Project Title: Final Elementals

Use Case Description:

Final Elementals is a classic turn-based RPG game set in a fantasy world where the elemental kingdom is on the brink of chaos. Players can play as an elemental hero, representing either one of the three elements: fire, water, and earth.

Prepare your hero and embark on a journey to restore balance to the elemental realms. Battle fierce monsters and master the powers of the elements to save the world from impending doom.

The game provides basic functions to heroes like random element attack and dodge against the enemy attack. Enemies are imbued with elemental properties, Players have to make a decision to make use of player health to guarantee the ultimate attack on the enemy or taking a chance in using elemental attack against the enemy that can be weak or strong against the enemy. The victory is determined based on the player taking advantage of the ultimate attack with the cost of health or random element attack.

The game has three stages with normal enemies and bosses. Win the first stage to progress to the next stage.

The project's goal is to develop a functional turn-based RPG game that allows players to use simple strategies and a bit of luck to progress.

// The value below is an example and will expect to change during coding.

- **A player (User)**
    - Player default attribute:
        - Health point: 100 —> 200 —> 300 as the stage progress respectively
        - Skill Point: 40
        - Attack value: 20
    - Player Action: (user have to decide these 3 action below)
        - Dodge: 50-50 chance to dodge (hit or miss). Recommend use when enemy about to use ultimate attack (boss)
        - Elemental skill (randomly chosen either water, fire, earth and consume 10 sp respectively): water (consume 10sp), fire (consume 10sp), and earth (consume 10sp) to cast. Attack imbued with element and scale with attack value.
        - Ultimate Attack: Consume 20 hp of the player to deal fixed 50 Damage to the enemy (Guaranteed Hit)
- **Enemy**
    - Basic enemy attribute:
        - Health point: 20 —> 40 —> 60 as the stage progress respectively
        - Element type: Fire (hidden to player)

        - Attack value: 5
    - Basic enemy action:
        - element basic attack: deal dmg to player scale with attack value.
    - Boss enemy attribute:
        - Health point: 100 —> 200—> 300 as the stage progress respectively
        - Element type: Fire (hidden to player)
        - Attack value: 20
    - Boss enemy action (Computer act randomly):
        - Element basic attack: deal dmg to player scale with attack value.
        - Ultimate: deal massive damage to players (will warn the incoming attack).

- element gameplay: water -> fire -> earth -> water
    - **Example scenario 1:** fire enemy (Normal)
        - Players use water against fire enemies, enemies take 100% of 20 attacks of player attack value. Players use everything else (fire, earth) against fire enemies, the enemy will take 20 attacks of player / 2 = 10 attacks.

Class List:

1. **Game_menu class:** This class is a parent class which has level, start, or finish features.
2. **Game_Stage class:** It is inherited from the main class, and serves as a framework for different levels of games. Each stage has a different enemy and boss with a set value to determine the difficulty of the level.
3. **Entity Class:** It is a base class which represents a character attribute such as Name, info and state attributes such as health, skill point, and attack value.
4. **Enemy Class:** It is inherited from entity_base class and includes the attributes and features of the enemy like attack action and element types.
5. **Normal_enemy class:** This is a subclass from enemy_class and it includes basic enemy details and functions and basic attributes. The normal enemy class is only able to do basic attack action.
6. **Boss_enemy class:** This is a subclass from enemy_class and it includes a very powerful enemy with very advanced attributes and features. Boss can do basic attack and ultimate attack randomly.
7. **Player class:** It is inherited from entity_base class and includes the attributes and features of player like points,hp and types. Player which is the user and interacts and chooses different types of action to fight against the enemy.
8. **Save_Game class:** It is used to save the game information on a file. It’s inherited from the game_menu class.

