SYST 17796 TEAM PROJECT

Team Name: Sunday Afternoon

Deliverable 2

July 23, 2021

Group Members:

Jinyoung Jeon

Juyoung Jung

Tamara Dang

Winston Martinez

Contents

Use Case Diagrams and Narratives

Start Game

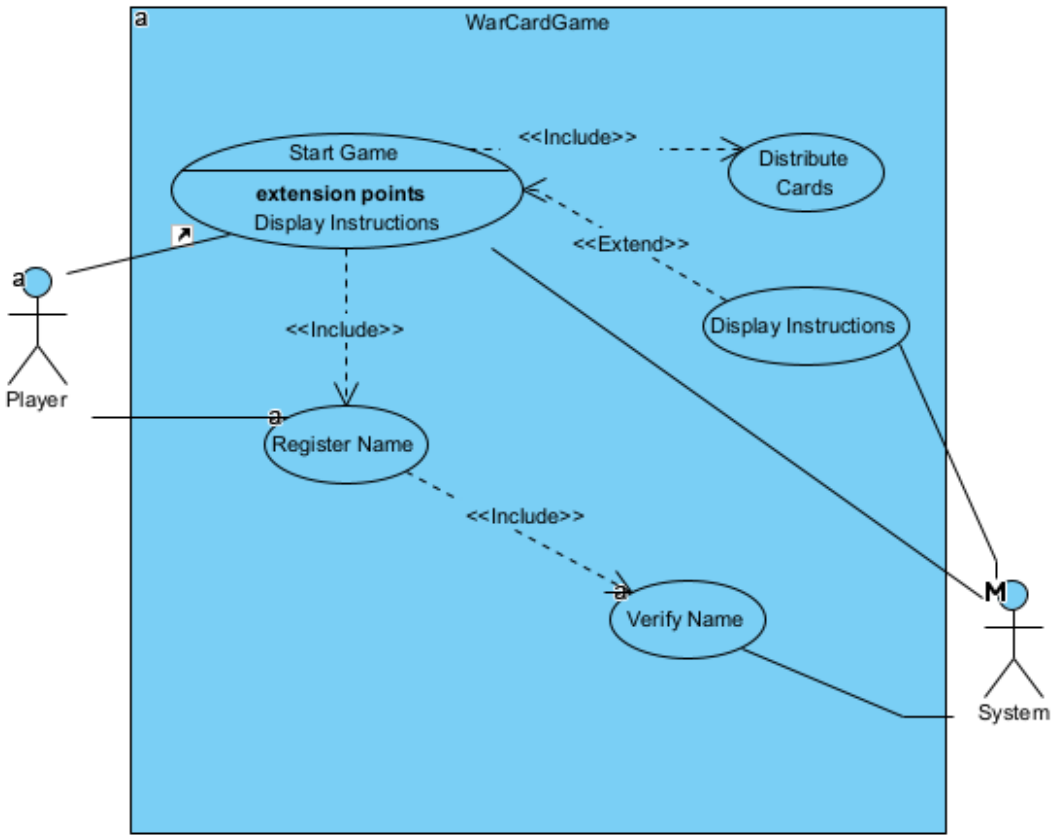

Figure 1. Start Game Use Case

1. Player wishes to start new game
2. SYSTEM Display start game introduction
3. if Player needs instructions
   3.1. Player selects display instructions option
   end if
4. Player selects start new game option
5. SYSTEM Prompt player for name
6. Player enters name
7. loop
   7.1. SYSTEM Verify Name
   7.2. if User enters an invalid name.
      7.2.1. SYSTEM asks player to reenter name
      end if
   until User enters valid name
8. SYSTEM Display verified name of player
9. SYSTEM Distribute cards to Players

10. SYSTEM Display action options
11. if Player selects display # of cards
    11.1. SYSTEM Display # of cards in win pile and in hand
    end if
12. if Player wishes to forfeit
    12.1. Player selects forfeit option
    12.2. Forfeit Game
    end if
13. if Player decides to play card
    13.1. Play Card
    end if

