Design Document

Version 1.0 – October 24, 2024

Dragon Slayer the Game

CPSC-2720

-

Dorgee Lama
John Holloway
John Nisperos

Introduction

Project Overview and objective of this document:
This project is aimed towards applying concepts of software application design and requirements we learned during our lectures.

The objective of this document is to establish a design for the project to act as a guideline towards implementing planned features in our text-based adventure game.

Timeline:
October 1, 2024 – October 20, 2024

2.Component Design

Overview of Main Components:

Game Engine:

The main controller of the game, managing the flow of the game. Not only is it responsible for loading all other components into memory, it is also responsible for managing user input and displaying the appropriate output.

The game engine will initiate a pair of windows using the ncurses library, one for receiving user input and the other for displaying the appropriate output. User input is in the form of text data typed in by the user. This information is parsed by the engine and the appropriate function is called based upon the input. If applicable, the parser will also pass the appropriate arguments to said functions. The function call will update the display ncurses window in response. The game loop will continue until the user exits the game.

Room:

Represents individual locations in the game, holding items and enemies. Items within the room can be picked up by the user and added to the player's inventory. The user can "look" while in a room to see what objects and enemies are in the room. For each of the four cardinal directions a room will either have a pointer to another room object if the user can go in that direction. If not, the direction will have a nullptr. When a user tries to go in a direction assigned a nullptr they will be informed that they cannot go in this direction.

Each room has a bool flag indicating if it is accessible to a user. By default, standard room objects are set as accessible to the player.

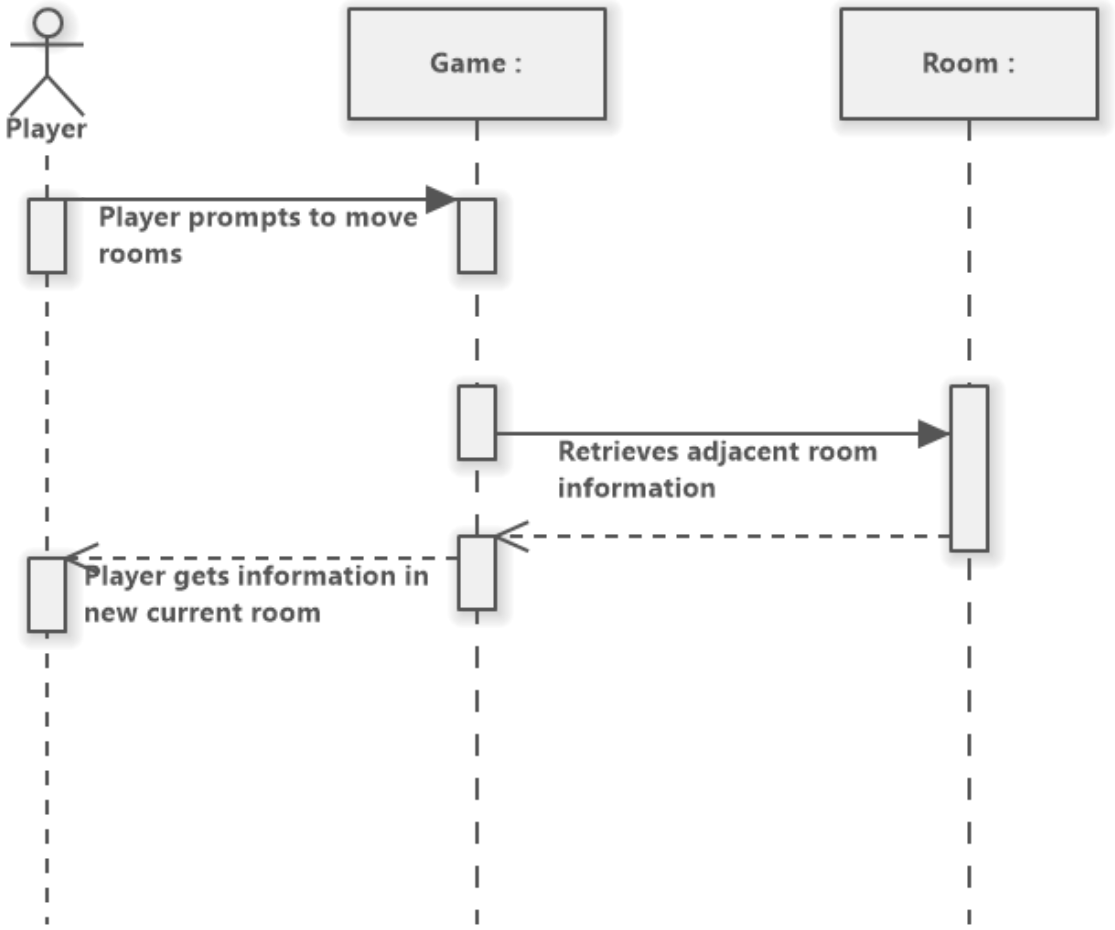

Figure 1. An example of the player input being parsed by the game engine, which is then used to retrieve information from the objects in the game and displayed to the user.

