

OBJECT ORIENTED PROGRAMMING

Final Project



SUBMITTED BY:

Danyal Zahid - (0026)
Moiz Nadeem Lone - (0064)

SUBMITTED TO:

Revered Prof. Huda Sarfaraz

Date of Submission: 08-Dec-2024

Beaconhouse National University LAHORE

Game Project

1. Introduction

- **Project Overview:** A brief description of what the project is about. In this case, the project is a simple game system where users can play various games like Math Challenge, Anagrams, Missing Number, Odd One Out, and Hangman. Players can sign up, sign in, and earn badges based on their game performance.
- **Objective:** The goal is to create a program where users can enjoy various fun challenges / games while keeping track of their scores and badges.

2. System Architecture

- **Components:**
 - **Game Class:** The base class for all games. It has a virtual play() method that must be overridden by subclasses.
 - **Derived Classes:** Each game (MathChallenge, Anagrams, MissingNumber, OddOneOut, Hangman) is derived from the Game class and implements its own version of the play() method.
 - **Player Class:** Stores player information like username, password, badges, and provides authentication and badge management.
 - **Leaderboard:** Displays the list of players and their badge counts, sorted by the number of badges.
 - **Main Program:** The entry point of the application where players can sign up, sign in, play games, and view their badges and the leaderboard.

3. Game Flow

- **Sign Up:** A new player can sign up by providing a username and password. The player's information is saved to a file.
- **Sign In:** A player can log in by providing their username and password. If valid, they can access the game options.
- **Game Selection:** The player can play one of the available games. A random game is selected for the player to play.

- **Badges:** Each game is associated with a badge. Players earn a badge after completing a game.
- **Leaderboard:** After playing, players can view their earned badges and check the leaderboard to see where they stand relative to other players.

4. Classes and Functions

4.1 Game Class

- **Attributes:**
 - **Title:** The title of the game.
- **Methods:**
 - **Play():** A virtual method to start the game. This will be overridden in derived classes.

4.2 Anagrams Class (Derived from Game)

- **Attributes:**
 - **WordList:** A list of words from which one is selected and scrambled for the player to unscramble.
- **Methods:**
 - **Play():** Scrambles a word from the list and prompts the player to guess the correct word.

4.3 MathChallenge Class (Derived from Game)

- **Methods:**
 - **Play():** Displays a math addition challenge and checks if the player's answer is correct.

4.4 MissingNumber Class (Derived from Game)

- **Methods:**

- **Play()**: Displays a sequence of numbers with one missing and asks the player to identify the missing number.

4.5 OddOneOut Class (Derived from Game)

- **Methods:**

- **Play()**: Displays a list of words with one "odd" word and asks the player to identify the odd one out.

4.6 Hangman Class (Derived from Game)

- **Attributes:**

- **Hangman_state**: It consists of an array representing the different stages of the hangman figure based on incorrect guesses.

- **Methods:**

- **Play()**: Implements the classic hangman game where the player guesses letters to reveal a hidden word.

4.7 Player Class

- **Attributes:**

- **Username**: The player's username.
- **Password**: The player's password.
- **Badges**: A list of badges earned by the player.

- **Methods:**

- **Authenticate ()**: Authenticates the player based on the username and password.
- **addBadge()**: Adds a badge to the player's profile.
- **displayBadges()**: Displays all badges earned by the player.
- **getUsername()**: Returns the player's username.
- **getBadgeCount()**: Returns the number of badges the player has earned.
- **getPlayerCount()**: A static method that returns the total number of players.

5. File Handling

- **savePlayerData()**: Saves player data (username and badge count) to a file players.txt.
- **loadPlayers()**: Loads player data from the file players.txt to retrieve previous player information.

6. Leaderboards

- **displayLeaderboard()**: Displays the leaderboard, showing the players sorted by the number of badges they have earned.