Play Game

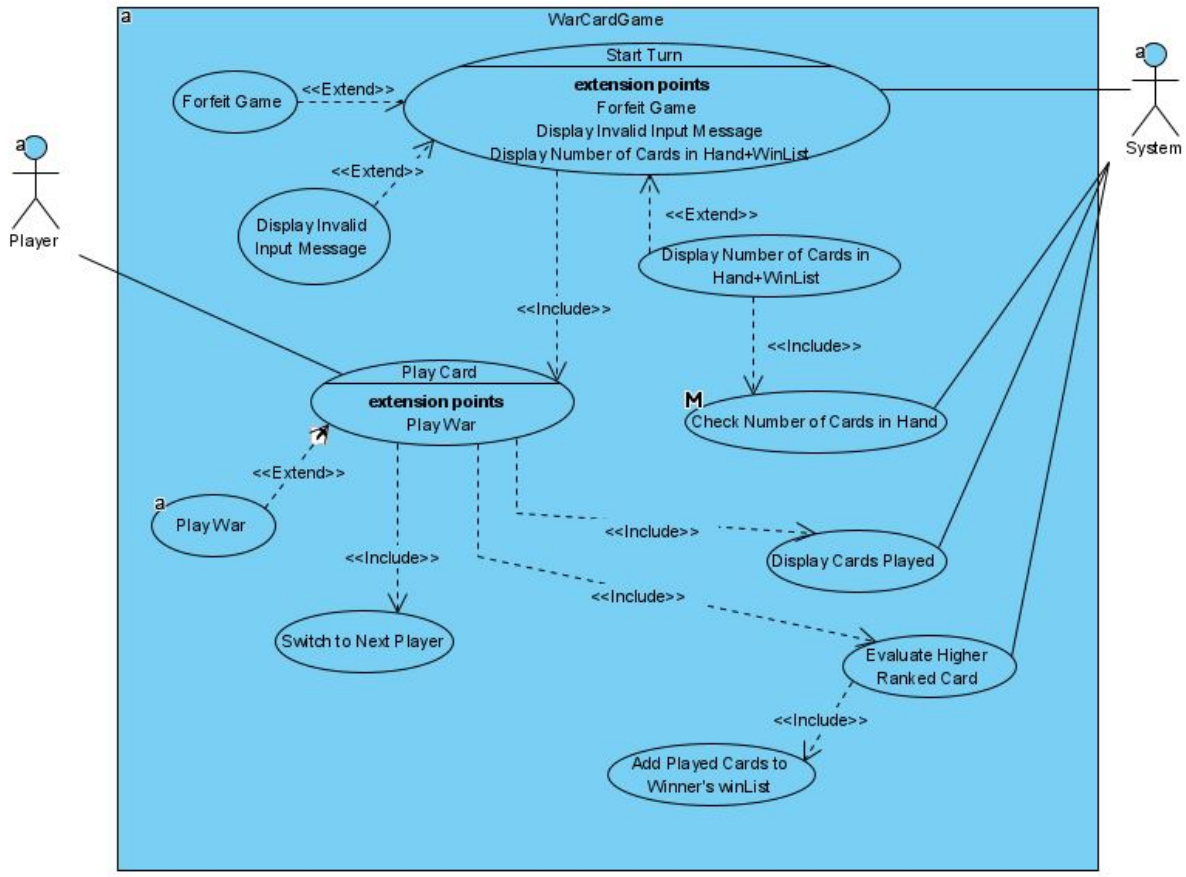

Figure 2. Play Game Use Case

1. SYSTEM Start turn and display instructions to player 1
2. Player plays card
   2.1. Player checks number of cards in hand
   2.2. Player forfeits game

2.3. Player enters invalid input
3. SYSTEM Switch to next player and display instructions
4. SYSTEM Evaluate the value of the cards played by players
4.1. if Value of card is equal
4.1.1. Play war
end if
5. SYSTEM Display winner of turn and displays card faces
6. SYSTEM Add cards played to winner's winList
7. SYSTEM Check number of cards in each player's hand
7.1. if Number of cards is 0
7.1.1. SYSTEM Goes to End Game use case
7.2. else
7.2.1. SYSTEM Start Play Card use case again
end if

