DUNGEON ESCAPE GAME

System Documentation

Mercer University
SSE 554 Object-Oriented Design II
Mr. Steve Hamby

Alyssa Bonifacio, Kandice Estrella, Joshua Seepersaud

Table of Contents

Problem Statement

The need for modularized game structures with an emphasis on optimization techniques is the problem this project aims to solve. More specifically, console applications written in C++ which heavily rely on proper memory management for optimal performance must find ways to efficiently structure, search through, and sort data.

Solution: Product

Our solution to this complex software engineering problem is a C++ text-based console game. By utilizing a variety of data structures such as binary trees, undirected graphs, stacks, queues, and singly linked lists, we were able to implement search and sort algorithms to demonstrate an efficient approach to modularized game management.

The final product is the C++ console application: Dungeon Escape Game. The player must navigate the dungeon collecting items and acquiring skills to defeat monsters to escape the dungeon.

Solution: Product Requirements and Use Cases

Requirements:

Once the game begins, the player is required to enter their name. From there, the player will navigate through the dungeon by choosing what to do next based on the displayed menu options.

Use Cases:

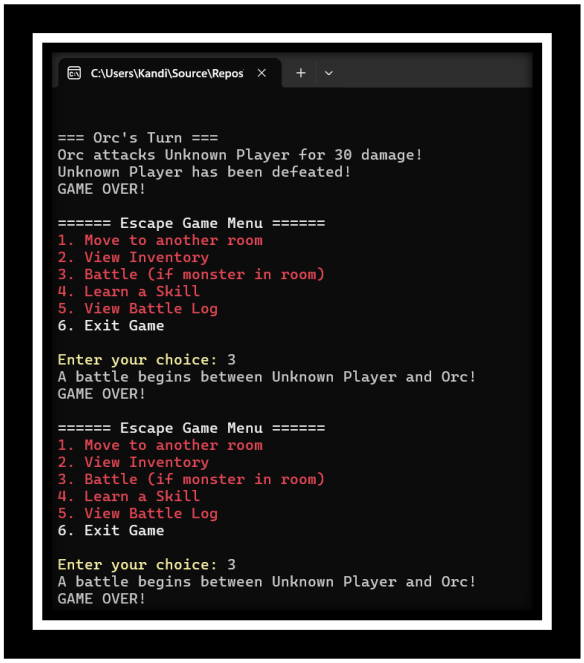
As a developer, anyone seeking a foundational blueprint for simple, modularized game management code can utilize this project as a starting point.

Software Quality

TEST CASE 1 – Tests Battle Logic

Test 1 Execution – 5/1/2025 4:30pm

Test Case 1 Failed – Game did not update player health to allow battling the monster again.

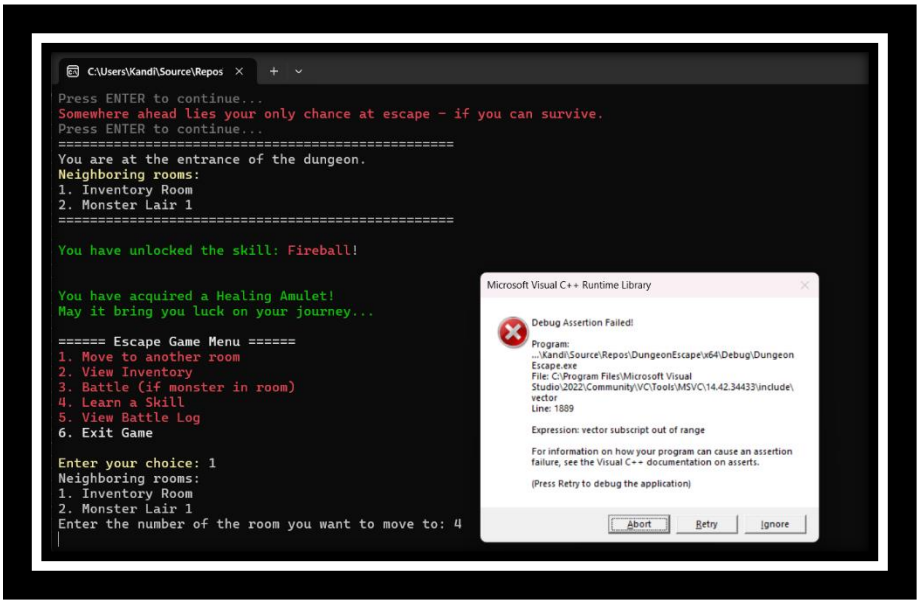


Test Case 1 Fix – To fix this error, the player's health and mana as well as the monster's health was reinitialized to starting values in the battle method inside the Player Class.

