Plan of Attack - CC3K+

CS246 A5
July 15, 2022

Howard Zhu, Yangfan Wang, Chris Qin

Introduction

The main components of the project are floors and entities. There are five floors and five chambers per floor. Entities are composed of treasures, items, potions, enemies and players. Enemies, treasures and potions are generated on the floors, and the player can move on the floor to slay enemies to get treasures or items or to pick up treasure and potions. Enemies can move one square randomly within the confines of the chamber.

When implementing the game, we have two controller classes. We use the Interpreter class to control all basic features of Floor, Chamber, and Cell class, whereas the Entity class controls all basic features happening on Floor.

Project Breakdown

1. Create UML and plan.pdf
2. Create Code skeleton/interface
3. Implementation of code
   3.1 Implement Cell, Chamber, Floor classes
   3.2 Implement Treasure, Item, Potion classes
   3.3 Implement Player, Enemy classes
4. Code Compile/Test
5. Debug/Finalize
6. Finish final documentation and UML

Responsibility/Task Division

Due to the difficulty of transportation and traits of OOP, our team has decided to implement different classes separately according to the UML design. Our main goal is to finish core classes/basics as soon as possible, the team will create a code skeleton first and compile/test it individually to ease the difficulty of the merge. After rounds of revisions and testing/debugging, we will finalize the code and produce proper documentation.

Cell, Chamber, Floor	**Chris**
Treasure, Item, Potion	**Howard**
Player, Enemy	**Candice**

Schedule

Date	**Task**
July 13	UML
July 14	Plan.pdf
July 15	Revision
July 18	Code Skeleton/Interface
July 20	Class implementation
July 22	Compile/Test
July 25	Debug/Finalize
July 26	Demo Presentation Practice/Final Revision

Question 1: *How could you design your system so that each race could be easily generated? Additionally , how difficult does such a solution make adding additional classes?*

We could use the Factory Method pattern to generate our race, as it features both high cohesion and low coupling. We have two abstract classes, Player and Enemy, which serve as interfaces for subclasses – all the races are inherited from the abstract classes, and we only need to override the core methods to adjust how each class interacts with items, potions, cells, etc. If we want to add another class, we could simply create a new subclass to extend our factory method (without needing to adjust anything else), which is a key advantage of the Factory Method pattern.

Question 2: *How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?*

We still use the Factory Method pattern for extensibility, since Players and Enemies share many common fields and the same parent class. Generation

of Enemies is largely identical to generation of the player characters, as the underlying structure of both classes are greatly alike. If we want to add a new enemy, we do not need to change large sections of code, and can simply extend the existing Enemy class to fit our requirements/goals. If we do not use Factory Method pattern, it would be very tedious to add a new class, since we do need to consider all the dependencies again in that case.

Question 3: *How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?*

Since we are planning to use the Factory Method pattern, each subclass of Enemy can store differentiated versions of methods, thus allowing for special abilities and interactions unique to certain subclasses. We would simply need to implement a virtual method within the Entity class, which we then override. Another thing to mention is that we decided all fields of classes should be protected for encausplation purposes – thus, for enemies to mutate player's fields, we have decided to apply polymorphism and the Visitor design pattern, so each will call the corresponding get_attacked method of the Player for different effects.

Question 4: *What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?*

We do not utilize any specific design pattern to model the effects of temporary potions. Instead, since all temporary potions only boost attack and defense, we store an additional base attack and base defense statistic within the Player class. When the player advances a floor, the attack and defense stat are reset, removing the need to track potions used.

Question 5: *How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? That is for example, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code? How could you reuse the code used to protect both dragon hordes and the Barrier Suit?*

We generate items like Treasure, Potions, and major items as subclasses of the Entity class, which allows them to have parameters like HP and protectors. For instance, this allows us to track protectors for both dragon hordes and barrier suits using the same inherited field and methods. Items having HP also allows us to add in additional features, such as all items being destroyable by player attacks, without needing to reuse code. Furthermore, as the Cell class (which represents an individual tile within the game map) is designed to store a single Entity object, we can easily attach any item (like a Potion of Treasure) to a Cell. All these factors allow us to minimize the amount of code reused between types of items.