DELIVERABLE 2

Guided under: Sivagama Srinivasan

Submitted by:Thinkers

Date: 12 March 2020

Table of Contents

Roles

Thinkers group consists of:

Names and roles

- Kirandeep Kaur: Use cases, UML diagram, document design template, formatting and proofreading

- Moses Oduwole: Use cases, UML diagram, formatting and proofreading, document design template

- Zohaib Kasmani: Use cases, UML diagram, formatting and proofreading, document design template

- Han Thao Nguyen: Use cases, UML diagram, document design template, formatting and proofreading

Part-1 Use Cases

1. User will run the application.
2. Users will be prompted to enter the number of players, possible entries can be 2,3,4.
   2.1. If a user enters any other number or nothing, it will display a message that players should be 2, 3 or 4.
   2.2 User will again prompted to enter the number of players.
3. Game will ask the user for the username , and that username will be used to set the player id. Username cannot be empty or cannot be a number.
   3.1 If a user gives invalid inputs for a username,a message will be displayed and the user will be asked to re-enter the username.
4. The cards will be distributed to all the players, five cards per player. The balance of the pack is placed face down in the center of the table and forms the stock.
5. A card will be drawn from the stock and revealed to all players.
   5.1. If an eight is turned, it is buried in the middle of the pack and the next card is turned.
6. All players will draw one card each, that matches with the revealed card drawn from the stock either by rank or suit.
   6.1. An eight is a wild card and can be played in any given cases, additionally, owners of this card can specify the next suit to be matched with.
   6.2. If a player doesn't have any matching card at hand (neither in suit nor rank), they have to continuously draw cards from the stock until a matching card appears, which can then be played.
7. Whoever disposes all of their cards first will be the winner.
   7.1. If everyone still has cards remaining while the matching stock runs out, whatever left on their hand will be counted towards the final scores.
8. The rest of the players will also reveal their remaining stack and add up the points corresponding to each suit and rank of their cards. The one with the largest sum in the group loses.

Part-2 UML class diagrams

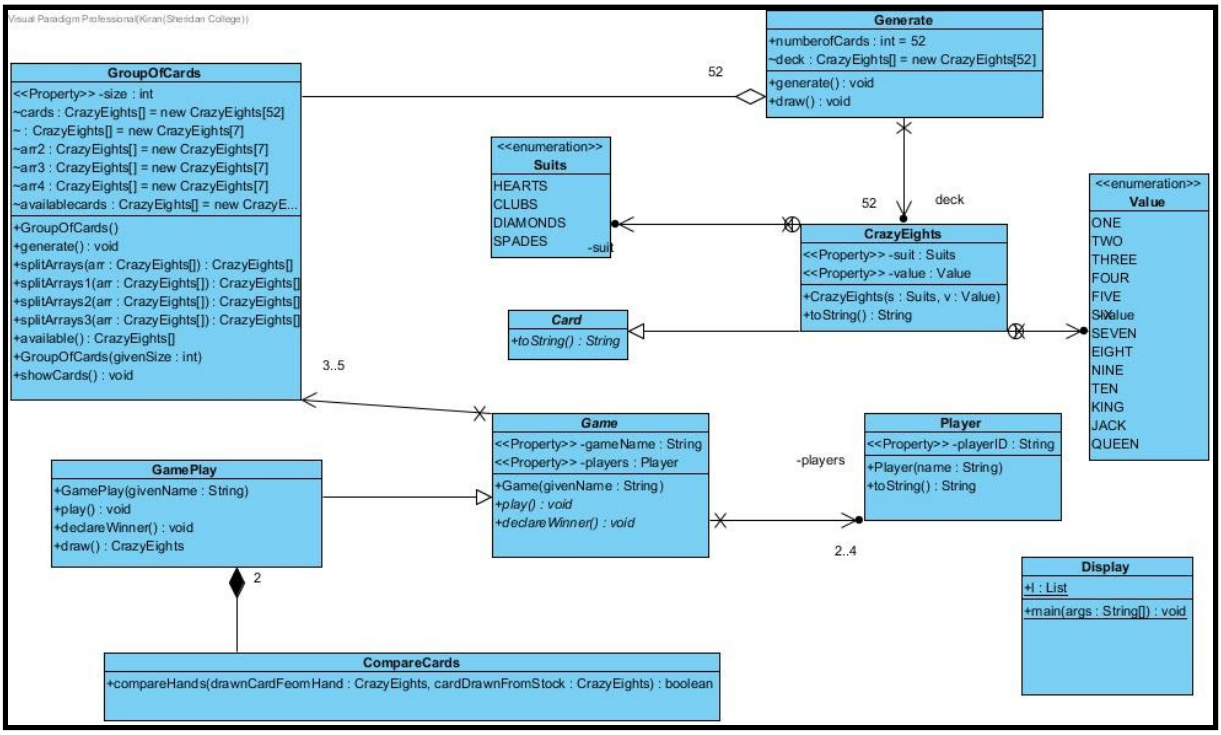

Figure 1

Part-3 Design document template

The OOD principles used:

Inheritance:
Inheritance is used to achieve code reusability, or to extend the properties of an abstract class so that all the child classes should implement the abstract methods defined in the parent class. In our game, we are using inheritance to extend Game class , to reuse play() and declareWinner() methods.

