# Assignment 2 Report: *Counting Up*, ABC Game$^{\text{TM}}$

**SWEN30006 SEMESTER 2 2023**

Evan Liapakis, Toby Guan, Claire Tosolini

October 22, 2023

## Contents

## 1 Introduction

The objective of this report is to outline the process undertaken to develop an improved version of the card game ***Counting Up***. This involved a full analysis and a comprehensive understanding of the requests and requirements outlined by ABC Games™, then revising and extending upon the existing implementations. Static and dynamic models were created to clearly outline our proposed implementation, and decisions are explained with regards to GRASP principles and Gang of Four (GoF) Design Patterns.

## 2 Requirements Analysis

Our primary focus was to understand the business requirements of ABC Game™. This was accomplished by first conducting an in-depth review of the current outlined rules, followed by an extraction of relevant domain components. From this we constructed a partial domain model (1). We intentionally kept this simple to facilitate a clearer understanding and also took note of ABC Game™'s current development status, including their request for revisions and an extension of the existing design.

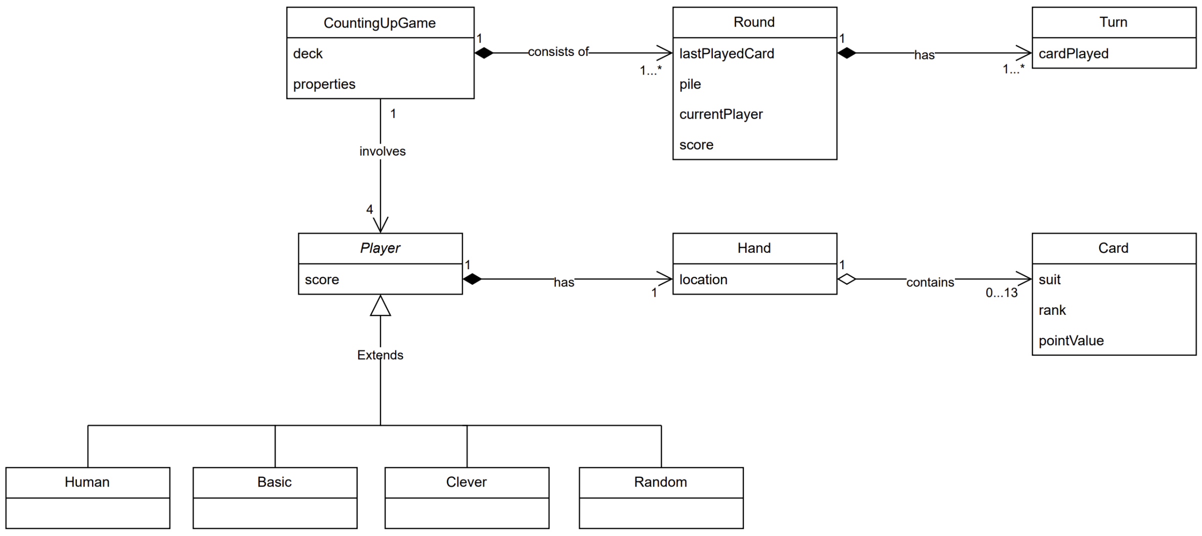

Figure 1: Primary Partial Domain Model

## 3 Current Implementation Analysis

We undertook an analysis of the existing version and identified weaknesses with respect to GRASP principles. Since ABC Game™ did not provide any diagrams, all observations were based solely on the codebase.

### 3.1 General Structure

The exitsing implementation consisted of only one extremely large class. This class alone contained all attributes, methods, information, and responsibilities related to the game. By definition, having a single class strongly violated the principle of High Cohesion. The *CountingUpGame* class contained all information required to execute the game, and must also itself undertake all execution. The focus of the class was unclear, and resulted in code that was difficult to understand, manage, and extend upon.

The existence of only one class meant other GRASP principles were not relevant to discuss, as they were technically not violated. However, note that this is a technicality and should not be considered a positive reflection on the design and framework; it is due to the fact that this program has been designed in clear infraction of a number of primary principles of Object-Oriented Programming, namely: abstraction, modularity, and delegation.

