Ahmed Abdul Rahim
30609

Dungeon Crawler RPG

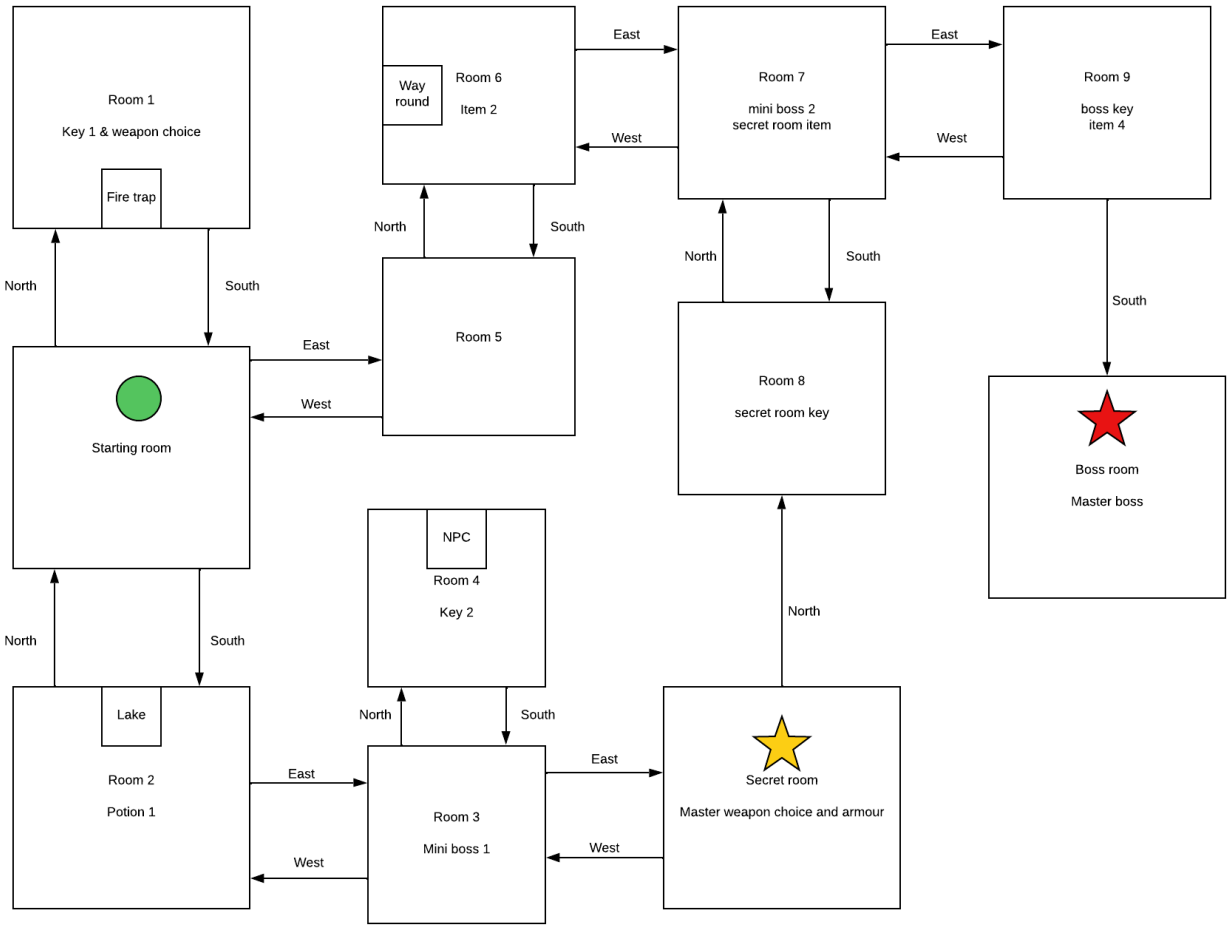
Name: Ahmed Abdul Rahim

ERP: 30609

Course: Object-Oriented Programming

Assignment: Dungeon Crawler RPG (Homework)

Submission Date: 7 June 2025



Ahmed Abdul Rahim
30609

Introduction

This document provides a comprehensive technical overview of the C++ SFML Dungeon Crawler project. It details the software architecture, a class-by-class breakdown of the implementation, and the core features that make up the complete gameplay experience. This documentation is intended for developers, instructors, or anyone interested in understanding the technical design and object-oriented principles used to build the game. The following pages contain a full index of the project's source code and assets, followed by detailed explanations of each component, a summary of how the project fulfills its academic requirements, and a look at potential future improvements.