Data and function Members:

- **Game_Menu class:**
  - Attribute:
    - StartOrExit: bool //Return 1 for start and Return 0 for exit
    - stageMenu: int (stage 1,2,3) //Number of stages
    - Game_Stage: //Create a stage or level of the game
  - Function:
    - void set_startExitGame (startOrExitGame : bool) //Ask player to start or exit
    - void set_stageMenu (stageMenu : int) //Ask player to select which stage to play
    - bool GamePlay() //Use a while loop to repeat the player and enemy action and calculate the battle status and condition to determine the winner and loser. Return 1 if the player wins and 0 if the enemy wins.
    - get_startExitGame(); //return startExit value
    - get_stage(); //return stage number

- **Game_Stage class:**
  - Attribute:
    - level_of_game: int
    - *Entity: player
    - **Entity: enemy //as an idea we can create an array of entities, for example level 2, arrays of 2 enemies (basic and boss and player needs to kill both of them).
  - Function:
    - add_player (Player) //add player or enemy
    - add_enemy (Enemy) //add enemy
    - get_player() //return player class
    - get_enemy() //return player class
    - Get level_of_game : int //return level of stage

- **Entity class:**
  - Attribute:
    - Name: string //name of entity
    - Info: string //info of entity
    - health_point: int //health point of entity
    - skill_Point: int //skill point to use skill and ultimate
    - attack: int //value of attack that dealt to enemy
  - Function:
    - Int get_Health(); //return health value
    - Int get_SkillPoint(); //return skill point value
    - Int get_Attack(); //return attack value
    - Int get_Element(); //return element of the entity
    - String get_name(); //return name of the entity
    - String get_info(); //return the info of entity

        - Virtual int ActionResult() = 0; //return the action result of the entity.

- **Player class:**
    - Attribute:
        - playerActionOption : int //player option to choose to act
    - Function:
        - Int get_playerOption(); //return player option
        - Player (Name : string, Info : string, health : int, skillPoint : int, attack : int, element : string) //create player class
        - Void set_playerActionOption (playerOption : int) // prompt for player option of action (attack, ult, skill, def)
        - int ActionResult (); //return value of damage dealt by player or dodge action
        - get_ActionResult //return player action result function value

- **Enemy class:**
    - Attribute:
        - EnemyOption: int //enemy option
        - element : int //they have a name but they are integer
    - Function:
        - void set_EnemyAction(EnemyOption); //can use rand() to randomise action
        - Int get_enemyOption(); //return enemy option

