Troy Johnson
Tyler W Belair

Table of Contents

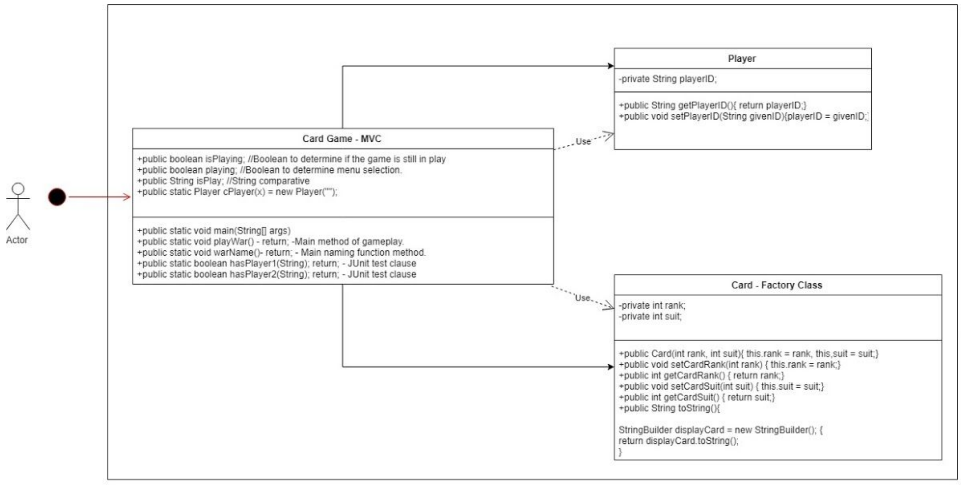
Overview

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 (by cardface NUMBER) 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 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.

Project UML - Class Diagram (Final)



Main Path/Execution 0

The user will begin the game following the program execution rules.

-User runs application
-User inputs Player 1 name
-User inputs Player 2 name
-User validates "game begin? y/n"
-User inputs "y"
-Game method executes
-Game is over
-Display winning party

Alternate Path 1/Execution 1

The user will begin the game following the program execution rules.

-User runs application
-User inputs Player 1 name (non-string)
-System throws new input mismatch argument
-SOUT("Please input valid string literal for player 1);
-User inputs Player 1 name (string)
-User inputs Player 2 name (string)
-User validates "game begin? y/n"
-Game method executes
-Game is over
-Display Winning Party

Alternate Path 2/Execution 2

The user will begin the game following the program execution rules.

-User runs application
-User inputs Player 1 name (string)
-User inputs Player 2 name (non-string)
-System throws new input mismatch argument
-SOUT("Please input valid string literal for player 2);
-User inputs Player 2 name (string)
-User validates "game begin? y/n"
-User input "y"
-Game method executes
-Game is over
-Display Winning Party

Alternate Path 3/Execution 3

The user will begin the game following the program execution rules.

-User runs application
-User inputs Player 1 name (string)
-User inputs Player 2 name (string)
-User validates "game begin? y/n"
-User inputs "n"
-Program exit

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(s)

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 the player class sends this information as privatized information which the CardGame class 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 CardGame class utilizes the information provided from the Card class and creates two dynamic arrays that allocate the total deck of cards (52) amongst both the player, it then distributes this information to the CardGame class in order to keep track of which entity controls which cards.

The CardGame class will control all the necessary information regarding actual gameplay mechanics.

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 Player 1 string name
User inputs valid Player 2 string name
System(“Are you ready to play? y/n”)
User inputs “y”
Application executions
GameState initializes
Game ends
Display winner
Program Exits

Alternate Path/Execution 1
Invalid type on Game round Input

User runs application
User inputs incorrect type for Player 1 name
System catches error
System outputs error message “INVALID TYPE MISMATCH - ERROR”
User is prompted to input correct Player 1 name
User inputs valid Player 1 Name
User inputs valid Player 2 Name
System(“Are you ready to play? y/n”)
User inputs “y”
GameState initializes
Game ends
Display winner
Program Exits

Alternate Path/Execution 3
Invalid type on Player name Input

User runs application
User inputs valid Player 1 name
User inputs invalid Player 2 name
System catches error
System outputs error message "INVALID TYPE MISMATCH - ERROR"
User is prompted to input correct type for name
User inputs valid Player 2 name
System("Are you ready to play? y/n")
User inputs "y"
GameState initializes
Game ends
Display winner
Program Exits

Alternate Path/Execution 4
Game play prematurely interrupted

User runs application
User inputs valid string for Player 1 name
User inputs valid string for Player 2 name
System("Are you ready to play? y/n")
User inputs "n"
Program exits

System Tests
Done with JUnit 4.12 in IntelliJ

Requirement	Use Case	Test Method (Class name, method name..)	Status (Date)
Has Player 1 been given an appropriate name	"Regular Play"	CardGameTest.has Player1()	Pass August 10th, 2019
Has Player 2 been given an appropriate name	"Regular Play	CardGameTest.has Player2()	Pass August 10th,2019

Manual System Tests

WarGame() - After the user has input 2 valid player names (or 1 valid player name in the case of a NPC player 2), simply run the warGame() method. It's complete execution results in a game conditional win for either Player 1, Player 2 or Computer.

WarName() - This method follows a strict rule set of string-literal inputs only, and will not accept anything other than aA-zZ characters. Proper functionality is contained within the method to check each string input against the regex .matches() built-in library.

To test, simply input a STRING LITERAL set of characters for Player 1/Player 2.
To test against, simply input anything OTHER than a STRING LITERAL, the resultant error should throw an Input Mismatch Exception and repeat the loop as necessary until the user either inputs correctly the player names, or exits the application.

GIT REPO URL : https://github.com/17796-GROUP4/SDF_WarProject-TermProject

GIT REPO URL - Test Source : https://github.com/17796-GROUP4/SDF_WarProject-TermProject/tree/master/src/SDF_WarProject

End