End Game

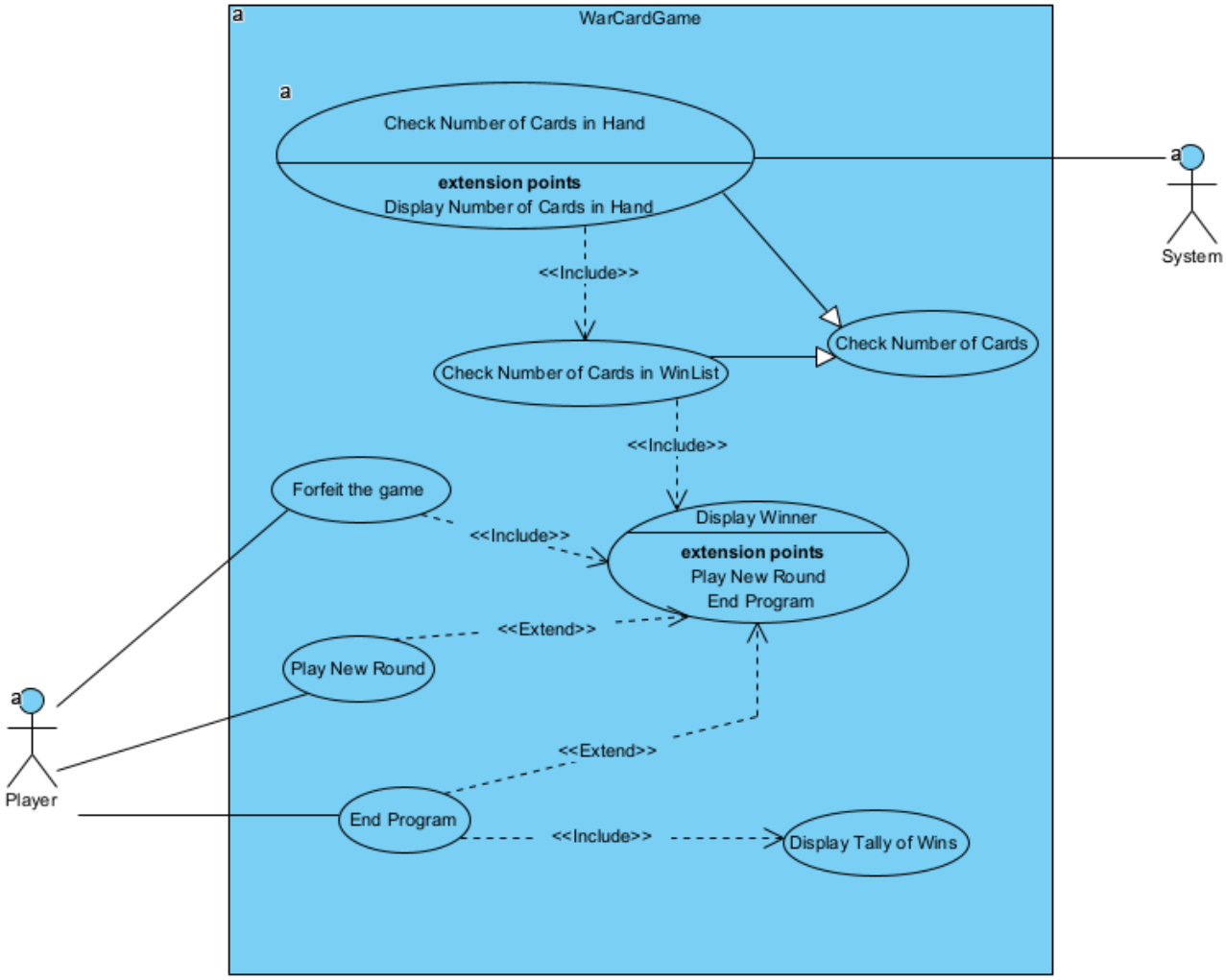

Figure 3. End Game Use Case

1. SYSTEM checks the number of cards in hand each time players deal a card
2. if A player has no more cards in hand
   2.1. SYSTEM displays winner who has most cards in WinList
3. else if A player wishes to end program in the middle of game
   3.1. SYSTEM terminates the entire game, and displays winner
   end if
4. Players finish five rounds
5. SYSTEM displays tally of wins and final winner
6. Player wishes to end game
   6.1. Player wishes to play new game
7. SYSTEM terminates the program

