Final Project Report

Jing Huang
Yuan-Hao Cheng
Shuhan Fan

Objective

Our proposed domain is to create a maze adventure game in C++. Text adventures are regarded as a first role-playing game and were published in 1976. You, as a player, is to escape the forest by typing the direction you want to go. Our project is a simple version of the game. We have four main modules, which are item, room, character, and command. The item includes the information of the item name, item value, and item type. The room includes the information about the item and the character. The character includes the information of the character and the monster. The command addresses all the actions that the character can do. The player's goal is to collect items that can strengthen your strength and defeat the final two bosses in the final room. If you lose the fight, you lose the game.

The modularity scenarios are extending the rooms of the maze, add new types of characters, add items with different effects, and add things to the room and extending the command that the player can do. We set monsters in the maze and the items that the player can hold and utilize. Since our program is a game-based program, a game-company especially module-game-company would like to have this system and extend some commands, items, characters, and rooms without editing the original program or code. This program is reusable because nowadays the reprint version of games is developed constantly and is popular in gamers. If someone wants to reprint a similar game like ours, they can refer to ours and create a new one.

System Overview

We have four main modules. In the command module, we create the commands that the system can call and package them to be ready for dispatch. Our design is looking like this: Characters have their own attributes and behaviors, while Monsters and Human(protagonists) are subclasses of the Characters, inheriting the characteristics of Characters.

In our project, we practice the extensibility by adding the "Drop" command for the character to drop the first item in their bag back to the room. The receiver so-called Player is the actual place to define what the command does. We also design Rooms with names and connections, and Items with names and its usage.

There is only one interface in our project, which is Command. One idea is that a Player can visit rooms, grab items or interact with monsters by command. Besides, we make extensions on the basis of the interface. For the Drop command, for instance. Because there is no drop command provided in the “Player” module, we claim a “newPlayer” module inheriting the “Player” module to extend the command “Drop” without changing the code in the “Player” module. In the end, each item will have its own effect with the extension.

Below, we take some examples from our system to help us to explain how we did.

For example, we want to extend our system to an item that is food and affects the health of a game character. The extension of this dimension is simple. We simply add a new class Food to extend the class Item and override its beUsed () method.

```
class Item {
protected:
    string    name;
    int       value;
public:
    Item(string aname, int avalue)
    {}
    string    getName();
    virtual void beUsed(Player* aplayer) {}
};



class Food : public Item {
public:
    Food(string aname, int avalue) : Item(aname, avalue) {}
    void beUsed(Player* player);
};
```

This allows the player to use the food without modifying the existing code.

On the other dimension, as mentioned above, we want to add a Drop command. At this time, we only need to inherit/implement the "Command", and then encapsulate the code to execute the command as an object (is class NewPlayer here), and then set it to the caller.

```
class NewPlayer : public Player {
public:
    NewPlayer(int hp, int power, string name) : Player(hp, power, name) {};
    void drop();
};

class Drop: public Command {
private:
    NewPlayer *player;
public:
    Drop(NewPlayer *player)
    {
        this->player = player;
    }
    virtual void excute();
};
```
In our system, we didn't design a Caller class, and the main() function sort of plays that role.

This allows the player to throw away the contents of his backpack. And we don't need to modify the existing code.

Techniques

- Information hiding
  In the first week of the course, we discussed the importance of information hiding, abstraction, and other concepts to modular programming. Moreover, we have learned that interfaces should be highly abstract and frequency of transfers between modules should be least. Our Command interface is a good example. We discussed which information should be hidden and which should be disclosed in each class. It's very difficult, we don't do it perfectly, but we still try to make good design and show it in our code. We intend to use an object-oriented approach to the design of the program to achieve information hiding. We have a class realized by using ADT. One example of our program is the position and package variables in the Player module. The two variables are protected in the Player module. They can only be used if the class is a subclass of the Player.

- Modular
  As discussed in class, we divided our system into many small modules. All we have to do is define our classes as declared in the.h file.This has greatly improved our work efficiency.