TEST CASE 2 – Tests User Selection with UI Options

Test 2 Execution – 5/1/2025 4:45pm

Test Case 2 Failed – Game did not handle room selection choice being out of range.



Test Case 2 Fix – There was an issue with the conditional logic for checking invalid user input. The issue was fixed by correctly declaring the intervals for user input.

TEST CASE 3 – Tests Exit Game Logic

Test 3 Execution – 5/1/2025 5:32pm

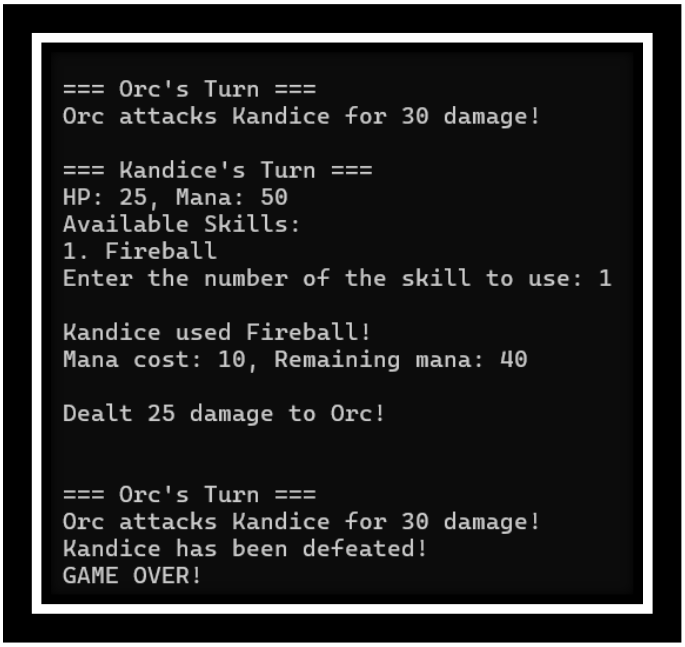
Test Case 3 Failed – Exiting the game early incorrectly displayed that the user completed/won the game.

Test Case 3 Fix – Added conditional logic to option 6 to test for whether the second monster is defeated before displaying end game output.

TEST CASE 4 – Tests Player Health and Mana Replenishment

Test 4 Execution – 5/1/2025 6:27pm

Test Case 4 Failed – The player's mana did not replenish after defeating the first monster.