7. Dependencies and Libraries

- **C++ Standard Libraries:**
 - **<iostream>** for input and output operations.
 - **<vector>** for storing player data and badges.
 - **<cstdlib>** and **<ctime>** for random number generation.
 - **<algorithm>** and **<random>** for shuffling words and other random operations.
- **File Handling:**
 - **<fstream>** for reading from and writing to files.

8. User Interface

- **Main Menu:** The user is prompted with a menu where they can sign up, sign in, play a game, view badges, view the leaderboard, or exit.
- **Game Prompts:** After logging in, the player is randomly assigned a game and prompted with the challenge.
- **Endgame:** After completing the game, the player is informed whether their answer was correct or not and awarded a badge.

9. Example Run

Sign Up:

- Enter username: player1
- Enter password: pass123
- Sign-up successful!

Sign In:

- Enter username: player1
- Enter password: pass123
- Sign-in successful!

Play Game:

- Unscramble the word: apelp
- Guess: apple
- Correct! You earned the Anagrams Badge.

Leaderboard:

- Leaderboard:
- player1: 1 badge

10. Significant Design Decisions

In designing the game system for this project, several key design decisions were made that had a significant impact on the functionality and structure of the code.

These decisions were driven by considerations of simplicity, scalability, maintainability, and user experience.

1. Object-Oriented Design (Inheritance and Polymorphism)

- **Decision:** To create a flexible and extensible system, we opted for an object-oriented approach, using inheritance and polymorphism to structure the various games.
- **Rationale:** The game system needed to support different types of games, each with distinct rules and logic (e.g., math challenges, anagrams, hangman). Rather than writing separate logic for each game in isolation, we designed a base Game class with a virtual play() method. Each specific game (e.g., MathChallenge, Anagrams, Hangman) inherits from the Game class and implements the play() method.
- **Impact:** This design allows the addition of new games in the future with minimal changes to the existing codebase. By relying on inheritance, the system can scale easily by adding new types of games without disrupting the overall structure.

2. Player Class Design

- **Decision:** The Player class was designed to handle user registration, authentication, and tracking of badges earned through games.
- **Rationale:** We wanted a centralized structure to manage player data and ensure that each player's progress (e.g., badges earned) could be saved and accessed consistently. The Player class includes attributes for storing the player's username, password, and badges, as well as methods for authenticating users and adding new badges.
- **Impact:** By centralizing player management in a single class, it became easier to handle sign-ups, log-ins, and badge tracking. This modular approach also allowed for the introduction of features like the leaderboard, where each player's badge count can be accessed in a uniform way.

4. Encapsulation

- **Decision:** We decided to use private attributes with public getter and setter methods to manage access to sensitive data like player passwords and ensure controlled modifications.
- **Rationale:** This enhances security by preventing unauthorized access to sensitive data and ensures changes to data follow defined rules.
- **Impact:** It made the system more secure and modular.

5. User Experience

- **Decision:** We focused on keeping things simple for the user by using a text-based interaction model where players are prompted with clear instructions for each action.
- **Rationale:** Since this is a console-based game system, we wanted to ensure that users would not be overwhelmed by overly complicated interfaces. A clean, simple interface helps players focus on the games and rewards without distraction.
- **Impact:** The simplicity of the interface makes it accessible and easy to navigate. The current design serves its purpose for a basic, console-based game system.

6 . Game Difficulty and Complexity

- **Decision:** The games were kept simple and relatively straightforward in terms of gameplay mechanics.
- **Rationale:** We wanted to ensure the games were not too difficult or time-consuming. This decision focuses on fun and quick interaction, making the games accessible to all players.
- **Impact:** Keeping the games simple ensures that they remain light-hearted and enjoyable. However, it also means the system might not appeal to more hardcore gamers seeking complex challenges.

