Plan of Attack

Geoffry Song and David Choi

# Estimated Program Structure

The Game class, which extends a CommandHandler, will be the base class of our game. It will handle input from a UI class (a basic one for the specified display, and a curses one) which displays input and reads commands. Game will control the Player and run the game loop (reading input, processing everything, and calling display). Because there can only be one Game and one UI, they are singletons.

For display, there will also be a Display and Displayable class. The Display will contain all objects that need to be drawn each iteration, and will sort them by z-index to decide what to draw on top of what. The objects that need to be drawn each iteration will be Displayables, which simply represent objects to display and have methods that aid in display (like linking and unlinking with the Display).

To help with this, we will have a class called Level which represents a floor in the game and handles all enemies, potions, treasures, and the actual dungeon layout itself. These will be represented by subclass Monster, Potion, Gold, Staircase, and Dungeon respectively. Monster, Potion, Staircase, and Gold will be LevelObjects (object contained within a level). A LevelObject would have an x-position, y-position, movement, and a tile, all things shared by objects within a level. A Dungeon would generate the layout when created, which would be used when Level generates Monsters, Staircases, Potions, and Gold to add to the floor. Level would every iteration of the loop call each LevelObject’s step method, which allows the LevelObject to take its turn. Notably, Level would not control the player, instead the game would.

When Potions are used or Gold is stepped over, it would be passed a Player pointer through its use method, and it will operate on it. When a Staircase is stepped on, it will send a message to its observer Game to tell it to generate a new level.

Monsters and Players will extend Character (representing something alive) which extends LevelObject. Monsters and Players will then be extended by races specifying the features of each race (like Vampire or Elves). For a Monster, each iteration its step method would be called, and it will attack (getting its neighbours from Level) or wander randomly. If a Monster dies, it will notify its observer Level to report its death. Players will be controlled by the player, so they will not be explicitly stepped each iteration, and will not notify Level of its death. Instead, the Game will tell Player if it should move, attack, or use a potion, and Player will notify Game of its death.

To determine if a Monster wants to attack a Player, each character will have a Team. Team will be a multiton, each of which has its alliance status (Ally or Enemy) with every other team. A Monster would then check each neighbouring Character’s team and attack if and only if the two teams are enemies. Merchants, Monsters, and Players would be the 3 teams, with alliances specified in the requirement and the Merchants and Players teams will un-ally if the Player kills a Merchant.

# Estimated Schedule

|  |  |  |
| --- | --- | --- |
| Estimated Completion | Part of the Project | Partner |
| November 16 | Initial framework planning | Both |
| November 16 | Initial framework, class structure, and curses display | Geoffry |
| November 17 | Add a display of player status (bottom bar) | Geoffry |
| November 17 | Add a way for the game to handle commands | David |
| November 17 | Implement player movement | David |
| November 17 | Add the display specified in the requirement | David |
| November 18 | Plan how players, monsters, gold, and potions work | Both |
| November 18 | Add a way to represent how stats are stored | Both |
| November 18 | Add monsters and their movement | Both |
| November 19 | Implement player and monster attacking | Geoffry |
| November 19 | Add potions | Geoffry |
| November 19 | Implement player and monster death, and restart | David |
| November 19 | Add teams | David |
| November 19 | Add gold | David |
| November 20 | Implement Field of View algorithms | Geoffry |
| November 20 | Add races | David |
| November 20 | Add staircase | David |
| November 20 | Add dragons | David |
| November 20 | Add victory/death screens | David |
| November 21/22 | Finalize, bugfix, and document basic game version | Both |
| November 23 | Field of view/lighting | Geoffry |
| November 23 | Colour and title screen | David |
| November 24 | Enemy AI | Geoffry |
| November 25 | Classes | David |
| November 26 | Floor generation | Geoffry |
| November 27 | Level up mechanics | David |
| November 27 | Plot | David |
| November 27 | Final floor (boss fight) | Geoffry |
| November 28/29 | Finalize, bugfix, and document final game version | Both |

# Questions:

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 classes?

We could create a base player class from which races inherit. This system would allow the us to simply add a new subclass of player to create a new race, so we could reuse the code for player. Depending on what classes are the difficulty varies, but in our planned implementation (a class is an array of skills), it would be easy because we could simply add a class variable to the base player class.

Question 2:

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

Our system has a base “Monster” class from which specific monster inherit if necessary. Because of this, we can reuse most of the code of monsters and only implement methods when necessary. Since the only difference between many of our monsters is their HP/Atk/Def, this means that for most monsters, there is no subclass necessary. This is different from how we add player races only because not all monsters have special abilities so we do not always need to have a subclass.

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.

We could create subclasses of the “Monster” class that override certain methods (like those that relate to attacking for goblins) to add special ability functionality. For example, in the step function for trolls, they could regain health after calling Monster::step().

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 could use the Decorator pattern. We made an AttributeProvider class from which there is an AttributeDecorator, which returns the statistics of the base class modified by the potion’s effects. A slight extension of the pattern was a strip() method which removes all decorators and returns the base provider. This method could be called when changing floors, to not explicitly track which potions were consumed.

Question 5:

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 do not duplicate code?

We could have a base LevelObject class, which represents any object on a level and includes placement methods like setPos(), and a Dungeon class, which can return a valid placement following the given rules. Then, for any LevelObject, a Level would simply ask Dungeon where it should put that LevelObject, the place it there. If treasure and potions extend that LevelObject, we could avoid duplicating code.