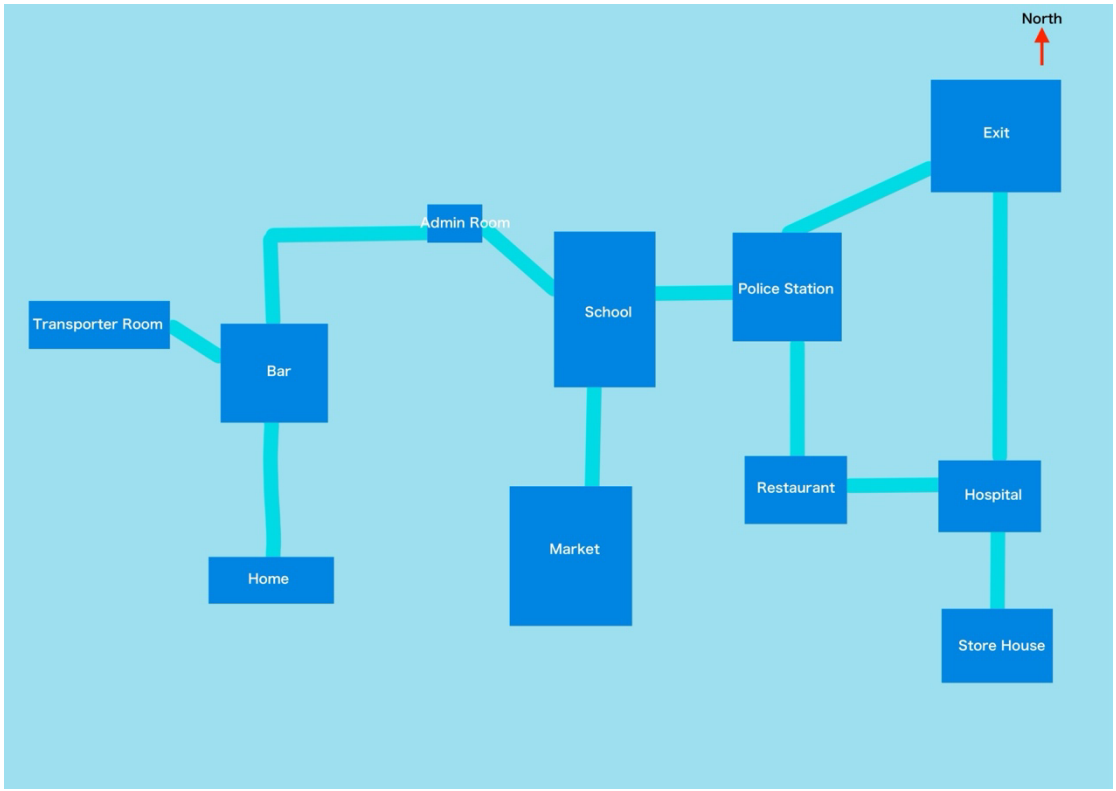
The Depths of Tomorrow

Coursework 2: World of Zuul

This Report is to describe the function of the game “The Depths of Tomorrow” and the process of game creation by Java.



Imagine1: The map of The Depths of Tomorrow

Jiangjing.Xu
Jiangjing.Xu@kcl.ac.uk
21134158
30/11/2023

Description of The Depths of Tomorrow

"The Depths of Tomorrow" is a text-based adventure game set in a dystopian world of 2087, where societal status is determined by wealth, and players are striving to escape from the lowest, chaotic floor within three days to avoid government elimination. The game features 12 distinct rooms, including a unique transporter room. Players engage with items through actions like "pick up," "carry," "put down," and "give." To win, players must strategically carry a key and an aircraft to the Exit, using the "escape" command. The game tracks room navigation history, enabling a 'back' command for revisiting rooms. The player can interact with non-player characters by giving items to them and having the chance to know the best way to escape.

Implementation

1 For the base task, I completed all the requests as below:

1.1 I developed a game world of 12 rooms, including a magic transporter room. The game's architecture utilizes a HashMap for efficient room management, eliminating the need to create multiple objects in the Game class.

1.2 I created two new classes, Item and ItemFactory, which were introduced to manage the properties and functionalities of game items. The Item class defines the basic attributes like name and weight, while ItemFactory streamlines the addition and management of new item abilities.

