SYST 17796

Section : 32045

Group 3 :Deliverable 2

Professor: Dr. Nagma

*INDEX*

Title and Index: page 1

Use Case Diagram : page 2

Use Case Narratives: page 2

Class Diagram : page 4

Design Document: page 4-6

Rubric : page 7

<https://github.com/radadime/CourseProject.git>

|  |  |  |  |
| --- | --- | --- | --- |
| Team Member Names | Signatures | Initials | Student ID |
| ***Project Leader:*** Jashanpreet Kaur | Jashanpreet Kaur | JK | 991708849 |
| Meetkumar Ramnikbhai Radadiya | *Meet* | MR | 991702529 |
| Sojaldeep Singh | Sojaldeep | S.S. | 991704170 |
| Sukan Sundaramoorthi | ***SUKAN SUNDAR*** | S.S.M | 991598890 |

USE CASE Diagram and Narratives :

CLASS Diagram:

A screenshot of a computer

Description automatically generated with medium confidence

The UML class diagram provided illustrates the structure and relationships of the classes involved in the Java code. The Cards class serves as the entry point, containing a main method to initiate the program. It has a static method createDeck() that generates a list of PlayingCard objects representing a complete deck of cards. The createDeck() method utilizes the Rank and Suit Enums to iterate over all possible combinations and create the cards. The Card class is an abstract class with an abstract toString() method, serving as the base class for the PlayingCard class. PlayingCard is a concrete class that extends Card and contains private attributes rank and suit. It also has getter methods to access these attributes. The PlayingCard class overrides the toString() method to provide a textual representation of the card&#39;s rank and suit.

The Rank and Suit Enums represent the possible ranks and suits of a playing card, respectively. They provide a set of predefined constants for each rank and suit.

Overall, this UML diagram depicts the hierarchy and relationships between the classes involved in the Java code. It showcases how the Cards class utilizes the PlayingCard class and how PlayingCard is composed of Rank and Suit objects.

SYST 17796 Deliverable 2

Design document

# Overview

## Project Background and Description:

Elaborate on the game you chose, and the Description provided in Deliverable 1 by providing more detail on the exact scope of your project (i.e. “the game will terminate after four rounds, giving each player a total score”).

Description:

The Chosen Game is ***Go Fish*** A traditional 52-card deck is used to play the card game Go Fish. By requesting cards from other players, the goal is to amass sets of four cards with the same rank. The person with the most sets wins the game when all the cards have been gathered.

Elaboration on the Rules of Go Fish:

1. A hand of cards is handed to each participant.
2. The first player begins by requesting cards of a certain rank from another player.
3. The player who is asked must hand over all cards of that rank if they own any.
4. The requesting player says, "Go Fish," and the asking player pulls a card from the deck, if they don't have any cards of that rank.
5. The player who asked receives another turn if the drawn card matches the requested rank.
6. The next player takes the field if the chosen card does not have the specified rank.
7. A player sets the set-in front of them if they ever gather four cards of the same rank.
8. Once all the cards have been gathered, the game is over.
9. At the end of the game, the player with the most sets of cards wins.

## Design Considerations:

Describe the Class Diagram you delivered above (it should be described as Figure 1 or Figure x if you have more than one Figure), explaining the associations and multiplicities depicted.

Comment on each of the following as it pertains to the class groupings you have decided upon and if you have included methods, modifiers and return types, comment on those here as well. You may wish to describe any data structures you wish to use (i.e. an enumeration) when you are explaining your design choices. Be specific about earning full credit.

The current code demonstrates some adherence to the following object-oriented principles:

***Encapsulation:***

The properties ('rank' and 'suit') and actions (getters, toString ()) associated with a playing card are all included inside the PlayingCard class. Encapsulation and data hiding are ensured by the attributes' declaration as 'private' and their access via public getter methods.

The logic necessary to produce a deck of playing cards is included in the 'createDeck ()' function. It provides a simple interface to get a deck while hiding the technical implementation details.

*Potential for improvement:*

It would be advantageous for the 'PlayingCard' class to provide more behaviour pertaining to card-related activities like testing for equality or card comparisons.

For improved encapsulation and a clearer separation of concerns, it would be advantageous to place the game-related logic and operations in a different class.

***Delegation:***

The 'PlayingCard' class extends the abstract 'Card' class, which allows it to inherit common behavior and attributes defined in the parent class. This is an example of delegation through inheritance.

The 'GoFishGame' class delegates the responsibility of creating a deck to the 'createDeck()' method. It relies on the method to provide a complete deck of playing cards.

*Potential for improvement:*

Additionally, several classes that manage the players, turns, and rules of the game, such as the 'Game' class, can be given control over the game logic. Better code separation and organisation would result from this.

***Cohesion:***

A class should represent one logical entity only and all operations should be able logically fit together in the context of that one entity. The Classes of Card, Suit and Rank are the best examples of high cohesion design principle in the Go-fish game.

Ranks and Suit have a similar task of populating the attributes of Playing cards with rank and suit objects to be feed into the PlayingCard() method.

***Coupling:***

The previously suggest addition of Game class will be used to better separate and reduce the level of dependency between classes. The abstract class Cards has no other denpencies on any other classes and their attributes to function, it interacts with PlayingCard and Playing uses it to help create objects for the createDeck method. This help keep the system loosely coupled.

***Inheritance:***

The class: PlayingCard extends the abstract class Card to inherit its toString() method.

***Aggregation:***

The ArrayList List <Playingcard> in our createDeck method demonstrates a form of aggregation as it can contains multiple object instances of PlayingCard.

***Flexibility/Maintainability:***

The use of camel case for method and variable names ('createDeck()', 'getRank()', and 'toString()'), for example, makes the code easier to understand and maintain.

The card game rules may be easily extended and modified thanks to the usage of enums ('Rank' and 'Suit'). By simply altering the Enum values, ranks or suits may be added or changed.

*Potential for improvement:*

Design patterns, like the Factory pattern, might be incorporated into the programming to create various card game kinds. This would increase flexibility by making it simple to integrate other game logic and rules without significantly changing the present code.

The use of interfaces and polymorphism can provide flexibility by enabling the replacement of various card kinds or game iterations.

**Rubric:**

| Item | Criteria | Feedback | Weight |
| --- | --- | --- | --- |
| Use Case Diagrams and Narratives | Fully developed use case diagram[s] and narratives produced, including alternate paths. The scope of the code is well defined given the use cases. The use cases are functionally correct based upon the rules of the game cited in Deliverable 1. |  | 4 |
| Class Diagram | Diagram is notationally correct and complete, showing all necessary classes, associations, and multiplicities. Also shows an understanding of aggregation, inheritance, and composition where appropriate. The diagram also depicts an architecture that follows the principles of OOD such as cohesion, coupling, delegation, and encapsulation. |  | 4 |
| Design Document Template | All sections of the template are complete. The design choices described represent the principles of OO Design studied in class and are well articulated and presented. |  | 2 |