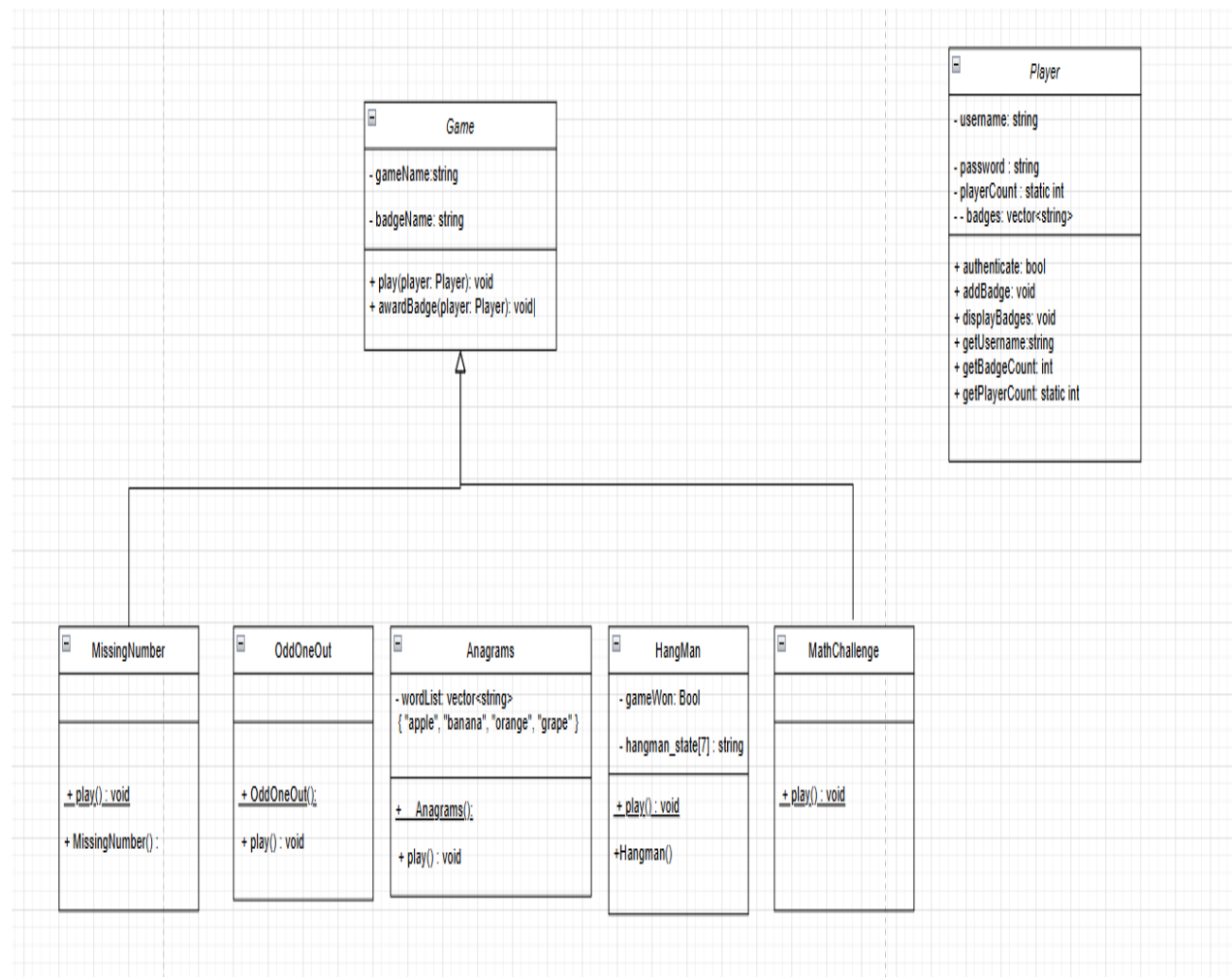
7. Random Game Selection

- **Decision:** To add variety to the user experience, We implemented random game selection, where a game is chosen randomly for each player upon login.
- **Rationale:** By randomly selecting a game from a predefined list, players are exposed to different types of challenges, keeping the experience fresh and engaging. This decision

