Project CC3K

August 1, 2023

Justin Wang (j238wang), Kevin Zhu (k4zhu), Tracy Hua (t2hua)

Introduction

ChamberCrawler3000 (CC3k) is a simplified rogue-like game where players explore a dungeon, battle enemies, collect treasure, and progress through five floors. The game features floors with five chambers connected by passages. Players can choose to be different races with unique abilities. Their goal is to defeat various types of enemies through turn-based attacks to collect gold and reach the end of the dungeon. The game incorporates items such as potions which provide various effects on player characters. The design uses various patterns to implement and command line arguments to control the creation of characters, the procedural dungeon generation, and embracing interactive environment elements.

Overview

From a programming perspective, the project revolves around the interaction between game elements. The program incorporated strategies of object-oriented programming, inheritance, and various design patterns, to promote code modularity and reusability. The program uses a model-view-controller type of pattern to relate all parts of the program. The game mechanism involves a main game loop to take in commands from the user, in order generation of items and characters on the game map, random movements of non-player characters, and visual display of the state of the game word into the console.

From a play's perspective, the game opens with a welcome page with the title of the game. Then it instructs the player to select a race. After that, a map is shown on the screen with coloured elements on the basic map. The player can then enter the commands that control the movement of player characters and try to get to the last floor of the game map.

Updated UML class model

Please see the last page of this pdf.

Design

The design of our game focuses on the phrase “high cohesion, low coupling”. That is, have clearly organized modules with parts that are related to each other or have the same goal together, and at the same time, separate unrelated parts of the code into different modules with less dependence between modules. Modules with high cohesion are easier to understand, modify, and maintain because their functionality is self-contained, and changes to one part of the module are less likely to affect other parts. Modules with low coupling are more flexible and resilient to changes, thus they can be modified, replaced, or extended with minimal impact on other modules. In order to achieve the goal of “high cohesion low coupling”, we also emphasized the single responsibility principle in designing the structure of the program. Overall it makes the debug process, maintenance, making changes, and reusing codes for the program easy. The program has a few key modules: main, start, map, cell, item, and character. The program uses a model-view-controller type of pattern, in which the start module is the controller, and the map class is the viewer.

Main

The module is responsible for reading the command line arguments and calling corresponding functions to start the game. There are three options. The player can either choose to just start the game, which will randomly generate the items on the map with a random seed. Or the player can choose to start the game with a known map, or with a map and a seed. The information about if a map or seed was given would be passed to the start module, this technique allows the program to handle different inputs and accommodate various user requirements.

Start

This module is responsible for starting and exiting the game, as well as read in commands during the game. It is the controller module of the game. As the game starts, it initializes the game with command line arguments read in from the main function. Then it introduces the game and guides the player through the process to select their player character. It then is responsible for reading each command during the game and calls corresponding functions in the map module to perform actions. By having all commands processed in a single module, it enhances readability of the code and allows easy modification when changes to commands are needed.

Map

This module is the viewer of the program. The map module is responsible for generating elements on the map and directing most actions for the game. There are different mechanisms for setting up the map depending on the arguments read in from the main. If a map is not given by the user, the program reads in an empty map in .txt form, then the class randomly generates

elements on the map following the given mechanism: with all chambers and each cell in the chamber having an equal chance following the order of player character location, stairway location, potions, gold, and enemies. As these elements are being generated, their location information is being saved. On the other hand, if a map is given, the program just reads in the given map and saves all elements on the map as it reads. Utilizing file input and output for reading map data from external files allows for easy map design and modification without the need to modify the code.

The map has an aggregation relationship with a few other classes, which are the map cells (or map grids) and elements on the map, such as the potion, the gold, the player character, and the enemies. The class contains a single shared pointer to the player character, and vectors holding shared pointers of instances of potion, gold, and enemies elements. The shared pointer is a smart pointer that retains shared ownership of an object through a pointer. As a smart pointer, it automatically deallocates after the last shared pointer goes out of scope, thus preventing memory leaks. Vector was used instead of the array because it allows dynamic memory allocation at run time. When we are reading the customized map, the number of elements like potion or gold are not fixed, thus its feature of automatic memory management can be helpful. The map class also holds a 2D array of “cell” objects which represent the game map grid. The reason we used an array for this is because the map’s size is fixed, so there is no need to dynamically allocate memory.

In addition, the class provides methods for the movements, interactions, printing, controlling game logic, and ending the game. The class provides methods to move the player and enemies on the map. It supports player attacks, enemy attacks, using potions, and dropping gold upon defeating an enemy. The “print_map()” function is used to display the current state of the map onto the console. In addition to the map, the class also outputs messages in text form to the console to inform users of the current state of player characters. Furthermore, the class contains various methods to handle game logic, such as checking if two positions are adjacent, and finding enemies and items around a given position. The class also includes a method to check if the game is over and sets flags accordingly.

