Plan of Attack

CC3K+

CS246

Malcolm Yeh
Zayaan Moez
Kai Lu

Project Summary

The project comprises of the game core (main.cc and game.cc), the floor, characters and items. The floor class includes the rendering the map, generating enemies and items and storing game data including player object, stairs and levels. The characters include the enemies and the player. The characters will include all the movement, combat mechanics and special attributes and buffs. The items include gold hoards, potions and barrier suit. The game core is divided into main.cc and game.cc including the main driver and the functions for input, output and all other game mechanics.

Project Breakdown

1. Creating project structure/ UML
2. Creating Interface files for all the classes
3. Implementation
    - 3.1. Creating the floor implementation including basic map
    - 3.2. Creating character and player class and implementing basic movements
    - 3.3. Creating enemy classes and items
    - 3.4. Finish random generation for all characters and items (finish floor)
    - 3.5. Implementing game core and main.cc
    - 3.6. Creating Makefile and compiling
4. Testing and debugging
5. Documentation and final UML

Estimated Completion Dates

Date	Task
July 23rd	Create UML and Plan of Attack
July 24th	Create interface (.h) files
July 25th	Complete basic Floor and player

	movement
July 26th	Finish all class implementation
July 27th	Finish Floor and game core (main.cc)
July 28th	Compile and Debug
July 30th	Testing, final documentation

Responsibilities

For core components of the program such as the Floor and Chamber class, and the game core, we will work closely together as it is important to get the basic functionalities of the program working as soon as possible. From there, we will divide the rest of the program as follows and work separately.

Malcolm	Characters: Enemies
Kai	Characters: Player
Zayaan	Items

Once the main core of the program is working, it will be easier to test out smaller functions individually. If the group member is finished with their section, they will add bonus features and/or work on the final documentation. Once all the parts are implemented, we will once again work together to debug and test the complete program.

Q&A

Q: *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: The base class Character will be an abstract class, with Player and Enemy being subclasses of Character. Races will be implemented as subclasses of either Player or Enemy (*see UML.pdf for more details*). Adding additional races would be very easy as it inherits the core members and methods from either Player or Enemy.

Q: *How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?*
A: The system could generate different enemies using rand() and the given probabilities. Each chamber has a given range implemented within the chamber class and the enemies would spawn within the chamber walls because of that. An enemy is created randomly and spawned in a random chamber, where as a player is created based on the type the user chooses but spawned similarly to the enemies in a random quadrant.

Q: *How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration from trolls, health stealing for vampires, etc?*
A: Assuming that the special abilities are triggered by attacks (vampires have life-stealing attacks, goblins have gold-stealing attacks, trolls regenerate fixed amounts while attacking), the implementation of special abilities can be done by overriding the attack method inherited from the Character class (specifically, *dealDamage(opponent: Character *)*). If special abilities were to be implemented as a move separate from the normal attack, we would create a new method for the individual ability and add a new mechanic where the enemy has a chance to use an ability in combat instead of using a basic attack.

Q: *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: Using the decorator pattern provides the best solution here. We created a decorator class called buffs with subclasses hp, attack and defense. Each of the concrete decorators can either increase or decrease the given attribute based on what potion was used. Everytime a potion is used, a decorator can be applied to the player. At the end of each floor, a getPlayer() function is used to get the original

undecorated player. The changes on health, attack and defense could be implemented by overriding the getHP(), getAtk() and getDef() functions.

Q: *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 hoards and the Barrier Suit?*
A: Not unlike how races will be handled, the Treasures, Potions would be subclasses, inheriting from the parent abstract class Item. We have decided to streamline how the other major items work. The Compass was removed altogether as an Item. We opted instead to treat it as a boolean member of the enemy class to track who holds the compass. Upon the death of the enemy that holds the Compass, the visibility of the Stair will be directly altered in the Floor class. The spawning of a Dragon, along with either the Dragon Hoard or Barrier Suit will be handled in the game core as a helper function, thus reducing any redundancies.