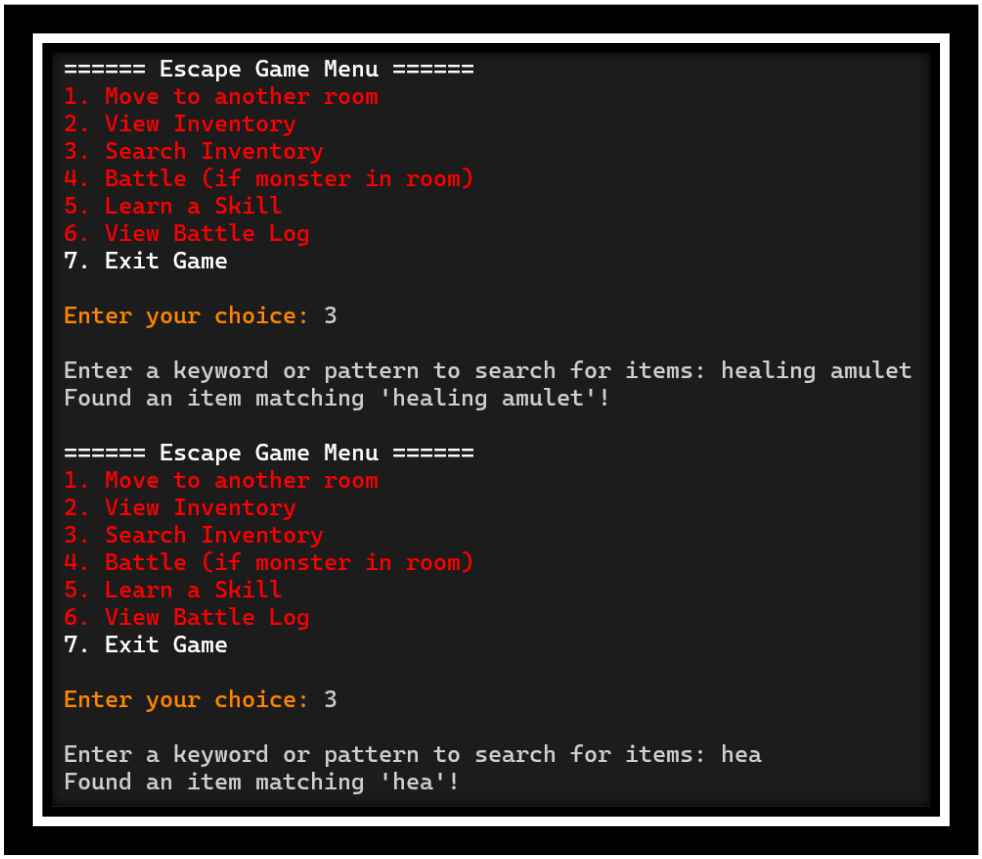


Test Case 4 Fix – The player's mana and health needed to be reinitialized within the Player Class logic for checking whether the player was defeated.

TEST CASE 5 – Tests Regex Search for Searching Inventory

Test 5 Execution – 5/1/2025 7:40pm

Test Case 5 Failed – The search only returned what the user searched for instead of displaying the inventory item.



Test Case 5 Fix – Added a line to show exact matching item name(s) from inventory.

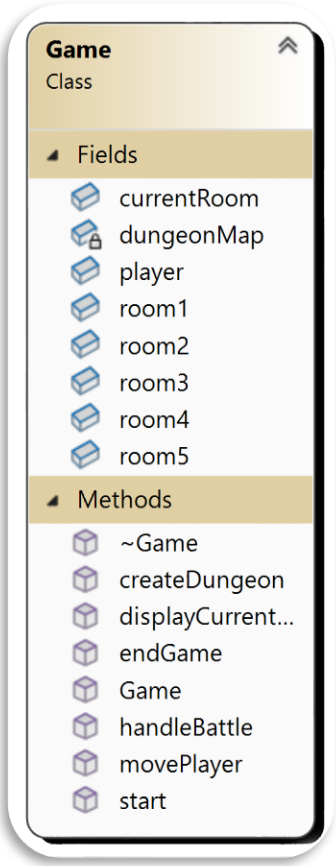
TEST CASE 6 – Tests Final Gameplay

Test 6 Execution – 5/2/2025 2:50pm

https://drive.google.com/file/d/1wJhTrTDd17hQaUCTgsRYBvvh33oP7kc0/view?usp=sharing

Test Case 6 Passed – Final gameplay met gameplay expectations. See Demo link above.

Design Artifacts and Code Explanations

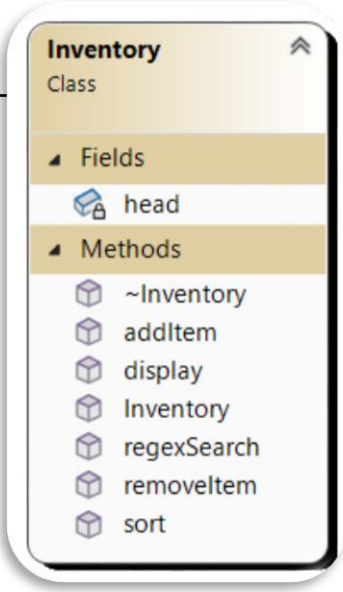


Software Architecture Overview

GAME CLASS – Alyssa Bonifacio, Kandice Esrella, Joshua Seepersaud

The architecture for our game consists of a shared Game class that incorporates the major classes for the game setup: Inventory, Player, Room, and UI.