Project Documentation Index

1. Headers (Headers/)

- CustomQueue.h
    - QueueNode<T> Class
    - CustomQueue<T> Class
        - Public: enqueue(), dequeue(), peek(), isEmpty()
        - Private: front, rear
- CustomStack.h
    - StackNode<T> Class
    - CustomStack<T> Class
        - Public: push(), pop(), peek(), isEmpty()
        - Private: top
- Dungeon.h
    - Dungeon Class
        - Public: startRoom, currentRoom, backtrackStack, enemyQueue, bossRoom, itemEffects, Dungeon(), enterRoom(), backtrack(), combat(), useItem(), handleTraps(), displayVictoryScreen(), setGUI()
        - Private: gui
- Enemy.h
    - Enemy Class (inherits from Entity)

        - Public: attack1, attack2, sprite, texture, hurtTexture, Enemy(), attack(), getSprite(), update(), triggerHurtEffect()
        - Private: animationClock, animationSpeed, currentFrame, frameWidth, frameHeight, isHurt, hurtClock, hurtDuration, originalPosition
- Entity.h
    - Entity Class (Abstract Base Class)
        - Public: name, hp, Entity(), ~Entity(), takeDamage(), isAlive(), attack() (pure virtual), update()
- Game.hpp
    - Game Class
        - Public: Game(), run()
        - Private: gui, dungeon, player, GameState enum, currentState, processEvents(), update(), render()
- GameGUI.hpp
    - GameGUI Class
        - Public: GameGUI(), run(), setRoomInfo(), setPlayerStats(), setInventory(), pushMessage(), clearMessages(), setEnemySprite(), clearEnemySprite(), isWindowOpen(), getLastKeyPressed(), handleInput(), render(), drawStartScreen(), drawRulesScreen(), drawGameOverScreen(), drawVictoryScreen(), close()
        - Private: window, font, all sf::Text objects, all internal state strings/vectors, lastKey, vectorToString()
- Player.h
    - Player Class (inherits from Entity)
        - Public: stamina, inventory, equippedWeapon, Player(), restoreHealth(), addItem(), hasItem(), displayStats(), viewInventory(), attack(), update()
- Room.h
    - Room Class
        - Public: description, sound, smell, enemy, north, south, east, west, fireTrap, waterChallenge, vineTrap, item, Room()

---

2. Main (main/)

- Main.cpp

- main() function: Entry point of the program. Creates a Game object and calls game.run().

---

3. Source Code (Source/)

- Dungeon.cpp: Implementation of Dungeon class methods.
- Enemy.cpp: Implementation of Enemy class methods.
- Entity.cpp: Implementation of Entity class methods.
- Game.cpp: Implementation of the main Game class and its game loop.
- GameGui.cpp: Implementation of GameGUI class methods for SFML rendering and input.
- Player.cpp: Implementation of Player class methods.
- Room.cpp: Implementation of Room class methods.

---

4. Assets (assets/)

- **Images:**
  - goblin.png
  - skeleton.png
  - elder.png
  - hurt.png
- **Fonts:**
  - LEMONMILK-Medium.ttf

EXPLAINATION

CustomStack.h

The CustomStack.h header file defines a generic, template-based Stack data structure implemented as a singly linked list. By using templates (template <typename T>), this custom stack is made versatile, allowing it to store any data type needed within the project.

Ahmed Abdul Rahim
30609

Its underlying structure consists of StackNode objects, where each node contains a piece of data and a shared_ptr to the next node in the stack. The use of std::shared_ptr is a key feature, providing modern C++'s automatic memory management to prevent memory leaks. The entire stack is managed through a single private member: a shared_ptr named top, which always points to the most recently added element. The class provides all the standard stack functionalities with LIFO (Last-In, First-Out) behavior. The push() method adds a new node to the top of the stack. The pop() method removes the top node, returning its value, and includes a check to prevent errors on an empty stack. Additionally, peek() allows for inspection of the top element's data without removing it, while isEmpty() provides a simple boolean check to see if the stack contains any elements.

CustomQueue.h

The CustomQueue.h header file defines a generic, template-based Queue data structure, also implemented using a singly linked list architecture. As a template class, it can be instantiated to hold any data type required by the application. Each element in the queue is a QueueNode that holds a data value and a shared_ptr to the subsequent node, leveraging smart pointers for robust memory management. Unlike the stack, the CustomQueue class manages its list using two private shared_ptr members: front and rear. This two-pointer system allows for efficient, constant-time additions to the end of the queue. The class implements standard FIFO (First-In, First-Out) behavior through its public methods. The enqueue() method adds a new element to the rear of the queue. The dequeue() method removes an element from the front. The peek() function allows for viewing the data of the front element without modifying the queue, and isEmpty() checks if the queue is empty by verifying if the front pointer is null.

Entity.cpp (abstract Base Class Implementation)

This file provides the implementation for the base Entity class. It contains the core, shared logic that applies to every entity in the game.

- **Entity::Entity(const std::string& n, int h)**: The constructor simply initializes the name and hp of the entity using a member initializer list.
- **void Entity::takeDamage(int dmg)**: This function reduces the entity's hp by the given damage amount. It includes a check to ensure that hp cannot go below 0.
- **bool Entity::isAlive() const**: A straightforward function that returns true if the entity's hp is greater than 0, and false otherwise.
- **void Entity::update()**: This is the default implementation for the virtual update function. It is intentionally left empty. This means that if a derived class (like Player) does not provide its own update method, this safe, empty version will be called instead of causing an error.

Ahmed Abdul Rahim
30609

Player.cpp (Derived Class Implementation)

This file implements the specific functionalities for the Player class.

