The Fundamentals of Software Design and Development

Deliverable One: Design Document

Prepared by Shibrah Misbah, Vincent Ursino, Zain Iqbal

Students of SYST 17796

Professor: Amandeep Sidhu

Sheridan College

Submitted:

February 23rd, 2021

Table of Contents

Project Background and Description 3
    Classes and the MVC Pattern 4
Project Scope 5
High-Level Requirements 5
Implementation Plan 6
Design Considerations 7
    Encapsulation 7
    Delegation 7
    Flexibility/Maintainability 8
    Enumerations 8
UML Diagram 9
Citations 10

Project Background and Description

Our project seeks to fulfill the simulation of a GoFish card game by implementing four base classes that each perform a different operation belonging to the gameplay.

The gameplay can consist of many players in which each player is dealt with seven cards that are shuffled by a single dealer. If there are 2 to 5 players, then the number of cards assigned is seven, otherwise a dealer should assign 5 cards to each player when there are 4 or more players. When the card assignment is done, the dealer will set the rest of the cards face down in the middle. This is known as the *fish pond* that players will use later on to draw from. The objective of the game is to create a set of four matching cards, called *books*, and the player with the most sets of identical cards wins! (Wikihow Content Management Team, 2020).

We will be demonstrating the game with two players using the template provided with just seven cards. First, player one will ask the next opponent for a card rank (e.g., kings, ones) that they may have plenty of and if player two has any amount of that type of card, their task is to lend all of the cards of that rank to player one. If player two does not have any of the cards asked for, they must tell player one to *go fish* from the pond and draw one card. If they're lucky, that card might be the one they're looking for and a set of four can be made. Otherwise, the card is simply added to their group. The game ends when a player is left with no cards at all and has fulfilled their set of *books*.

The classes provided are set up in a way that establishes a separation of concerns so each class has a unique responsibility with respect to the game.

1. **Model Classes:** Card and GroupOfCards
    a. Used as a means to connect the view and controller classes.
    b. GroupOfCards sets the size of each player's deck, ensures that given size is not exceeded, and implements showCards() and shuffle(cards) method
2. **View Class:** Game Class
    a. Handles user interaction and manipulates input/output
    b. Specifies the name of the game and returns the players using an ArrayList
    c. Implements play() method to play the game and declareWinner() method to give corresponding output of user's score
3. **Controller Class:** Player Class
    a. Setting the player's unique ID
    b. Contains abstract play() method template

Classes and the MVC Pattern

1. **Card:**
The goal is to implement the abstract class Card that acts as a template for the children classes that will be used to return a String representation of a “Go Fish” card. The objective of the base code is to have the children classes or the objects of the children classes to share the methods defined in the base code so that each card can be converted into a String. The overridden abstract toString() method will be used when we extend the code into its respective branches.

2. **Game:**
The abstract game class is a generalized base code that will be implementing a private and final String game name that will be set to “Go Fish”. The final keyword is used as well because the String name is the only variable that needs to be protected from instant modification. The second data field seeks to create an ArrayList object of type Player where the players will be stored. The setters and getters are also provided below to allow the modification of the game name, as well as the ArrayList of players. The end of the code includes abstract void methods that will not return anything and be invoked later in the extensions of this class in order to play the game and declare the winner.

3. **GroupOfCards:**
This is to be a base code that represents the groupings of any cards, which in our scenario, is 2 groupings for 2 players that will represent the deck of cards with a certain size that is equal to the givenSize using the set size method. The non-private constructor for GroupofCards ensures that the size is what we set it to, while the public ArrayList of object type Card will return all the cards by using its showCards() method. These cards will be passed as a parameter in the shuffles method of the Collections class.

4. **Player:**
Finally, the controller class will be in charge of giving the player its piece of identification with setters and getters for the unique ID the player will have (String name), as well as a constructor that ensures the player ID is set to the name provided in the parameters of the Player object.

Project Scope

This project consists of 4 developers who will collectively ensure that all stages of development have been completed. Vincent Ursino is the Leader and Repository Manager who will oversee all aspects of the code, the pdf files, diagrams, and any other contributions of the group members. Shibrah Misbah and Zain Iqbal, the reporters, will handle all write ups/documentation while Hardeep Kaur will be the designer for the use cases and UML diagrams.

The interface will be a fairly simple “console” game where the user enters the input such as their ID, the card ranks they are asking for, and receives output from the console, all while interacting with a computer generated player with a randomized group of cards. The game will be seen complete when the requirements are met, all classes with instantiated objects have been stored in their appropriate folders, the right visibility modifiers are used and the code accomplishes a general run of the card game. The requirements outlined in the next section such as the user being able to receive seven or less cards, must be achieved before we can move forward with the execution of the program.

