CS246 Spring 2022 Final Project

The Game of ChamberCrawler3000+

Final Design Document

Ruolan Mao / Tianlan Shen / Chun Ye

Table of Contents

Introduction

The Game of ChamberCrawler3000+ (CC3k+) is a simplified rogue-like where the player character moves through a dungeon, slays enemies, and collects treasures until reaching the end of the dungeon (the end of the 5th floor). Each floor consists of 5 chambers connected with passages. In this simplified version of CC3K+, the display is going to be terminal-based, and the player needs to type in specific commands in which the command interpreter supplies to play the game. In addition, CC3k+ differs from other rogue-likes in a significant way: it only updates the terminal/window whenever a new command has been read in, rather than updating the display in a real-time manner.

Overview

This project is implemented in a way that structures around and maximizes the advantages of object-oriented software design. Classes are implemented to separate and classify the various types of objects within the game (i.e., enemies, players, treasures, etc). More specifically, to accommodate changes in a clean and flexible way, the observer software design pattern is constructed to model the changes happening to both the player and the enemies within the cells. The decorator software design pattern is being employed such that the effects of potions on players need not to be explicitly tracked. Detailed explanations of the employed techniques and reasonings behind the implementation are included in the "Design", "Resilience to Change", and "Answer to Questions" sections.

Updated UML

See "uml-final.pdf" for an updated UML.

Design

We have incorporated OOP design principles throughout the implementation. To clearly outline the blueprint of the program, the best approach would be to give a broad overview of our UML. First and foremost, three enumerated classes are defined to enhance readability: the eight direction types, human race types, and some miscellaneous cell types. There is an attribute under the Cell class (the highest superclass) named CellType to keep track of each individual cell. Nearly all methods within this class are pure virtual, such that its subclasses can easily implement their own versions.

Directly inherited from Cell is the Character class, which serves as the parent class for Player and Enemy base classes (will be explained in “Answer to Questions” section). Organizing Player and Enemy under one superclass prevents the duplication in the construction of the shared attributes and methods, to name but a few: HP, Atk, Def, and their respective accessors.

Going down, we have classified each player and enemy types into classes as well. By following this design hierarchy, we could generalize all the methods in the base class and override the respective versions in the concrete subclasses. For the enemy types, only the beAttacked function needs to be overridden as their attributes/functionalities remain stationary unless they get involved in combat.

The decorator software design pattern is being utilized when temporary potions are consumed by the player character. Implementation-wise, a decorator class is constructed as a subclass of the Player to model the temporary effects (corresponding to the requirement of implicit tracking).

The other subclass directly inherited from Cell is the Thing class. The Thing class serves as an organizer of different types of treasures to facilitate their effects on the player and the game as a whole. As mentioned in the previous paragraph, potions are being registered under the decorator class in a sense that when the player goes out of the current chamber, the temporary effects could be easily withdrawn.

In order to correctly print out the layout of each floor after random generations, the observer software design pattern is employed. A TextDisplay class is constructed to accommodate this feature. There is a two-dimensional vector embedded within this class to store the actual characters being displayed onto the screen. To further improve game visualization, ASCII color codes are used to segregate different types of objects. TextDisplay is therefore being registered as an observer to each cell object to perceive the changes happening on the floor.

Next, we have three standalone classes named Chamber, Game, and Floor, who are friends with each other. Floor is where the generation of all other objects occurs, with the exception of the player character (happens within Game). Since the player is the only object that should not be destroyed when the old floor is deleted, its corresponding operations (attack, pick up potion, set race, etc) are being separated to the Game class. The Chamber class is there to aid random generation.

Finally, we get to main.cc, which acts as a controller of our program, and handles the acquisition and deallocation of inputted resources.

Resilience to Change

We have designed the program in a way to facilitate the easy addition/subtraction of functionalities to the game, to name but a few: the addition of new races and enemies, and modifications to the rules. Specifically, the general structure of our program follows the principle of object-oriented software design while adhering to the concepts of low coupling and high cohesion.