- **Player::Player()**: The constructor uses an initializer list to call the Entity base class constructor, setting the player's name to "Player" and starting health to 200. It also initializes player-specific stats like stamina and equippedWeapon and adds a starting item ("Arrow") to the inventory.
- **void Player::attack(Entity& target)**: This function is an override of the pure virtual attack function from Entity. Its body is **empty**. This is because the player's attack logic is uniquely handled by the QTE (Quick Time Event) system inside Dungeon::combat. However, the function must be defined here to fulfill the contract of the Entity base class, allowing the Player class to be a concrete, instantiable object.
- **void Player::update()**: Similar to the attack function, this is an override that is intentionally left empty, as the player character does not require continuous animation updates.
- **Other Methods**: Functions like restoreHealth(), addItem(), hasItem(), and displayStats() are all unique to the Player class and provide the logic for managing inventory, health, and stats.

Enemy.cpp (Derived Class Implementation)

This file contains the rich implementation for the Enemy class, focusing on its combat AI, graphics, and animations.

- **Enemy::Enemy(...)**: The constructor for the enemy is more complex.
    - It calls the Entity base constructor to set its name and health.
    - It loads two sf::Texture files: one for the enemy's main appearance and one for its "hurt" effect. It prints an error to std::cerr if a texture fails to load.
    - It sets up the sf::Sprite, including its scale, origin point (for rotation and positioning), and its initial position on the screen.
    - It configures animation parameters like animationSpeed and frame dimensions.
- **void Enemy::attack(Entity& target)**: This is the enemy's concrete implementation of the polymorphic attack function. When called, it calculates damage by randomly choosing between its attack1 and attack2 values, prints the result to the console, and calls the takeDamage() method on its target.
- **void Enemy::update()**: This overridden function contains all the animation logic for the enemy.

  - It first checks an isHurt flag. If the enemy was recently damaged, it will display the hurtTexture and jitter the sprite's position for a short duration (hurtDuration).
  - If isHurt is false, it performs a simple two-frame idle animation by cycling currentFrame and updating the texture rectangle of the sprite periodically, based on the animationClock.
- **void Enemy::triggerHurtEffect()**: This is a helper function that sets the isHurt flag to true and restarts the hurtClock, initiating the hurt animation in the update loop.

GameGui.cpp Implementation

Constructor: GameGUI::GameGUI(...)

This is the primary setup function for the entire graphical interface. When a GameGUI object is created, the constructor performs several crucial tasks:

- It creates the main application window with a specific width and height.
- It loads the font file (LEMONMILK-Medium.ttf) from the assets folder. If the font fails to load, it throws a std::runtime_error to prevent the game from running without its necessary resources.
- It initializes all the sf::Text objects that will be used throughout the game. This includes setting the font, character size, color, and position for:
  - The main gameplay HUD (room info, player stats, inventory, message box).
  - All menu and state screens (menuTitleText, rulesBodyText, gameOverText, and victoryText).

Screen Drawing Functions

These functions are responsible for rendering the different full-screen game states, like menus and end screens. Each one follows a simple pattern: clear the screen, draw the specific text for that state, and display the result.

- drawStartScreen(): Renders the initial screen with the game title and a prompt to start.
- drawRulesScreen(): Displays the title "Game Rules" and the multi-line text explaining how to play.

- drawGameOverScreen(): Displays the "You Have Been Defeated" message in red when the player loses.
- drawVictoryScreen(): Displays the "You have Escaped!" message in green when the player wins.

Core Interface and Gameplay Loop

These methods form the main interface for the Game class to control the GUI.

- handleInput(): This function polls the SFML window for events. It specifically listens for the sf::Event::Closed event (to close the window) and the sf::Event::KeyPressed event, storing the code of the last key pressed.
- getLastKeyPressed(): This allows the main game logic to retrieve the last key that was pressed. After returning the key, it resets the internal variable to sf::Keyboard::Unknown to ensure the same key press isn't processed multiple times.
- render(): This is the main drawing function for when the game is being played. It draws the complete gameplay HUD, including room descriptions, player stats, the inventory list, the message log, and the current enemy's sprite if one is active.
- close(): A simple public method that calls the window's internal close() function, allowing the main Game class to safely terminate the application.

Data and State Management Functions

These are "setter" methods and helpers that allow the rest of the program to update what the GUI displays.

- setRoomInfo(), setPlayerStats(), and setEnemySprite(): These functions take data from other parts of the program (like Dungeon and Player) and store them in the GameGUI's private members, ready to be drawn by the render() function.
- setInventory(): This method receives the player's inventory as a vector of strings. It then uses std::sort to alphabetically organize the inventory before displaying it.
- pushMessage(): This function adds a new message to the message log. To keep the log from becoming cluttered, it ensures that only the last four messages are stored and displayed.
- vectorToString(): A private helper method that converts a vector of strings into a single string with newlines, making it easy to display the inventory and message log.

Room.cpp (Room Implementation)

This file contains the implementation for the Room class. A Room object acts as a node in the dungeon map, holding all the information about a specific location.

Ahmed Abdul Rahim
30609

Constructor: Room::Room(...)

This is the only function in the file. Its purpose is to initialize a new Room object to a clean, default state using a member initializer list.

- It takes the room's description, sound, and smell as arguments and assigns them to the corresponding class members.
- All shared_ptr members (enemy, north, south, east, west) are initialized to nullptr. This means that when a room is first created, it is isolated and has no connections to other rooms.
- All boolean trap flags (fireTrap, waterChallenge, vineTrap) are initialized to false.
- The item member is initialized to an empty string ("").