Restricted Room:

Similar to Rooms, but are set as inaccessible until a specific enemy has been killed. A pointer to an enemy will be passed in as an argument to the Restricted Room. The room will return *false* for the method isAccessble(), thereby preventing the room being accessed by the player until they have killed the designated enemy.

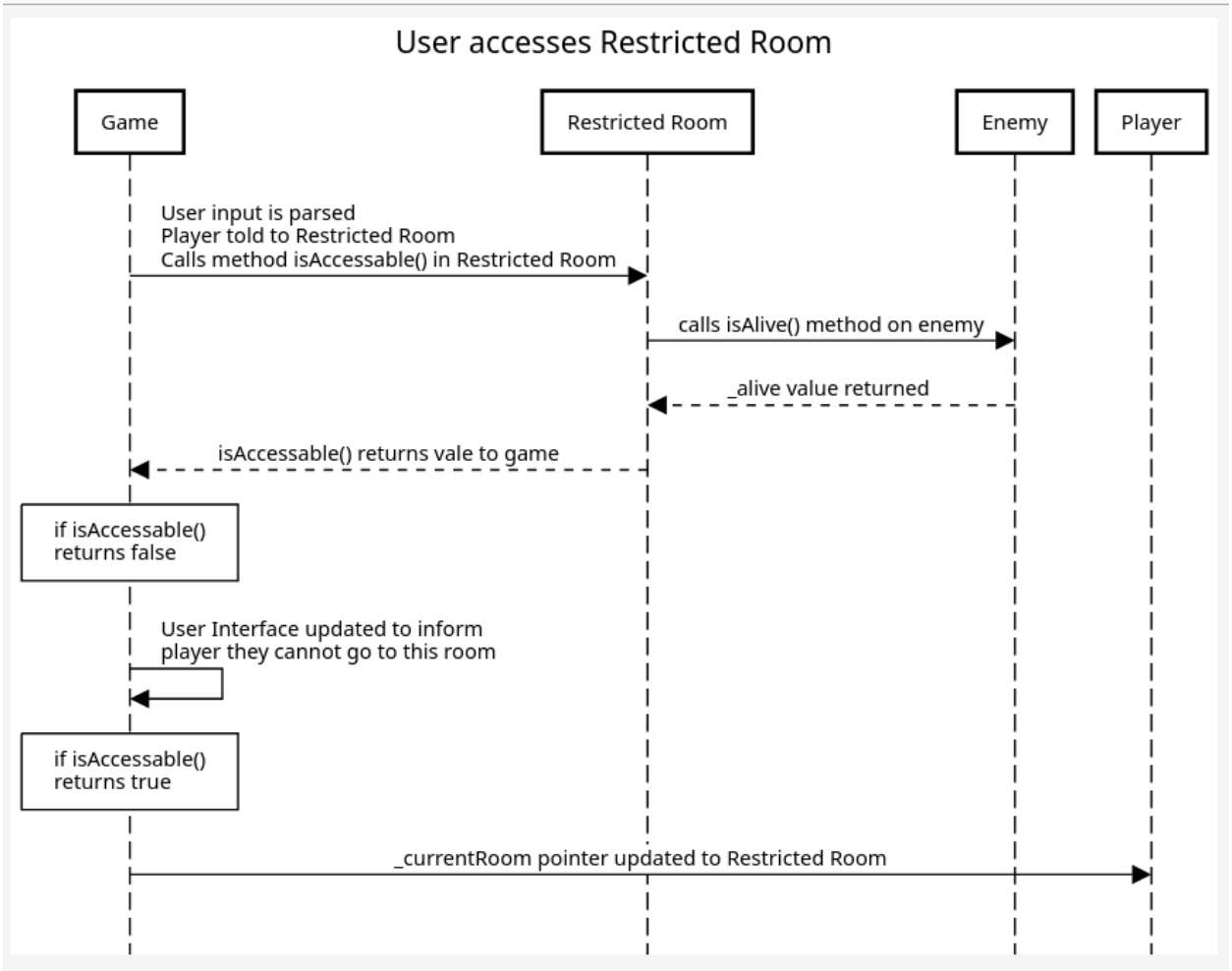

Figure 2: User tries to tell player to enter a Restricted Room

Exit Room:

Similar to Room objects, but will trigger a specific function upon entering the room. Because of the short nature of this game, it will be used to load a congratulations screen informing the player they have won the game.

Player:

The user-controlled entity, capable of interacting with rooms, items and enemies. The player will be able to pick up items and store them in their inventory. Players have a default health set at 100 points. Health can be increased by consuming food items, or decreased via attacks from enemies. When the player's health reaches 0 or lower the game will end and a game over screen displayed.