1.3 I set that players must first "pick up" an item to carry it. Items are designed with 'pickAbility' and 'carryAbility' properties; only items with both properties set to true can be carried by the player. There is a weight limit for carried items, and exceeding this limit prompts a warning, requiring the player to "put down" items before carrying more. The "put down" command allows players to leave items in any room.

1.4 Winning the game requires the player to carry a key and an aircraft to the Exit, where they can use the "escape" command to win, regardless of subsequent commands. The route is designed so players can visit at least five rooms to achieve victory.

1.5 Navigation history is tracked using a 'roomHistory' stack, supporting the 'back' command functionality. Popping elements from the stack allows players to return to previous rooms, with an empty stack indicating the starting room.

1.6 I add “pick up”, “carry”, “put down”, “give”, “drink”, “eat” and “escape” seven commands. The player can ‘drink’ or ‘eat’ items to get one more time using ‘go’, drinking beer will waste one day.

2 For the challenge task, I completed all the requests as below:

2.1 I created a new class Character to manage the character and made 4 characters who can move in different rooms randomly by using random.nextInt(n), n is the size of specific rooms, therefore, it generates a random index within the range of the list.

2.2 I divide inputLine into 3 parts, commandWord, descriptionWord and characterWord. Then, I use List to store inputLine searching the character names in inputLine to determine whether has characterWord by for-each loop, if has, descriptionWord will be extracted by replacing characterWord using “” and jump command word by String join method.

2.3 I added a new class TransporterRoom, which uses extend Room. If the player enters this room, they will be randomly transported to any room. I select a random room from the room list by selecting a random index within the range of the list’s indices and retrieving the room at that index.

Consideration of Code Quality

1. **Coupling:**

I created the TransporterRoom class by extending the Room class, which means it inherits the characteristics and functionalities of the Room class. Due to this inheritance, whenever I change the constructor of the Room class, I also need to make corresponding adjustments in the TransporterRoom class. This is because in the TransporterRoom's constructor, I use super() to invoke the constructor of the Room class. So, any modifications in the Room class, particularly its constructor, require careful consideration of their impact on TransporterRoom, to ensure everything functions cohesively.

2. **Cohesion:**

In my project, I have embraced this principle by separating concerns among different classes: Room, Item, and Character. The Room class is responsible for defining the attributes and behaviors of a room, like its exits or description. While a room can contain items and characters, it doesn't directly manage the intricate details of these entities. The Item class is dedicated to defining the properties and functionalities specific to items, such as their name, weight, and any other item-specific attributes or actions. Similarly, the Character class is focused on managing the properties and behaviors pertinent to characters, like their name, current room,

and movements. By segregating these responsibilities, I ensure that each class remains focused and manageable and makes high cohesion.

3. **Responsibility-driven design:**

In applying the principles of responsibility-driven design in my project, I ensure that each class is responsible for its specific domain. For instance, when integrating a new 'eat' command in the Game class, I recognize that the determination of whether an item is edible should not be the responsibility of the Game class. Instead, this responsibility falls to the Item class. The Item class is designed to manage all properties and behaviors related to items in the game. By adding an eatAbility property (or a similar attribute) to the Item class, I can encapsulate the characteristic of edibility within the items themselves. This attribute would define whether a particular item can be eaten or not. The Game class, responsible for handling game commands and interactions, would then check the eatAbility of an item to decide if the 'eat' command is applicable. This approach maintains a clear separation of concerns: the Game class handles the game logic and user commands, while the Item class manages the specific properties and state of the items.

4. **Maintainability:**

I create the class ItemFactory and Character which can help me keep the code readable when I need to add more property in Item and character. Using ItemFactory makes adding new properties to items much simpler and keeps the item's creation process clean and organized. Similarly, having a separate Character class means that any changes to characters are easy to handle and don't mess up other parts of the code. Both of these help keep the code neat and easy to update. Furthermore, I added some comments which explain the code so that when I need to revise it, I can easily understand what the code is doing, and I use readable names of variables. These all gain the maintainability of the project.

How to win

Winning Input: go north----go east----pick up Key----carry Key----go east----go east----pick up Aircraft----carry Aircraft----go east----escape