This default setup ensures that every room starts as a safe, empty, and unconnected space. The connections, items, and enemies are then configured manually within the Dungeon class constructor.

Dungeon.cpp (Game World and Logic Implementation)

This file contains the implementation for the Dungeon class. It is responsible for building the world, defining game rules, and handling all major player interactions like movement, combat, and item usage.

Constructor: Dungeon::Dungeon()

This constructor's primary role is to build the entire game level from scratch when the Dungeon object is first created.

- **Room Creation**: It begins by creating every Room in the game as a std::shared_ptr, giving each one a unique description, sound, and smell.
- **Map Generation**: It then meticulously connects all the rooms by setting their north, south, east, and west pointers. This hard-coded process forms the static map layout of the dungeon.
- **Item Placement**: It places items throughout the dungeon by assigning an item's name as a string to the item member of specific rooms.
- **Enemy Placement**: It populates the dungeon with enemies by creating instances of the Enemy class and assigning them to the enemy member of specific rooms.
- **Trap Placement**: It sets the boolean flags (fireTrap, waterChallenge, etc.) to true for specific rooms to activate their traps.

- **Lambda-Based Item Effects**: In a key demonstration of modern C++ features, it populates the itemEffects map. For each usable item, it defines a **lambda function** that encapsulates the item's effect (e.g., restoring health, boosting stats) and assigns it to the map with the item's name as the key.

Dungeon::combat(...)

This function manages the entire turn-based combat system.

- **Initiation**: It starts by displaying a "BATTLE START" message and setting the enemy's sprite in the GUI.
- **Fight or Run**: It enters a loop presenting the player with the choice to either fight or run.
    - **Run Logic**: If the player chooses to run, a random chance determines success. On success, the enemy is added to the enemyQueue to act as a "guard," and the combat ends. On failure, the player takes damage, and the enemy attacks.
- **QTE Attack Mechanic**: If the player chooses to fight, the main combat loop begins. In this loop, a random sequence of WASD keys is generated and shown to the player. The player must input the correct sequence to land a successful hit. The damage dealt is proportional to the length of the combo.
- **Turn Flow**: After the player's turn, the enemy performs its own attack if it is still alive. This continues until either the player's or the enemy's health drops to zero.
- **Post-Combat Cleanup**: After the fight, if the player won and the defeated enemy was the "guard" at the front of the enemyQueue, it is dequeued.

Dungeon::enterRoom(...)

This function handles all logic related to player movement.

- **Validation**: It first checks if the desired direction is a valid path (i.e., not a nullptr).
- **Key Restrictions**: It then checks if the destination room requires a specific key by using the player.hasItem() method. If the player lacks the required key, movement is denied.
- **Backtrack and Trap Handling**: If movement is valid, it pushes the previous room onto the backtrackStack and calls handleTraps() to trigger any traps in the new room.
- **Enemy Encounter Logic**: It checks if the new room contains an enemy. If there is a "guard" in the enemyQueue, it will only allow combat if the new room's enemy is the correct guard; otherwise, it blocks the player. If there is no guard, it initiates a standard combat encounter.

- **Item Pickup**: After all other events, if the room contains an item, it prompts the player to pick it up.

Dungeon::useItem(...)

This function implements the flexible, lambda-based item usage system.

- It displays a numbered list of usable items from the player's inventory.
- It waits for the player to select an item with a number key.
- It uses the name of the selected item to find the corresponding lambda function stored in the itemEffects map.
- It executes the found lambda, which applies the item's specific effect to the player.

Dungeon::handleTraps(...)

This function contains the logic for all the interactive traps in the game.

- It has separate if blocks for each trap type (fire, water, vine).
- For each trap, it presents the player with choices, calculates the outcome based on random chance or item possession, and applies the consequences (damage or stamina loss).
- Once a trap is resolved, its boolean flag is set to false so it does not trigger again.

Dungeon::backtrack(...)

This function executes the backtrack mechanic.

- It pops the previous room from the backtrackStack.
- It moves the player to that room and reduces their stamina.
- It then checks if the room the player returned to is guarded by an enemy in the enemyQueue and initiates combat if necessary.

Dungeon::displayVictoryScreen()

This function's sole responsibility is to print the large, colorful ASCII art victory message to the console when the game is won.

Game.cpp (Main Game Engine Implementation)

Constructor: Game::Game()

- The constructor initializes the Game object by using a member initializer list.
- It creates the GameGUI object with a window size of 1000x800 pixels.

- It sets the initial game state to GameState::StartScreen.
- Crucially, it calls dungeon.setGUI(&gui), passing a pointer of the gui object to the dungeon. This allows the Dungeon class to communicate with the GUI (e.g., to push messages to the screen).

Game::run()

- This is the main public method that starts the game after the Game object has been created.
- It contains the primary while (gui.isWindowOpen()) loop, which keeps the application running until the window is closed.
- Inside the loop, it repeatedly calls processEvents() to handle all per-frame operations.

Game::processEvents()

- This private function orchestrates all the actions that occur in a single frame of the game.
- It begins by calling gui.handleInput() to check for user input and then gui.getLastKeyPressed() to get the most recent key press.
- It checks if the Escape key was pressed and calls gui.close() to terminate the game if it was.
- It calls the update(key) method, passing the user's input to update the game's logic and state.
- After the logic is updated, it calls the render() method to draw the current state to the screen.
- Finally, it pauses for 100 milliseconds using std::this_thread::sleep_for to manage the frame rate and reduce CPU usage.