Enemy:

A non-playable creature in the game. Each enemy will have its own health value and can also hold items, as well as use items to attack the player. Once the enemy's health reaches 0 or lower, the enemy's items will be dropped into the respective room, thereby allowing them to be picked up by the player. When attacked by the player, the enemy will retaliate and attack the player.

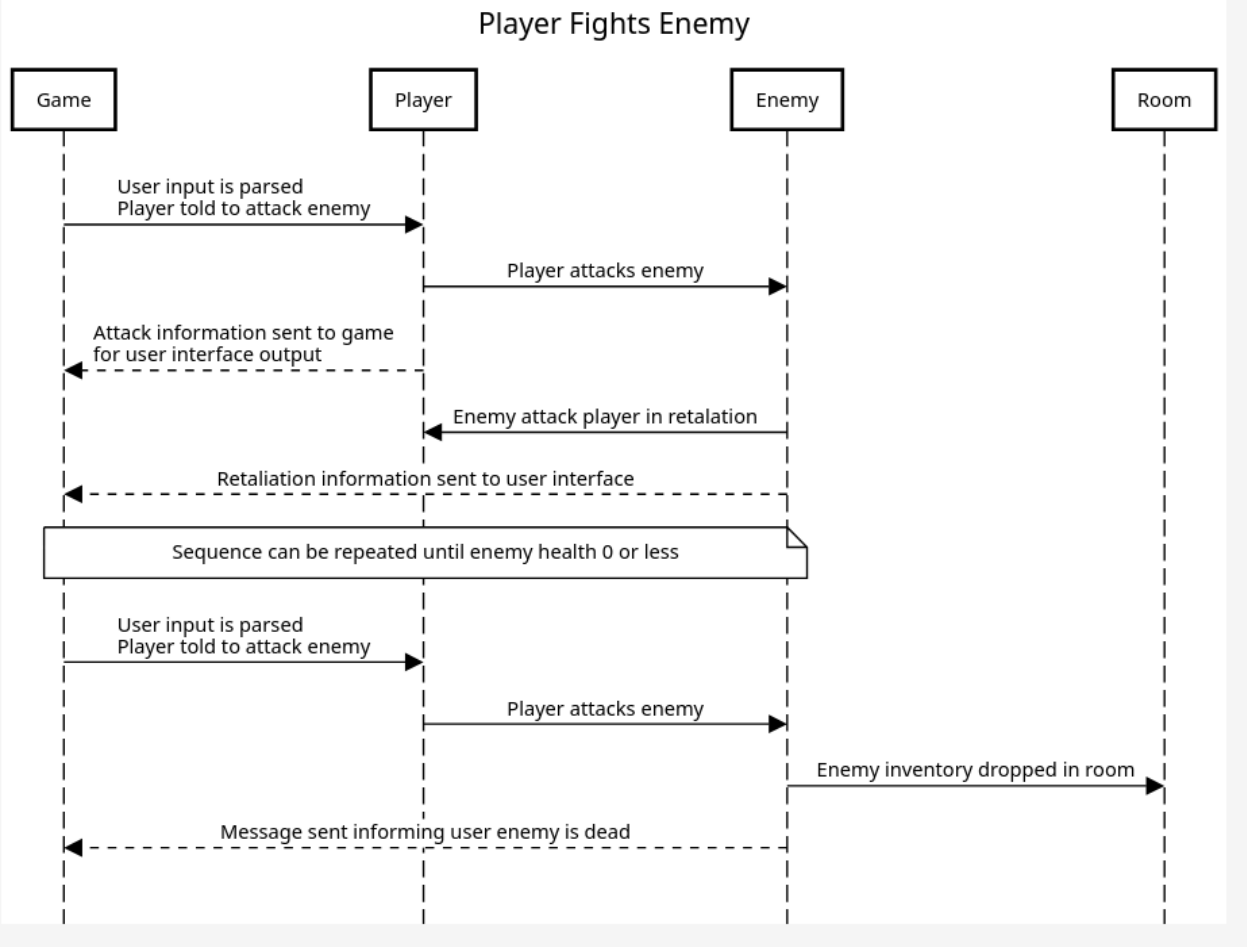

Figure 3: Player attacks an enemy

Item:

Items are objects that the player can interact with, such as pickup, use, etc. Items can belong to either the player, an enemy, or to a room (see the section on Inventory) and their owner will be tracked by the application. This is necessary to ensure that if an item is transferred from the inventory of one object to another, such as a player picking up an item from the room, the object is automatically removed from the previous owner's inventory. This will avoid creating a

duplicate object or multiple objects having a pointer to the same object in their respective inventories.

The player will be able to use items, such as attacking an enemy with an item or eating/consuming the item. If the player tries to attack an enemy with an item, the game will need to check if that is a valid item to use in this use case and respond appropriately.

