Design Report:

The Game of ChamberCrawler3000

Feiyang Li (f89li)

Jody Zhou (j433zhou)

Wendy Zhang (y3447zha)

Introduction

Long long ago, in the universe far far away, there existed a game. Behold the almighty Chamber Crawler3000. Here, villains can be heroes, heroes can be villains, and anything is possible. Start the game and rewrite your destiny!

The game Chamber Crawler3000 is an imitation of the video game Rogue written by Michael Toy and Glenn Wichman. It is a D and D(dungeon of dragon) type of game where players are given protagonist, game environment, enemy, and then he/she sets on an arduous and mighty advantage with his/her character. In this project we planned to design and implement a similar game as Rogue, recreate the classic game. However, there will be some differences between the real game and our implementation of the game. There are quite a few special features that we plan to implement to spice things up, such as a newly designed end of game dashboard that shows the game statistics, as well as new characters to impersonate and new enemies to defeat. We also plan to implement different difficulty levels, where the powers of enemies differ accordingly.

Overview

We wrote this project strictly following the Object Oriented Programming principles we learned in this course. We used inheritance and polymorphism to implement our Player, Enemy, and Item classes. We used the Observer pattern to update the game after each Player action, and we used the Visitor pattern to implement combat. We achieved encapsulation of classes; all our class fields are private, and we use accessors and mutators when we need to get and mutate them. We also opted to use smart pointers, instead of raw pointers, as an extra feature to prevent memory leaks.

Design

In this project we used two design patterns that we learned in this course: the Visitor and the Observer.

Observer:

The Observer pattern is a design pattern that contains an observer and a subject. The subject contains the memory address of all observers, and when we modify the subject, we trigger the notifyObserver function in the subject. NotifyObserver() then calls the notify() functions of all observers whose addresses are contained in the subject. As a result, any change in subject will

call notify() in all observers, which will achieve the desired result depending on the content of notify() function in each observer. Here in our program, Player actions are the subject. When the Player make any actions (movement, attack, pickup potion), the following will occur:

1. Conditions of the character change this either or both change the character's health, Atk, Def, gold, position, temp Atk, temp Def, and some action will change the stat of enemy.
2. When character action occurred, we might need to update character location inside the map; thus, there action of character will call notify function side the map, which will change the character location inside the map.
3. When a character performs any action, all enemies(except dragon) will move in one random direction; thus, the enemy class is an observer of character, so there is notify inside.
4. When a character performs any action, we need to print the map, and the stat out; thus, floor is an observer.

As a result, any change to character action, will cause change of character stat, or location, will change character inside map, will change location of enemy, will print out map and stat.

Visitor:

We used the visitor pattern to implement combat. Dynamic dispatch helps accomplish the desired effects in an organized way. The Enemy abstract class has a virtual attack() method for each Player race, and one virtual beAttacked() method. Each concrete Enemy class overrides these methods. Since both Players and Enemies can attack and be attacked, we applied the same technique to the Player races as well. The Player abstract class has a virtual attack() method for each Enemy race, and one virtual beAttacked() method. Each concrete Player class overrides these methods.

If we wish to add a new Enemy or a new Player, we only have to add additional attach() and beAttacked() methods to our classes. We don't have to change our existing methods at all. This is resilient to change.

Change in Design from DD1 UML:

Cell

In the code implementation process, we found that each location inside the map could potentially consist of many different components (for example: a person can stand on top of the money, which is on top of the empty floor.) Thus, we created a vector of cells (which is the previous component in DD1) inside components, it is implemented for stacking items, or character, or structure.

Connection between player and vector of enemy
Interface is like the wrapper for the game, it is what users would use to play the game. Therefore, it contained the player, the enemies. Wrapper is like a global environment, where calling methods is made easy, so we can directly access the methods for players, enemies and so much more to achieve our goal.

Utility_tool
We changed the name of display to utility_tool, where it handles printing to the standard output. The functionality remains the same, only the name is changed.

God & Weakling
Since we added new characters as an extra feature, the changes will also reflect on the UML diagram as two new classes that inherits from Characters are added and they are 'God' and 'Weakling'. It also demonstrates how we followed the open close pattern as adding new classes does not need to change the previous implementations, and they are modularized.

Resilience to Change

We strictly followed the open close design principle. We modularized our code so that adding new sub modules will not affect the general functionality of the program. We maximized cohesion and minimize coupling, so if any new specifications are added, we only need minimal changes to our existing program. We used inheritance, polymorphism and design patterns diligently. For example, if we wanted to add extra features like new characters, potions, or enemies, then we can just modify the enum to include the types and then add the corresponding .h and .cc class files. By doing so, it would be easy to add new features, but it will be closed for modification.

