Introduction

We have chosen to build ChamberCrawler3000+ (CC3k+), which is a simplified rogue-like video game. In this game, the player must travel through 5 floors in a dungeon and win the game by reaching the end of the dungeon at the 5^{th} floor. During the playthrough, the player will encounter different types of items, treasures, and enemies, and the score is counted at the end of the game.

Overview

In terms of the overall structure of the project, we first have the Game class which is responsible for processing user inputs and determining whether the inputs are valid commands. The Game class is also responsible for race selection of the player character, as well as resetting and quitting the game.

Next up is the Floor class, which is the central processing class for our implementation of CC3k+. The Floor class has a grid of GameObjects that acts as the current state of the board, and it is responsible for randomly generating all components of the floor, which includes PC location, stairway location, potions, gold, and enemies. Moreover, the Floor class will process player commands like moving, attacking, and using potions, and send appropriate information to the Player class and other related classes to be processed further. Lastly, the Floor class has a TextDisplay field that will be printing the current state of the board onto the terminal, as well as displaying PC's current HP, Atk, Def, and actions.

The TextDisplay class is inherited from the Observer class, and its sole purpose is to output the current state of the floor onto the terminal. To do this, it has a 2D vector of chars and this grid is updated whenever the grid in Floor is modified.

Our version of the game makes heavy use of the Observer design pattern. As such, our program has an Observer class as well as a Subject class that stores the basic attributes of a GameObject, which are its row, column, and ObjectType. The GameObject class is inherited from the Observer and Subject class, and it is a collection of in-game objects which includes Character, Compass, BarrierSuit, Treasure, Potion, and MapComponent.

The Character class is a representation of characters in the game, which includes both the playable classes in the PlayerRace class and the enemy class in the Enemy class. The Character class contains all the attributes of an in-game character, which are previous (the MapComponent that they are currently stepping on; e.g. Tile, Door, Passage), current & max HP, and base Atk & Def. The PlayerRace subclass inherits from Character, and it contains the functionalities that are unique to the playable classes, which are collecting gold/compass/barrier suit and drinking potions. In our version of the game, the playable classes are Human, Dwarf, Elves, Orc, and the hidden playable class God. On the other hand, the Enemy subclass also inherits from Character, and it contains the attack functionality as well as special skills for some types of enemies. The enemies in our version of the game are vampire, werewolf, goblin, merchant, phoenix, troll, and dragon.

The Compass, BarrierSuit, and Treasure class represent the in-game items that the player can collect by stepping over the item. Since the Barrier Suit and gold hordes of value 6 cannot be picked up until the dragon guarding them has been killed, an extra field has been added to BarrierSuit and Treasure. The Potion class is an abstract base class that represents the potions player can drink during the game, and it is a generalization to the six types of potions in the game (PH, BA, BD, RH, WA, WD). Since PH and RH are permanent potions, their effects are handled in the PlayerRace class. On the other hand, since the effects of BA, BD, WA, WD need to be cleared when the player enters the next floor, each of these classes have their own getAtk() and getDef() function that overrides the player's current Atk and Def.

The final subclass of GameObject is MapComponent, which is a generalization of HWall, VWall, Door, Passage, Tile, and Stairs. Since Stairs are not visible until the player collects the compass, it will have an extra field that represents its visibility to be processed by TextDisplay.

For a visualization of the structure of our project, please refer to the UML at uml-final.pdf.

Design

There are several major design challenges that we encountered during the project:

Random Generation of the Floor

One of the greatest challenges we've encountered during the project is the random generation of the locations of player, stairways, gold, and enemies. In particular, it's difficult because it's stated that "a larger chamber should be no more likely to spawn the PC/stairs than a smaller chamber" in the project specification. This means that for every tile on the floor, we first need to figure out which chamber the tile belongs to before we can proceed onto the generation. Other elements about the random generation are also quite difficult:

- Making sure the player and the stairway are generated in different chambers
- Making sure the number of enemies, gold hordes, and potions are correct
- Making sure that a gold horde of value 6 & barrier suit is spawned with a dragon

Making this process even more difficult, our program needs to have the ability to process a command-line argument that specifies the floor layouts.

To solve this, we use 5 vectors to store the locations of tiles in each chamber. This is because all in-game objects can only be spawned on the tiles. After we have all the tile locations, we start the generation process by shuffling each of the 5 vectors so that the locations are now randomized. When we generate an object, we first randomly choose the chamber that the object will be placed in. Since each of the 5 vectors have an equal chance of getting selected, this satisfies the condition that an object has equal probability to be spawned in any chamber. Next, select the 1st element in the vector as the spawn location, and remove that element from the vector so that no other objects can be spawned in the same location. We repeat this process for the correct number of enemies, gold hordes, and potions, as well as ensuring that gold hordes (value 6) & barrier suit is spawned with a dragon.

Handling Different Interactions using the Observer

As discussed earlier, our version of CC3k+ makes heavy use of the Observer design pattern. This is because there are a ton of interactions that can happen between different components of the game. We will discuss the different use cases of our observer and how it handles the interactions between different elements of the game:

1. Character & Map Components

When a character decides to move, they will notify the map component that they are currently stepping on. The map component will update its row and column to the player's original location.