Game::update(sf::Keyboard::Key key)

- This function contains all the game's logic and state management, driven by a switch statement based on the current GameState.
- **Menu States (StartScreen, RulesScreen):** In these states, the function checks for an Enter key press to transition the currentState to the next screen (from Start to Rules, and from Rules to Playing).
- **Gameplay State (Playing):** This is the most complex state.
    - It first checks for loss conditions (!player.isAlive() or player.stamina <= 0) or the win condition (!dungeon.bossRoom->enemy->isAlive()) and changes the currentState to GameOver or Victory accordingly.

    - It then uses a nested switch statement to handle player actions based on the keyboard input (W, A, S, D, B, U), calling the appropriate methods in the dungeon object.
    - After the player's action, it calls the update() method on the current enemy to process animations.
    - It finishes by sending the latest player stats and inventory to the GUI to ensure the display is up-to-date.
- **End States (Victory, GameOver):** These cases are empty, as no further game logic needs to be updated once the game has ended.

Game::render()

- This function's sole responsibility is to handle drawing operations.
- It uses a switch statement based on currentState to determine which screen to draw.
- Each case makes a single call to the appropriate drawing function in the GameGUI class (e.g., gui.drawStartScreen(), gui.render(), gui.drawVictoryScreen(), etc.), cleanly separating the rendering logic from the game logic.

Fulfillment of Assignment Requirements

This section details how the project successfully meets the specified academic requirements, with code snippets provided as evidence.

Core Object-Oriented Programming

- **Inheritance & Polymorphism:** The project is built on an inheritance hierarchy. The Player and Enemy classes both inherit publicly from the Entity class, inheriting its base properties and behavior.

/ In Player.h

class Player : public Entity { ... };

// In Enemy.h

class Enemy : public Entity { ... };

- **Abstract Classes & Pure Virtual Functions:** The Entity class is an abstract base class. It cannot be instantiated directly because it contains a pure virtual function attack(). This enforces a contract that any derived class *must* provide its own implementation of attack(), which is a core tenet of polymorphism.

// In Entity.h

virtual void attack(Entity& target) = 0;

This is fulfilled by the child classes:

// In Enemy.cpp

void Enemy::attack(Entity& target) { ... }

Data Structures & Algorithms

- **Stack:** A custom, template-based CustomStack class was implemented from scratch. It is used in the Dungeon class to power the game's backtrack mechanic, storing a history of visited rooms.

// In Dungeon.h

CustomStack<std::shared_ptr<Room>> backtrackStack;

// In Dungeon.cpp

backtrackStack.push(previousRoom);

- **Queue:** A custom, template-based CustomQueue class was implemented. It is used to manage the "guard" mechanic for enemies. When a player flees combat, the enemy is added to the queue, ensuring major fights happen in a specific order.

// In Dungeon.h

CustomQueue<std::shared_ptr<Enemy>> enemyQueue;

// In Dungeon.cpp

enemyQueue.enqueue(enemy);

Advanced C++ Features

- **Templates:** Both the CustomStack and CustomQueue are generic classes built using templates, allowing them to store any data type.

Ahmed Abdul Rahim
30609

```
// In CustomStack.h

template <typename T>

class CustomStack { ... };
```

- **Exception Handling:** The GameGUI constructor uses a try...throw block to handle critical errors. If the required font file cannot be loaded, it throws a std::runtime_error, which safely terminates the program and prevents it from running in a broken state.

```
// In GameGui.cpp

if (!font.loadFromFile("../assets/LEMONMILK-Medium.ttf")) {

    throw std::runtime_error("Failed to load font!");

}
```

- **Lambdas:** The item usage system is powered by a std::map that links an item's name to a lambda function. This provides a highly flexible and scalable way to define item effects directly where the dungeon is created.

```
// In Dungeon.cpp

itemEffects["Potion of Copeium"] = [&](Player& player) {

    player.hp = 200;

    if (gui) gui->pushMessage("You drank the Potion. Health fully restored.");

    //...

};
```

---

1. Class Inheritance Diagram

This diagram illustrates the "is-a" relationship between your core gameplay classes. The Player and Enemy classes are both specialized types of the Entity class.

Ahmed Abdul Rahim
30609

Explanation:

- **Entity** is the abstract base class. It defines the core attributes (hp, name) and functions that are common to all characters. The attack() function is "pure virtual", meaning any class that inherits from Entity *must* provide its own version.
- **Player** and **Enemy** are concrete derived classes. They **inherit** everything from Entity and add their own unique properties and behaviors. They both provide an override for the attack() and update() functions, demonstrating **polymorphism**.

2. Architecture & Logic Flow Diagram

This diagram shows how the main components of the game are connected and how the game flows from one state to another.