# 4 Proposed Design Changes

In this section, we outline the changes made to the current software design. Through thorough and systematic analysis of all provided information, as well as our own preliminary models, we aimed to produce a solution that would satisfy both the business requirements and good software design practices. Each design change is justified and discussed mainly with regards to GRASP principles and Gang of Four Design Patterns (see Appendix A for more information), however general object-oriented principles are also referenced.

## 4.1 Classes Extracted from the Problem Domain and Business Requirements

Our approach was to build and extend upon our partial Domain model (1) produced in the Requirements Analysis. After some discussion, we elected to only retain a select few of the classes originally extracted, given the context of the full design and our aims.

### 4.1.1 *Round*

The *Round* class primarily supports the principle of High Cohesion through being delegated the tasks and data within *CountingUpGame* that related to the cards played. This gave both classes clearer, more specific responsibility. The decision for *CountingUpGame* to instantiate *Round* objects is justified by Creator, since *CountingUpGame* contains all of the initialising information for *Round*.

### 4.1.2 *Player*

The abstract *Player* class was also preserved, though the design details differed slightly. The Design Pattern Template was applied here, with two concrete subclasses extending *Player*: *HumanPlayer* and *ComputerPlayer*. This decision is supported by the principle of Polymorphism, as *Player* objects are able to behave differently depending on their subclass type. Additionally, this design supported the implementation of only *HumanPlayer* objects responding to keyboard input. Furthermore, this design stands strong with reference to the principle of Low Coupling, as interactions between *Player* instances and other classes are independent of the inner details of *Player*.

### 4.1.3 *Hand* and *Card*

We retained both classes, and used the implementations provided in the JCardGame package. This decision supported High Cohesion through containment of specific tasks and data; the delegation of these responsibilities consequently improved the cohesion of Round as well.

In order to represent the finite set of cards in a deck, Enumerations were used to capture the permitted suits (*Suit*), ranks (*Rank*) and card values in respect to different ranks, according to the specifications. This allowed *Card* objects to be created with the knowledge that their attributes would always be within the set of pre-determined values, giving the implementation consistency and readability.

## 4.2 Fabricated Classes

### 4.2.1 *GameManager*

Initially, by Information Expert, we assigned the responsibility of validity checking cards to *Round*, since it had access to all the cards that had been played (pile). However, this reduced the Cohesion (High Cohesion) of *Round*, and associations with *Player*, *CountingUpGame*, and potentially more classes would have also been required, violating Low Coupling.

In order to avoid these issues, we applied Pure Fabrication to create the *GameManager* class. It was then assigned all responsibilities related to Player turns (see DSDs); most critically, validity checking, achieved via storing the last played card (i.e., at the top of the pile) for comparison to the next requested move.

Another design problem we encountered was other classes (namely *Round*, *Logger*, and *CleverComputer*) needing to know when a new card was played, and what that card was. Since *GameManager* now stored the lastPlayedCard, a potential solution in prior to our current design was for these classes to check whether the value of the attribute had changed. However, not only would this be extremely inefficient, it would result in strong violation of Low Coupling as each of the classes is now dependent on the attribute. Instead, we utilised the Observer Design Pattern, where classes who want to be notified when a card is played implement a common interface, *PlaySubscriber*, supporting Polymorphism as they are capable of reacting differently to the same method call by *GameManager*.

#### 4.2.2 *DisplayManager*

The game involves rendering information to the screen, which would require many different classes having to handle display responsibilities, violating High Cohesion. We wanted to design the system such that these tasks are handled independently from the classes that contain the information.

To achieve this, we first applied the principle of Pure Fabrication to create the *DisplayManager* class. In this way, the game system itself is entirely detached from how it is relayed to the screen, and classes do not have to be concerned with how their data is represented, supporting Low Coupling.

#### 4.2.3 *Logger*

The existing implementation involved logging and storing much of the runtime game information. To preserve this functionality, the *Logger* class was created using Pure Fabrication, and subscribes to GameManager through implementing the PlaySubscriber interface. This allowed us to maintain the original data logging structure in a way that promoted Low Coupling, as it is not directly associated with any classes, and High Cohesion, as it encapsulates all related logic into one focused class.