2. Player & Enemies/Items

When the player kills an enemy that is not a dragon or a merchant, the enemy will notify the player and the player will collect 1 gold for the kill. Similarly, when a player walks over a gold horde/compass/barriersuit, the item will notify the player and the mutator corresponding to the item will be called.

3. Dragons & Gold Hordes/Barrier Suit

When a dragon is killed, it will notify the item (either a gold horde of value 6, or a barrier suit) that it is guarding. This will make the item available for the player to walk over/pick up.

4. Stairs & Compass

When the compass is picked up, it will notify the stairs so that the stairs can now be visible.

5. Floor and TextDisplay

Whenever a change occurs on the grid in Floor, whether by character movement, enemy death, disappearance of potions and items, or the appearance of stairs, TextDisplay will be notified and update the changes to be displayed on the terminal.

The Decorator Dilemma and Designing a new Potion-handling System

Potion was not the hardest aspect when implementing the game, but it is one of the most interesting ones. Since the effects of Atk/Def potions are temporary and we do not want to explicitly track the potions that the player has used, we decided to use a decorator at first. The decorator was certainly functional, but we found out that since the Player is now a Potion class, all of their other functions (ex. collecting gold, getting compass/barrier suit, etc.) need to be overridden in the Potion class. This will drastically increase the coupling of our program and it's something that we want to avoid.

Our solution is to ditch the Decorator design pattern and construct a “pipeline” structure to implement the effects of the potions. The potion drunk by the player will be a decoration on player: we input the original stats, and it outputs the updated ones. Therefore, Potion can have a component of potion representing the potion player drunk before. If the potion hasn’t been drunk, its pointer will indicate a null pointer. When the potion receives an Atk/Def, it lets its component process this value and updates it. This structure is efficient as it’s easy to manage the memory and once the player goes to the new floor, potions can just be directly recursively deleted.

Resilience to Change

We have organized our modules in a way such that changes to specifications can be easily supported, and this is done by achieving low coupling and high cohesion. We’ll discuss how this goal is accomplished in detail in the paragraphs below.

We have taken several general procedures to maximize our project’s resilience to change. We have extensively used constructors, accessors, and mutators in almost all of our modules instead of allowing multiple modules to have access to the same data. This effectively lowers the coupling of our program as modules are less interdependent. Another measure that we have taken to reduce the coupling of our modules is separating the different player commands into different elementary functions, instead of writing them in one giant function. For example, player movement, attack, and potion usage are handled via separate functions, which reduces dependencies to a minimum.

As discussed earlier in the “Design” section, our project makes heavy use of the Observer design pattern, which reduces the coupling one step further. Take the relationship between the grid in Floor and TextDisplay as an example, if a state in grid is changed and we want TextDisplay to be updated, if the Observer is not present, Floor needs to explicitly call a function in TextDisplay, increasing the coupling between the modules. However, with the Observer pattern and through the notifyObserver() function, an explicit function call is not required, and Floor is not coupled to the concrete class of TextDisplay. As such, through our heavy use of the Observer pattern, we are effectively lowering the coupling of our project.

Our program has achieved high cohesion by using multiple levels of inheritance for the collection of in-game objects, each with an increasing level of cohesion. We first have the GameObject class, which is a generalization of all the in-game objects including characters, major items, potions, and map components. These objects are related since all of them can show up on the game floor and having this GameObject class is extremely beneficial when setting up the floor grid.

The Character class is inherited from GameObject and it represents the characters (both playable and enemies) in the game. As such, it has several fields that represent a character's basic attributes like HP, max HP, and base Atk/Def. One level down and we have the PlayerRace class, which holds the collection of playable races. At this point, the cohesion is very high as the playable classes are very closely related. For example, all playable classes have the ability to attack, collect items, and drink potions, so we can have the implementation of these functions done in the PlayerRace classes instead of writing them separately in their own subclasses. Similarly, the Enemy class has very high cohesion since all enemies share very similar attributes.

This design enables us to support the possibility of change to a great extent. Most of the character functionalities are covered in either the Character or the PlayerRace/Enemy class, which makes the creation of new character races extremely easy. All we need to do is to build a new subclass that inherits from PlayerRace/Enemy and set up its base stats in the constructor. Moreover, changes to a player/enemy's stats can be easily modified by changing the number in its constructor and nowhere else, which is an indicator of low coupling. This is what enabled us to create new playable classes like "God" as a bonus feature. In fact, since this design is consistent throughout GameObject, it's very easy for us to set up new types of in-game objects, whether it be new types of weaponry/armour or different types of potions.

As discussed above in the "Design" section, we have chosen to use a "pipeline" system to handle the effects of temporary potion to lower the coupling of our program. The system uses data coupling, where the player's original stats are passed into the function in Potion, and the updated stats are returned. In contrast to using the Decorator, which has to override every function in PlayerRace, the Potion class only has to override the getAtk() and getDef(), drastically reducing the coupling of the program. This is very effective because adding a new function to PlayerRace will not have any effect on Potion.