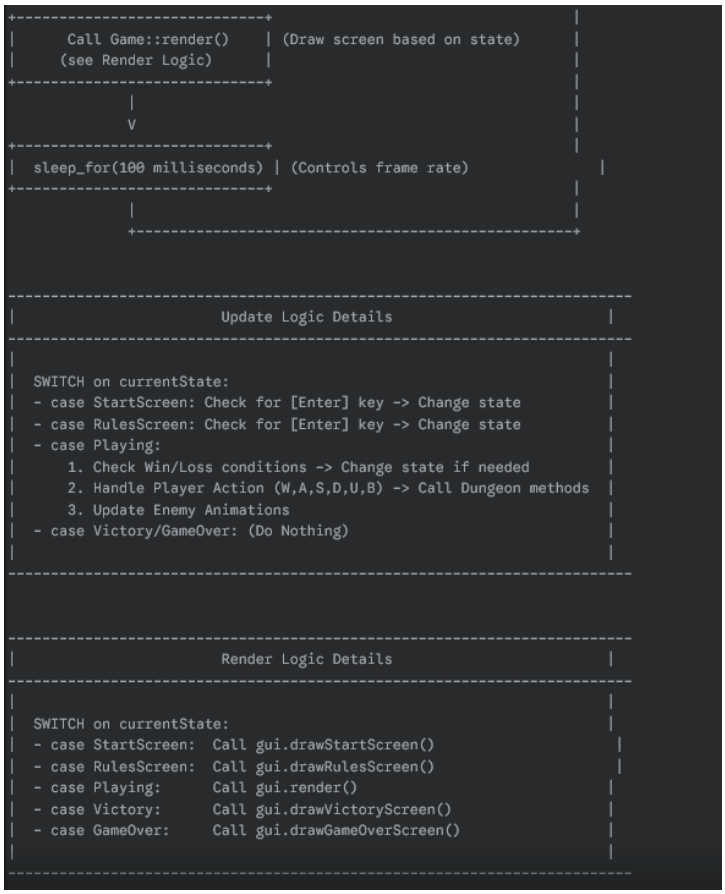
Ahmed Abdul Rahim
30609

Explanation:

- **main()**: The program's entry point. Its only job is to create the Game object and call its run() method.
- **Game Object**: This is the central engine. It "owns" the other major objects and contains the main game loop.
  - The **run()** method loops continuously.
  - In each loop, **processEvents()** gets input, **update()** changes the game's state and logic, and **render()** draws the result.
- **Component Interactions**:
  - The Game object is the "controller," telling the Dungeon what logical actions to perform and telling the GameGUI what to draw.
  - The Dungeon can also directly send messages to the GameGUI to update the message log.
- **Game State Flow**: This shows how the player progresses through the different states of the game.
  - The game begins in StartScreen.
  - Pressing Enter transitions through the menus to the main Playing state.
  - From the Playing state, the game can end by transitioning to either GameOver or Victory.
  - The game can be closed with the Escape key from any of the final states.

FLOW-CHART

Ahmed Abdul Rahim
30609



Explanation of the Flow:

1. **Start & Condition Check**: The run() method starts a loop that first checks if the GUI window is still open. If not, the program ends.
2. **Input**: The loop then processes all user input. It checks if the Escape key has been pressed to close the window.
3. **Update**: It calls the update() function, which is the "brain" of the loop. This function uses a switch statement to perform different actions based on the current GameState. For example, in the Playing state, it handles player movement and checks for win/loss conditions. In menu states, it just checks for the Enter key.
4. **Render**: After the game's logic has been updated, it calls the render() function. This function also uses a switch statement to call the correct drawing function from the GameGUI class, ensuring that the right screen is displayed to the user.
5. **Delay & Repeat**: Finally, a small delay is added to control the speed of the game before the loop repeats.

Project Report: Development of a C++ SFML Dungeon Crawler

1. Project Overview & Objectives

This document provides a comprehensive summary of the development process for the C++ SFML Dungeon Crawler, a 2D graphical role-playing game. The primary objective of this project was to design and implement a complete game from the ground up, with a strong focus on fulfilling a set of core academic and technical requirements.

The principal goals were to demonstrate a mastery of:

- **Object-Oriented Programming (OOP):** Utilizing concepts like inheritance, polymorphism, and encapsulation to create a robust and maintainable character system.
- **Custom Data Structures:** Implementing and applying fundamental data structures like stacks and queues to create unique gameplay mechanics.
- **Modern C++ Features:** Leveraging advanced features such as templates, smart pointers, lambda functions, and the Standard Template Library (STL) to write efficient and modern code.

Every major architectural decision was made with these core objectives in mind, aiming to produce a final product that is not only functional but also well-designed and technically sound.

---

2. Architectural Design Decisions

Ahmed Abdul Rahim
30609

This section explains the rationale behind the key architectural choices made during the project's development.

Decision 1: Encapsulated Game Engine (Game Class)

Initially, a game's logic might reside entirely within the main() function. However, this approach quickly becomes unmanageable as a project grows.

- **Decision:** To move the entire game loop, state management, and ownership of all major subsystems (Player, Dungeon, GameGUI) out of Main.cpp and into a dedicated Game class.
- **Rationale:** This decision adheres to the **Single Responsibility Principle**. The main() function's sole responsibility is now to be the entry point of the applicationto create the Game object and start it. The Game class's responsibility is to manage the game's lifecycle. This separation makes the code significantly cleaner, more organized, and easier to debug.

Code Evidence (Main.cpp): The final main function is extremely simple, demonstrating this encapsulation.

```
int main() {
    std::srand(std::time(0));

    Game game;
    game.run();

    return 0;
}
```

Decision 2: State-Driven Architecture (GameState Enum)

A game is rarely in a single mode of operation. It has menus, gameplay, pause screens, and end screens. A simple way to manage this is with boolean flags, but this leads to complex and brittle if-else chains.