For each race type of the player, respective subclasses are constructed under the Player base class. Similar implementations have been done for both treasures and enemies. By following this methodology, adding a new race/enemy/treasure only requires constructing a subclass under the respective base class. It would be advantageous both implementation-wise and efficiency-wise as most of the methods can be directly inherited or overridden.

Our implementation of potions is exceptionally resilient, in a sense that the addition of new potion types only requires modifying the existing Potion class (the rest of the program remains inertial). By examining from a high-level viewpoint, our implementation is being realized through the cooperation of multiple classes. The program is therefore highly cohesive and maintainable by definition.

Encapsulation is an utterly crucial component to software programs as well; therefore, we have endeavored to use classes as opposed to structs and create fewer public methods. All the attributes of each class are private, such that only the subclasses would have access to these properties, enhancing program safety. Friend functions are declared only when it is absolutely necessary.

In addition, global variables are avoided such that no modules are sharing global data. By adopting this design pattern, the detrimental effects of coupling would be minimized.

We have thus justified that our program is resilient to changes.

Answers to Questions

- **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?**

  The inheritance mechanism was adopted such that each race could be easily generated. According to the project specification, different races only differ in terms of Health Points (HP), Attack (Atk), Defense (Def), and abilities. Other methods and attributes would be similar amongst all races throughout the game. For instance, enemies are indifferent about the race of the character; therefore, relatable methods can be shared across all races. To address these conditions, we implemented a Player class which serves as the base class of all race classes and includes all shared features defaulted to human. If the player is not human, only the features changed need to be re-implemented (other ones are inherited from the base class), saving tons of time.

  The Template Method design pattern which we included in the previous submission, turned out to be inapplicable as the functions we need to override are uniform across all race types with the exception of the players' abilities; however, their abilities are being reflected in different circumstances accordingly (for instance, the reversing potion effects of elves are reflected whenever a potion is being picked up, and halving gold effects of orcs are reflected whenever gold is applied to the player), which are not under the domain of the Player class.

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

  We used similar methodologies to handle the generation of both enemies and the player character; however, as the player character is the only object that should not be destructed when a new floor was spawned, it is being generated in a different class such that when the destructor of the Floor class (where all other generations take place) runs, it remains present.

  Apart from the previous keypoint, their generations are fairly interchangeable due to the fact that both players (specific race types) and enemies (specific types) do inherit from their respective base classes. Each time a new object is spawned, adding a corresponding subclass would suffice.

- **How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?**

  The special abilities for different enemies can be implemented by inheriting and overriding the beAttacked function in the Enemy base class, such that they could be implemented

when attacking takes place. This methodology makes sense as their abilities do not need to be reflected unless the player character is being attacked.

- **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 used the Decorator design pattern to model the effects of temporary potions. Since this pattern allows us to "add functionality or features to an object at run-time rather than to the class as a whole", by registering each of the temporary potions as a decorator to the player character, we could implement the temporary effects within the decorator class. For instance, when the chamber number of the player character changes, the effect is going to be withdrawn. In this way, we do not need to explicitly track which potions the player has consumed on any particular floor; in other words, the player does not need to "remember" what is in effect, which makes the implementation easier and cleaner.

- **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?**

  The generation of treasures and potions do share many similarities in terms of the algorithm. On one hand, the process of their generation in each chamber is the same, except for their differing probabilities. Thus, we could use the same algorithm to generate random positions for both treasures and potions: First and foremost, we select one of the five chambers with equal probability. Secondly, we pick a random position in this chamber for treasures or potions. The only difference would be the number they need to generate, which could be solved by changing the number of loops that generate one object. As opposed to our answer in the first submission, treasures are no longer the decorators of the player character. The reason is that while the player is picking up the treasure, simply adding the gold value to the gold attributes of the player would be enough to document all the changes. In this case, making the treasure to be a decorator could be redundant. On the contrary, potions are much more complicated to be effective. They change various aspects of the player.

  In response to the second question, the main difference between the Barrier Suit and the dragon hoard is that the Barrier Suit is not associated with a gold value. While the Barrier Suit is under the Item class, we consider the Barrier Suit to be a subclass of the Treasure class by setting

the attribute value to 0. In this case, the Barrier Suit will not affect the gold of the PC. Before the dragon is defeated, the Barrier Suit/the dragon hoard will be noted. Thus the dragon hoard/the Barrier Suit could be accessible.