High-Level Requirements

The requirements of the project are what the system must do in order to function properly and communicate with the user. The set of requirements that need to be fulfilled include the system (game) and user (game player) requirements. First, we need to ensure that the principles of OOP are fulfilled while designing the code so that the developer can implement them correctly to ensure a user friendly experience free of any potential runtime issues.

A user should be able to:
- Set and retrieve their unique ID
- Be assigned a group of cards no more than seven and view their cards
- Remove cards (when asked for) and add cards to their deck (from player one) if the requested ranks are available
- Pick up cards from *fish pond* when the requested ranks are not available

The game should be able to:
- Return the cards in their correctly formatted rank (e.g., Queen of Hearts)
- Print title of game and finalize it
- Modify list of players by adding and removing objects
- Display and declare a winning player at the end of the game
- Generate a group of cards through a given size and shuffle the cards
- Randomize cards for the opponent as well as the player
- Award the player points for getting a set of 4

Implementation Plan

Link to Go Fish Project Repository: https://github.com/vu-193/go-fish-project

The code files will be found under src/ca/sheridancollege/project in the GitHub repository.

The git repository managed by our project leader will be used to access all the code files. We will be using the base codes to export to our local repository whenever changes are necessary, as well as committing and pushing any changes made in the local repositories to the remote git repository. Each member has the responsibility to make a *“Modified by”* note at the top of each class that may have been modified. Another use of this is to ensure that each developer is able to oversee the commits that have been added, remotely at the end of every day, and to test out the program to ensure its smooth development.

While designing and planning the execution of our program, our tools will prove useful in the overall development of the game system. Apache NetBeans IDE Version 11.3 will be used in conjunction with Visual Paradigm, both which will assist us in the creation of the use cases and UML diagrams that represent our base codes.

In order to increase the readability of the code, our goal is to implement proper coding conventions and standards such as using a consistent programming style, end-of-line or next-line, as well as being mindful of block and line comments. The organization of Java source files must be followed throughout all the files by starting off with the necessary beginning comments, then the package and import statements, and finally class and interface declarations. While coding, it is often a good practice to keep the number of characters in a line to no more than 80 and in the event that it does overflow, a break after a comma or before an operator is ideal (*Code Conventions for the Java Programming Language*, n.d.).

Keeping the code organized, well-structured, and visually clean will allow us to follow along with the testing procedure later on so that during our trial and error processes, all developers are able to experiment with the potential solutions an issue can have, and identify what may be causing any compile or runtime issues.

Design Considerations

Encapsulation

In each base class, encapsulation is used to ensure that the user of the classes is unable to see or modify the properties directly.

This can be seen in the following examples:

- In the **Game** class, there are **gameName** and **players** properties which have private visibility modifiers. The **players** property has both an accessor and mutator method, which allows the private property to be changed, while the **gameName** property has only an accessor method. Since it is a constant (final), having a mutator method is unnecessary as it cannot be changed after assignment.
- In the **GroupOfCards** class, there are **cards** and **size** properties which also have private visibility modifiers. The **size** property has an accessor and mutator method to allow the max size of the group of cards to be seen/modified. This increases the flexibility of the class as it can be used for any game. The **cards** property also has an accessor method called **showCards()** which is invoked to get the **cards** ArrayList.

Delegation

Delegation is a widely-used concept in the base classes as it seeks to create a separation of concerns and produce high cohesion within classes. As we update the base code with our own additions, we hope to maintain its loosely-coupled structure, which will save a lot of time while debugging or updating the project in the future as we will not need to make changes to every class.

- Instead of merging each method and property together in one big class, which would cause confusion, errors, and unconventional code, the project was divided into four classes: **Card**, **Game**, **GroupOfCards**, and **Player**.
- The **Card** class will act as a Model class for each card used in our Go Fish game. At this time, it only contains an abstract toString() method which will be overridden in our subclasses to return the full name of any given card, i.e. “King of Hearts.”
    - We plan on implementing suit and value properties and enums, along with accessor and mutator methods for both the suits and values in order to encapsulate our private data.

- The **GroupOfCards** class will also act as the Model class and is expected to have subclasses that will contain the appropriate groupings that are returned through an ArrayList. A maximum size option is given so that any card game can reuse this code respectively.
- The **Game** class will act as the View class. In this class, the players will be represented as individual **Player** objects stored inside an ArrayList. It contains an accessor method for retrieving the current players, as well as a mutator method that permits reusability so that in each round, a different set of players can be generated.
- The **Player** class holds the role of a controller that is supposed to collect all the functional components, or controls so that when a control is activated, it calls a method in a Model to change its state accordingly (Pattis, n.d.).