- **Decision:** To implement a formal **finite state machine** using a simple enum class GameState to manage the different modes of the application (StartScreen, Playing, GameOver, etc.).
- **Rationale:** A state-driven design provides a robust and scalable way to manage application flow. The main loop's update() and render() functions use a switch statement to execute code that is relevant only to the current state. This prevents, for example, player movement logic from running while the start menu is active. It also

makes adding new states (like a Paused screen) trivial, as it only requires adding a new value to the enum and new case blocks to the switch statements.

Code Evidence (Game.cpp): The render logic is cleanly separated by state.

```
void Game::render() {
    switch (currentState) {
        case GameState::StartScreen:
            gui.drawStartScreen();
            break;
        case GameState::Playing:
            gui.render();
            break;
        // ... etc.
    }
}
```

Decision 3: Separation of Logic and Rendering

A common pitfall in game development is mixing game rules directly with drawing code. This makes the codebase difficult to change or expand.

- **Decision:** To strictly separate responsibilities. The Dungeon and Player classes handle pure game logic (stats, rules, outcomes), while the GameGUI class handles all rendering and input processing via SFML.
- **Rationale:** This **decoupling** is a critical design principle. The game logic classes have no knowledge of SFML. They simply update their internal state. The GameGUI class then reads this state and translates it into a visual representation. This means we could replace the entire SFML GUI with a simple console interface (or a different graphics library) by only rewriting the GameGUI class, without touching the core game rules in Dungeon, Player, or Enemy.

Decision 4: Abstract Entity Class for Polymorphism

The game features different types of characters (Player, Imp, Skeleton) that share common traits (like having health) but behave differently (they attack in different ways).

- **Decision:** To create an abstract base class Entity with a **pure virtual function** for attack(). The Player and Enemy classes then inherit from Entity and provide their own implementations.

- **Rationale:** This design leverages **polymorphism**. The core game logic (like the combat function) can interact with any character through a generic Entity pointer or reference, without needing to know its specific type. When attack() is called, C++ automatically determines the correct version to run at runtime. This prevents the need for if statements to check the character type and makes adding new character types incredibly easy.

Code Evidence (Entity.h): The pure virtual function creates a "contract".

```
class Entity {
public:
    // ...
    virtual void attack(Entity& target) = 0;
};
```

Decision 5: Lambda-Based Item System

The game needed a system for items with unique effects. A switch statement in the useItem function would work, but it would be rigid and require modification every time a new item is added.

- **Decision:** To use a std::map that links an item's name (std::string) to a **lambda function** (std::function<void(Player&)>).
- **Rationale:** This is a data-driven approach that provides immense **flexibility and scalability**. To add a new item, a developer only needs to add a new entry to the map in the Dungeon constructor, defining its effect within a new lambda. The core useItem function never needs to be touched. This isolates changes, reduces the chance of introducing bugs, and is a powerful demonstration of modern C++ features.

Code Evidence (Dungeon.cpp): Defining an item's effect is clean and self-contained.

```
itemEffects["Potion of Copeium"] = [&](Player& player) {
    player.hp = 200;
    if (gui) gui->pushMessage("You drank the Potion. Health fully restored.");


    auto it = std::find(player.inventory.begin(), player.inventory.end(), "Potion of Copeium");
    if (it != player.inventory.end()) player.inventory.erase(it);
};
```

Ahmed Abdul Rahim
30609

3. Project Timeline & Milestones

This represents a plausible development timeline, breaking the project into logical phases.

- **Milestone 1: Core Engine & SFML Setup**
    - **Tasks:** Initial project scaffolding, creating the main SFML window, and implementing the foundational, encapsulated Game class and its main loop. Basic asset loading (fonts) was established to ensure the build environment was configured correctly.
    - **Outcome:** A running application that displayed a blank window, ready for features to be added.
- **Milestone 2: OOP & Character Implementation**
    - **Tasks:** The Entity, Player, and Enemy class hierarchy was designed and implemented. This involved creating the abstract base class and the derived classes, defining their core properties, and implementing the polymorphic attack() function.
    - **Outcome:** The ability to create Player and Enemy objects in memory. Basic sprites were loaded and drawn to the screen to verify the Enemy class's graphical components.
- **Milestone 3: World Building & Gameplay Logic**
    - **Tasks:** The Dungeon and Room classes were created. The static map of the dungeon was built by creating all room objects and connecting them via pointers. Logic for player movement (enterRoom), item placement, and trap placement was implemented.
    - **Outcome:** A navigable world where the player could move from room to room.
- **Milestone 4: Data Structures & Core Mechanics**
    - **Tasks:** The CustomStack and CustomQueue classes were implemented from scratch. The CustomStack was integrated into the Dungeon to power the backtrack mechanic. The CustomQueue was used to create the enemy "guard" system for fleeing combat.
    - **Outcome:** Two unique, core gameplay features were now functional, both fulfilling specific data structure requirements.
- **Milestone 5: Combat System & Advanced Features**
    - **Tasks:** The complex Dungeon::combat function was implemented, including the QTE attack mechanic. The lambda-based item system (useItem) was developed. The std::sort algorithm was added to the inventory display.