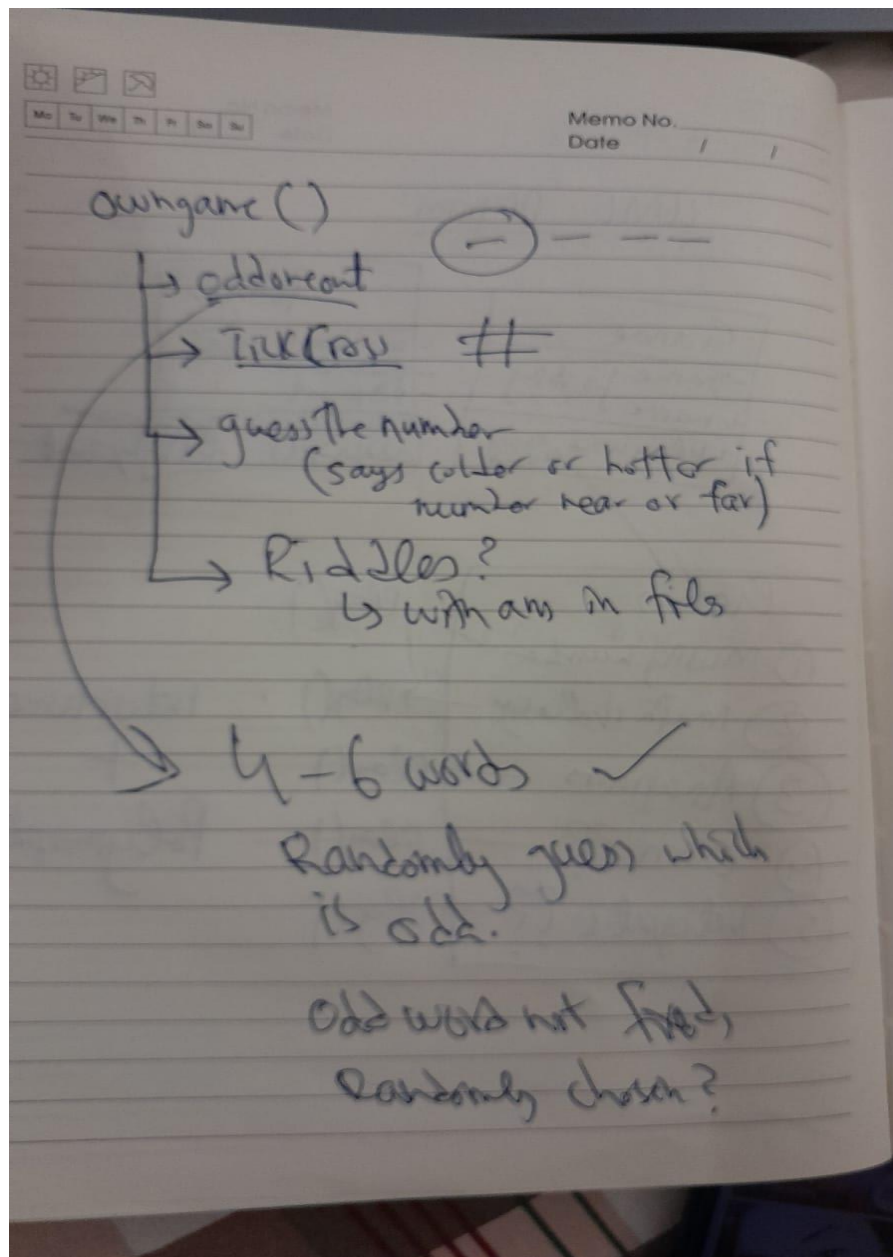
also allows for the easy addition of new games in the future, without altering the structure of the main gameplay loop.

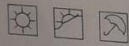
- **Impact:** This simple randomization approach enhances replay value, as players are likely to experience different games each time they sign in, which increases the overall engagement and longevity of the system.

11. UML DIAGRAM



12. Rough Work





Mo Tu We Th Fr Sa Su

Memo No. _____

Date / /

anagram

↳ To take words ~~from~~ and
making jumble

↳ but how?