### 4.3 *Player* Instantiation

The responsibility of instantiating *Player* objects was assigned to the class *PlayerFactory*, a class that was created by applying Pure Fabrication. The functionality of this class utilises the Design Pattern Factory, specifically Concrete Factory. Through removing the responsibility from *CountingUpGame* and reassigning it to *PlayerFactory*, High Cohesion is supported as the focus of each class is better defined. Additionally, the design promotes Low Coupling by encapsulating potentially complex creation logic; *CountingUpGame* does not need to know the inner details of *Player* objects.

### 4.4 *ComputerStrategy*

In order to accommodate different Player types as per the additional features request, we first determined that what differed between each was the 'strategy' i.e., algorithm, they employed to choose what card to play. We then identified this to be conducive with the Strategy Design Pattern. Since we had already defined the Human and Computer players as separate classes, this strategy would only apply to the latter. This design hinges on the principle of Polymorphism, as the various computer strategies (basic, random, clever) can be executed using their common interface. The algorithm is removed and the ComputerPlayer does not need to be concerned with the actual algorithmic logic, supporting Low Coupling; additionally, this separation and delegation inevitably creates High Cohesion within each class that implements the interface.

### 4.5 Changes to *CountingUpGame*

Having delegated different responsibilities to newly created classes by Pure Fabrication as discussed above, the *CountingUpGame* was now solely focused on the game's core logic flow, showcasing the benefits of High Cohesion. By deploying the Singleton Design Pattern, we could guarantee that only one globally accessible instance of *CountingUpGame* existed at a given time. This enabled a consistent configuration for the game and prevented conflicting initialisations.

## 5 Domain Model

Following the changes outlined in Section 4, we reconstructed our partial domain model (1) to more accurately reflect the design. This model does not include GRASP or implementation details.

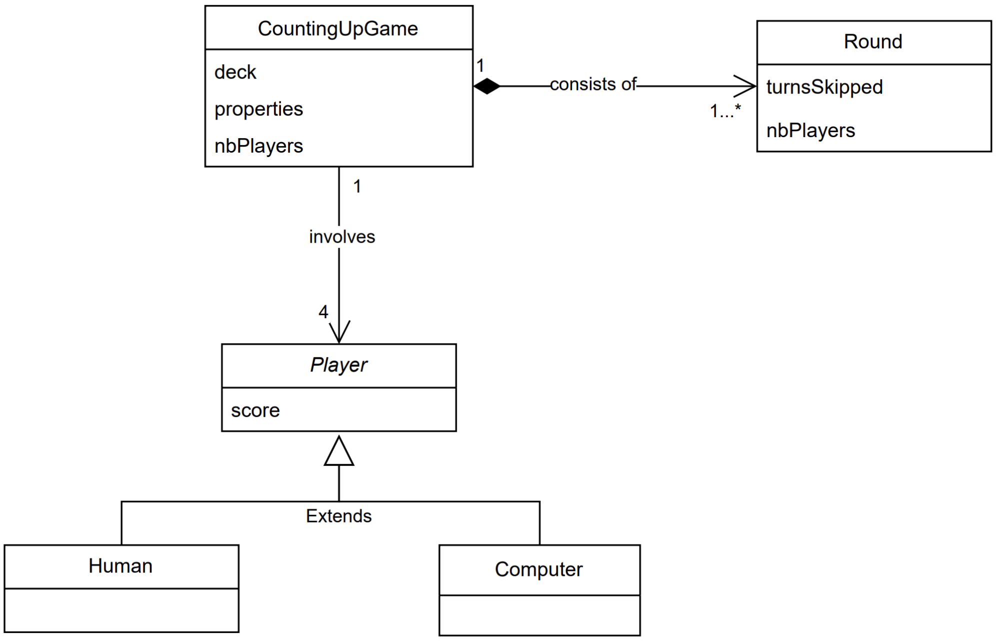

Figure 2: Updated Domain Model

## 6 Design Class Diagram

The diagram below provides a more detailed and thorough depiction of our proposed implementation, including all applied GRASP principles, design patterns, and logic.