Class Diagram

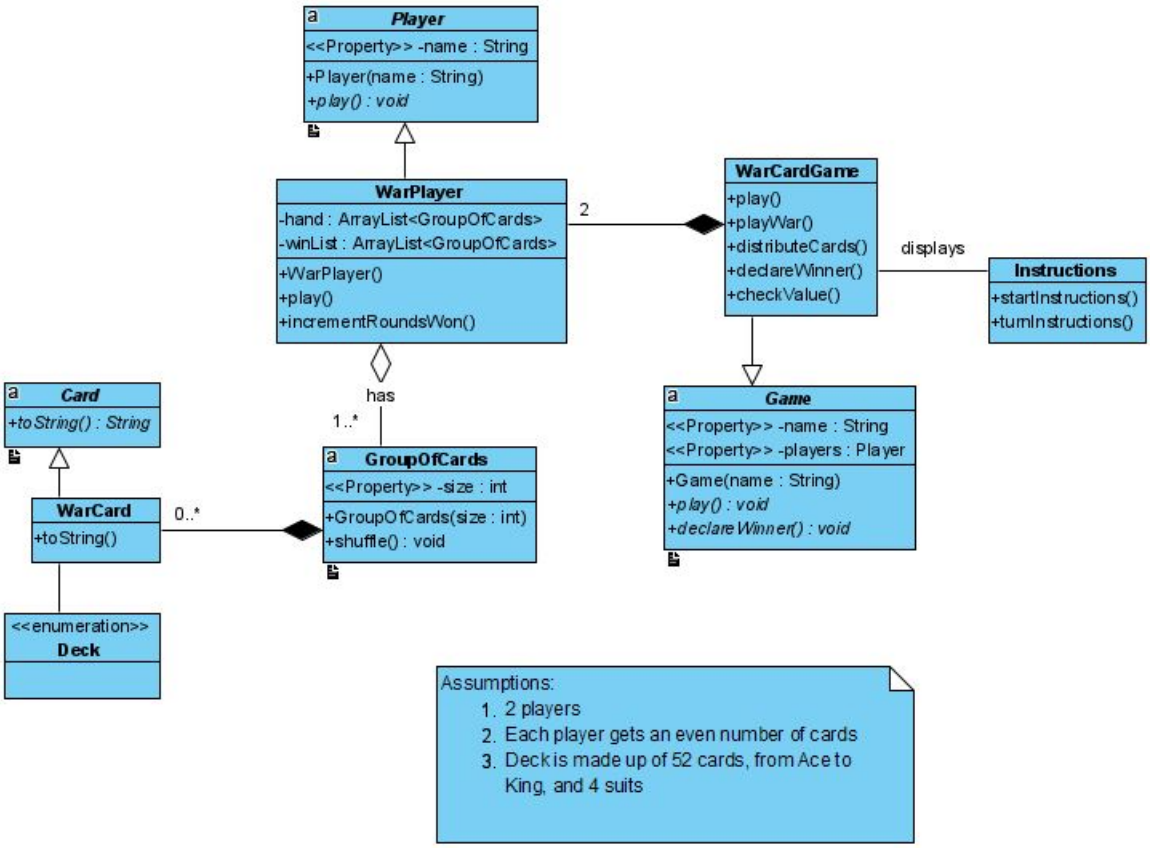

Figure 4. Class Diagram

Design Document

1. Project Background and Description

Our team will create the console card game based on the War card game. In the original game, the game will continue until one player has no cards in `hand` and in their `winList` deck as well. However, we decided to add more rules such as “terminate game at any point”, “set a certain number of rounds”, “complete a round when players have no cards in hand”, and so on. Here are more details divided by game process:

- Register player
    - Our War card game will require two players to register with their names.
    - Their names will be verified to not have the exact same name to distinguish players while the game is running.
    - Players cannot change their name unless they start over.
- Play game
    - Players can check the number of cards in hand.
        - When it is their turn, they will have multiple choices. For example :
            - Type in “c”: check in hand,
            - Type in “p”: put card on play deck,
            - Type in “t”: terminate game (forfeit), and
            - Type in “i”: show instruction
    - Once players put cards onto play deck, system will show their values and determine who wins those cards by comparing their values.
- End game
    - There are two ways of finish one round of the game:
        - If there are no cards left in player’s hand, system will complete the round automatically.
        - If one player wants to end the game at any point in time, they may do so by terminating—i.e. forfeiting—the game.
            - By default, player who forfeits the game will lose the entire game.
    - The game terminates after five rounds, system will :
        - Display tally of wins. For example:
            - Timmy : Tommy = 3 : 2, Tommy won the game!
        - Ask if players want to play new game or terminate the program.
            - If players choose to play a new game, everything will be reset from registration.
            - If players choose to terminate the program, display “Thanks for playing” message and close the program.

2. Design Considerations

Description of the Class Diagram

Figure 4 shows the relationships among classes to conduct the card game: War. There are three base classes that are defined as a more general class in **Figure 4**: `Player`, `Card`, and `Game`. They derive more specialized classes (derived class, subclass or "child class").

First, `Player` class, which is an abstract class, has a derived concrete class: `WarPlayer`. `WarPlayer` inherits all the non-private code from the base class, which is `Player`. This causes `Player` and `WarPlayer` to have an inheritance relationship as a special case of an association. `WarPlayer` has relationships with `WarCardGame` and `GroupOfCards` as well. On the one hand, `WarPlayer` is part-of `WarCardGame`. This is a composition relationship (whole-to-part relationship), which means that `WarPlayer` cannot exist without a `WarCardGame`. When it comes to multiplicities, `WarCardGame` has two `WarPlayer` objects to implement the game. On the other hand, `GroupOfCards` is-part of `WarPlayer` (part-to-whole relationship). This is an aggregation relationship. `GroupOfCards` can exist without `WarPlayer`. In terms of multiplicities, `WarPlayer` "has" more than one `GroupOfCards` object in the relationship.