Inventory:

The player, enemies and the room. will have a vector of items that form the inventory. The player can look at the items they are holding by entering the command “look inventory” or “look player” into the text input. Once an item has been entered into the player’s inventory it can be used by the player either for an attack or eaten for consumption.

The room will also have an inventory and these items will be displayed on the user interface when a user types “look” in a room. Only items in the room’s inventory can be picked up by the user.

Items in the enemy’s inventory are inaccessible to the user unless the enemy has been killed. At this point the enemy’s inventory will be “dropped” and items transferred to the room’s inventory, thereby making items accessible to the player.

Component Interaction Flow:

- Game Engine interacts with the UI to receive user input and display game states.
- Game Engine processes the Action by interacting with Player, Map, Room, and Inventory components.
- Map contains Room objects, which hold Item objects and Player interactions.
- Player manages their Inventory and interacts with Room and Item objects.

3. Class/Object Design

Creature:

An abstract class that is the parent to both Player and Enemy objects. Contains the common fields, properties and methods for the for the child objects such as the std::vector<Item *> _inventory object, int _health field, and methods such as hurt(), attack(), etc.

Figure 4: Creature family of classes

All Creature objects will have a pointer to a Room object named _currentRoom. This variable is to be used to determine where in the game world the object is located. This will be used by the Game object to display the correct Room information to the user, as well as by the Enemy objects to drop their inventory in the correct room in the event of their death.

Player:

The Player class will extend Creature class. The Player class will have additional methods, some of which will override those of the Creature class, specifically designed to handle user input. The method listInventory() will be used to display the player's current inventory to the screen when called. Multiple attack() methods will exist to facilitate different commands from user (ie. attacking without a target, attacking target, attacking target with specific item).

Enemy:

The Enemy class will extend the Creature class. The Enemy class features similar methods to those of the player, such as attack(), but with fewer parameters. Unlike Player objects, an Enemy can only attack with a weapon and not any item.

Item:

The item class is an abstract class that will be the parent to both Weapon and Food classes. It contains several fields, the most important of which is the std::optional<std::variant<Creature*, Room*>> _owner. This field is responsible for determining what object currently has the item in its inventory, and will be a nullptr if it has neither a Creature nor Room object as its owner.

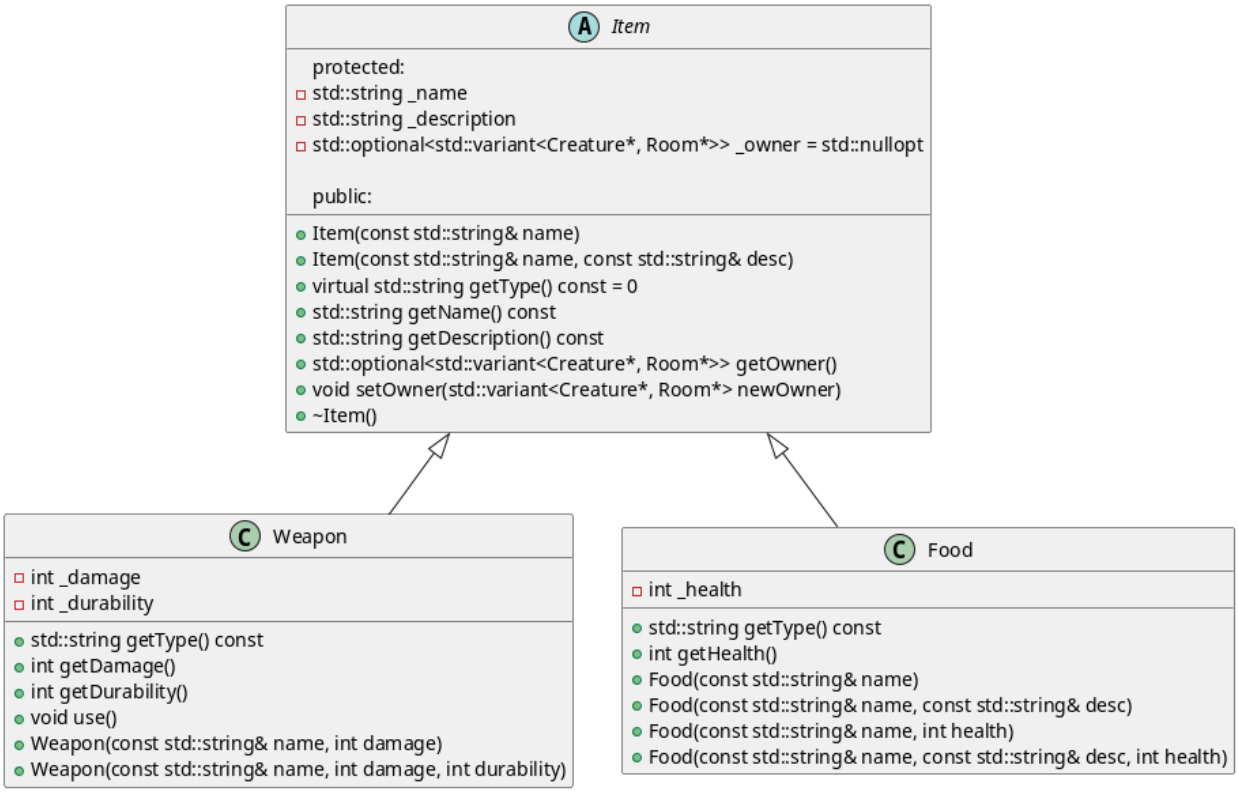

Figure 5: The Item family of objects

Calling the setOwner() method will use the _owner field to determine who the current owner is and remove the item from the current owner's inventory. Once removed from the previous owner's inventory, the item can now be added to the inventory of the new owner.