Inventory handles adding and removing items needed to escape the dungeon. The Player Class handles player initialization including learning/unlocking skills, storing a battle log, and using skills to fight monsters. It also includes the Skill, Skill Node, and Skill Tree Classes which handle the construction of the player's skill tree. The Room Class contains the logic necessary for constructing the rooms in the dungeon map. It also includes the smaller Monster class which allows the Room class to add and remove monsters once defeated. Finally, the UI handles displaying a menu to the user for navigating the game.



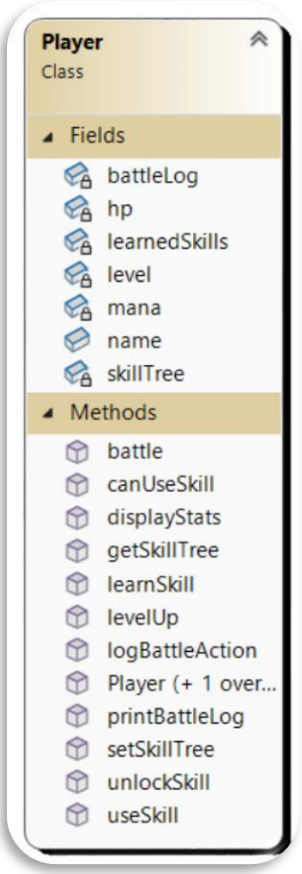
LINKED LISTS – SINGLY LINKED LIST

The inventory system in our Dungeon Escape Game is implemented using a singly linked list, coded by Alyssa. Each item collected by the player is stored as a node (InventoryNode) that holds the item's name and a pointer to the next node. The Inventory class manages the singly linked list and supports operations such as adding new items (addItem()), displaying the current inventory (display()), searching through items

(regexSearch()), and sorting the inventory alphabetically (sort()). The head pointer always points to the most recently added item, and traversal is performed by iterating from the head through the next pointers until the end of the list is reached. Using a singly linked list was an efficient choice for this project because it allows dynamic growth of the inventory without needing to predefine a fixed size.

```
// Inventory constructor
Inventory::Inventory() {
    head = nullptr;
}

// Add a new item to the inventory
void Inventory::addItem(std::string name) {
    InventoryNode* newNode = new InventoryNode(name);
    newNode->next = head;
    head = newNode;
}
```

STACKS, TREES, AND SEARCH ALGORITHMS – DEPTH FIRST SEARCH / BINARY TREE



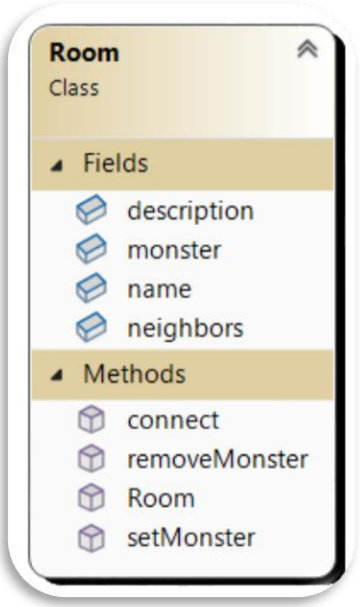
Stacks were implemented in our code in the algorithm used for the battle log in the Player class, coded by Joshua. The battle log stack is used in the program to store any skill used by the player against the monster, as well as the result of the player's battle against monsters.

```
std::stack<std::string> battleLog; // History of battle actions


// logs the player's skills used and monsters defeated
void Player::logBattleAction(const std::string& action) {
    battleLog.push(action);
}

// prints the battle history of the player
void Player::printBattleLog() {
    std::cout << "=== Battle Log ===\n";
    std::stack<std::string> tempLog = battleLog;
    while (!tempLog.empty()) {
        std::cout << tempLog.top() << "\n";
        tempLog.pop();
    }
}
```

