Plan of Attack

Before we explain in detail what our specific plan of attack is, please read our schedule and rationale for why that partner is developing that specific class. Please note the Date to be completed chart has a ±0.5 day uncertainty.

Schedule

Classes TBI*	**Date to be Completed**	**Partner & Rationale**
Game, Controller, View, GameNotification, Main	Nov. 28	Both. Completing these files are crucial since they control all aspects of the program and hold everything.
Floor, Game	Nov. 29	Both, Refine floor and its relation to game and all other objects on the map, game is a very important class thus both members should work on it
Cell, Floor	Nov. 30	Brian, Created UML for floor and Cell, understands algorithms for both classes
Enemy	Dec. 1	Zack, worked on UML part for this
Player	Dec. 1	Brian, worked on UML part for this
Enemy Subclasses, EnemyVisitor	Dec. 2	Zack, worked on UML part for this, worked through logic of Visitor Pattern for enemy
Player Subclasses, PlayerFactory	Dec. 2	Brian, worked on UML part, worked through logic of Factory pattern for Player
PotionEffect, PotionEffect Subclasses	Dec. 3	Zack, worked on UML part, worked on designing Decorator Pattern for PotionEffects
Item, Potions	Dec.3	Brian, worked on UML part, has understanding of items, potions
Treasure, DragonTreasure	Dec. 3 - 4	Zack, worked on Enemies, can implement DragonTreasure easier
DLC or Finish base program	Dec. 4 - 5	Either add finishing touches or 1 or 2 DLC, buffer day for program.
Prepare Final Documents	Throughout	Preparing the final document and updating UML are both of our jobs

*TBI: To be Implemented

Why this order?

We take a more top down approach to developing this piece of software. Starting from the classes which control many different objects, such as controller, game, view, and main. Since there are **required** for the program to run properly and due to our limited timeframe, we want to have our program run first. Then we move on to some of the essential parts of the game, such as the floor and cells. These classes helps map out everything on the map, and handles the loading of the map, thus these two classes are very important. Then we proceed to do more crucial elements that make this game what it is. This includes Enemies and Player, as well as their subclasses, which are needed for the game to be a game, and thus are prioritized over item. After those classes are done, we we can then create the items, and create the dragon and dragon treasure. Developing in this order helps us create a runnable program, regardless of whether or not we reach our deadlines.

Questions

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?

The Player class has a function called playerFactory() which enables the user to use the factory method when choosing a player race. This function calls a race constructor depending on the argument passed and returns a pointer to the newly constructed race. This type of implementation would make implementing additional races extremely easy because all you would need to do would be to create the class and then add the constructor to the list of ones that can be called in playerFactory().

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

In our code, we will have an array of vector of cells. Each element in the array will indicate one chamber and the vector will contain pointers to all the cells in that chamber. We will start of by randomly choosing a chamber to spawn the monster in. Then we will randomly choose a cell within the chamber to spawn the monster. The cell is checked to see if it is occupied or not. If the cell is occupied, randomly choose another cell within the chamber to spawn it. Once a chamber is completely full, we will remove it from the list of chambers which are still capable of spawning enemies. Once the cell is chosen, we will randomly choose a monster to spawn based on the probabilities provided to us.

The special case is dragon. We will spawn dragon within a radius of one of a dragon hoard. The dragon will spawn immediately after the dragon hoard is spawned. If the

dragon hoard is spawned and happens to be completely surrounded (in other words, dragon is unable to be spawned within a radius of 1), then that spot will be removed from the list of valid candidate spawning cells as there must always be a dragon within 1 radius of a dragon hoard. The game keeps on randomizing until it finds a spot where a dragon hoard can spawn and there is at least one spot within a radius of 1 that a dragon can spawn.

The similarities between the generation of the player and the enemies is that both will have to randomly choose a chamber to spawn in and then randomly choose a cell to spawn in. The differences occur in the fact that the player will get to choose the race of the “hero” that they spawn whereas the type of enemy spawned is always random. Additionally, the player will never have the problem of not being able to spawn in a certain cell since the player is the first object spawned in the game.

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 implemented a Visitor pattern for the attack and defense of different enemy characters since there can be a large number of enemies each with their own unique attack or defense pattern based on the enemy type. By using visitor pattern it allows us to easily define attack and defense patterns and be able to use them when the player attacks them or when they’re defending an attack from player. If the ability is on that activates every turn, that is handled by EnemyTurn method in game in every game loop.

We don’t use visitor for player because there is only 1 player, and can incorporate their ability if the ability modifies attack or defense in each respective class. If the player has an ability every turn, that is handled by the PlayerTurn method in game in every game loop.

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.
We chose to use the Strategy Pattern over Decorator Pattern for one simple reason. Since a Potion Effect Decorator would need to inherit Player in order to overload methods getAtk and getDef in the Player class, we would also have to define the pure virtual functions attack, defend, and move, which are irrelevant to the class and makes the Decorator unclear on its intent. Furthermore, resetting the player using decorator pattern would involve creating a new player with the players base stats, and deleting all

the decorators which would increase complexity and decrease readability. Instead, using a strategy to keep track of the player's temporary attack and defense which resets at the end of the floor is much or clearer, and requires less coding with the same degree of effectiveness.

In general, some of the advantages of Decorator is that it is very useful when developing variants of the same thing that won't necessarily be included, such as a TextWindow which may require or may not require a horizontal and vertical scroll bar and a border to the window. It's very difficult for the Strategy pattern to apply itself here, but the Decorator gets it done instead of defining multiple subclasses with every combination of the different variants.

In the end, we used a modified strategy pattern to store a player's temporary attack and defense, and provided the functions getAtk and getDef for the player's getAtk and getDef functions, and setAtk, and setDef to allow potions to edit the strategy and to reset the fields easily after the player has cleared the level.

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?

Both treasure and potion are subclasses of a class called item. To reuse as much code as possible when spawning items, we will have a private class called spawnItem which takes in a pointer to item. What this function does is choose a cell on the grid and spawns the items of the passed argument at 10 different locations. When it's time to spawn potions, a for-loop will execute the following 10 times:

1. randomly choose potion type
2. create a potion with said type and create a pointer to said potion
3. call spawnItem with pointer specified above as the argument

The following process above will be repeated for treasures. If the treasure created is a dragon hoard, a dragon will be created as well and placed within 1 space of the dragon hoard. This process reuses spawnItem 20 times, removing the necessity to create 2 separate spawn functions and eliminating code duplication.