Chamber Crawler 3000 - Villain

Vibhanshu Bhardwaj, Rut Patel

April 4, 2017

1 Introduction

Chamber Crawler 3000 - Villain is a rogue-like video game that runs on Linux powered terminals. Written in C++, this game was made by Rut Patel (20647619) and Vibhanshu Bhardwaj (20607705).

Players can choose between a variety of characters and fight an eclectic mix of monsters, not unlike the members of CS246 staff, with special powers while coming up with appropriate strategies. Players can take risks by picking up potions or play it safe.

There are a few cool hidden features, so play to find out!

2 Overview

We followed various design patterns and good object-oriented-programming principles throughout our code.

At the heart of our game, we have a game class. The player essentially only interacts with the game object, which 'owns-a' floor, which is where all the action on that floor takes place. We also have a Cell class to allow easy maintenance of the grid and also lets us read in custom layouts for each of the 5 floors.

All Characters inherit from an abstract base class 'Character'. We used design patterns like Model-View-Controller (MVC), factory pattern, pattern, observer design patterns to greatly simplify coding, and visitor design pattern. This allowed us to maximize cohesion and minimize coupling while enabling good separate compilation.

3 UML

Submitted separately at a5-dd2uml as uml2.pdf. (It couldn't be fit to this size)

4 Design

Our design adopts versions of Model-View-Controller (MVC), factory pattern, observer design patterns, and visitor design pattern. This provides us a solid foundation.

The game object is our 'Controller'. It 'owns a' floor object . The game also 'owns' the player object. The game object handles the input the player provides and relays all the actions to the player or the floor accordingly.

The Floor is our Model. All of the logic of the code is implemented in the Floor class. The Floor 'has a' player, and it 'owns' all the enemies since the enemies should be destroyed and regenerated for the next floor.

We use the factory pattern for generating various enemies, and the generator function is overloaded to allow both random generation of characters and specific generation. The latter is used for generating characters that will be spawned on a floor based on the layouts in the given file, and the former is used to randomly generate characters for the default behavior.

Using the factory patter has numerous advantages. It provides a central place for creating different types of characters, allowing us to abstract the commonalities in the constructor of each subclass (different types of players and enemies) to the constructor of the super class Player or Enemy.

Factory pattern is especially great because it allows us to pass default arguments to subclasses' constructors depending on the type of character to be created. Perhaps the greatest advantage of this pattern here is that we can very easily accommodate change. Adding new characters is really easy and we can also change all the players/enemies with one change should the need arise. Thus the factory pattern is very flexible when it comes to accommodating change.

The Floor 'owns' a two-dimensional vector of pointers to Cell objects, and each cell (named grid) has its co-ordinates. So essentially the vector has the entire detailed map of the floor.

We use the observer pattern to allow the movement of characters. When a movement action is passed to the floor by the controller, we dispatch a method which notifies the observer (grid, essentially the Cells) of the desired action. This lets the grid be a perfectly accurate and detailed map of the floor at any given instant. The subjects are the characters themselves.

We chose the observer pattern here because of the following advantages that it provides in our case

1. Seamless Transmission of data. This allows us to supply the grid with the updated position with no difficulty at all.

2. Character Death and Generation. When an enemy dies, we want it to disappear from the floor without affecting any other characters. Since we follow the observer pattern, we can simply remove it from the array of subjects without changing any other part of code. And when a character is created, we just add it to the array of subjects. This is especially convenient.

3. Loose coupling. This allows us to have little coupling between cells and characters.

We use the visitor design pattern to implement attack between player and enemies. When an attack action is passed to the floor by the controller (or as determined by the enemies), we simply dispatch a method that handles it. We want attack to mostly be the same regardless of race of the character. So our interface is rather static. The visitor design pattern also lets the individual races to implement their own functionality and have it work well with the static functionality.

This was the perfect choice for us because that's exactly how attack works- it's mostly the same for all races, but the races can have slightly different attacks.