Association:
Association is a relation between two seperate classes, where two classes are connected to use the functionalities of each other using their objects, or without inheriting the properties. In the project, each class has an association with another class , so that code can achieve the OOD principles of delegation and cohesion. For instance, Player class is associated with a group of cards, to provide player id to the group of cards .

Composition:
Composition relationship describes the strong dependency of classes on each other.
In the project code, GamePlay class and CompareCard class have composition relationships with each other, because GamePlay class needs a CompareCard class to compare cards and to get the output. Similarly, CompareCard class cannot exist without getting two comparable sources from GamePlay class.

Aggregation:
In aggregation, two objects can exist independently .
For example, the Generate class has an aggregation relationship with the GroupOfCard, as both classes can exist without each other. If we eliminate GroupOfCard class, that doesn’t affect

Generate Class() , similarly if we remove Generate Class, we can still have a GroupOfCard class that will contain a group of references to card objects. This can also be considered as an ownership and uni-directional relationship as GroupOfCard class ***has*** references to the data members (Rank and Suit of cards) created in Generate class, without which GroupOfCard will still exist yet contains only ***empty*** references. Thus, Generate class is necessary but not indispensable to GroupOfCard class.

Cohesion :

Cohesion is an OOP's principle associated with the idea that each class is designed to perform a particular function. We have different classes for each function or purpose. Hence, each class in this project has its own well-focused purpose making it more effective and reliable. For instance: We have generate.java class which performs the action of generating cards. We have a Display.java class whose sole purpose will be to call the methods and display the outputs. We even have GroupOfCards.java class used to group the cards for each player.

Coupling:

Coupling means properties of two classes are directly related with each other, so that changes in one class forces the changes in the second class. Coupling can be loose or tight. Tight coupling refers to the strong dependency between classes, whereas loose coupling means that classes are approximately independent of each other.

In game code, we strived to achieve loose coupling, to maintain code flexibility. We declared card’s rank and suit as an enum, instead of String, to avoid tight coupling.

Delegation:

In delegation, one class generates an event and other classes, especially the main class make use of that function to perform some tasks. As the name suggests, delegation is related to the

delay in performing duty. CrazyEights games have multiple supplier classes , to support the working of the main class. For instance : We have Dispaly.java which calls generate() method to generate and shuffle cards of the deck and getSuit() , getValue() to initialize the card object with Suit and value.

```
public class Display {
    public static List l;

    public static void main(String[] args){
        Generate ch = new Generate();
        ch.generate();



        for(CrazyEights c : ch.deck){

            System.out.println("suit is "  + c.getSuit() + " and value is "  + c.getValue());
        }
```

Figure 2

Encapsulation:

This concept is also often used to hide the internal representation, or state, of an object from the outside.The data fields of the class are declared as private so they are protected from outside classes. We use getter and setter methods to make the data members hidden or encapsulated.

```
public abstract class Game
{
    private final String gameName;//the title of the game
    private ArrayList <Player> players;// the players of the game
```

Figure 3

As mentioned above data members can only be accessed through getter and setter methods (refactoring) as demonstrated in the following screenshots:

```
public String getGameName()
{
    return gameName;
}

/**
 * @return the players of this game
 */
public ArrayList <Player> getPlayers()
{
    return players;
}
```

Figure 4

```
/**
 * @param players the players of this game
 */
public void setPlayers(ArrayList <Player> players)
{
    this.players = players;
}
```

Figure 5

Flexibility/Maintainability :

Flexibility can be achieved by reducing the dependency of classes on each other. In the project code , we declared Suit and Value, two enums and generated data members for each of that (suits for Suit and value for Values).Introducing Enum in the code makes it independent , resulting in a more efficient program.

```
public class CrazyEights extends Card {


    public enum Suits{HEARTS,CLUBS,DIAMONDS,SPADES};
  private Suits suit;
    public enum Value{ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN,KING,JACK,QUEEN};
    private Value value;
    public CrazyEights(Suits s, Value v){
```

Figure 6

Since we abide by the principles of cohesion and delegation, each class is created better-focused and responsible for only one specific task. If the compiler issues an error, it’s easier to pinpoint the source of the problem since the type of error can be narrowed down to a specialized class, and troubleshooting it first can save more time and resources. As a result, the maintainability of the program is improved.

.