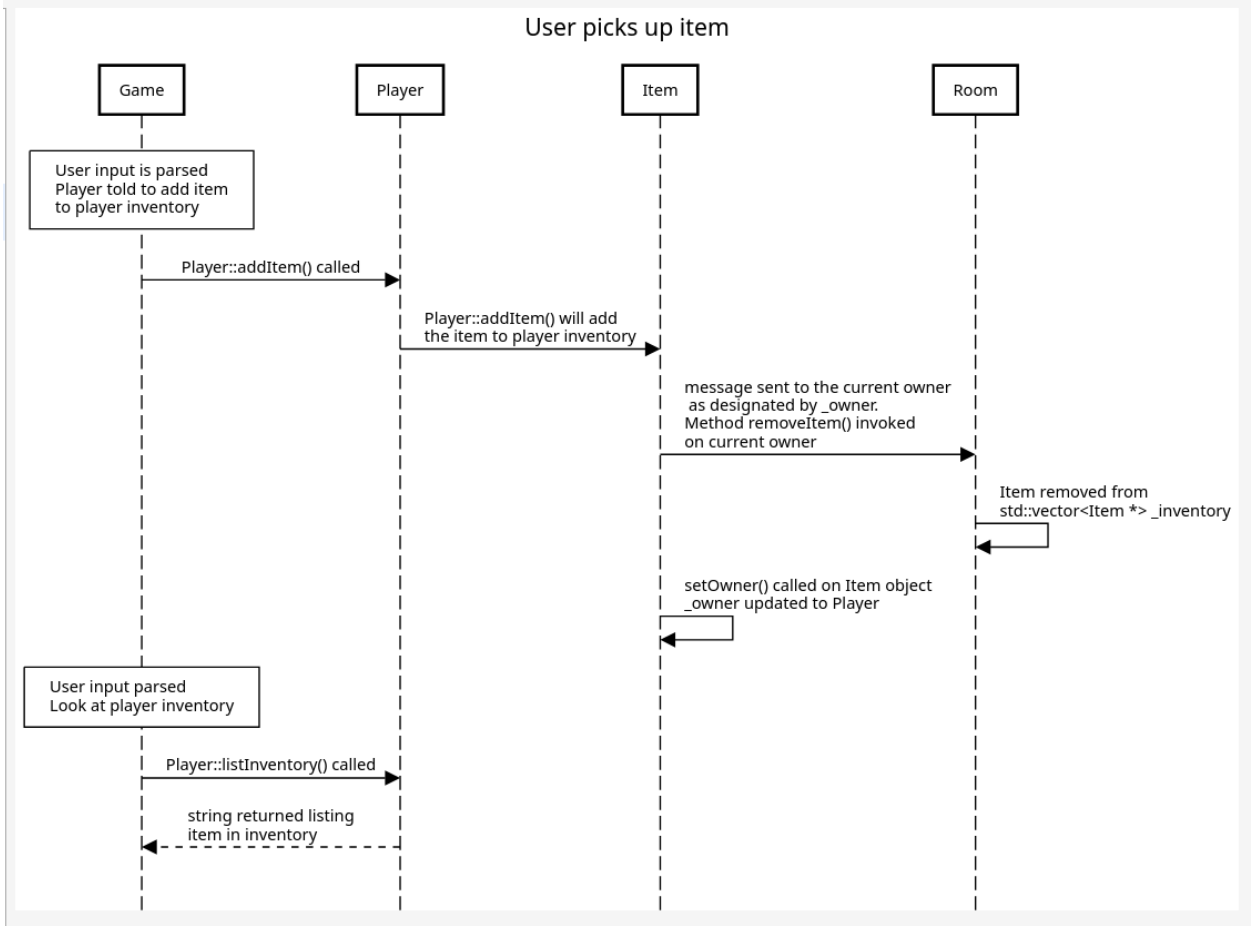

Figure 6: Example sequence of updating Item's current owner

Weapon:

A child class of Item. Contains the int field _damage to specify it's damage output. To be used when Player or Enemy attack another Creature.

Food:

A child class of Item. Contains the int field _health, representing the health that will be added to the creature's health when consumed by the creature. Can be a negative value for poisonous food.

