Jesus Rosalez
12 - 03 - 24
Professor K. Kanemoto
CPSC 39 - 12111

TableTop RPG Loot Generator Report

What it does, and how is it useful?

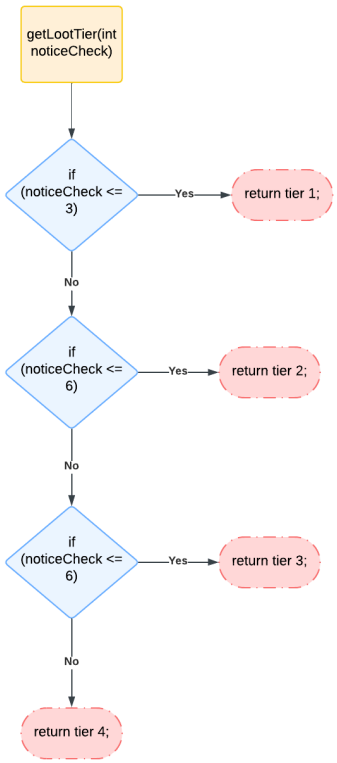
The TableTop RPG Loot Generator is a terminal-based application designed to enhance tabletop RPG games by generating loot based on location within the narrative and a player's Notice Check roll. It offers an engaging and dynamic loot generation experience, providing variability and efficiency within the game. The program supports features such as managing loot by location, determining loot rarity and tier, and tracking loot in a player's virtual backpack. This application is beneficial in a tabletop RPG game because it helps the Dungeon Master (DM) speed up the process of looting since at times it could take upwards of 10 minutes per player since the DM has to select randomly, make up on the fly and look up items to give, this optimizes it and offers suggestions for loot to give based of the players rolls. Furthermore, the DM can add their own items as they see fit since it is customizable. I specifically created this for a zombie apocalypse setting, so rarity, durability, amount of items found, using the skill of searching, and limiting carrying capacity all matter and I made sure to incorporate that into the code.

Algorithms

The program includes three primary algorithms, as detailed below:

1. Determining Loot Tier:

- **Purpose**: Determines the loot tier based on the player's notice check roll. This is used multiple times in other algorithms to determine durability, item roll, and the quantity of each item.



FlowChart:

- **Big O Analysis**: **O(1)** (Constant Time) - The operation involves simple conditional checks.

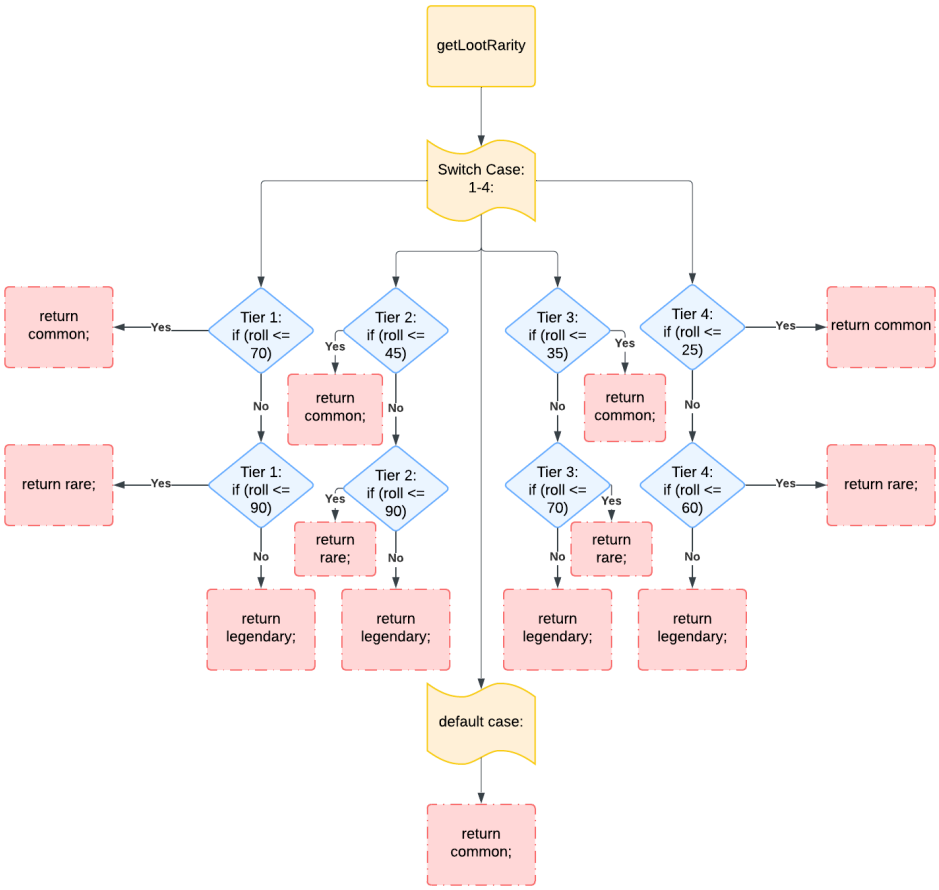
Implementation:

```
// Determine loot tier method
public static int getLootTier(int noticeCheck) {
    if (noticeCheck <= 3) return 1;  // Tier 1
    if (noticeCheck <= 6) return 2;  // Tier 2
    if (noticeCheck <= 8) return 3;  // Tier 3
    return 4;                        // Tier 4
}
```

2. Determining Loot Rarity:

- **Purpose**: Assigns a rarity level to loot based on the tier and notice check roll. This is used in other parts of the code to determine the chances of the user finding a lower-rarity item vs a higher-rarity item.