Extra Credit Features

- In the project specification, there are no specific requirements with respect to print winning messages and classifying the player into different tiers based on their final scores. To further enhance user experience and entice user interests/motivation, we have implemented this extra functionality following the following rules:
    1. If final score = 0, print “REALLY? THAT'S ALL U GOT?”
    2. If 0 < final score <= 5, print “Hey don't give up, man.”
    3. If 5 < final score <= 15, print “Yo, u r getting better at this.”
    4. If 15 < final score <= 50, print “U did a grrrrrreat job.”
    5. If final score = 246, print “Difficulty set to: HELL!”

- As the difficulty level of the original game might be high. We added a developer mode for this game. The developer mode increased the player’s Atk and Def in large scale. It also makes the stairs visible. Using this mode, the developer of the game is not required to be highly proficient at this game to test and debug. In addition, the developer mode is beneficial for the demo. It would be much easier to demonstrate all the aspects of the game without trying many times.

Final Questions

The answers to the following two questions are agreed upon unanimously among group members after thorough introspection, contemplation, and discussion.

- **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?**

    This project is designed in a comprehensive and pedagogical manner such that we are able to both utilize the knowledge gained in class and learn to work with people (soft skills that are going to be helpful at the workplace). The lessons the project taught us about developing software in teams are as follows:

1. Conduct Unit and Integration Tests on Key Functionalities

   As this project is relatively large-scale (compared to the assignment questions), the return on investment of conducting unit and integration tests on key functionalities would be high. Throughout the entire project, multiple unit tests are performed to ensure the correctness of the current components before moving on to the next ones.

2. Conduct Performance Tests of the Program

   Performance tests are essential in a sense that they verify the run-time performance of the final deliverable. The grey-box testing approach has been used to better carry out the testing process. An extensive list of test cases (including edge cases and error handlings) have been prepared to test the robustness of the program.

3. Collaborate Remotely via GitHub and VS Code

   To facilitate remote collaboration, a version control system, namely GitHub, came into play. Before doing the group projects, all of us have zero to little knowledge about remote collaboration; therefore, we took the time to explore and familiarize ourselves with these platforms. The learning curves turned out to be steep, and a significant amount of time has been saved.

4. Synchronize Coding Styles

   Group members have different preferences in terms of coding and documentation styles. To ensure code maintainability and documentation consistency, all our group members have negotiated and agreed upon the style to employ amongst all three. Line-by-line re-examinations are done by each group member after the entire program has been coded.

5. Communicate and Allocate Tasks

   All group members have learned about the importance of communication and allocating tasks wisely within a software team.

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

  If we had the chance to start over the project, the followings are the areas in which we would have done differently:

1. Use the Observer Software Design Pattern in TextDisplay Class From the Beginning

   When designing the UML, we thought about registering the TextDisplay class as the observer to each Cell on the floor, but we ended up abandoning this methodology due to efficiency concerns. During the implementation process, however, we have experienced various unexpected barricades which made us decide to adopt the original design structure. If we had the chance to start over, we would utilize this pattern at the beginning as it would save us an abundance of time.

2. Writing Pseudocode Before Designing UML

   For the deadline 1 submission, we did not informally write out any pseudo code before designing the UML. We simply brainstormed as a group and used mind maps to organize the general hierarchy of the program; however, if we ever got the chance to redo the entire project, all of us would prefer to write some pseudo code before drafting the UML, as the overall structure would not be fully sorted out until the completion of the coding part. This effectively prevents process loss.

Conclusion

In conclusion, this project serves as an excellent consolidation of the knowledge covered in CS246. Throughout the project, we had the opportunity to utilize various object-oriented programming design patterns and learn to collaborate in a software project. The project is designed in a way that adheres to the rules of the NVI idiom, polymorphism, low coupling, high cohesion, and so forth. Not only could we enhance our understanding of the core concepts, but we also could acquire a solid project experience which could be beneficial to talk about on various occasions.

This project marks the end of one story (journey of CS246), yet the beginning of another one (journey of computer science, or whatever our career endeavors are). Cheers.