    - **Outcome:** The main gameplay loop was now feature-complete, with interactive combat and item usage.

- **Milestone 6: UI/UX & Finalization**
    - **Tasks:** The state-driven menu system was implemented, creating the StartScreen, RulesScreen, Victory, and GameOver states and their corresponding rendering functions in GameGUI. The final documentation (README.md, this report) was written.
    - **Outcome:** The project was wrapped in a complete, user-friendly package from start to finish, fulfilling all remaining objectives.

End Of Report

---

4. Conclusion

This project successfully integrates a wide range of C++ programming concepts to create a complete, playable Dungeon Crawler game. The architecture effectively separates concerns, with the Game class managing the main loop, GameGUI handling all SFML-based rendering, and the Dungeon class overseeing the game's world and logic. Key OOP principles like inheritance and polymorphism form the backbone of the character system, while custom data structures and modern C++ features like lambdas and smart pointers provide robust and scalable gameplay mechanics. The result is a finished application that meets all specified technical requirements and serves as a strong foundation for future expansion.

---

5. Potential Future Improvements

While the current game is a complete experience, there are many avenues for future improvement to expand its depth and replayability.

- **Gameplay Enhancements**
    - **Dynamic Combat System:** The current QTE system could be expanded with more player abilities, such as magic spells, special weapon skills, or defensive options like dodging and blocking. Status effects like poison, stun, or buffs could also be added.
    - **Procedural Generation:** The dungeon map is currently static and hard-coded. A major improvement would be to implement a procedural generation algorithm to create a new, unique dungeon layout every time the game is played, offering infinite replayability.

  - **Deeper NPC Interaction:** The world could be populated with non-hostile NPCs, such as shopkeepers to buy and sell items, or quest-givers who provide narrative objectives and rewards.
  - **Save/Load System:** Implementing a system to save the game state (player stats, inventory, room location) to a file and load it later would allow for longer, more engaging play sessions.

- **Code and Architecture Improvements**
  - **External Data Files:** A significant architectural improvement would be to move all hard-coded game data (e.g., enemy stats, room descriptions, item properties) out of the C++ code and into external data files like JSON or XML. The game would then load this data at startup. This would allow for easy modification, balancing, and addition of new content without recompiling the entire program.
  - **Advanced State Machines:** The enemy animation, currently handled by a simple boolean isHurt flag, could be refactored into a more formal state machine (Idle, Attacking, Hurt, Dying states). This would make handling more complex animations and behaviors much cleaner.
  - **Component-Based Architecture:** For ultimate scalability, the inheritance-based entity system could be transitioned to an Entity-Component-System (ECS) architecture. In ECS, an "entity" is just an ID, and its properties (like Health, Sprite, Inventory) are separate data "components." This is a more advanced design used in many modern game engines that provides greater flexibility than deep inheritance chains.

Conclusion

This project successfully integrates a wide range of C++ programming concepts to create a complete, playable Dungeon Crawler game. The architecture effectively separates concerns, with the Game class managing the main loop, GameGUI handling all SFML-based rendering, and the Dungeon class overseeing the game's world and logic. Key OOP principles like inheritance and polymorphism form the backbone of the character system, while custom data structures and modern C++ features like lambdas and smart pointers provide robust and scalable gameplay mechanics. The result is a finished application that meets all specified technical requirements and serves as a strong foundation for future expansion.

Potential Future Improvements

Ahmed Abdul Rahim
30609

While the current game is a complete experience, there are many avenues for future improvement to expand its depth and replayability.

- **Gameplay Enhancements**
    - **Dynamic Combat System:** The current QTE system could be expanded with more player abilities, such as magic spells, special weapon skills, or defensive options like dodging and blocking. Status effects like poison, stun, or buffs could also be added.
    - **Procedural Generation:** The dungeon map is currently static and hard-coded. A major improvement would be to implement a procedural generation algorithm to create a new, unique dungeon layout every time the game is played, offering infinite replayability.
    - **Deeper NPC Interaction:** The world could be populated with non-hostile NPCs, such as shopkeepers to buy and sell items, or quest-givers who provide narrative objectives and rewards.
    - **Save/Load System:** Implementing a system to save the game state (player stats, inventory, room location) to a file and load it later would allow for longer, more engaging play sessions.
- **Code and Architecture Improvements**
    - **External Data Files:** A significant architectural improvement would be to move all hard-coded game data (e.g., enemy stats, room descriptions, item properties) out of the C++ code and into external data files like JSON or XML. The game would then load this data at startup. This would allow for easy modification, balancing, and addition of new content without recompiling the entire program.
    - **Advanced State Machines:** The enemy animation, currently handled by a simple boolean isHurt flag, could be refactored into a more formal state machine (Idle, Attacking, Hurt, Dying states). This would make handling more complex animations and behaviors much cleaner.
    - **Component-Based Architecture:** For ultimate scalability, the inheritance-based entity system could be transitioned to an Entity-Component-System (ECS) architecture. In ECS, an "entity" is just an ID, and its properties (like Health, Sprite, Inventory) are separate data "components." This is a more advanced design used in many modern game engines that provides greater flexibility than deep inheritance chains.

Ahmed Abdul Rahim
30609

THE END