# **Design Document - Overview**

## **Project Background and Description**

### **Chosen Game: War**

War is traditionally played between two players, but can be expanded for up to four players.

Our vision is as follows:

The player faces off against one to three computer-controlled opponents. The deck is split as evenly as possible, with each player receiving a specific number of face-down cards. Each side flips the top card of their deck, and whichever player has the higher card wins. If the highest cards match, then the players with the highest cards flip their next cards; repeat until one card is higher, in which case the winner takes all flipped cards. The game ends when one person is left with every card, making them the winner.

Highest Card: Ace

Lowest Card: 2

Face/Court cards rank below Ace (King = 13, Queen = 12, Jack = 11)

The start of the game will take user inputs for the following:

* The number of players (two to four)
  + If player count = 1, then the computer-controlled opponent becomes Player 2
  + The number of cards in each player’s hand is dependent on the number of players:
    - Two players = 26
    - Three players = 17
    - Four players = 13
* Each player’s name
  + Computer-controlled opponents are still named by the player

At the start of a round, any player may choose to:

1. Flip a card; this happens for all active players simultaneously. If trying to beat matching cards, then only players who had matching cards during this round will flip again.
2. Check round count and player card totals. This information is printed as a text block.
3. Quit game. This aborts the program and closes everything down.

### 

### **Starting Code Structure**

This is a Java project where we will build on starting template code. This code includes the following classes, variables or arrays, constructors, and methods:

**Class: Card.java**

This abstract class defines a card object. It has the following features:

1. Public abstract toString() method, which returns the card’s value as a String

**Class: Game.java**

This abstract class models the game’s basic functionality. It has the following features:

1. Private String variable name: This is the name of the game, which is final (immutable)
2. Private ArrayList players: This holds each player
3. Constructor for the Game object, which includes the name variable and players ArrayList
4. Public Getter methods to return the name variable and players ArrayList
5. Public Setter method for setting the Game object’s player list
6. Public abstract methods play()and declareWinner(), used to play the game and declare the winner

**Class: GroupOfCards.java**

This concrete class represents any grouping of cards for a game. It has the following features:

1. Private ArrayList cards: This stores the group of cards
2. Private int size: This sets the size of the group as an integer
3. Constructor for the GroupOfCards object
4. Public Getter method for fetching the cards ArrayList
5. Private method to shuffle the cards
6. Public Getter and Setter methods to fetch or store the size variable

**Class: Player.java**

This abstract class creates a player object. It has the following features:

1. Private String variable name: This is the Player’s name, meant to be used as a unique identifier for that player
2. Public Getter and Setter methods for fetching or storing the name variable
3. Public abstract play() method, which can be overridden when/if we need to subclass this later

## **Project Scope**

### **Team Composition and Expectations**

As a group, we have committed to working on project tasks together, rather than assigning specific roles to each member (except for Slate submissions). This will ensure that we can oversee and collaborate on each other’s work in real time, making it easier for us to understand what we are doing and why we are doing it.

Team members are as follows:

* Arthur Sharipov (leader: will submit through Slate)
* Sean Sumangil
* Andrew Elliott

The goal will be to arrange a meeting at least once per week and decide on what needs to be done before we can proceed with other tasks. While we are not delegating specific roles to each member, we may still decide that a given person might take the lead on given tasks (e.g. if we decide that one member should prioritise work on a Card subclass and another should prioritise work on a Player subclass). This will be determined on a case-by-case basis.

### **Technical Scope & End Goals**

The program is a success if it can do the following:

1. Give every player (2-4 players total) an even number of unique cards; there is only one deck, so only one player can have a given card at any time.
2. Execute each round, one at a time. In the event of a matching pair of cards, the round will continue until only one player wins all played cards.
   1. Players that did not flip a matching card will not flip again until the round is over.
3. Check each player’s card count at the end of every round, eliminating players from the game when they run out.
   1. When only one player has cards, the game will end and the winner will be announced.
4. When the game is over, the player(s) may choose to continue onto a new game with the same players; total wins will be tracked and tied to each player.

## **High-Level Requirements**

|  | *Ability for each player to register with a unique, unrepeated name*  *Ability for player to choose the number of opponents*  *Ability for the game to communicate a win or loss*  *Ability for player to know how many cards each player has (i.e. their score)*  *Ability for the game to handle a “war” (when two or more players have the same card)*  *Ability to continue playing with the same players*  *Ability for game to shuffle deck between games* |
| --- | --- |

## **Implementation Plan**

### **GitHub**

TheTitansDeliverable (GitHub Repository): <https://github.com/Sharipov-dev/TheTitansDeliverable>

We will be meeting once per week to decide what needs to be done, aiming to push code changes within the week between meetings. In all likelihood, this could mean that we meet on a specific day (e.g. Wednesday), then work on code over the weekend and push on Sunday. The specific days are not set in stone, but the process will be consistent throughout this project’s life.

In addition to the main sections used by the project (e.g. /src/ca/sheridancollege/project/ for code), we will also have the following folders for storing supplementary documentation:

* text-files
  + Including the Class Contract and this Design Document
* diagrams
  + Including the UML Class Diagram and others, as needed

### **Tools**

We will be utilising the following tools, in addition to GitHub:

* Apache NetBeans IDE 17
  + For coding
* Visual Paradigm (17.2)
  + Used to create UML diagrams as needed
* JUnit
  + For testing
* Spring
  + For performing different processes

## **Design Considerations**

### **Encapsulation**

* + Card class will have the internal properties such as the suit and number of cards provided access through a toString method ensures that the card internal state remains hidden while allowing the program to interact with its value controlled through the controlled interface.
  + GroupOfCards class stores a private arrayList that provides getters and setters. We will also have the shuffle method private, Means the shuffling logic will be hidden and only accessible internally by this class; this encapsulation makes sure other classes don't interfere with the deck's integrity.

### **Delegation**

* + We will be using one of the SOLID principles which is Open for Extension, Closed for Modification. In the future we will add some new methods or variables inside of a class, so we should do it by creating a new class and extending it from the common one.For example, In our game, we could add new game rules (such as "war" on a draw) or new functionality (such as a scoring system) without changing the core logic, leaving the existing code unchanged. We expand the game with new components or methods, rather than modifying old ones.
  + We can have different types of players: human and computer. Both players must be interchangeable during the game, which means they must have the same interfaces and methods. This control governs player control based on abstractions, making the system flexible for expansion. So we would need to have a common abstract class which is Player, and then extend from it classes like: HumanPlayer, AIPlayer.

### **Flexibility/Maintainability**

* + In the future we may add factory patterns which will help us easily create such objects as card decks, players or even AI opponents. It would help us not to decrease the boilerplate code of creating each object, so it will be easier to maintain this type of code.
  + We can also add an observer pattern which would help us to notify players or other services that player won the round or the game. For instance, In the case of winning, there would be a message printed to a console: “Congratulations, you won!”
  + For the sake of logic of our application, we could divide it into states like: “Beginning of the game”, “Round is happening”, “War”, “The end of the game”, so it would be better to support, as the part of the logic will be divided. So using State Pattern, we can create separate methods or classes for each of the states, which is easy to maintain.