- Command Pattern
  In our program, we need to send requests to our object, but we don’t need to know what the receiver is, and what the operation request is. We just need to send the command to the specific receiver within the runtime.
  The command pattern is a software design pattern. It encapsulates a request as an object, allowing us to parameterize customers with different requests. The command pattern can be completely decoupled from the sender and receiver, with no direct reference between the sender and receiver, and the sending object only needs to know how to send the

request, not how to complete it. This is the motivation for command pattern. This is also a suggestion from our professor.
Because of the advantage that the Command Pattern method has, we choose to use it to practice the extensibility feature without editing the original code, which could decouple the sender and the receiver. “Drop” module is an example that we use in our program. The module is declared in another module which inherits “Player” modules and can be functioned successfully.

Challenge

1. When we tried to decrease the connection between modules, we had a hard time making modules separate completely. During this term, we discussed the paper *"Structural Design" (Stevens, Myers, & Constantine, 1974)*. The key to the structural design is to reduce the complexity of the program by making modules have functional binding and reducing the coupling between them. We have taken this principle into account in our design. For example, we use the command design pattern to reduce the coupling of the system. But we don't think we're good enough. For instance, in class Room, we need to put in an Item list, a Character list. In class Player, we designed a Room type of variable *position* because we needed to record its location. In this way, class Room and class Player are a highly coupled situation that is referenced to each other. We had a lot of discussion about this situation, and we think that design was as important as implementation and that we could use tools like UML to help us at design time. In our later work, we need to optimize our structure to reduce the coupling of our system.

2. We had a hard time deciding which data should be protected, which could make our program unsafe. In the paper *“On the Criteria To Be Used in Decomposing Systems into Modules” (Parnas, 1971),* It mentions that one of the criteria of decomposing is “information hiding”, which means the interface or definition of a module is chosen to reveal as little as possible about its inner workings. And the paper *“Revisiting Information Hiding: Reflections on Classical and Nonclassical Modularity” (Ostermann, Giarrusso, Kastner, Rendel, 2011)* also mentions if developers have carefully chosen to hide those parts most likely to change, most changes have only local effects: The interfaces act as a kind of firewall that prevents the propagation of change. As a result, we didn’t define any global variable and did set up private variables for each class. Any modification in each module will only affect itself.

3. When using the Command Pattern, our code includes a lot of concrete command class because there is a need to design a concrete command class for each command, which may affect the use of the Command Pattern. Our response is that for some simple

command, we don't have to make it a real "command." For example, Q/ q, which forces to exit the game, simply sets the loop condition to *False*.

4. We tried to apply more ADT method in our program. In the paper *"Programming With Abstract Data Types" (Liskov, Zilles)*, they mentioned that the abstract data types prevent misuse of implementation details which can avoid type errors. In Cook's paper *"On understanding data abstraction"*, it mentions that an ADT has a public name, a hidden representation, and operations to create, combine, and observe values of the abstraction. However, when we tried to use more of ADTs to practice abstract structured class, it is controversial to the Command Pattern method, which holds a lot of concrete command class.

Reference

Cook, W. R. (2009). On understanding data abstraction, revisited. *ACM SIGPLAN Notices, 44*(10), 557-571. doi:10.1145/1639949.1640133

Liskov, B., & Zilles, S. (n.d.). Programming With Abstract Data Types. 50-59.

Ostermann, K., Giarrusso, P. G., Kästner, C., & Rendel, T. (2011). Revisiting Information Hiding: Reflections on Classical and Nonclassical Modularity. *Lecture Notes in Computer Science ECOOP 2011 – Object-Oriented Programming*,155-178. doi:10.1007/978-3-642-22655-7_8

Parnas, D. L. (1971). On the Criteria to Be Used in Decomposing Systems into Modules. doi:10.21236/ad0773837

Stevens, W. P., Myers, G. J., & Constantine, L. L. (1974). Structured design. *IBM Systems Journal, 13*(2), 115–139. https://doi.org/10.1147/sj.132.0115

Parnas, D. (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), 1053–1058. https://doi.org/10.1145/361598.361623