Lastly, changes of specifications in terms of random generation are also made easy with our design. We have constructed helper functions in the Floor class that specifically handle the random generation of a single enemy/treasure/dragon/potion (randomPotion(), randomEnemy(), randomTreasure(), geneDragon()). This simplifies the generation process by a significant margin, and this is what enabled us to create different difficulties of the game (with different number of enemies per floor, etc.) as a bonus feature. This is especially crucial because while creating a new difficulty of the game was made effortless with our design, they can bring brand-new experiences to players and it adds a lot more playability to the game.

Answers to Questions in Project Specification

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 implemented a Character superclass that is a generalization of PlayerRace and Enemy, which are generalizations of specific character races like humans, dwarfs, vampires, and dragons. When we want to generate a character of a specific race at a specified location on the grid, we can just call "grid[i][j] = new [insert Character Race]{…}". Implementing such a solution requires making a new subclass for each character race, but it reduces the complexity of character generation by a great extent, and this is the approach that we decided to use. Another advantage of using this approach is that it's very easy to build additional character races, since it can be done by adding a new class with only its constructor.

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 is slightly more complicated than generating the PC, but the process is very similar. When generating the PC, we first pick one of the five chambers on the floor. After that, we choose one of the tiles in that chamber to be the spawn location of the PC. We follow the same procedure of choosing one of the five chambers on the floor. However, since the enemies are generated after other components are generated, we first determine the available tiles and then randomly select one as the spawn location. After that, we randomly generate an enemy race based on the specified probabilities (2/9 werewolf, etc.).

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:

We implemented special abilities for different enemies by adding a virtual method specialSkill() in the Enemy class. Enemies that have a special skill, such as vampires and goblins, will have this method overridden. SpecialSkill() is called whenever an enemy attacks, and the effect will apply whether or not the enemy's actual attack lands. As an example, when a vampire attacks the player, the attack has a 50% chance of missing, but its special ability of blood stealing will activate, taking 5HP away from the player to add onto its own HP.

We also have special abilities that only activate under specified conditions. For example, our Phoenix class has the ability to be reborn, which activates when the Phoenix dies and returns the Phoenix with half the original max HP, increased Atk and decreased Def.

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 originally thought that using the Decorator design pattern was a good idea to model the effects of temporary potions (stated on DD1), but as discussed above in the "Design" section, we ultimately decided that the coupling would be too high for our likings and we used a "pipeline" system instead. Each of the temporary potions have a getAtk() and getDef() function, which takes in the original Atk/Def of the player and returns the updated Atk/Def. Implementing this system was actually a bit easier than implementing the Decorator, and it has the same effect of not needing to explicitly track which potions the player has consumed on any floor.

Question: 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 hordes and the Barrier Suit?

A:

We reused a lot of code for the generation of different items, as they all follow a very similar procedure of randomly selecting a chamber and then randomly selecting one of the available tiles as it spawn location. To avoid as much duplicate code as possible, we have a 2D vector "gene" that contains the remaining available spawn locations, and it is used by all generation functions. To reuse the code for protecting both dragon hordes and Barrier Suit, we created a geneDragon() function that generates a dragon within one-block radius of the horde/barrier suit. Moreover, Dragons will have a private field indicating the item it's guarding, and the player cannot pick up the item until the dragon is slain.

Extra Credit Features

There are several bonus features that we implemented in our program:

- The hidden "God" playable class – totally overpowered class with 1000 base HP and 500 base Atk/Def. Since we did not make a random seed argument, this playable class speeds up the playthrough and makes testing a lot easier
- Varying difficulties – we have made 4 modes of difficulty (easy, medium, hard, impossible), each with different number of enemies and items on each floor. Obviously, going hard mode means facing against more enemies, but it also means more potential kills which leads to higher scores

Final Questions

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

The most important lesson that I learned about software development in this project is the importance of UML and the planning that occurs before coding. The UML and plan document that we made for DD1 helped us a lot as it sets up the foundation for our implementation of the game. Another crucial lesson is the importance of effective communication. We found that actively communicating sped up our development & debugging process by a significant margin, as we can each provide a new perspective/approach to the other member. This project was time-consuming to say the least, and having a partner really helped out for both of us.

2. What would you have done differently if you had the chance to start over?

If we have the chance to start over, we'd definitely follow our planned schedule. There are several days where we slack off and made little to no progress and catching up to schedule was an absolute pain. Time was very limited even till the end, and we'd certainly need to work on our time management. Another aspect that we can improve on is the use of smart pointers. We tried adding them after we finished the program using normal pointers, but it didn't work as planned so we ended up just using normal pointers. If we have the chance to start over, we'd definitely use smart pointers from the beginning.

Conclusion

Just like how the game ends when the player reaches the end of the dungeon, this project comes to a close as we submit our version of CC3k+. This project was certainly a memorable experience with mixed feelings, but in the end, it seemed to be turning out alright. Keep adventuring.