A5 Project Final Design Document

Timac Wang, Zheng Pei

Overview

Controller
The logic of the game such as taking turns, starting the game, printing action messages, and ending the game will be handled by the Controller class, the Controller class has access (directly or indirectly) to all the class objects and is responsible for handling all data exchange between different classes. The role of the Controller is similar to the controller in an MVC model (and it is inspired by the model).

The Controller owns Floor and Character, which subsequently owns almost all other classes or is the superclass of other classes. The floor is responsible for initializing itself and can print itself.

Floor
The Floor class owns the Chamber class, which has a vector of Tiles and has methods that allow the floor to get a random empty tile from it. This will come in handy when spawning objects randomly with equal probability in each chamber.

FloorGenerator

The class that's responsible for generating random floors.

Tile
Many classes have instances of the Tile class; the Tile class can access private fields of Floor since it's a friend of Floor. It represents a coordinate, and it can be a coordinate on any floor since it's not tied to a specific floor, which makes it very flexible to use. One class must have access to the floor (which will be given by the controller) to be able to use the Tile class to perform operations on the floor. Tile class provides many useful methods for checking the type of Tile and the type of objects on it. It is a utility class specifically designed for the Floor.

Util
The Util class is independent of the class hierarchy and provides methods for getting random numbers, setting random seeds, and receiving input with built-in error checking. This class is not logically tied or related to any class in the hierarchy.

Item
The item class is the parent class for the gold and potion class and it holds some common fields for these two children, the field currently will be used by the Player class to modify the display of the tile after the player either picks up the gold or potion.

Gold
A subclass that belongs to the Item class.

Potion
An abstract class that forms the observer pattern with RH, BA, etc. (concrete observers), Character (abstract subject), and the Player (concrete subject) classes. The Potion class itself acts as the abstract observer in the pattern, it provides an abstract method notify() which is typically used in observer patterns.

When attaching observers, triggerPermanentEffect() will be called, when proceeding to the next floor, all observers will be detached except the potions stored in the player's inventory if the "Inventory DLC" is enabled. This ensures that permanent potions get to preserve their effects (already done by triggerPermanentEffect), and all temporary potions lose their effects (no longer observing the subject), RH BA etc.

Character

The abstract class Character holds common fields of all player races and enemies and acts like the abstract subject in the observer pattern. The Character class will have some virtual member functions that would be used in common Observer Patterns such as attach(), detach(). One virtual method this class possesses is the attack() function which will be used for combat. The tile field is what the controller will use to get information about the position of the character and move the character around.

Player
As the name indicated, the "Player" class is designed for the player the user would control. It is a child class of the "Character" class and it is a concrete "Subject" class. The most important method it has is the takeTurn() method which will accept some commands from the Controller and do corresponding things in conditional statements such as moving the player through the chamber, attacking the enemies, and picking items such as "Potions" or "Gold". The "Player" class is the parent class for all the different race classes.

Race classes (Goblin etc.)
These classes will be the children class of the Player class and they have special overridden functions. Take the example of Vampire, which will gain 5 HP for every successful attack and has no maximum HP, but it is allergic to dwarves and loses 5 HP rather than gain. To achieve this, we have overridden the attack function provided in the Character class to add an if statement that checks the type of enemy, if it is Dwarf, then subtract 5 from the current HP field, otherwise add 5.

Enemy and all its subclasses
These classes are very similar to Player and Race classes. Its most important method is also the takeTurn() method. It receives commands from the Controller and lets the enemies do corresponding things such as attack the player if the player is in attack range or perform random moving if the "f" key is not pressed. Some subclasses also have overloaded functions because of their abilities and behaviour logic. Due to their similarity to the Player subclasses, we will omit details here.

Design

Encapsulation:

The design of the Controller and Floor class promotes encapsulation – the Floor class and the Character class have no direct access to each other. Furthermore, the Floor is friends with class Controller and class Tile only. This means that if any class other than the controller want to modify the grids on the floor, that class must use the Tile class which only allows modifications of one tile at a time. This is again another level of encapsulation

Loose Coupling:

The classes we designed are highly independent, for example, the tile class is not tied to a specific floor and can react on any floor that's passed in as a parameter. The methods that receive or print messages to the player do not directly use cout or cin but use the ostream and instream references they receive instead. The list goes on.

High Cohesion:

We adhered to the single responsibility principle, that is, every class is only responsible for one thing and has one reason to change. The controller class is responsible for the interactions between different classes. The player class is responsible for all the logic relevant to the player. The util class provides methods that are commonly used in the other classes but are not tied to the logic of one class.

Observer Pattern:

As described in the overview section.

Class Inheritance:

This can be easily observed in the UML diagram. All the fields and methods shared by the child classes are moved to the superclasses (parent classes), significantly simplifying the implementation.

Resilience to Change

Due to our usage of abstract classes and inheritance. If we want to add more races for the player character or the enemies, or new kinds of potions, we can simply create more concrete children classes of the abstract class and override some methods to implement the new abilities. For example, if we want to add a Berserker character, we simply create a Berserker class that inherits the Player class and overrides the attack() method where the Berserker increases its ATK every time it slains an enemy.