Trees were implemented in the program in the form of a Binary Skill Tree also coded by Joshua. The structure of the player's skills is in the form of a tree with nodes that each have a skill object as its data. This makes use of a Skill class that contains all of the skill's information. This tree makes use of Depth-First Search when displaying the tree and finding a skill. The display function uses in-order traversal while the find function uses the pre-order traversal.

```
void SkillTree::display(SkillNode* node, int indent) const {
    if (!node) return;

    if (node->right) display(node->right, indent + 4);

    if (indent) std::cout << std::string(indent, ' ');
    node->skill.display();

    if (node->left) display(node->left, indent + 4);
}



SkillNode* SkillTree::find(SkillNode* node, const std::string& skillName) {
    if (!node) return nullptr;
    if (node->skill.name == skillName) return node;

    SkillNode* leftResult = find(node->left, skillName);
    if (leftResult) return leftResult;

    return find(node->right, skillName);
}
```

GRAPHS



To demonstrate the use of graphs, Kandice created the Room class; it constructs rooms and connects them accordingly. The result of the connection is an undirected graph. Using an undirected graph allows the player to move back and forth between rooms rather than only moving forward. The rooms are initialized in the Game Class constructor to serve as the dungeon map. The following snippet of code portrays the basic structure of a room node and the logic for connecting rooms.

```
Room::Room(std::string n, std::string desc)
{
	name = n;
	description = desc;
	monster = nullptr;
}
// Function to connect dungeon rooms as an undirected graph
void Room::connect(Room* other)
{
	// Avoid connecting the same room to itself
	if (this == other) return;
	// Connect the other room to this room if it is not already connected
	if (std::find(neighbors.begin(), neighbors.end(), other) == neighbors.end())
	{
		neighbors.push_back(other);
		other->neighbors.push_back(this);
	}
}
```

TIME COMPLEXITY BIG-O -

Tree Traversal Time Complexity – O(n)

As for time-complexity, we found that the runtime in the Player and Skill Tree classes are heavily influenced by the tree traversal functions such as display and find. These functions make

the overall runtime of the program linear. This is because these are the most time-complex functions regarding the player and the functions that these methods are nested in share their runtime of O(n).

Battle Log Traversal Time Complexity – O(n)

The stack operation for the battle log is constant. However, since the function to print the battle log is linear, its runtime is also O(n). This is because as the number of logged items increase, the longer it will take to print the battle log.

Graph Traversal Time Complexity – O(n)

The time complexity for traversing the dungeon map (i.e., room neighbors) is linear. Since the map construction is small, it makes sense to use a simple linear traversal for finding and displaying the neighboring rooms. The following snippet shows the implementation of the graph traversal in the shared Game Class.

```
// Display neighboring rooms
if (!currentRoom->neighbors.empty()) {
    std::cout << ANSI_WARNING_YELL "Neighboring rooms:" ANSI_RESET << std::endl;
    for (size_t i = 0; i < currentRoom->neighbors.size(); ++i) {
        std::cout << i + 1 << ". " << currentRoom->neighbors[i]->name << std::endl;
    }
    std::cout << "================================================\n";
}
else {
    std::cout << "There are no neighboring rooms." << std::endl;
}
```

Overall Time Complexity – O(n)

The worst case scenario for any of the search, sort, or traversal algorithms was linear due to the simple nature of the resulting data structures used in the game. In each case, the time complexity depended on the number of items to be iterated over.

REGULAR EXPRESSION SEARCH

To enhance the inventory management system, Alyssa implemented a regular expression (regex) search method in the Inventory class. Instead of requiring the player to enter the exact item name to find an item, the regexSearch() function allows players to input keywords or partial patterns to search for matching inventory items. The function traverses the linked list of inventory items and uses the C++ <regex> library with case-insensitive matching to check each item name against the user's input pattern. If a match is found, the matching item names are displayed to the player. This feature improves user experience by making the search process more flexible and forgiving, helping players quickly locate items without needing to remember the exact spelling or capitalization.

```
// Search for items using a regular expression and display matching names
bool Inventory::regexSearch(std::string pattern) {
    InventoryNode* current = head;
    bool found = false;

        std::regex re(pattern, std::regex_constants::icase);

        std::cout << "\nMatching items:\n";

        while (current != nullptr) {
            if (std::regex_search(current->itemName, re)) {
                std::cout << "- " << current->itemName << "\n";
                found = true;
            }
            current = current->next;
        }

        if (!found) {
            std::cout << "No items matched your search.\n";
        }

    return found;
}
```

SORTING ALGORITHM – BUBBLE SORT

To organize the inventory system, Alyssa implemented a sorting algorithm using a basic bubble sort. The inventory in our game is structured as a singly linked list, where each node stores an item name. To make it easier for players to browse their collected items, Alyssa created a sort() function in the Inventory class that orders the items alphabetically. The function works by repeatedly traversing the linked list, comparing adjacent items, and swapping them if they are out of order. This process repeats until the entire list is sorted. Although bubble sort is not the most efficient algorithm for large datasets, it is appropriate for this game because the inventory is expected to contain a manageable number of items, making the performance acceptable.