Minimal recompilation
We divided our program into many files, and we tried to put one object in one file, so the purpose is clear.
We made sure that each module only contains the things that are immediately relevant to it. So whenever we make a change to our program and recompile, we compile as few files as possible. When in doubt, we will run the command "make clean", and that is the only time we recompile the whole thing.

Inheritance

Many of the classes in our program share similarity, and the differences are very little. For example, all our Player and Enemy classes need health, location, atk, and def fields. However, the exact value of these fields differ, as each race has different starting statistics. Therefore, to reduce repetition of code, we used inheritance. For example, both Human and Merchant inherit from the abstract class Enemy. This keeps things organized and saves us a lot of time. Most importantly, in the future we could easily create additional classes; for example, we can add "superman" as one of the enemies and we need little implementation to do so. **In fact, we did add two additional races as extra features after we finished the core program, with minimal changes to existing code. These are the two characters God and Weakling, which inherit from the Player class, which then inherits from the Character class.** Another example of this is for items and gold, where they inherit from items, but the implementation is very similar for characters and enemies, so we will not go into detail on this.

Encapsulation

We have many classes, which each mean different things. However, each class belongs to a group, which the implementation can be seen above in the Inheritance section. We used private, and protected for variables that should not be used by other classes. Thus each component works as a package, in which each groupmate can use parts of the package without understanding what is actually inside the class. In the human interaction section, we created a large class called interface, which handles all smaller packages; as a result, players, and coders who want to modify the code, by adding packages can add to the interface, which would change the behaviour of the game, easily.

Answers to Questions

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 races?

We will use inheritance to create our race classes, because all characters share common attributes such as HP, Atk and Def regardless of race. We will have an abstract class Character, from which both our Player and Enemy abstract classes inherit from. Then all Player races are derived from the Player abstract class. This allows each race to be easily generated when needed. If we wish to add an additional race to players, all we need to do is to add another derived class for the class Player. Then, we also follow the Open/Closed Principle because new enemies or player races can be added without modifying existing code, which makes it easier to add on new features. For example, new player races can be added as a new feature with minimal effort as the general structure is similar compared to other existing ones.

(This question is the same as our DD1 answer.)

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

We use inheritance to design different enemies. The way enemies are generated is quite similar to the way that players are generated. We have an abstract Enemy class deriving from the Character class, and all Enemy races are derived from the Enemy class. This allows each Enemy race to be easily generated when needed. If we wish to add an additional race to enemies, all we have to do is add another derived class for the Enemy class. One vector in the Floor Class fields contains all empty coordinates on the map. We will use srand(), rand(), and this vector to randomly generate positions to insert each of our enemies. Before we generate enemies, we will create a vector of 18 enemies, according to the possibility of the occurrence of each enemy. For example, the possibility of generating a Dwarf is 3/18, then the vector of 18 enemies would contain 3 Dwarfs. Then, we randomly select an enemy from this vector of 18 to append to the map.

(This question is the same as our DD1 answer.)

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

We have decided that we don't need to implement the various abilities for the enemy characters within the Enemy classes. We will use the Visitor design pattern between the Enemy and Player classes and have double dispatch take care of performing the desired effect. The Enemy abstract class will have a virtual attack() method for each Player race, and one virtual beAttacked() method. Each concrete Enemy class will override these methods. Since both Players and Enemies can attack and be attacked, we apply the same technique to the Player races as well. The Player abstract class will have a virtual attack() method for each Enemy race, and one virtual beAttacked() method. Each concrete Player class will override these methods.

(This question is the same as our DD1 answer.)

Question. The 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. In your opinion, which pattern would work better? Explain in detail, by discussing the advantages/disadvantages of the two patterns.

We ended up not using a design pattern to model the effects of potions. We felt that the Decorator pattern is not necessary for implementing such a straightforward and relatively easy feature. We wrote less lines of code this way without using the Decorator pattern. We modified the statistics of the Player through the Player pointer in the interface.

(This question is different from our DD1 answer.)

Question. 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?

We designed an abstract Item class, containing a virtual method pickup(), from which our Potion and Gold classes derive from. Each concrete Potion class and the Gold class will then override pickup() and modify the powers of the player accordingly. This way, code duplication is reduced. Also, there will be a dedicated method for generating potions and gold randomly, so these methods can handle the location.

(This question is the same as our DD1 answer.)