Room:

The Room class will contain information about the current location. In each Room object there are four pointers, one for each cardinal direction, to other Room objects. When a Room is set as a pointer for another room, the opposite cardinal direction of the target room object will automatically be set to that of the first room. I.e: if the Room object *home* receives the method call setWest(*field)*, inside of the object *field*, the method setEast(*home)* will automatically be called. This ensures that all pointers are bi-directional, creating a graph data structure that forms the world map

Figure 7: The Room family of objects

RestrictedRoom:

Restricted Rooms extend the Room class. They contain a pointer to an enemy object set during the object's initialization. The method *isAccessable*() in a restricted room overrides that of the parent object, and instead checks if the enemy represented by the pointer is alive. If the enemy is alive the value for _accessable will remain false. If the enemy is not alive, then _accessable is set to *true* and the player can enter the room.

ExitRoom:

ExitRoom is simply an extension of the Room class. It exists only to differentiate this specific object from the parent class. When loading a room, the game engine will check if the room is an ExitRoom. If so, the game can be set to call functions, such as a method to display a congratulation screen.

Game:

The game class forms the heart of the DragonSlayer game. Its constructor calls several methods to build the game, such as setDefaults(), loadTitle(), initWorldMap(), and initPlayer(). These functions will build the game world. After building the game world, the user interface is built using the ncurses library. Two WINDOW pointers will be constructed in the terminal. _display_win is the larger of the two, consuming the upper 80 of the terminal window. It will be used to display output to the user. The second WINDOW, _input_win, is smaller and will be where the user can type in text commands for the game. After initializing the game a loop begins, continuously calling getCommand().

GetCommand() waits for an input from the user over *stdin* on the unix terminal. Once the command is received from the user the string is tokenized and its words parsed. If the user input is invalid _display_win will be updated to inform the user that an invalid command was received. If the command from the user is valid the appropriate function will be called; the Game object will either call one of its own methods to update the display, or it will call a method in one of the fields that compose a Game object.

The loop checks the value returned by getCommand(). If the user enters the command "exit" or "quit" getCommand() will return -1 which will terminate the loop. At that point the application will call the destructor on the game object. The destructor will then call the destructor on all rooms that have been build, which will in turn call the destructors of all Creatures and Items within the Room. This ensures proper clean up of all objects in the game that have been allocated on the heap.