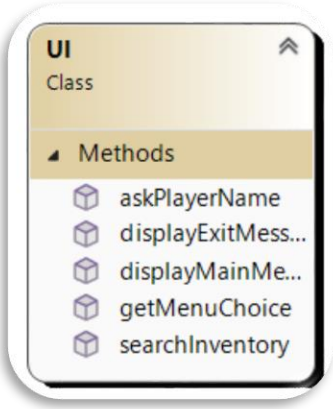
```
// Sort the inventory using a simple bubble sort
void Inventory::sort() {
    if (!head || !head->next) return;

    bool swapped;
    do {
        swapped = false;
        InventoryNode* current = head;
        while (current->next != nullptr) {
            if (current->itemName > current->next->itemName) {
                std::swap(current->itemName, current->next->itemName);
                swapped = true;
            }
            current = current->next;
        }
    } while (swapped);
}
```

DATA REPRESENTATION

To demonstrate data representation using UTF-8 character encodings, Kandice added the windows header in the main source file for outputting emojis to the console. Emojis are used to display skills, rooms, monsters, and the player.

```
#include <windows.h>

int main()
{
	SetConsoleOutputCP(CP_UTF8); // Ensures correct character encoding
```

USER INTERFACES – UI



To create a smooth and player-friendly experience, Alyssa developed the User Interface (UI) system for the Dungeon Escape Game. The UI handles all player interactions, including displaying menus, gathering user input, and guiding players through their choices during gameplay. Functions such as askPlayerName(), displayMainMenu(), getMenuChoice(), and searchInventory() are responsible for prompting the user at different stages of the game. Input validation was implemented in the menu selection to ensure that invalid entries do not cause the program to crash or behave unexpectedly. By structuring the interface carefully and handling user input safely, the UI helps maintain the overall flow of the game and makes it more accessible for players.

```
// Display the main menu options for the player
void UI::displayMainMenu() {
	std::cout << ANSI_BRIGHT_WHITE "\n====== \xF0\x9F\x8C\x9F Escape Game Menu
\xF0\x9F\x8C\x9F ======\n" ANSI_RESET;
	std::cout << ANSI_BLOOD_RED "1. \xF0\x9F\x9A\xAA Move to another room\n"
ANSI_RESET;
	std::cout << ANSI_BLOOD_RED "2. \xF0\x9F\x8E\x92 View Inventory\n"
ANSI_RESET;
	std::cout << ANSI_BLOOD_RED "3. \xF0\x9F\x94\x8D Search Inventory\n"
ANSI_RESET;
	std::cout << ANSI_BLOOD_RED "4. \xE2\x9A\x94\xEF\xB8\x8F Battle (if monster
in room)\n" ANSI_RESET;
	std::cout << ANSI_BLOOD_RED "5. \xF0\x9F\x93\x9A Learn a Skill\n" ANSI_RESET;
	std::cout << ANSI_BLOOD_RED "6. \xF0\x9F\x93\x9D View Battle Log\n"
ANSI_RESET;
	std::cout << ANSI_BRIGHT_WHITE "7. \xF0\x9F\x9A\xAA Exit Game\n" ANSI_RESET;
}
```

Getting Started

If not already installed, download and install the latest version of Microsoft Visual Studio:

Download Visual Studio Tools - Install Free for Windows, Mac, Linux.

To run the console application:

1. Open Visual Studio and click “Clone a repository”
2. Copy and paste the following link in the first text box:

   https://github.com/alyssamaebb/DungeonEscape.git

   a. Be sure to save the repo to the location of your choice
3. Build and run the application