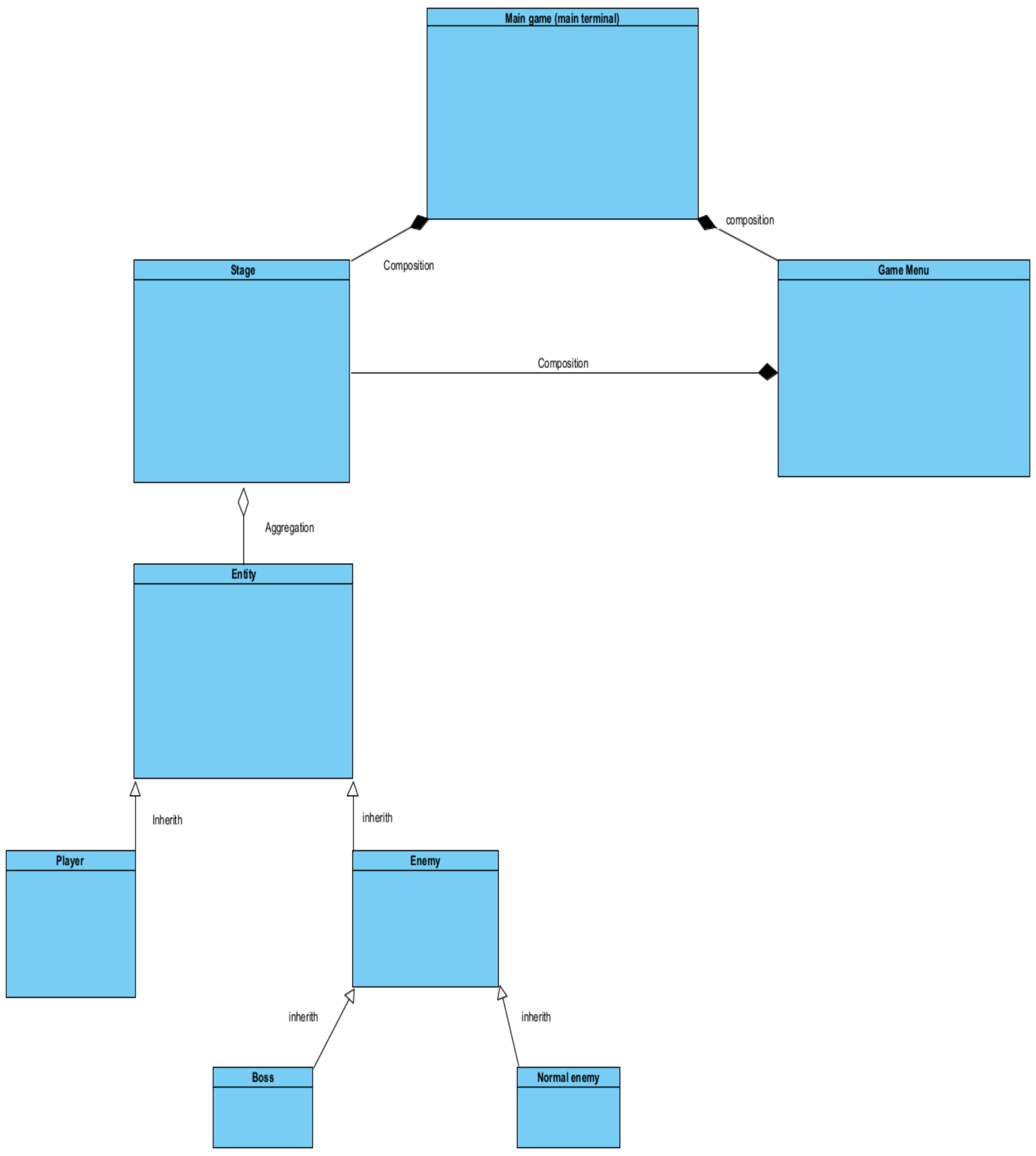
- **Normal_Enemy class:**
    - Attribute:
    - Function:
        - normalEnemy (Name : string, Info : string, health : int, skillPoint : int, attack : int, element : string) //create normal enemy sub-class
        - int ActionResult (); //only for basic attack
        - get_ActionResult //return normal enemy action result function value

- **Boss_Enemy class:**
    - Attribute:
    - Function:
        - bossEnemy (Name : string, Info : string, health : int, skillPoint : int, attack : int, attack : int, element : string) //create boss enemy subclass

        - int ActionResult (); //basic and ult
        - get_ActionResult //return boss action result function value

Relationships between Classes:

The relationship between classes is shown in UML below.



- Player and Enemy classes are inherited from entity classes.
- Entity class has an aggregation relationship with game_stage class.
- Game_menu class and game_stage class have a composition relationship.
- Normal_enemy class and boss_class both are inherited from enemy class.

Project Task List and Timeline:

1. Design the class structure and relationships. (2 days) (Sophana, Ajay Ramesh, Mohammed Ali)
2. Implement the Game menu for user options. (2 days) (Mohammed Ali)
3. Implement Entity class and its derived class which are player and enemy sub class. (1 day) (Sophana)
4. Implement the stage by adding player and enemy, including the gameplay between user interaction (player) versus enemy(Computer). (2 days) (Ajay Ramesh)
5. Run the whole program includes all the classes, execute and test the program (1 day)(Sophana, Ajay Ramesh, Mohammed Ali)
6. Implement the user interface(terminal) menu, using CIN and COUT(1 day) (Sophana, Ajay Ramesh, Mohammed Ali)
7. Write unit tests for all classes and functions. (2 days) (Mohammed Ali)
8. Debug and refine the code. (1 day) (Sophana, Ajay Ramesh, Mohammed Ali)
9. Write user documentation including the error handling. (2 days) (Sophana, Ajay Ramesh, Mohammed Ali)

User Interaction Description:

The terminal is used as a way of interaction for users with programs. When the game is launched, the terminal displays various options to the player through a menu. The player can start the game, select a stage, or exit the game.

The terminal will print out the options and the user can select from the menu option. Players interact with the game by typing their inputs into the terminal. These inputs include choices like selecting actions during gameplay or navigating the game menu.