5 Resilience to Change

As explained in design, using proper design patters allows us to accommodate changes very easy. Proper modularization and little coupling allows for great separate compilation. We could make changes to how we generate a floor, and it would have absolutely no impact on any other aspect of code.

Similarly, adding new characters is completely painless. If you want to use different commands, only one part of the code is changed; if you want to attack differently, only the attack method is changed in exactly one place; if you want to move differently, you just change the move function.

Essentially, for the most part, we can make changes to the behavior rather easily without modifying too much code.

We ourselves changed quite a few things in a very short time period.

6 Answer to Questions

1. Q: How could your design your system so that each race could be easily generated? Addition-ally, how difficult does such a solution make adding additional races?

(i) We used Factory Method calls to generate Enemies and Player while initializing a Floor. We've created several sub-classes for each race which inherits from its abstract Parent class.

(ii) Player and Enemy are abstract classes which inherits from an Abstract Super class named Character and every Player character and Enemy character inherits from Player class and Enemy Class. This kind of connection between the 3 abstract classes (Character, Player and Enemy class) allows us to add additional class easily.

With the help of this design structure allowed us to create new characters with minimal effort. (Created 2 new Characters, 1 Enemy and 1 Player character very easily).

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

   As per given guidelines, every enemy has a specific probability of getting spawned. We used the random integer generation which generates a random number between 1 to 18 inclusive. Enemy is spawned depending on its probability and the length of interval the generated random number 'r' is in.

   For instance:
   - Human: [1, 4]
   - Dwarf: [5, 7]
   - Halfling: [8, 12]
   - Elf: [13, 14]
   - Orc: [15, 16]
   - Merchant: [17, 18]

   i.e. If 'r' is 7, Dwarf is spawned on the floor. Dragon is spawned near a Dragon hoard only.

   Yes, generating player character is different from that of enemy. The client/user selects the race of Player and that Player character is then placed on to the floor (no randomness is associated with the player character selection). After the selection, it will be assigned to a tile in any chamber just like Enemy.

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

   Answer: All Enemy and Player races will use the default abstract character implementation of defend or attack unless it is overridden by that specific race class of attacker/defender. Therefore, all abilities for both Enemy and Player Character will be implemented using the Visitor Design Pattern.

   In our code, attack is a Virtual method in Player and Enemy Class. The default attack action is implemented in both classes: Enemy and Public. And

this attack method is overridden by a derived class which has different action in a specific condition. eg. Orcs do 50% more damage to Goblin

4. Q: The Decorator and Strategy 8 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.

   Answer: 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 but while implementing this pattern, we came up with an alternate and efficient solution.

   Every potion derived class has a static field which is switched to true when the player consumes the potion for the first time, and this bool then becomes true for every Potion of that type. This allow us to keep track of what type of potions a player has consumed at any given instant of the Game.

5. Q: 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?

   Answer: We have implemented Factory method to generate both, Potions and treasures. That is, Factory method is called every time a Potion or a treasure has to be generated. This structure of our system enables us to generate Potions and then generation of treasure without duplicating code.

7 Extra Credit Features

We implemented a couple of new characters, but we also implemented cheat codes (can be activated when the user runs the game). Initially, our program required significant changes in order to make the cheat codes work, and that helped us catch bad design quite early in the coding process.

The user can choose a custom race for its player and it can also use cheat codes to boost its health, attack, and defense.

We came really close to using smart pointers entirely, but we couldn't debug it in time.

8 Final Questions

1. This project was a fabulous learning experience for both us. We both think that this course was the key to actually understand the concepts covered in the course.

This was also the first time both of us worked in a team, and the experience couldn't have been better. We had a ton. We learned how important it is to have tasks yet collaborate and help each other. We also realized how great pair programming can be. It allowed us to make fewer errors, catch errors quicker, and have more fun at the same time. When we were coding together, it hardly seemed like work. We both highly prefer working together over working alone.

2. We would've collaborated more from the very start. We would also have liked to use smart pointers from the beginning now that we know how they work.

9 Conclusion

This was an amazing experience for us. We got to learn a lot while making a really cool game. It doesn't get any better. Thanks, CS 246 staff!