Due to our implementation of the Tile, Floor and Chamber class, the dimensions of the dungeon and the number of potions can be customized by modifying the const values and do not need modification of any logic.

The template method requestInput() in the util class makes adding different commands become very easy. The function provides built-in error checking using cin.fail(). The type of input that it can receive and the logic that it uses to check if the input is valid can be easily customized using lambda functions.

All the methods that receive or print messages to the player do not directly use cout or cin but use the ostream and instream references they receive instead. If we want to change the output to a file, we can easily do that by switching cout with ofstream.

Many classes use constant values to define the parameters used when performing random generation. Some even allow change during the run time (FloorGenerator). These parameters defined using constant values allow a program to modify the way the game runs without having to look at the underlying logic. For example, you could change the number of potions generated on every floor by modifying the NUM_POTIONS field in floor.h.

Answers to Questions

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

We decided to use Abstract Class & Class Inheritance to make the generation of each race easier since all races have some common fields such as HP, ATK, and DEF, and some common methods such as attack. This information will be stored in an abstract class called Character, with two concrete subclasses, Player and Enemy. All the races will be concrete children of either of those two classes.

Some races are born with special abilities or weaknesses, we will implement these traits by overriding the corresponding methods provided by the superclass. The example of Vampire is provided in the Breakdown section. To create an instance of a race, all one needs to do is instantiate the corresponding class, such as the Vampire class. The constructor would only need a tile which represents the location of the player and nothing else. The placement of the Player onto the board is handled by the Floor class, which holds a list of chambers and will randomly select a chamber, and a random tile within that chamber to place the Player (Will be passed in the Player's constructor), if there's a stair in that chamber, then reselect a random chamber and repeat.

Adding additional races would also be easy as one just needs to add an additional concrete class and override some superclass methods as we did in the case of Vampire; most methods and fields will not need any modification.

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

There will be some additional field for the enemies which indicates whether they have moved already or their movement is paused or not. The move method would also be different since enemies move in

random directions unless they are fighting, or it is a dragon. Other than that, implementing an Enemy class like Human or Halfling is no different from implementing a Player class. All one needs to do is override methods to accommodate special abilities.

The placement of enemies onto the board would also be handled by the Floor class, different from players, enemies have no restrictions when being generated, with the special case of a dragon that must spawn next to a dragon hoard. This process will be handled by a for loop that randomly selects a chamber with a random empty tile to place the enemy, and randomly selects an enemy type from an array, it will run until 20 enemies are generated.

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

The techniques will be nearly identical, with some exceptions such as the ability of Dwarf and Halfling. Their abilities will be implemented in the attack method of player characters because that is really when their ability triggers. Other abilities such as elf's and orc's abilities will be implemented by overriding the attack method provided in the enemy class, just like how player characters are implemented.

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 decided to use the Observer pattern to implement potions, the pattern is composed of Potion (abstract observer) RH, BA, etc. (concrete observers), Character (abstract subject), and the Player (concrete subject) classes. The Potion class has two virtual methods, triggerPermanentEffect() and notify(). The potion with permanent effects like RH and PH would provide a meaningful implementation in the triggerrPermanentEffect() method (directly modify the field of the player pointer), while does nothing in the notify() method. Temporary potion classes do the exact opposite of this. When attaching observers, triggerPermanentEffect() will be called, when proceeding to the next floor, all observers will be detached. This ensures that permanent potions get to preserve their effects (already done by triggerPermanentEffect()), and all temporary potions lose their effects (no longer observing the subject).

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 does not duplicate code?

Both potion and item will inherit from the Item class which will provide all common fields for easier implementation. Their interaction with the player will be implemented in the Character class with the pickGold() and consumePotion() methods (Both are private so not shown in the UML diagram).

The Floor class will handle the generation of Treasure and Potions, and just like how Floor generates enemies, it will use a for loop that randomly selects a chamber with a random empty tile to place the Potion, and randomly select a type from an array. The probability of generating different gold piles can be

done by generating random numbers between 1 and 8 and associating each number with a type of gold pile, if a dragon hoard is selected, then we need to check to make sure that there's an empty adjacent tile for the dragon to spawn.

Extra Credit Features

Selection of DLC

During the runtime, the option of selecting the DLCs will be available to the users after selecting the race of their characters and the game seed. The users can choose to enable one or more of the DLCs according to their preferences. All DLCs are compatible with each other and can run as a standalone DLC.

Random Floor Generation DLC

This DLC is algorithmically difficult to implement, the algorithm can be broken down into 3 steps. First, we use a random number generator to generate random room count, room dimensions, room positions. Then we select a random point on the map as the top left corner of one room, attempting to fit the room onto the floor without collapsing with other rooms. Step 2 has 20 attempts to place the room, if a room can't be placed after 20 random attempts, the whole algorithm starts from step 1 again. Repeat step 2 n times, n being the number of rooms.