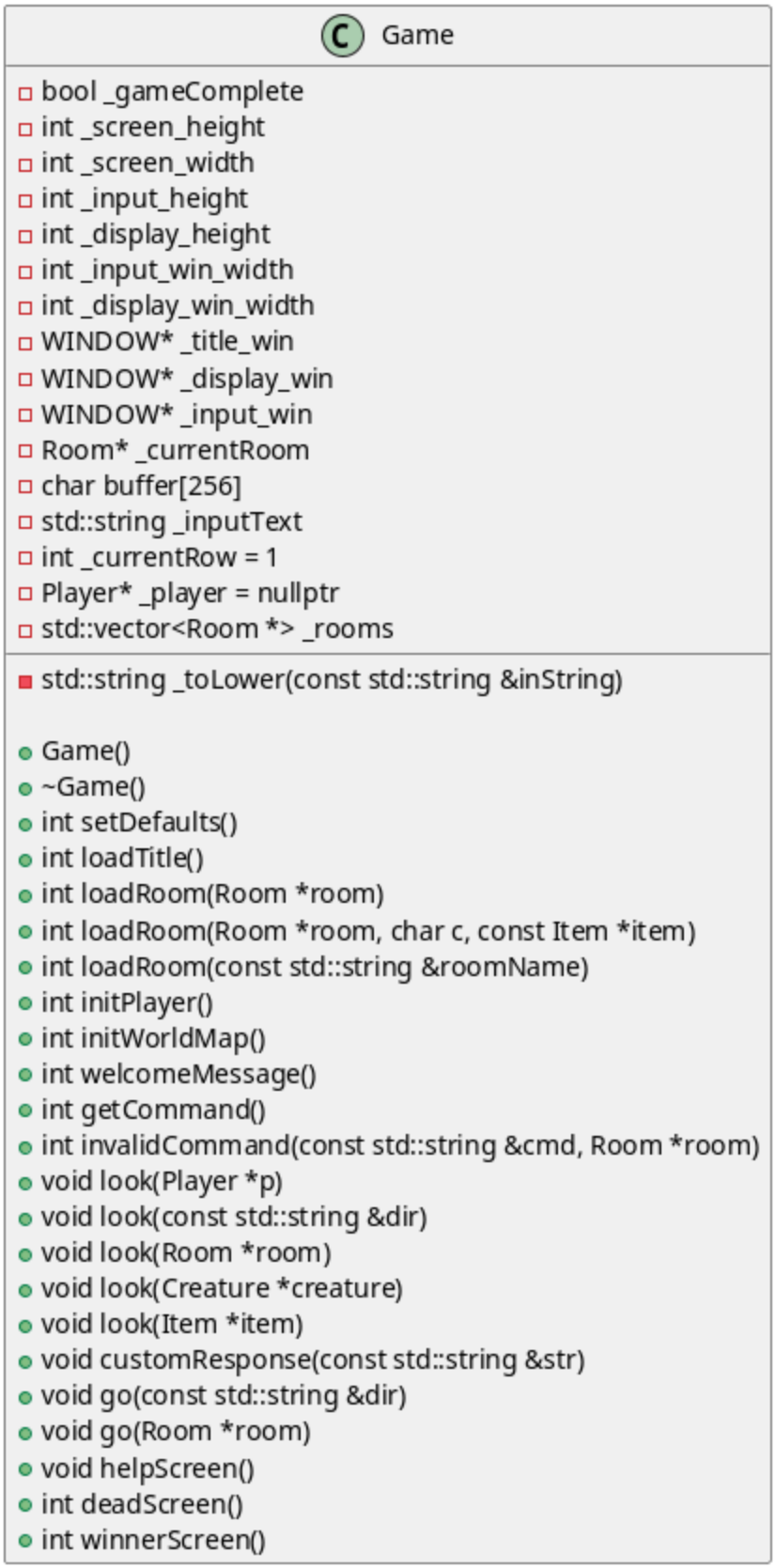

Figure 8: Game class

4. User Interface Design

User Interface Design Description for Text Adventure Game

The **User Interface (UI)** of the text adventure game will be **text-based** and designed to provide clear communication between the player and the game world. It should be simple, efficient, and intuitive, allowing players to focus on the narrative and interactions without complex commands or distractions. Here's a breakdown of the UI design:

1. Layout Design

The UI will consist of three main sections:

- **Game Output Window**: Displays the current location description, available items, exits, and events.
- **Command Input Area**: Where players input commands to interact with the game world.
- **Player Status Area**: Displays the player's inventory, current location, and any key information like health, objectives, or actions performed.

2. Game Output Window

- **Purpose**: Displays the current game environment, room descriptions, items, and any important events happening in the game world.
- **Details**: The game will print descriptions of the room when the player enters or when any new events occur. This window also shows item details and exits available to the player.
- **Behavior**:

    - Each time the player enters a command, the game will update this window with the result of the action, such as:
        - Picking up an item
        - Moving to a new room
        - Examining an object

3. Command Input Area

- **Purpose**: Provides a place for the player to type their actions.
- **Details**: Players will type simple text commands like "move north", "pick up sword", or "use key". The system will parse these commands and translate them into actions within the game.
- **Behavior**:
    - The UI should clearly prompt the player for input and provide instant feedback once a command is entered.
    - For any invalid commands, the game should respond with helpful error messages like: “I don't understand that command” or “You can't do that right now.”

4. Player Status Area

- **Purpose**: Keeps the player informed of their current state and progress.
- **Details**: This section shows the player’s inventory, current location, health status, and any important items they are carrying.
- **Behavior**:
    - The inventory updates automatically as the player picks up or uses items.
    - Location and health information update dynamically based on game events (like if the player takes damage or heals).
    - Players should always be able to check this information while navigating the game.

---

5. Interface Features and Usability

- **Command Shortcuts**: The interface will support abbreviations and synonyms for common actions. For example, "go north" and "move north" both work for moving to the north.
- **User Feedback**: After each command, the UI will provide feedback in the output window to ensure the player understands what happened. For example:
    - If the player picks up an item: "You picked up the sword."
    - If the player tries to do something impossible: "There is nothing to pick up here."
- **Text Styling**: Use simple text styles to differentiate key elements in the output:
    - **Bold** for room names (e.g., *Dimly Lit Room*).
    - **Italics** for object names (e.g., *Sword*, *Key*).
    - **Colored text** for important actions, such as exits being in **blue** and items in **green**.

6. Error Handling

- Clear and concise error messages for invalid inputs or actions.
- If the player enters a command that doesn’t make sense, such as "fly", the game should give a meaningful response like "You can't fly here."

---

7. Accessibility Considerations

- **Readable Fonts**: The text will be displayed in clear, readable fonts with appropriate spacing to ensure ease of reading for players of all ages.

- **Keyboard-Only Control**: Since it's a text-based interface, the game will rely entirely on keyboard input, making it accessible for players without requiring advanced navigation or mouse use.

8. User Flow

- Upon starting the game, players are greeted with a short introduction in the **Game Output Window** that sets the scene.
- The **Command Input Area** prompts the player for their first action, and the player begins interacting with the game world.
- As the player moves through rooms, picks up items, and interacts with objects, the game updates the **Game Output Window** with narrative descriptions and feedback.
- The **Player Status Area** provides a constant snapshot of the player's current status, ensuring they can make informed decisions throughout the game.