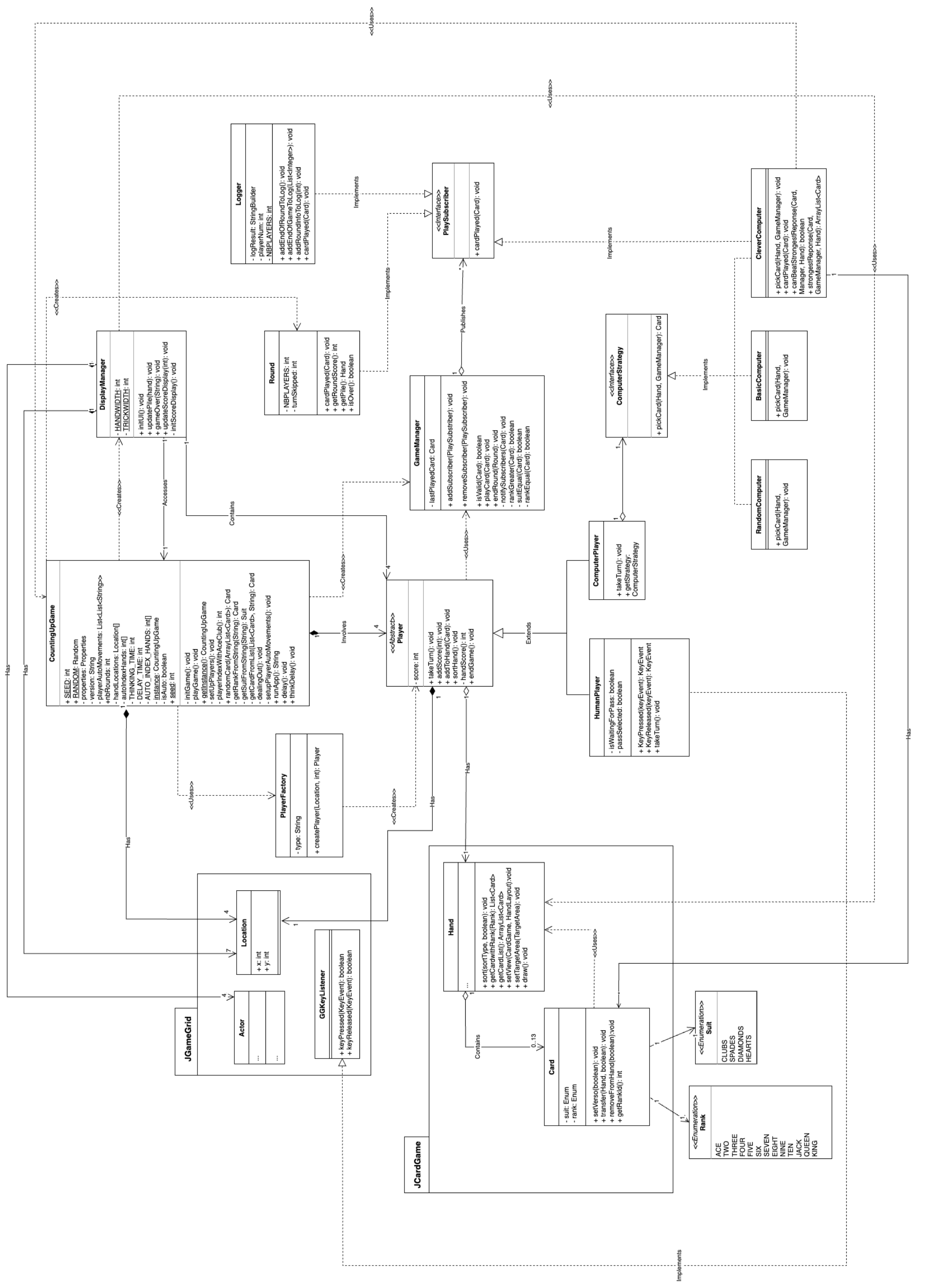

Figure 3: Design Class Diagram

## 7 Design Sequence Diagrams (DSD)

After gaining a clearer understanding by laying out the different classes and familiarising ourselves with the JGameGrid and JCardGame packages, we identified the following significant system events, and captured them in design sequence diagrams below.