All inputs must be entered in capital letters. This requirement ensures consistency and prevents input errors related to case sensitivity. If an incorrect input is detected, the terminal will display an error message along with instructions for correcting the mistake. This helps guide the player on how to proceed correctly.

Alongside real-time error messages, the game provides a booklet that contains helpful information on common errors and troubleshooting tips. This booklet is designed to assist players in understanding the game's input system and resolving issues independently.

Unit Testing and Debugging Plan:

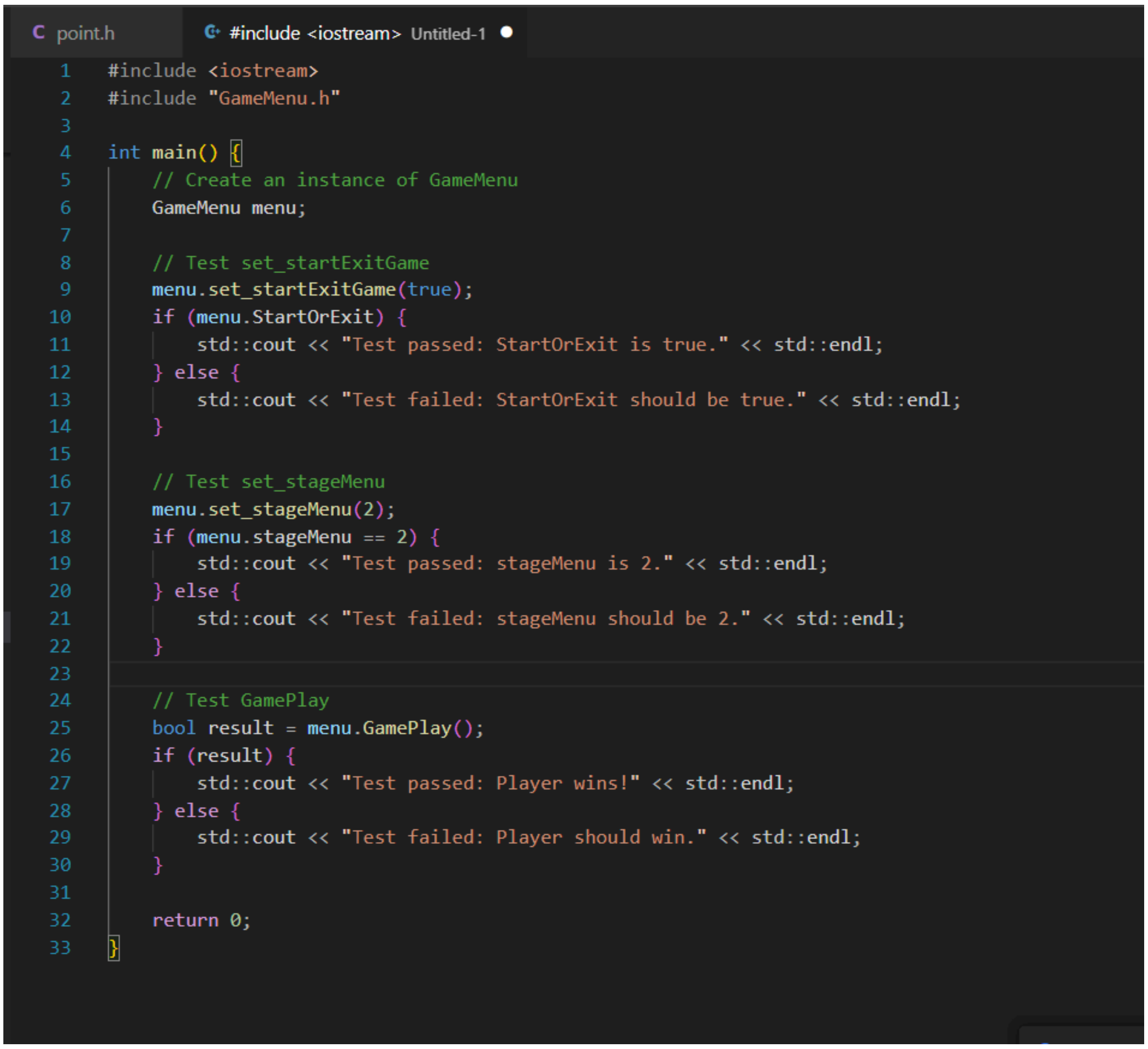
Unit Testing:

Testing the program is very crucial and must be inclusive of testing all aspects of the program, every single component.