The last step is to generate passages between all rooms. This is the most tricky step, as we need DFS to find an efficient path between two rooms and use backtracking to find another path if the current path runs into a dead end. While doing these, we also need to ensure that the passages are "slim", that is, there are never two passages that form a 2x2 square. The most significant challenge arises when using DFS, traditionally, DFS will randomly select directions to go and stop (or backtrack) until it finds the destination. However, the passages generated by this method are extremely twisted and too many doorways are generated. Therefore we switched the approach, if we want to connect two rooms, we'll randomly select one point in each room respectively as the starting point and the destination, then calculate the x differences and y differences between the points, and try to connect them with preferably only 1 twist and as little doorway as possible. However, sometimes two points can't be connected without extra twists, so we added one extra boolean parameter to the function that turns on the random DFS algorithm. We'll first run the algorithm without the random DFS, if we fail to find a path, then we'll turn on the random DFS to ensure no room is left without an outgoing passage.

Last of all, during testing, we have to spend lots of time to ensure that no matter how the player customizes the dungeon, the dungeon will have enough room to fit every item. It's quite difficult to find a balance point between fun, giving the player maximum freedom and ensuring the game runs properly without breaking.

Inventory DLC

The Inventory DLC, gives the Player class an additional field - a vector holding potion pointers which represents the potion PC picked up. Three methods are placed in the Player class for this DLC, The printInventory(string&) function will concatenate the names of all the potions stored in the Player

inventory and print to the action message. The usePotionInInventory(int) function will allow the Player to choose which potion in its inventory to consume. The storePotion(Potion*) method allows the player to store the potions he encountered in his inventory instead of having to drink it.

One problem that this DLC brings is memory management, because when exiting the game, instead of deleting all potions in the Floor and attached to the Player, we now have to delete the potions that are in the inventory of the Player.

Trading With Merchant DLC

To implement the Trading with Merchant DLC, the Merchant class is given an additional field called storage - a vector holding potion pointers. When merchants are initialized, two of each RHs, BAs, and BDs potions (Positive Effect Potions) will be generated and stored in the storage. The Merchant class has two additional methods: the printStorage() is literal and sellItem() will erase the sold potion from the storage. The Player class also has an additional method, the buyItem() method allows the Player to buy the potion at index i from the Merchant. The purchase will be successful only if the Player has not attacked a Merchant before and it has sufficient gold.

Again like the Inventory DLC, Merchant DLC brings memory management issues but they are relatively easy to fix by adding destructors to the Potion class. The complicated part of this DLC is the numerous corner cases that must be considered during the testing phase. For instance, we must ensure that the Merchant won't trade with the PC if they are hostile to the PC, the Merchant can't move around when the PC is trying to trade, PC must have enough gold to trade with the Merchant etc.

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?

Planning & Design

From this project, we learnt about the importance status of proper planning and design in creating large projects because they would save time and effort in the long run. Having a well-designed UML chart allowed us to have a clear idea about the responsibilities of each class and the required methods that need to be implemented. It also allowed us to split the work reasonably.

Version Control

During the implementation of the project, we used Github as our version control system and it tremendously improved our efficiency. GitHub allows us to exchange updated codes with each other, track the history of code changes and revert to a previous version if a problem is identified in the new version of code.

Communication & Teamwork

Without effective communication and teamwork, we would never achieve such a high completion rate of the project as it is now. Having regular meetings allowed us to facilitate collaboration and decision-making. We chose the topic of the project the day the document guideline was released. And quickly started working on planning and UMLs. In the subsequent weeks, we have regular meetings around three times a week to ensure that we meet all the deadlines and finish our code with high quality.

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

Design

If we have the chance to start over, we would consider spending more time on the planning process. Even though our UML diagram successfully captured the class inheritance relationships, we were ambiguous on important implementation details. For instance, initially, we coded all the turn logic into a function in the Controller class since it has access to all fields relevant to the game. But we soon discovered this approach makes the Controller class ridiculously long and difficult to change. It is also a violation of the idea of loose coupling (Enemy and Player do not have any turn logic without the Controller). Therefore, we decided to add the takeTurn() method in both the Player and the Enemy class in our implementation which significantly reduced the responsibility of the Controller class. Lots of similar events happened when we started coding, which is reflected through differences of class methods in the first and final versions of the UML.

Smart Pointers & Factory Method

Since we started the project very early, the professors have not covered the implementation of the smart pointers and the factory method. We did not use any pointers factory method in our project. This causes us to use delete in multiple files and is often confused with “who” owns the pointer, this becomes the leading cause of memory errors during testing and is very hard to debug. Missing factory methods forced us to write lots of classes to provide different parameters to the constructor. This caused the file system to be quite overwhelming to look at. If we have the chance to start over, we would consider using smart pointers throughout the project instead of allocating memories using the “new” keyword so we don’t have to explicitly manage the memory. We would also consider using the factory method to code the generation logic of the different enemies which would make generation much simpler.

Static Methods

All the methods in the Util class should have been declared as static methods since they do not depend on any instances of the class.