A close-up of a logo

Description automatically generatedA blue circle with a white letter

Description automatically generated

Software Design

DELIVERABLE 2

Student ID : 991727895

Student Name : Ronish Makaju

Class : 1249\_83045

Professor : Nagma Nagma

Submitted on : <dd-mm-yy>

# 

# 

Contents

[1. Project Background and Description 3](#_Toc183201920)

[2. Diagrams 4](#_Toc183201921)

[Narratives 4](#_Toc183201922)

[B. Class diagram: 6](#_Toc183201923)

[3. Design Considerations 6](#_Toc183201924)

[4. Design Principles 7](#_Toc183201925)

[Figure 1: Use case diagram 4](#_Toc183201943)

[Figure 2: Narative for Registering for game 4](#_Toc183201944)

[Figure 3:Narative for starting the game 5](#_Toc183201945)

[Figure 4:Narative for selecting winner 5](#_Toc183201946)

[Figure 5:Class Diagram 6](#_Toc183201947)

# 1. Project Background and Description

This project involves building a digital version of Blackjack. The game is a card-based competition where players aim to beat the dealer by having a hand closer to 21 without going over.

The game will follow these rules:

* Both the player and dealer start with two cards. Players can choose to "Hit" (draw another card) or "Stand" (end their turn).
* The dealer must follow fixed rules: they must draw cards until their hand is at least 17 and stand at 17 or higher.
* Cards are scored as follows: face cards are worth 10, number cards are worth their value, and Aces can be worth either 1 or 11.

The system will track the scores for multiple rounds. After four rounds, the game will end and display the final winner. Players will see their score after each round. The scope includes implementing all the rules of Blackjack, managing the deck, automating dealer behavior, and calculating the winner. Players can register by providing their name.

# 2. Diagrams

a. Use Case Diagram

A diagram of a diagram

Description automatically generated

Figure : Use case diagram

Narratives

A screenshot of a computer

Description automatically generated

Figure : Narative for Registering for game

A screenshot of a computer

Description automatically generated

Figure :Narative for starting the game

A screenshot of a computer

Description automatically generated

Figure :Narative for selecting winner

# B. Class diagram:

A diagram of a game

Description automatically generated

Figure :Class Diagram

# 3. Design Considerations

Class Diagram Description (Referring to Figure 5)

The class diagram shows the relationships and responsibilities of the game's core classes:

The Card class is the foundation for all types of cards in the game. It's an abstract class, meaning you can’t create objects from it directly. Instead, it provides a blueprint with an abstract method called toString(). This method must be implemented by any class that extends Card.

Next, we have the Game class, which models the overall structure of any game. It’s also abstract, serving as a base class for specific game types. This class holds two attributes: name, which stores the game’s title, and players, a list of Player objects participating in the game. The class includes methods to play the game and declare a winner, but since it’s abstract, these methods must be defined by subclasses. Think of this class as the backbone for creating customized games.

The GroupOfCards class represents a collection of cards, such as a deck or a hand. It contains an array list called cards to hold the individual Card objects and an integer size to define how large the group can be. This class also includes methods to shuffle the cards and manage the size of the group. Importantly, GroupOfCards has a composition relationship with Card, meaning it owns the cards it holds. If the group of cards is destroyed, the cards within it will also be destroyed. This is represented in the UML by a solid diamond on the GroupOfCards side of the relationship line.

The Player class models individual participants in the game. It’s abstract too, meaning specific types of players will extend it. Each Player has a unique name attribute and an abstract play() method that subclasses must implement. The name ensures each player can be identified uniquely.

The relationships in the diagram are also important to understand:

Game and Player: A game can have multiple players, so there’s a one-to-many relationship between Game and Player. The multiplicity on the Player side is denoted as 1..\* (one or more).

GroupOfCards and Card: This is a composition relationship, which means GroupOfCards completely owns the Card objects. The solid diamond on the GroupOfCards side of the line indicates ownership. The multiplicity on the Card side is \*, showing that a GroupOfCards can hold many cards.

In short, this diagram connects the building blocks of a card game. It clearly shows how games, players, and cards interact and how ownership and responsibilities are distributed. The abstract classes lay down the rules, while specific subclasses (not shown here) bring them to life.

# 4. Design Principles

* Encapsulation

Classes have private attributes, which are accessed through getters and setters.For example, the Card class keeps its suit and rank private, ensuring better control.

* Delegation

The GroupOfCards class handles shuffling and dealing cards, so the Game class doesn’t need to manage these directly. The Player class delegates specific actions (e.g., Hit/Stand) to its subclasses.

* Cohesion

Each class focuses on a single task. For instance, GroupOfCards is only responsible for managing a deck of cards. This design makes the system easier to understand and modify.

* Coupling

The design minimizes direct dependencies. The Game class interacts with Player objects without needing to know if they are HumanPlayer or Dealer. This reduces the impact of changes in one class on others.

* Inheritance

The Player class is extended by HumanPlayer and Dealer to customize behavior. The Card class can also be extended for other card games, making the design reusable.

* Aggregation

GroupOfCards aggregates Card objects but does not own them permanently. Similarly, the Game class aggregates players to manage them during gameplay.

* Composition

Strong composition exists between Game and GroupOfCards. The deck is essential for the game to function. The Blackjack class combines game-specific logic with the parent Game structure.

* Flexibility/Maintainability

Abstract classes like Card and Player make the system flexible. Adding new types of cards or players will not affect existing code. The code is modular, so changes in one part (e.g., the deck) will not require changes in unrelated areas (e.g., player behavior).