**ChamberCrawler3000: Plan of Attack**

Kunlin Ran, Yuxuan He, Yujia Cao

**UML:**

Additional Information:

* We are planning to use vectors and smart pointers for all implementations. However we just write normal pointers in UML for now.
* Zombie and Midterm are two additional subclasses we will try to implement after we meet the basic requirement.

**Project Breakdown：**

* Interface for classes(Floor, Chamber, Character, Player, Enemy, Item, Potion, Gold)
* Implementation

1. Complete implementation for Floor and Chamber, be able to display the empty board
2. Implement methods for the rest of super classes and subclasses
3. Create the interaction between Floor and each classes, be able to display the board with all items and characters
4. Implement interactions between Enemy and Player, Play and Item
5. Implement main to control the game, be able to take input from users
6. Debug
7. Add DLC

**Timeline and Task Distribution:**

|  |  |
| --- | --- |
| Date | Task |
| July 15th | Writing UML, deciding design patterns for classes |
| July 16th | Complete interface(header files) |
| July 18th | Complete Floor and Chamber, and basic requirements for Character and Item |
| July 21th | Finish all implementation |
| July 22st | Compile the program, debugging |
| July 24rd | Adding DLC |
| July 25th | Testing, final documentation |

**Who is Responsible:**

Every group member is responsible for designing the programs. For the Floor and Chamber class, we will try to do the implementation together as much as possible as it is the main controller of our program. The rest of the features can be done separately as all the subclasses will be put into separate files.

**Project Summary:**

Our Implementation will be divided into three parts: Floor, Character and Item. For Floor class, an additional class Chamber will be implemented to separate the Floor into five separate parts. Therefore, it is easier for us to generate Character and Item within its range. Floor is our main controller of the entire program. Everything will be initialized and updated in Floor.

We have the base class Character containing all the values for characters, and all characters are implemented as subclasses of it. Same design applies to the Item class. We will use visitor pattern to implement the interaction between player and enemies.

**Q&A:**

**Question.** 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?

**A:** We will use an abstract base class Character and all different races can be implemented as subclasses. We will override the constructor, destructor and some methods of the base class in subclasses so that we can maintain the variety of each race. In this way, each race can be easily generated, as they will all use the base class constructor. Adding additional classes will be easy as well.

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

**A:** Generating enemies will be the mostly the same as generating player characters as they all share common field(HP, atk, def, .etc). However, player will be only generated once in one game. We will use a base class containing common fields for all characters. Different enemies will be initialized by using rand() function.

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

**A:** Special abilities will be achieved by overriding the attack(player \*) method in each enemy subclasses. We can add different features in this method so that player and enemies will be updated differently.

**Question.** 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?

**A:** We can use design pattern to model the effects of potions each floor. We can use the decorator class to update the fields of the player every time we want to change it. When the player goes to the next floor, we can regenerate an undecorated Player class to get rid of the effects of temporary potions.

**Question.** How could you generate items so that the generation of Treasure and Potions reuses as much code as possible? That is, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code?

**A:** We will use an abstract base class Item containing the common field of treasure and potion (potion, name, symbol,.etc). Treasure and Potion can be implemented as subclasses and different features will be achieved by overriding methods.