CCPS 406 Project: Text Adventure

Design Patterns

Group members:

- Aden Hersi (Aden.hersi@ryerson.ca)
- Charif Sukkar (charif.sukkar@ryerson.ca)
- Ervin Demnushaj (ervin.demnushaj@ryerson.ca)
- Chris Fontein (cfontein@ryerson.ca)
- Jeesoo Kim (j17kim@ryerson.ca)

Instructor: Ilkka Kokkarinen
Subject: CCPS 406 Introduction to Software Engineering
School: Toronto Metropolitan University
Date: June 12, 2022

This is a documentation of the design patterns used in this project, along with the anti-patterns encountered and the refactoring to make the design clean of any codes of anti-patterns.

1.Design Patterns

(1)Creational Patterns
This is to solve design problems or complexity to the design by controlling object creation mechanisms, which is trying to create objects in a manner suitable to the situation

Our system is mainly to fit the **Builder Design Pattern**.

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.
We use the Builder Design Pattern when we create our game level. We go through the rooms YAML file and construct each room one by one. As we construct each room, we go through the items and characters present in that room, we look them up in their respective character and items YAML files, and we build those objects. We have the player, NPCs, and monster.
A character consists of name, description, identifies, current_health, base_stats.etc, same structure of nodes.
The content of each character for each field(node) is various. But the construction process is the same regardless of whether a character is the player, NPCs, or monster.
The game controller directs the game flow to assemble all the fields. All the assembled objects are then delivered to the end user, game player.
This applies to Items, rooms and characters in the system.
All the creation decisions are made by a builder.
This allows for the creation of a large number of assets with similar functionality to be created with just a handful of classes.

Reference: https://sourcemaking.com/design_patterns/builder

(2)Structural patterns
To ease the design by identifying a simple way to realize relationships between entities, we are using a combination of the various design patterns, not just one. One of the patterns is named **Composite Design Pattern**, which is as follows for example.

Our game allows for nested hierarchies of objects through the use of Containers. Each Room contains various Items and Characters. The Items themselves can be Containers that hold other Items and can even contain other Containers. Characters as well can hold Items including Containers for Items in their inventory. Thus, the assets in the game can be represented as a tree structure of nested objects.

Reference: https://sourcemaking.com/design_patterns/composite

(3)Behavioral patterns

This game is implementing the **Strategy Design Pattern**.

All Characters are controlled by a Controller class. Each subclass of Controller contains different control systems that make decisions based on information contained within the Character they are connected to. A MonsterController attached to a Character will only move and attack. An AdventurerController will move, attack, pick up and equip items, etc. A PlayerController can do everything an AdventurerController can but is guided by user input. Any of these can be freely passed between different Characters to change their behavior without affecting their functionality. Depending on the Character subclass a Controller is assigned to, new options become available, such as a Monster being able to navigate connections that other Characters cannot.

2.AntiPatterns

AntiPattern is a recurring problem that is usually ineffective and risks being highly counterproductive. Software Development AntiPatterns that our system may have are duplicates of the same functionality for different objects.

To establish good software structure for easier system extension and maintenance, we have been applying software refactoring in the process of building the application.

We want to reuse a functionality for various objects/characters if the functionality is the same, instead of creating another method with similar functionality under a different name.

3.Refactoring

(1)Duplicate Code
Several functions of the same functionality for creating different objects are reduced to one method by having an argument(s). For example, there are methods to read data files.

E.g.1 This is before code-refactoring.

```
def room_yaml():
    """ A function to read YAML file"""
    with open('Data/rooms.yml') as file:
        #config = yaml.safe_load(f)
        roomsList = yaml.safe_load(file)
        #print(roomsList)
    return roomsList #config

def character_yaml():
    """ A function to read YAML file"""
    with open('Data/characters.yml') as file:
        #config = yaml.safe_load(f)
        charactersList = yaml.safe_load(file)
        #print(roomsList)
    return charactersList #config

def item_yaml():
    """ A function to read YAML file"""
    with open('Data/items.yml') as file:
        #config = yaml.safe_load(f)
        itemsList = yaml.safe_load(file)
        #print(roomsList)
    return itemsList #config
```

This is how the code is refactored.

```
def read_yaml(file_name):
    """ A function to read YAML file"""
    with open(f"Data/{file_name}.yml") as file:
        #config = yaml.safe_load(f)
        dataList = yaml.safe_load(file)
        #print(roomsList)
    return dataList #config
```

Reference:https://sourcemaking.com/design_patterns