↳ search

missing sequence

↳ To add lots of missing

seq → ~~words~~ ~~to~~ ~~find~~ with

are already stored.

(~~not~~ ~~making~~ ~~it~~)

Integrate games

- Bilal group
- Am group
- Ayeshah group
- Etasham/umar

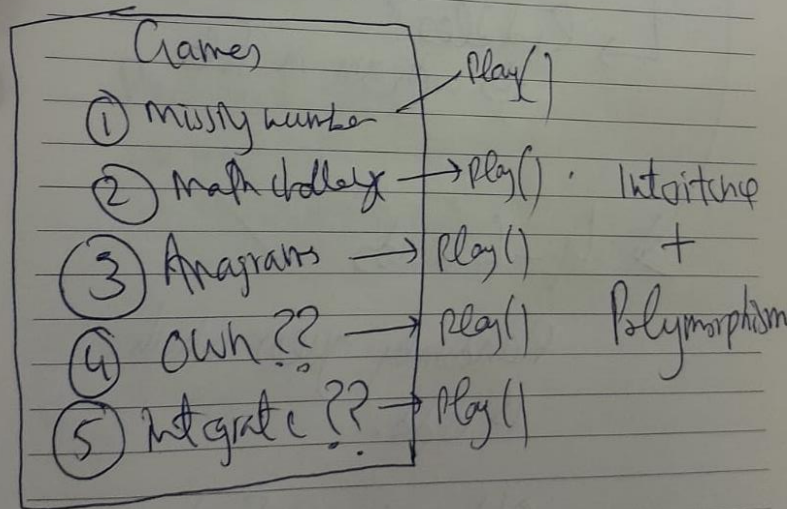
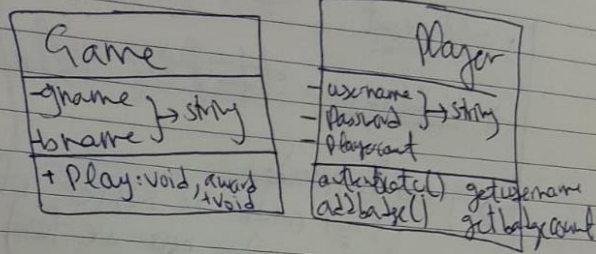
Predefined game

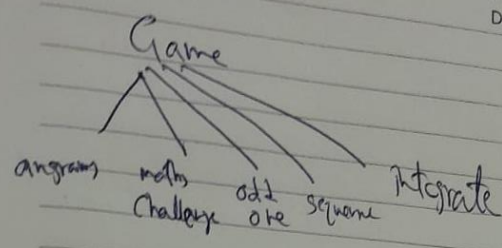
- math challenge → simple (+ -) ✓
- missing sequence → ✓
- anagrams ✓

math challenge

- generate 2 rnd numbers
- operator to make list of operators $(+, -, *, \div)$ which then chooses as Random from list?

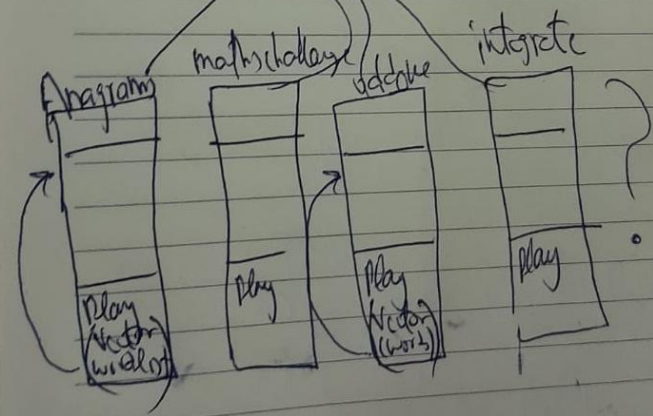
UML Diagram





| Game | |
|-----------|--|
| title | |
| Play Game | |

| Plays | |
|--------------|-------|
| user | count |
| password | |
| brides | |
| authenticate | |
| add bridge | |
| display | |