5. Plan Of Implementation

Week 1: Planning and Design (Days 1-7)

1. **Day 1-2**: Game Design Documentation
    - Define game objectives, rules, mechanics.
    - Outline basic story, player goals, and major features.
2. **Day 3-4**: Class Design and UML Diagrams
    - Define classes, relationships, and methods.
    - Create UML diagrams for game components and their interactions.
3. **Day 5-6**: Component Breakdown and Responsibility Outline
    - Identify the role of each component (player, room, item, map, etc.).
    - Finalize the abstract classes and action framework.
4. **Day 7**: Review and Adjust Design
    - Review the game design, UML diagrams, and component outline.
    - Make necessary changes or improvements.

Week 2: Implementation (Days 8-14)

1. **Day 8-9**: Set Up Project Structure and Implement Core Classes
    - Create the C++ project and implement the base classes: `GameObject`, `Item`, `Room`, `Player`, `Inventory`, `Boss`.
2. **Day 10**: Implement Map and Room Navigation
    - Implement the `Map` class with room linking and navigation methods.
3. **Day 11**: Implement Player Actions and Inventory System
    - Implement the `Action` and `MoveAction` classes.
    - Integrate the inventory system, allowing the player to pick up and interact with items.
4. **Day 12**: Implement UI and Game Loop
    - Implement the `UI` class to handle text-based input and output.
    - Create the main game loop to process player actions and update the game state.
5. **Day 13**: Testing and Debugging
    - Test all components and fix any bugs.
    - Ensure smooth gameplay and interaction.
6. **Day 14**: Final Adjustments and Review
    - Refactor code, add final touches, and review the entire project.

Gantt Chart Layout (Summary):

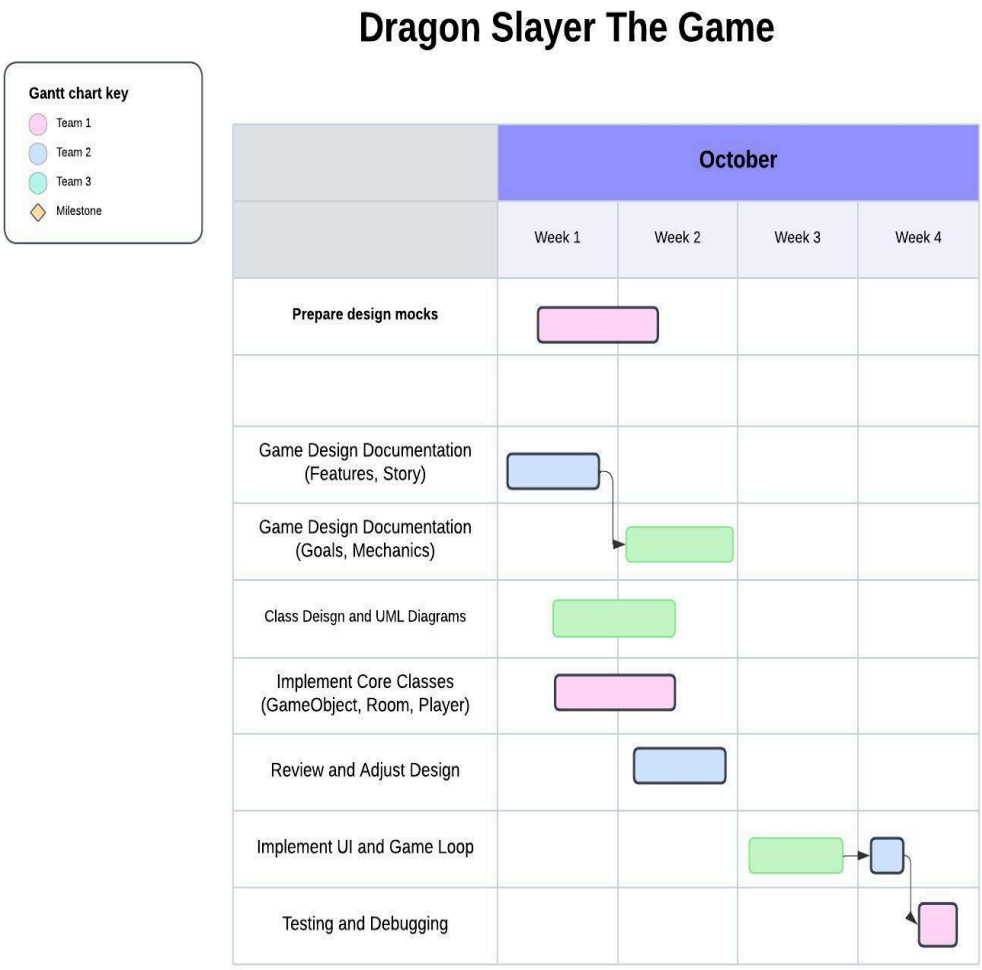

Figure 9: Estimated development of game