a new implementation faces the rest of the system. Interface Segregation is evidenced by granular interface design: player control and card abilities are split rather than merged into colossally bloated base classes. Finally, Dependency Inversion is achieved through in that high-level modules like the GameManager are designed against interfaces rather than concrete implementations, making future maintenance and evolution significantly easier. The resulting architecture preserves modularity, enhances flexibility, and maintains long-term clarity.