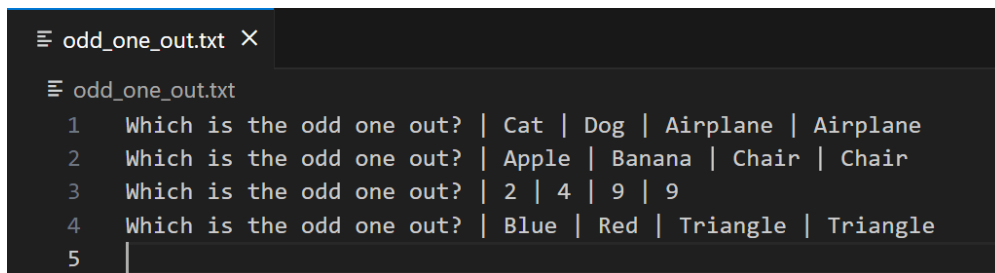
13. Update

The suggested updates were the following:

- The game of Odd one out should take statement and options from a text file instead of in the code.
- The game should save the previous user data in a text file.

1. Game Improvement

The game of Odd one out should take the statement and options from a text file



```
≡ odd_one_out.txt X
≡ odd_one_out.txt
1 Which is the odd one out? | Cat | Dog | Airplane | Airplane
2 Which is the odd one out? | Apple | Banana | Chair | Chair
3 Which is the odd one out? | 2 | 4 | 9 | 9
4 Which is the odd one out? | Blue | Red | Triangle | Triangle
5 |
```

The format is following

- Statement | Option1 | Option2 | Option3 | Correct Option

Code for this implementation is following:


```

OddOneOut.cpp > play()
6 void OddOneOut::play() {
7     std::ifstream file("odd_one_out.txt");
8     if (!file.is_open()) {
9         std::cout << "Could not open the questions file.\n";
10        return;
11    }
12
13    std::string line, statement, option1, option2, option3, correctOption;
14    std::vector<std::string> options;
15
16    std::cout << "Odd One Out Game:\n";
17
18    while (getline(file, line)) {
19        std::istringstream iss(line);
20        getline(iss, statement, '|');
21        getline(iss, option1, '|');
22        getline(iss, option2, '|');
23        getline(iss, option3, '|');
24        getline(iss, correctOption, '|');
25
26        std::cout << statement << "\n";
27        std::cout << "1. " << option1 << "\n";
28        std::cout << "2. " << option2 << "\n";
29        std::cout << "3. " << option3 << "\n";
30
31        int userChoice;
32        std::cout << "Enter your choice (1-3): ";
33        std::cin >> userChoice;
34
35        std::string userAnswer;
36        if (userChoice == 1) userAnswer = option1;
37        else if (userChoice == 2) userAnswer = option2;
38        else if (userChoice == 3) userAnswer = option3;
39        else {
40            std::cout << "Invalid choice.\n";
41            continue;
42        }
43    }

```

Figure 1: Odd one out.cpp

2. Save User Data

The program should save the user data, number of badges, and name of all the badges he won in a text file. Implemented is following:

```

odd_one_out.txt  userdata.txt X
userdata.txt
1  huda,3|Math Badge|Anagrams Badge|Missing Number Badge
2  moiz,1|Math Badge
3  danyal,0
4

```

The format is following

- Username, Number of badges | Badge1 | Badge2 | Badge3 | Badge4

Code for this implementation is following:

```

void savePlayerData(const std::vector<Player>& players) {
    std::ofstream file("userdata.txt", std::ios::out); // Open file in overwrite mode
    if (!file.is_open()) {
        std::cerr << "Error: Could not open userdata.txt for writing.\n";
        return;
    }

    for (const auto& player : players) {
        file << player.getUsername() << "," << player.getPassword(); // Save username and password
        for (const auto& badge : player.getBadges()) {
            file << "|" << badge; // Save badges separated by '|'
        }
        file << "\n";
    }

    file.close();
}

```

Figure 2: Player.cpp