Troy Johnson
Saima Ishaq
Tyler W Belair

Table of Contents

Overview

Project Description and Scope

The card game chosen is War, a replica of Bicycle, a similar card game. The key objective to War is to “win” all of the cards. This is done by dividing the deck evenly between the players (2 at the max) and placing all 26 cards face down (nuance doesn’t matter for coding practices), every turn each player draws 1 card and places it “face up” in front of them, the player with the highest card (ranked/unranked) wins both cards. After a turn is concluded the player that is victorious puts both cards at the bottom of their library, the game continues until either player has collected all 52 cards.

The game will continue until 1 player (or computer controller) has acquired all 52 cards. This is done by continuously playing out the game (aka a round) until either the player’s “container” has all 52 cards, these containers will be built using a dynamic array list so as to allow the game to either add or remove contents specific to the card won or lost. Since this game is modeled in such a way, there will be no such thing as a tie, nor will there be a time limit.

However to delimit the potential for “extra long games” there will be an option for maximum potential rounds to be played out during the life of the game so as to give the user the satisfaction of winning or the sorrow of losing, and to demonstrate proper coding standards and usable gameplay mechanics.

The total maximum for these rounds will be 10, allowing the opportunity to allow for competitive gameplay and chance mechanics. However should the user incline towards playing a full game, there is an option for “limitless” play that will enable the game to proceed until a winner is declared.

Project UML - Class Diagram (extended)

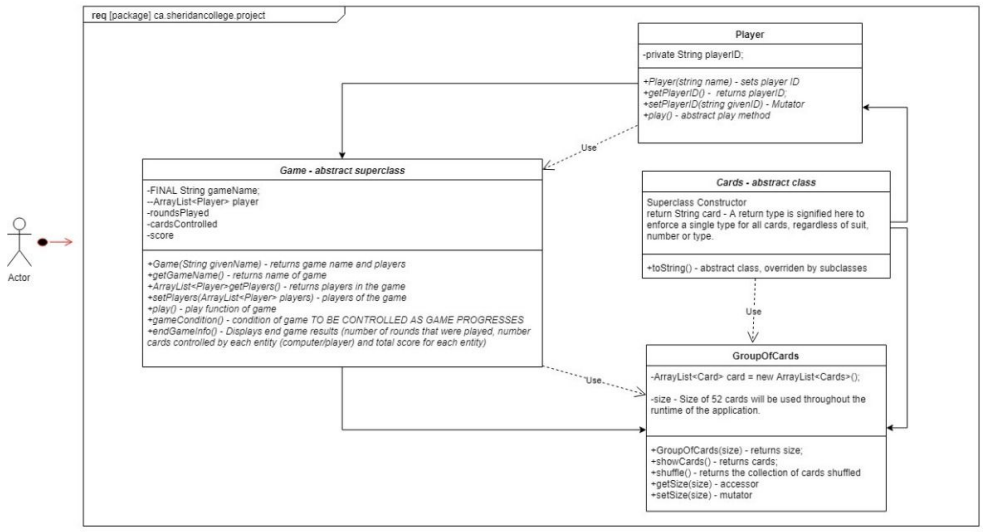

Figure 1A

The class diagram represented in Figure1 represents the totality and scope of our group project. Our game is known as “War’ and thusly does not have many rules and advocations regarding its design and functionality for application runtime.

Each class and data structure is dependant on each other for information received and exchanged during the runtime of the program as well as:

Game state changes
Information (or variable) changes
Error Catching and Exception Handling
User Input
Win/Loss conditions
Game functionality

Every byte of data involved in runtime will be handled in the GameEngine data structure.

Some of the return types involved in the game are :
String - for both the generation of the Cards and the Array List which will house the game objects creates as well as player name
Integer's - will keep track of player/computer score and rounds played/chosen

Design Considerations

Encapsulation:
Data handling is straight forward, each class (and subclass) will delegate private functions to written accessors and mutators. This allows each superclass the ability to draw data from handled sources only, and keep any non-essential data out of the scope of the game so as not to be manipulated out of chance.

Any invalid inputs from the user will result in error messages and/or pause gameplay.
Delegation:
The delegation of information is handled through appropriately named variables and associated accessors and mutators to handle data, memory and application logic.

Cohesion:
Each functioning data structure will have self-dependency provided the user has specified appropriate data types at given input intervals, otherwise data dependency is handled internally through accessors and mutators.

The card type is singular in nature, returning a string data type, to enforce strict code cohesion during application execution and application runtime.

Coupling:
The application itself enforces tight coupling in order to discourage any external data manipulation resolving in anomalous game play.

Inheritance:
The data structures all rely on inherited superclasses for data exchange and variable resolution for initial game play specifics.

Specifically once the player inputs their “name” as a string and the “number of rounds” as an integer type (no floating or double data types will throw a mismatch exception) the player class sends this information as privatized information which the GameEngine superclass will absorb and initialize the game with these variables in memory.

The Cards class enforces the data return type of each object within the ArrayList as a string data type, as our game does not utilize different game play for the suits or type of card, only the number of the card is considered.

The GroupOfCards class utilizes the information provided from the Cards class and creates two dynamic arrays that allocate the total deck of cards (52) amongst both the player and computer controlled entity, it then distributes this information to the GameEngine class in order to keep track of which entity controls which cards.

The GameEngine class will control all the necessary information regarding actual gameplay mechanics. From initiating a round (drawing cards), keeping track of which entity controls how many

Aggregation/Composition:
The project identifies as tight aggregation, as much of the data structures involved in the game must give and exchange information with each other in order to allow full functionality and smooth program flow.

Flexibility and Maintainability:
The code enables any coder the ability to modify, enhance and extrapolate their own designs and modifications at any given time to allow IDE or variable naming updates as well as to allow access to code modulation in order to conform to future coding standards.

Data Structures

Much of our project will utilize the dynamic ArrayList library which will allow us to customize how the data is created, manipulated, deleted and expanded upon in the coming weeks. Outside of the library functionality it will allow us to dynamically change and allocate memory according to the runtime of the application. This is done by manipulating the game object container while the application is being executed to store information about the game state.

Use Cases

As with a simple game comes simple use cases. This does not mean the code and project are not complex, but our written encapsulation allow for very few or no errors that are not immediately accounted for in the game logic.

Main Path

User runs application
User inputs valid round number
User inputs valid string name
GameState initializes
Game ends
Display score, number of rounds, total number of cards controlled by each entity

Alternate Path/Execution 1
Invalid type on Game round Input

User runs application
User is prompted to input integer type for number of rounds
User inputs incorrect type (anything but a flat integer)
System catches error
System outputs error message “INVALID TYPE MISMATCH - ERROR”
User is prompted to input correct integer type

Alternate Path/Execution 3
Invalid type on Player name Input

User runs application
User inputs valid integer type for round variable
User inputs invalid variable type for player name
System catches error
System outputs error message “INVALID TYPE MISMATCH - ERROR”
User is prompted to input correct type for name

Alternate Path/Execution 4
Game play prematurely interrupted

User runs application
User inputs valid round number
User inputs valid string for name
Game play initializes
Single round is played
User interrupts gameplay with escape option
Display score, rounds played, cards controlled and game condition

End