Flexibility/Maintainability

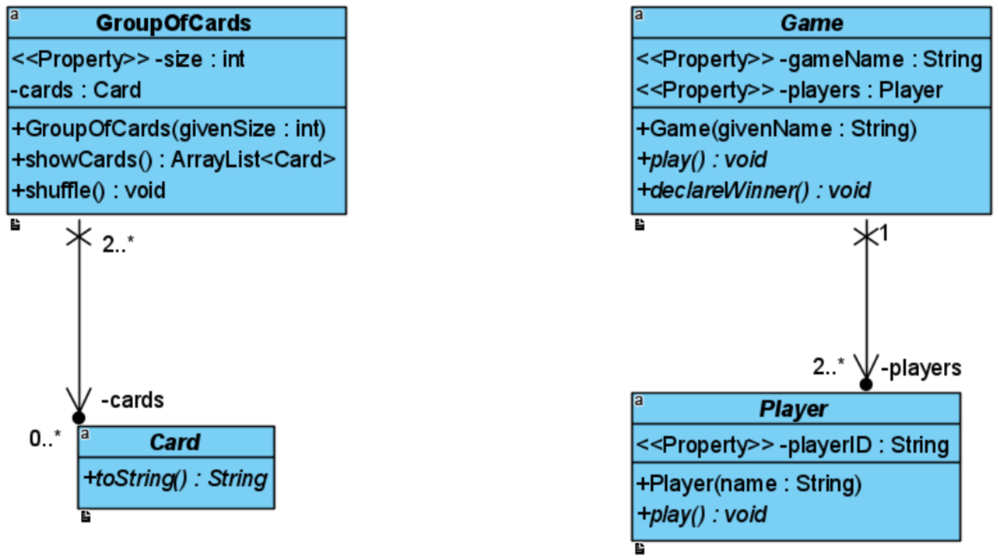
The base code of the project contains many instances of flexibility and maintainability. Instead of rewriting the same lines of code multiple times to achieve a goal, common tasks have been given their own methods which will prove useful in debugging and updating our code, as well as promoting readability. It is much easier to make a single change to a method than searching through a cluttered file to find and change every instance of a mistake.

- Inside the **Game** and **Player** classes, a method called **play()** exists, which is meant to hold all of the logic needed for playing the game. Instead of hard-coding all of that logic inside our Main method, all we have to do is invoke the **play()** method inside the Main method, which will contribute to a more maintainable and flexible design.
- The **GroupOfCards** class contains methods and properties which will help us achieve a flexible design in our code, no matter the game idea chosen. All card game ideas have common features such as needing a “deck” feature, which is accomplished through the **cards** ArrayList; a way to shuffle the cards, which is solved by the **shuffle()** method; and a maximum amount of cards per deck, which is carried out by the **setSize()** method and the **size** property.

Enumerations

A key part of the classes is using enumerated types that ensure type and value safety and this method of compiling similar objects can also allow one to divide the objects into their own classes, if needed. On the other hand, it minimizes the hassle that using constants would have created such as having to manually add a new String to the set of constants, while with enums, one can simply retrieve the values of a specified enum using the values() method without having to make any changes in the method that accesses those variables. To conclude, a major benefit of enums is that they can readily return a collection of values since there is only ever one instance of the enum, allowing it to be more efficient (Gorbeia, 2015).

UML Diagram


GroupOfCards
<<Property>> -size : int
-cards : Card
+GroupOfCards(givenSize : int)
+showCards() : ArrayList<Card>
+shuffle() : void
2..*
-cards
0..*
Card
+toString() : String
Game
<<Property>> -gameName : String
<<Property>> -players : Player
+Game(givenName : String)
+play() : void
+declareWinner() : void
1
2..*
-players
Player
<<Property>> -playerID : String
+Player(name : String)
+play() : void

Citations

Code Conventions for the Java Programming Language. (n.d.). Oracle.
https://www.oracle.com/java/technologies/javase/codeconventions-fileorganization.html

Gorbeia, G. (2015, March 11). Java Enums vs Constants. *Wordpress*.
https://gorbeia.wordpress.com/2015/03/11/java-enums-vs-constants/

Pattis, E. R. (n.d.). Model Classes in the MVC Pattern. *Richard E. Pattis.*
https://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/modelinmvc/index.html

Wikihow Content Management Team. (2020, November 17). *How to Play Go Fish.*
wikiHow. https://www.wikihow.com/Play-Go-Fish