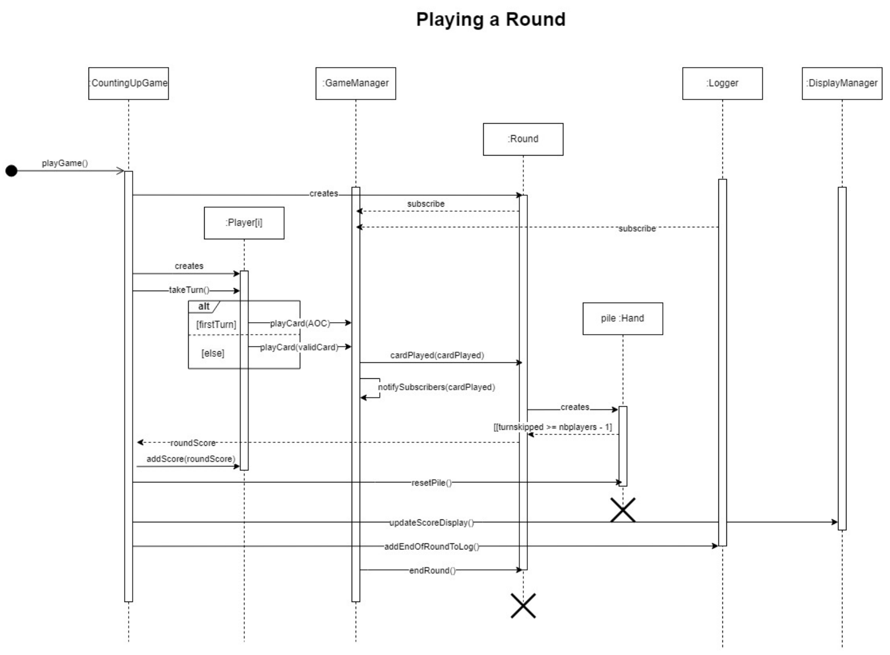

Figure 4: DSD: Playing a Round

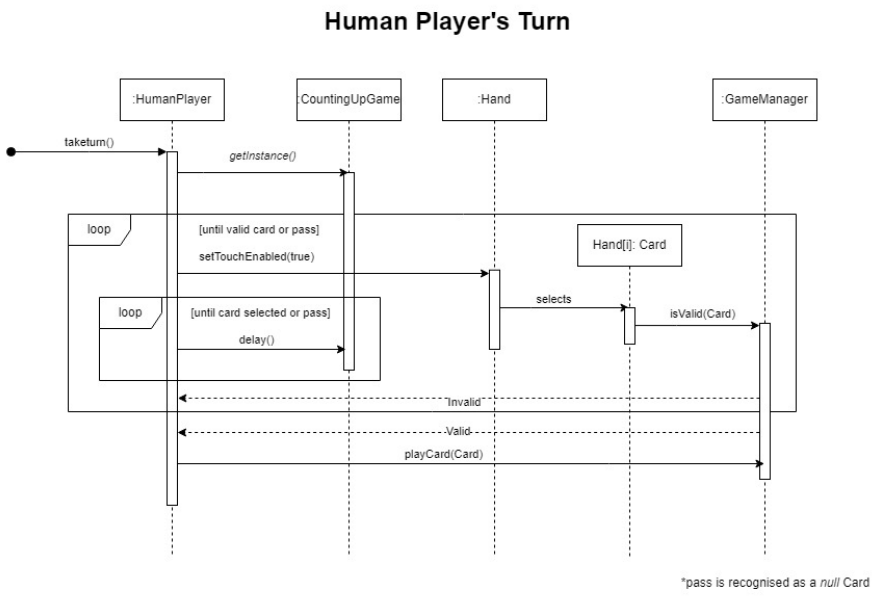

Figure 5: DSD: Human Player's Turn

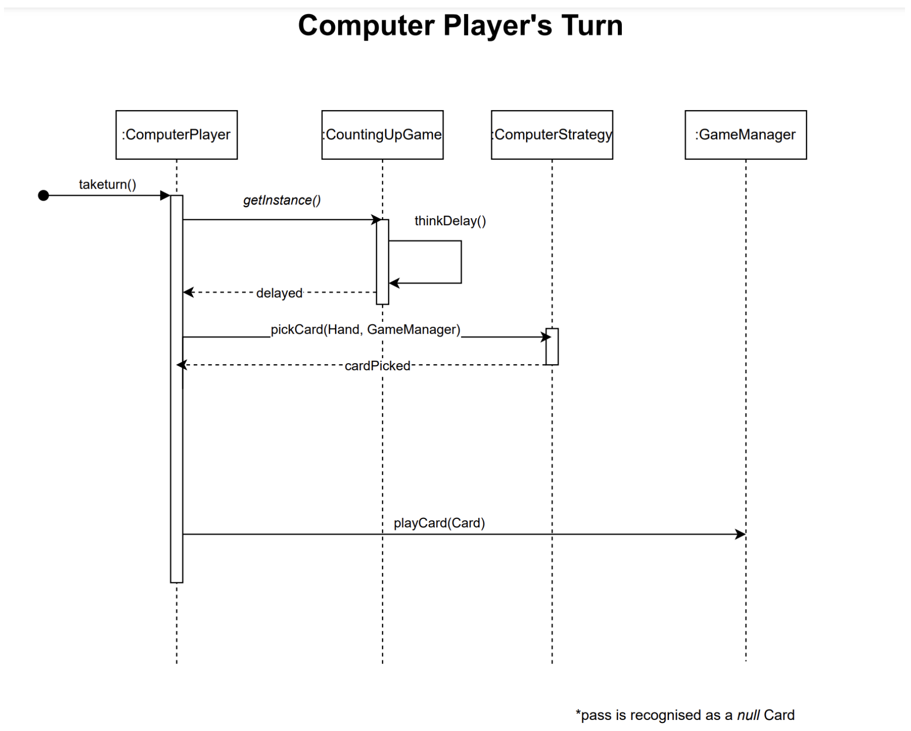

Figure 6: DSD: Computer Player's Turn

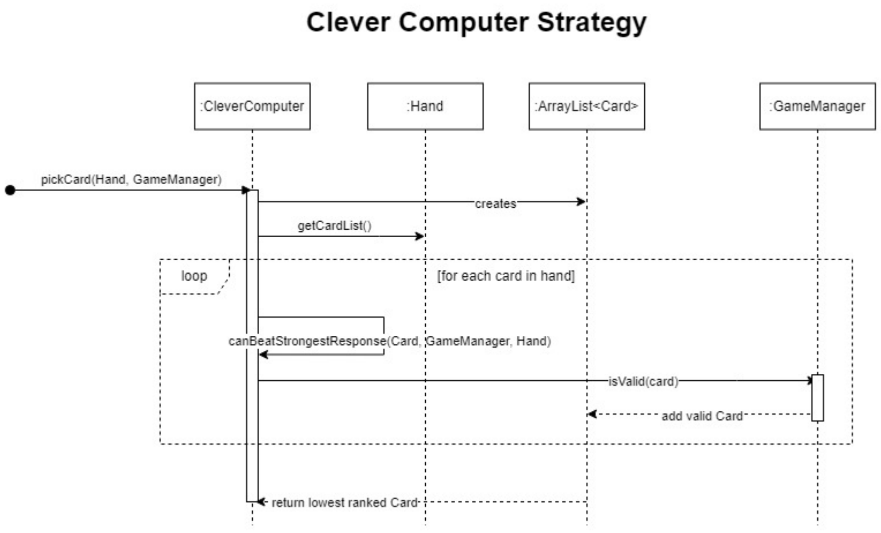

Figure 7: DSD: Clever Computer Strategy

# 8 Implementation Logic

## 8.1 *CleverComputer* Strategy Algorithm

The implementation of the *Clever Computer* strategy aims to maximise its own score while minimising the score of its opponents. The way it achieves this is by using the knowledge of all cards that have been played and trying to anticipate potential opponent moves to play cards only in situations where it believes it can win. The *Clever Computer* strategy keeps track of all cards that have not been played yet, and then for every card it could play for each turn, it looks for potential responses from opponents by looking at which cards have not been played yet (i.e. are in all opponents' hands). The strategy then looks for the strongest of all of these responses, only considering a card to be a good move if it can beat the strongest response to that card with another card in their hand. It then picks the lowest ranked card out of this list of cards. The reason it plays this way is because if a player plays a card but does not win the round, it is effectively giving another player free points. By withholding cards and only playing when it knows it can win, the *Clever Computer* strategy is able to maximise its own points by baiting other players to add more points to the round, while minimising its opponents' points by not giving away cards when it is impossible to win the round.

The *Clever Computer* strategy is demonstrably smarter than the *Random* and *Basic* strategy as in all games that we have tested, the *Clever* strategy beats both other computer strategies 100% of the time, and beats a human player around 50% of the time, often losing only when purposefully exploiting the strategy.