In this program we need to test every single component separately to make sure we have the right outcome. Different methods used to test the game such as a test set , covering different scenarios (edge cases, invalid inputs and …), using the testing framework like GTest.

For example, To test the functionality set_startOrExit() and other functions of game_menu class, we use the following unit testing code set. In the test compare the desired outcome with the generated outcome; if the answer is not true, an error will be generated. The code is written below:

Test_set code for game_menu class



```cpp
#include <iostream>
#include "GameMenu.h"

int main() {
    // Create an instance of GameMenu
    GameMenu menu;

    // Test set_startExitGame
    menu.set_startExitGame(true);
    if (menu.StartOrExit) {
        std::cout << "Test passed: StartOrExit is true." << std::endl;
    } else {
        std::cout << "Test failed: StartOrExit should be true." << std::endl;
    }

    // Test set_stageMenu
    menu.set_stageMenu(2);
    if (menu.stageMenu == 2) {
        std::cout << "Test passed: stageMenu is 2." << std::endl;
    } else {
        std::cout << "Test failed: stageMenu should be 2." << std::endl;
    }

    // Test GamePlay
    bool result = menu.GamePlay();
    if (result) {
        std::cout << "Test passed: Player wins!" << std::endl;
    } else {
        std::cout << "Test failed: Player should win." << std::endl;
    }

    return 0;
}
```

For all the classes, a similar method will apply to make sure they work properly.

Another unit testing for Entity_class:

We create one player in Entity_test.cpp with known information (name, age and ..)

And we write a code similar to above to compare the created information (by using setters and getters) with initial information; if they match “correct” is printed, otherwise “error” will be printed, pseudocode is shown below:

Unit Test for Entity Class:

```
// Define a test function for the Entity class

function test_Entity_class():

    // Create an instance of the Entity class with known information

    entity = new Entity("TestName", "TestInfo", 100, 50, 20)

    // Test the getter methods to ensure they return the correct values

    if entity.get_Name() == "TestName":

        print("Name test passed: correct")

    else:

        print("Name test failed: error")

    if entity.get_Info() == "TestInfo":

        print("Info test passed: correct")

    else:

        print("Info test failed: error")

    if entity.get_Health() == 100:

        print("Health test passed: correct")

    else:

        print("Health test failed: error")

    if entity.get_SkillPoint() == 50:

        print("Skill Point test passed: correct")

    else:

        print("Skill Point test failed: error")

    if entity.get_Attack() == 20:

        print("Attack test passed: correct")

    else:

        print("Attack test failed: error")

// Run the test function

test_Entity_class()
```

More Testing:

1. Write a scenario and expected output then we run the code if they don't match we need to fix the code.

    For example: In this program there are 3 stages and the user needs to enter one number from 1 to 3. It should be tested for invalid input such as 0 or 4 or even different types of characters such as letters; and ,make sure the right errors are generated.

2. Test Coverage:Do as many as different inputs and make sure we get the right output.

    Test the program for all valid inputs (level 1, 2 and 3) and make sure they all work properly.

Debugging:

Debugging is the process of identifying and fixing errors, or "bugs," in software code.
It can be summarised in 5 steps:

1. Find the root cause
2. Understand the problem.
3. Determine a fix.
4. Repair the issue.
5. Retest

We go through the code line by line and try to fix the error parts. Following steps will be taken:

- Go through the code part by part and fix the errors generated by IDE (syntax errors!)
- Using a debugger tool such valgrind or VScode tools to identify issues
- Isolating the error parts of code
- Temporary fixing the code contains error
- Using **cout** to print out different stages in program to make sure it works
- Reviewing the code with team members to double check it

Makefile will be included to automate the run process and README files will be included in the project, which will provide clear instruction on how to run and test the program.
Finally after going through debugging, and we retest the code, the code can be improved and meet our purpose of the project.