**Plan of Attacks**

**Breakdown:**

|  |  |  |
| --- | --- | --- |
| Task | Completion Date | Worker |
| Gold | July 16 | Yuxi |
| Potions | July 16 | Yuxi |
| PC | July 16 | Bruce |
| Enemies | July 16 | Shuheng |
| Factory | July 16 | Bruce |
| Observer | July 17 | Shuheng |
| Integration & Normal Controller | July 19 | Together |
| Grid | July 19 | Together |
| Testing & Debugging | July 20 | Together |
| Bonus - WASD | July 21 | Yuxi |
| Bonus - more PC | July 21 | Bruce |
| Bonus - more enemies | July 21 | Shuheng |
| Testing & Debuging | July 21 | who wrote the part |
| More Bonus? | depends on time |  |
| Final UML | July 22 | Together |
| Final Document | July 23 | Together |

**Questions:**

*Player Character:*

1. How could your design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional races?

* We are using “Factory Pattern” to generate different races. With an abstract class “Factory” and its subclasses to generate different races depending on what is entered by user.
* The function to generate the character returns a type of “Character” but not specific race so that adding additional races would only require to change the list selected from by adding a new subclass.

*Enemies*:

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

* Basically, there is little difference between generating players and generating enemies except that players are chosen and enemies are randomly generated according to specific probabilities. However, the remaining is quite similar, including we use same method to spawn characters randomly and use the same constructor from Character class to construct enemy objects. Note that we will place and initialize the Dragon separately from other enemies due to its special properties.

1. How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

* We will use inheritance for implement various abilities. We created an abstract superclass for the “Enemy”. With only the basic attributes in the superclass, every subclass inherited from it will have their own implemented abilities.

*Item*:

1. The Decorator and Strategy patterns are possible candidates to model the effects of potions, so that we do not need to explicitly track which potions the player character has consumed on any particular floor. In your opinion, which pattern would work better? Explain in detail, by weighing the advantages/disadvantages of the two patterns.

* The advantage of Strategy Pattern is that it requires less and much simpler encapsulation because in our case, we have 6 different types of potions, all of which have different behaviours. Thus, our base potion should have pure virtual behaviour and each subclasses have their own implementations, which make the max use of Strategy Pattern.
* The disadvantage of Decorator Pattern is that we shouldn’t have more than one effects on a single potion. Hence, it’s kind of a waste to have 6 decorators as we know we can only use one of them.
* In our own implementation, we will combine the Strategy Pattern with the Visitor Pattern in order to avoid code duplications. To be more specific, we will use six potions, namely the vistors, to visit two elements, i.e. Drows and Others.

*Treasure*:

1. 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?

* On one hand, both Potions and Treasures are inherit from the superclass Item, in which case they will have similar fields and methods. They will override to achieve their specific features and share the other methods for the sake of code reuses.
* On the other hand, when we randomly generate their positions in Grid, we will use the protected helper function randomSpawn from Item to generate a random position yet not occupied by others.