Extra Credit Features

Smart pointers

We used smart pointers throughout this project. We didn't use "new" or "delete" anywhere. Before smart pointers were introduced, debugging for memory leaks was time consuming and tedious. After using smart pointers, less compilation errors surface, which improves productivity for the whole team. Smart pointers are crucial to the success of this project as we basically used it everywhere from map to characters, and we don't need to worry about manually deleting it. Since, we passed the vector of the smart pointers around, then that means we cannot use unique pointers as it lacks the property of reference.

Custom map

Our project accommodates custom maps in addition to the default map we provide. The player can start the game with any custom map, as long as it's consistent in format with the default map (same characters for walls, hallways and empty tiles). The player can simply execute the program with the custom map file as an argument. This extra feature was relatively easy as the logic is the same as the default one except the extra step of reading from the command line.

If we had more time, we would auto generate new maps every game to make it more engaging and fun.

Multi-block movement

When writing the project, we found that using single block movements to navigate the floor is time-consuming and tedious. As a result, we implemented multi-block movement. This made testing a lot easier. We dedicated a lot of time towards this as it can save a lot of hassle when during regression testing, and speed up the process. In the original implementation, we ran into trouble with the while loop as the standard input stops reading, which results in an infinite loop. The original input format is like "direction #", so something like "ea 10" will prompt the PC to go east 10 blocks. Then, we realized the input format needed to be tweaked as commands like 'a' and 'u' are needed for attack and use potions. We then collectively decided to rewrite the whole main file to incorporate the new command format of "# direction", which is suitable for commands like '10 ea', 'a ea', 'u ea', 'f', 'ea' and so much more. Having this implemented and fixing the conditionals so it would break properly. The original approach is to read into a string and a number, but that will not work very well with single commands like 'f' or 'a ea' as the character's ascii number is read instead of as a character. The solution we came up with instead is having two strings and checking to see if the first string is a direction, if not the second string will be default to direction and the other string will be switched into cases and evaluated separately. Since, we used stoi to convert string into numbers for multi-block movement, that means sometimes if the input is neither the provided there will be a problem, which is a case that stoi is used to characters. Then, it is too annoying to break out of the program due to accidently inputting the wrong thing, so we added a try and catch block on the stoi to ignore the error and ask for the input again.

Local high score

Any game that contains a score will have some kind of score tracking system, so we will implement something similar. Since, we have no access to any apis or web servers, we will just store the high score locally in a text file. Originally, we thought the file can be read and changed at the same time, but it turns out it is not possible in c++, so instead a new file will be created and renamed to highscores.txt, and the old file will be removed, then the contents of the original file will be stored in a variable to compare with the user's high score and the higher one will be stored inside the new highscores.txt file. This feature is important for users to be engaged and continue to play the game and reach higher scores!

Secret characters

We realized how fragile the original player characters are, so we decided to add two secret PCs to the game, so it can help us test the game better, so we have "God" which has a lot of HP, ATK, and DEF, so it is undefeatable, then we have "Weakling", which is very weak with 0 ATK and low HP. We mainly used "God" to walk through the game and experience it ourselves and it is

denoted as “$” on the map. Then we have “Weakling”, which is just tilda “~” in the game. We mainly used “Weakling” to test for the end game after consuming the death potion of ph, which will immediately drop the health to 0, so the game should end immediately. It is also easy to check end game initiated by enemy attacks as “Weakling” has very low HP, so it will not be able to defend for a lot of rounds.

Final Questions

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?

Developing software in a team is a lot more fun than developing software alone. We resolved issues and bugs in our programs a lot faster when we encountered them, because all of us would contribute ideas. Working in a team also allows each person to work on what they are most comfortable with. Both of these things promote efficiency tremendously. We almost never got stuck on anything, whereas we often got stuck when doing individual assignments. We also had the chance to learn better coding styles from each other.

We learned that it is very important to come up with a detailed design strategy before starting to code. We spent three days coming up with our design strategy, discussing and working out all the little details. This hard work paid off, as our strategy ended up working flawlessly. We also wrote the main groundwork of the program (moving and updating) together, which sped up our process later working separately because we knew how the shared part works. As a result of our strategy, we were able to finish the project without breaking a sweat.

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

There’s not much that we would do differently if we had the chance to start over. We had a great collaboration. Everyone was very enthusiastic about the project and contributed wholeheartedly. The only thing that we would do differently is to start writing the demo and final design document earlier. We finished the code very early on, but were rushed in the end on writing the documents due to exams crowding near the due date.