Behind the terminal – Design report

Dungeon Escape is a text-based escape room game that integrates concepts of OOPs, memory management and data Structures. The game challenges players to navigate through a dungeon, solve puzzles, fight enemies, and collect treasures while managing their health and limited moves. This report explains the key design choices, the implementation of required data structures, and challenges encountered during development.

Player Class

The Player class represents the main character, managing health, inventory, and moves. It plays a central role in combat, exploration, and resource management.

Key Functions:

fight(Enemy& enemy) – Controls the battle mechanics. The player can choose to block, attack, or make a deal. This function determines how each encounter unfolds and ensures that combat is engaging rather than repetitive.

takeDamage(int damage) – Updates the player's health when attacked. Preventing health from going negative ensures proper game mechanics.

isAlive() – Checks if the player still has health left. This is essential to determine if the game should continue or end.

collectTreasure(const Treasure& treasure) – Adds treasure to the player's inventory. This is crucial for attacking enemies or making deals.

showInventory() – Displays all collected items, allowing the player to make strategic choices.

decreaseMoves() – Deducts moves when the player answers a challenge incorrectly or moves between rooms. This function prevents unlimited exploration, adding an element of urgency.

getMoves() and getHealth() – These getters provide the current move count and health status, which are used throughout the game for decision-making.

Together, these functions allow the player to interact with the game world, fight enemies, solve challenges, and manage their resources effectively.

Room Class

The Room class represents individual locations in the dungeon. Each room contains enemies, a challenge, and potentially a treasure. It is structured as a node in a **doubly linked list**, allowing smooth navigation forward and backward.

Key Functions:

triggerRoomEvents(Player& player) – The core function of the room. It controls the sequence of events:

Presents a challenge before the player can continue. Forces the player to fight enemies in order if they are present. Rewards treasure after clearing the room. Checks if the player still has moves left before proceeding.

addEnemy(const Enemy& enemy) – Stores enemies in a **queue** to ensure they are encountered in a fixed order.

hasEnemies() – Checks if there are any remaining enemies in the room.

getNextEnemy() – Retrieves the next enemy for the player to fight.

removeDefeatedEnemy() – Removes an enemy from the queue once it is defeated.

resetEnemies() – Restores the original enemies when a room is revisited, maintaining game consistency.

setNext(Room* nextRoom) and setPrev(Room* prevRoom) – Links rooms together, allowing forward and backward movement.

This class ties different game elements together, ensuring the player progresses logically through challenges, combat, and exploration.

Enemy Class

The Enemy class represents hostile creatures the player must face. Each enemy has health and an attack power, creating dynamic encounters.

Key Functions:

getAttackPower() – Returns the enemy's attack strength, which determines how much damage the player takes.

takeDamage(int damage) – Reduces the enemy's health when attacked.

isAlive() – Determines if the enemy is still in battle or has been defeated. Enemies are stored in a queue within rooms, ensuring structured combat. The player must fight them in the order they appear, preventing random skips or unfair fights.

Challenge Class

Challenges introduce a puzzle element into the game. The player must solve a logic-based question before progressing, adding variety beyond combat.

Key Functions:

isSolved() – Checks if the challenge has already been answered, preventing repeated attempts.

askQuestion() – Asks the player a riddle or logic question. If answered incorrectly, the player loses a move.

This class ensures players engage their minds rather than brute-forcing through the dungeon.

Treasure Class

Treasure acts as a collectible resource. Players can use it to attack enemies (weapons) or trade with them to avoid battle.

Key Functions:

getName() – Returns the name of the treasure, helping the player identify items in their inventory.

The inventory system makes treasure useful in multiple ways, giving players strategic choices.

Data Structures Used

The game relies on multiple data structures, each chosen for its efficiency and suitability for the task.

1. Vector (std::vector)

Used in Player to store inventory. Vectors allow dynamic resizing and easy access to items when selecting weapons or trading. The player adds treasures to the vector when collecting them, removes items when using them, and can view the full inventory at any time.

2. Queue (Queue)

Used in Room to manage enemies. A queue ensures that enemies are fought in the order they appear. The first enemy added is the first enemy fought. Once defeated, it is dequeued, preventing unfair skipping of battles.

3. Doubly Linked List

Used in Room to structure the dungeon. A doubly linked list allows movement between rooms in both directions, enabling forward progression and backtracking. Each Room points to the next and previous rooms, ensuring seamless transitions.

4. Stack

Used to track player movement history. A stack follows the Last-In, First-Out (LIFO) principle, making it perfect for tracking room visits when backtracking.Every time a player moves forward, the current room is pushed onto the stack. When backtracking, the stack is popped to return to the previous room.

Challenges Faced

1. Balancing Combat

In early versions, players could block infinitely, making battles too easy. Since blocking reduced damage without drawbacks, players could stall indefinitely.

- A limit was introduced on how many consecutive blocks a player could use before being forced to move on.
- Enemy attack power was adjusted to ensure they remained a real threat while keeping battles fair.
- These changes made combat more strategic rather than a repetitive cycle of blocking.

2. Handling Enemy Encounters

Enemies were stored in a queue, ensuring battles followed the correct order. However, a major issue arose: when a player revisited a room, defeated enemies were gone, leaving it empty.

- A backup list of enemies was maintained, storing the original set of enemies.
- If a player re-entered a room and the queue was empty, it was repopulated from the backup list, resetting the battle.
- This ensured consistency, preventing exploits where players could leave and return to avoid enemies.

3. Designing Challenges

The challenge system needed to be engaging but not overly frustrating. If riddles were too easy, they wouldn't be meaningful. If too hard, they could halt progress unfairly.

- Challenges were carefully balanced in difficulty to encourage problem-solving without overwhelming players.
- Move penalties were added for incorrect answers to prevent infinite guessing.
- This made puzzles a meaningful part of the game rather than something players could brute-force.

4. Validating input

The cin buffer would not be cleared, and hence the action loops would infinitely ask for input. This was fixed by clearing out the cin buffer and making numeric limit checks to ensure no unwarranted input was fed.

Structure of the Main Game Loop

The loop follows these key steps:

Display Room Information: The player enters a room, and its name is displayed.The triggerRoomEvents() function handles challenges, enemy encounters, and treasure collection. If the player fails a challenge or dies in combat, the game ends immediately.

Check for Game Over Conditions; If the player's moves reach zero, the game ends. If the player's health reaches zero, the game ends.

Player Movement Options: If the player is still alive, they are prompted to move. Move forward if there is a next room. Move back if they want to return to the previous room.

Process Player's Choice: If the player moves forward, the next room becomes the current room, and a move is deducted. If the player moves back, the previous room is restored, and a move is deducted. The stack is updated to track movement history, allowing proper backtracking.

Loop Repeats Until a Win or Loss Condition is Met: The player wins when they reach the last room. The player loses if they die in battle or run out of moves.