Second, `Card`, which is an abstract class, has subclass `WarCard`. Since `Card` class is a parent class of `WarCard`, it will be the basic template for `WarCard` in the hierarchy. In other words, `WarCard` inherits the attributes and operations of the parent class - `Card`. `WarCard` is associated to `Deck` which is an enum class: a special class that represents a group of constants. `Deck` contains all fifty-two cards that players need for the game. `WarCard` is part-of `GroupOfCards`, resulting in a composition relationship. `WarCard` cannot exist without a `GroupOfCards`. Regarding multiplicities, greater than or equal to zero `WarCard` is expected to each `GroupOfCards` in the relationship.

Third, `Game` is an abstract class and a superclass. `WarCardGame` is derived from `Game`—in other words, a `WarCardGame` is-a `Game`. This is a generalization/ inheritance relationship. There is an association between `WarCardGame` and `Instructions`. It shows that two classes need to communicate with each other. To be specific, when players need instructions, `WarCardGame` "displays" instructions to players by using a method.

Encapsulation

The `WarPlayer` class contains two ArrayList attributes, `hand` and `winList`, with each having private access modifiers. These will contain the cards the player has yet to play and the cards the player has won respectively. The `WarPlayer` constructor, `play`, and `incrementRoundsWon` methods have public access modifiers to allow for use within the `WarCardGame` class. The `play` method is an implementation of the `play` method from the abstract `Player` parent class. The `WarPlayer` constructor invokes the parent class constructor from the abstract `Player` parent class to initialize the `name` attribute, which has a private access modifier.

The `WarCardGame` class contains a `play`, `playWar`, `distributeCards`, `declareWinner` and `checkValue` method with public access modifiers. The `play` and `declareWinner` methods are implementations of the corresponding methods in the abstract `Game` parent class.

The `Instructions` class contains `startInstructions` and `turnInstructions` methods with public access modifiers.

The `GroupOfCards` class contains a `size` attribute with a private access modifier, as well as a public constructor and a public `shuffle` method.

The `WarCard` class contains a public `toString` method implementation of the parent `Card` class.

Delegation

We created a subclass for the `Game` and `Player` classes, meaning the delegation discussed in **Deliverable 1** can be fleshed out using concrete classes. `WarCardGame` is the delegator and a `WarPlayer` object is the delegate. In this case the `getPlayers` method can be called with the index of the desired `WarPlayer`, object and the `play` method from the `WarPlayer` class can be used.

This also applies with the concrete `WarCard` class we created. An ArrayList of `WarCard` objects is listed as an instance variable for `GroupOfCards`. To call upon the `toString` method in the `WarCard` class we could use the `getCards` method from `GroupOfCards` with the desired index to access the `WarCard` object. The `toString` method in this case can be used to display the `WarCard` object's data to the users.

Cohesion

We separated the game into multiple classes that are responsible for doing one thing. The `WarPlayer` class extends the `Player` class and is responsible for all things related to the players. Each `WarPlayer` instance will contain data such as the individual hand and cards won and methods to count the number of rounds won. The `WarCardGame` class, on the other hand, is responsible for the game progression and any action required to play the game.

Loose Coupling

In our class diagram, all our classes revolve around three central classes: `WarCard`, `WarPlayer`, and `WarCardGame`. Between these three classes, there are very few connections. This helps keep our code self-contained.

We are building the deck as an enum class, `Deck`, to help create more loosely coupled code. If we wanted to switch the deck from a standard card deck to a set of Mahjong tiles, we would need to change only the `Deck` enum class.

Inheritance

To extend the base code, we created the `WarCardGame` subclass to extend the `Game` class, the `WarPlayer` class to extend the `Player` class, and the `WarCard` class to extend the `Card` class. All these subclasses will inherit the methods of its parent classes.

Aggregation

`WarCardGame` "owns" two `WarPlayer` objects, and the game cannot exist without 2 players. Therefore, they have class relationship of composition. Each `WarPlayer` object will "own" or "have" one or more `GroupOfCards` object, but it can exist by itself. So, we define their relationship as aggregation.

Composition

Also, `GroupOfCards` "has" any number of `WarCard` which is using `Deck` enumeration. `GroupOfCards` objects are always composed of `WarCard`. Therefore, their relationship is composition.

Flexibility/Maintainability

Extending classes from abstract classes allows us to build any number of games from a basic set of properties and behaviours. Further, by adhering to OOP design principles as shown above, we ensure that our code is flexible and easy to maintain.