- **FlowChart**:



- **Big O Analysis**: **O(1)** - Uses simple calculations and conditional statements.

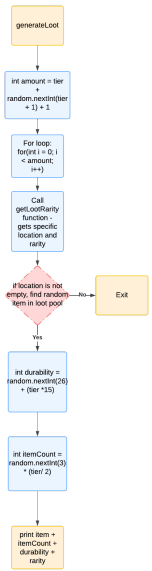
Implementation:

```java
// Determine loot rarity - changes ever tier
public static String getLootRarity(int tier, int noticeCheck) {  1 usage
    int roll = noticeCheck + random.nextInt( bound: 101);

    switch (tier) {
        case 1: // Tier 1 - low chance for legendary, higher chances for common
            if (roll <= 70) return "Common";      // 70% chance for Common
            if (roll <= 90) return "Rare";        // 20% chance for Rare
            return "Legendary";                   // 10% chance for Legendary
        case 2: // Tier 2 - higher chance for legendary and rare, common still most common
            if (roll <= 45) return "Common";      // 45% chance for Common
            if (roll <= 80) return "Rare";        // 35% chance for Rare
            return "Legendary";                   // 20% chance for Legendary
        case 3: // Tier 3 - 1/3 chance for legendary, equal chances for common and rare
            if (roll <= 35) return "Common";      // 35% chance for Common
            if (roll <= 70) return "Rare";        // 35% chance for Rare
            return "Legendary";                   // 30% chance for Legendary
        case 4: // Tier 4 - highest chance for legendary, more common rare, and less common common
            if (roll <= 25) return "Common";      // 25% chance for Common
            if (roll <= 60) return "Rare";        // 35% chance for Rare
            return "Legendary";                   // 40% chance for Legendary
        default:
            return "Common";                      // Default case, though it shouldn't happen
    }
}
```

3. Generating Loot:

- **Purpose**: Generates a list of loot items based on location, tier, and rarity.
- **FlowChart**:



- **Big O Analysis**: **O(n)** - Iterates over the number of items to generate (n is based on tier).

Implementation:

```java
// Generate loot for a given location and tier
public static List<String> generateLoot(Map<String, Map<String, List<String>>> locationLoot, String location, int tier, int noticeCheck){

    List<String> loot = new ArrayList<String>();

    // Determine number of loot items
    int amount = tier + random.nextInt( bound: tier + 1) + 1;

    for (int i = 0; i < amount; i++) {

        // Determine rarity of loot and get the list of items for the location and rarity
        String rarity = getLootRarity(tier, noticeCheck);
        List<String> items = locationLoot.get(location).get(rarity);

        // Random selection from loot pool (location specfic)
        if (items != null && !items.isEmpty()) {
            String item = items.get(random.nextInt(items.size())); // Pick a random item

            // Random durability percentage (based on tier)
            int durability = random.nextInt( bound: 26) + (tier * 15); // Durability is between 0 and 100%

            // Random number of items (based on tier)
            int itemCount = random.nextInt( bound: 3) + tier/2;

            loot.add(item + " (" + itemCount + " items, " + durability + "% durability, " + rarity + ")");
        }
    }
```

Data Structures:

1. Map:

I used a Map data structure to store loot categorized by location and rarity. I chose the map data structure because it allows efficient O(1) retrieval of location-specific loot pools based on rarity. I specifically like this because it allows me to have different loot pools in different locations since in the zombie apocalypse you'll find what you find depending on where you look.

2. List:

I used a list data structure to track the player's backpack inventory. It maintains an ordered list of loot items with a limit of 30 to account for limited carrying capacity during the apocalypse. Furthermore, I chose this because lists provide dynamic resizing and ordered insertion.

3. Arrays:

I used an array only to Parse and split loot data from CSV lines. It temporarily holds parts of a line until it is passed to the Map for storage. I liked using this array because it is simple and efficient at splitting and iteration.

Design and Development Insights:

Opportunities:

- When adding in the loot I saw an opportunity to add variability to loot attributes (e.g., durability and quantity) to improve gameplay depth I did this by Introducing a tier-scaled randomization for loot characteristics.

Errors and Resolutions

- When testing my code I incorrectly handled invalid lines in the CSV file which caused crashes so I added a validation check for line length before processing to prevent that.

Future Improvements:

If I had to update the version of the application I would enhance the backpack management to add sorting and filtering features for loot. Possibly giving the user the option to discard an item or even combine item durabilities if it is the same item. I would also integrate basic ASCII graphics for a more immersive experience.