# List of Figures

# A Appendix: Definitions of Referenced Terms

## A.1 General Responsibility Assignment Software Principles (GRASP)

### A.1.1 Creator

The Creator principle is applied when determining who should be responsible for instantiating a class. This responsibility of creating an instance class A can be assigned to class B if any of the following criteria is true, in order of priority:

1. B contains or compositely aggregates A.
2. B records A.
3. B closely uses A.
4. B has the initialising data for A.

### A.1.2 High Cohesion

The premise of High Cohesion is to design objects that have clear and focused responsibility. Highly cohesive design produces manageable, comprehensible and extensible code. Classes that do too much work, or have unrelated responsibilities and data, are undesirable and violate this principle.

### A.1.3 Information Expert

The Information Expert principle relates to determining which class should be assigned a certain responsibility. A responsibility may be assigned to a class if that class contains, or has access to, the information required to fulfil it.

### A.1.4 Low Coupling

Coupling refers to the strength of the connection between classes. This connection can refer to how much knowledge, dependency or reliance a class has on other classes. The premise of Low Coupling is to create independent, self-contained classes to produce a system where changes to one class have little impact on the system as a whole.

### A.1.5 Polymorphism

Polymorphism provides the ability to vary the behaviour of objects and systems based on their type, in response to input conditions.

### A.1.6 Pure Fabrication

The premise of Pure Fabrication is applied when there is no class that can be assigned a certain responsibility such that the principles of high cohesion and low coupling are not violated. The solution involves creating an artificial class, i.e., one did not exist in the problem domain.

## A.2 Gang of Four Design Patterns

### A.2.1 Factory

This design pattern addresses the problem of object creation responsibility by encapsulating all the related process. More specifically, the Concrete Factory implementation involves fabricating a new class, called a Factory, for this purpose, and using pure fabrication to delegate potentially complex logic to a highly cohesive class.

### A.2.2 Observer

The Observer pattern is used to achieve low coupling and low dependencies between classes. The design involves observers subscribing to the subject in order to be automatically notified when its state changes. Each observer can then respond uniquely to this change (polymorphism). This avoids observers needing to check directly and repeatedly for state changes, and only run responding events when required.

### A.2.3 Singleton

The Singleton pattern is applied when the requirements of a class within a system determine the following rules should be enforced: exactly one instance of this class should exist at any time, and this instance should be globally accessible.

### A.2.4 Strategy

The Strategy design pattern defines how to design for the ability to switch between algorithmic implementations. This is achieved through separating said algorithms from their use into their own classes, and having them share a common interface.

### A.2.5 Template

This design pattern provides the framework of a superclass through defining an algorithm as a set of steps. The premise is to allow subclasses to override parts and provide their own specific implementation without changing the general structure.