Cell

This class represents the fundamental building block that makes up the game map, which stores the information about each cell in the map, as well as if the cell can be stepped on. The cell as the basic building block of the game enhances the flexibility of the program. It makes it easy to read in and save item or character positions on the map. The boolean “can_step” field is also an important determinant for the user's movement. For instance, the player cannot step onto the wall, nor they cannot step onto the same cell as a potion. However, they can step onto the same cell as gold to pick them up, or they can walk over a dragon hoard when the dragon is still alive. In addition, the cell class enhances the ease of maintenance and accommodates changes in the

map. If we were to add additional features that allow multiple elements to occupy the same cell, or if we want to change the map size, the class can easily accommodate those changes.

Item

The class item is an abstract class with subclasses potion and gold. By using inheritance for this implementation, we get to reuse as much code as possible. The class encapsulates the properties of an item, such as its name, symbol, pickability, and position. This encapsulation ensures that the item's internal data is accessed and modified through appropriate member functions. By setting the pickable field, the class determines if an item can be picked up by the player's character in the game. This flexibility accommodates different gameplay mechanics and makes changes to the game easy. If we want to later allow users to hold a potion or a weapon, we can do that by setting the pickable field. The class also enables polymorphism, where objects of derived item classes can be treated as “item” objects, providing a unified interface to interact with items of various types. In addition, this way of implementation also makes adding additional items in the future easy – by simply creating classes that inherit from the “item” class. Each derived class can implement its specific behaviour while inheriting and reusing the common attributes and methods defined in the “item” class.

Character

This class is responsible for the definition and movements of player characters and non-player characters, which are enemies in this case. The implementation is very similar to the item class, which uses inheritance and polymorphism to increase the reusability of code and allow objects of different classes to be treated as objects of a common superclass. The character class has two subclasses – “player_character” and “enemy_character”. Each subclass then has more subclasses. For instance, the “enemy_character” class has subclasses like elf, orcs, and dwarf, that is where each enemy has their unique ability and properties implemented. The inheritance structure again enhances the reusability of the code and allows easy modifications to individual methods.

Resilience to Change

In order to ensure flexibility and adaptability in our program, we employed key object-oriented techniques such as polymorphism and inheritance. These methodologies allowed us to easily accommodate potential changes and additions to the system.

For instance, we implemented the "item" and "character" classes with careful consideration of inheritance, which facilitated seamless integration of additional features. This design approach ensured that modifying one class would not introduce unintended consequences in other classes. Using the "character" class as an example, adjustment of their abilities, and alteration of their defined fields, and even including the additional characters can be done easily.

Moreover, in the "map" class, we utilized vectors to store information about items and characters. This dynamic data structure enables us to accommodate varying quantities of these elements, thus accommodating changes in game specifications with ease. The "cell" class, serving as the foundational building block of the game, was meticulously designed for reusability. Even when adjusting the size of the game map or modifying available game elements on the map, the "cell" class remains versatile and adaptable.

Because we follow the goal of “high cohesion low coupling” throughout the entire program, it makes reusing code and making changes to the program easy. Changing the implementation of one class rarely affects the implementation of the others. In order to promote “high cohesion low coupling” we followed the single responsibility principle, that is, a class should have only one reason to change or a single responsibility.

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?

For generating a player character, the race is only chosen once, which is at the beginning of the game. At that time, the information about the selected race is saved. For each floor, the player’s position needs to be regenerated, using the mechanism of randomly choosing a chamber from the five, randomly choosing a cell from the chamber, and placing the character with the player object saved. In order to generate the race of the character, we will be implementing the character class using inheritance. The inheritance allows common functions and fields of player characters to be defined in the superclass. We can choose to override or add necessary fields and methods in the subclass without changing the structure. This would be beneficial to the program as we get to reuse as much of the code as possible, and keep a similar and organized structure for all of the characters. Also when changing the implementation of one character it would not affect other characters, thus making modifications easy. In addition, it makes it easy to add additional races to the game as it only requires creating extra subclasses.

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

The method for generating enemies is similar to player characters. For each level, 20 enemies are generated at the beginning. Their positions are first generated, then their races are generated. The mechanism for generating each enemy’s position is the same as generating the position of player characters. After the position is generated, their race would be generated using the given probability. By generating the position first, we can reuse the codes from the generation of the player character’s position. The information about generated enemies is saved in the form of a vector. This is different from player characters because player characters are only generated once

at the beginning of the game and used throughout the entire game, while the enemies are generated multiple times.

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 implementation of the ability of enemies is different from player characters. The ability of the player characters are relatively similar for each player — they are mainly related to their fields such as health points. On the other hand, non-player characters have various abilities. For instance, Dwarf causes Vampires to lose 5 health points for each attack, Elf gets two attacks for all player characters except for Drow, Orcs do 50% more damage to Goblin… From the above examples, we see that some enemies have abilities related to the player character's fields, while others relate to the attack mechanisms. The variety in abilities causes us to choose to implement their abilities in a different way. Instead of using a template pattern, we would just use inheritance. By using inheritance, we get to reuse codes as much as possible, such as the fields of health point, attack, and defence. Instead of overwriting methods, we would just have individual methods for the enemies.

4. 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.

The decorator pattern allows individuals to dynamically add new behaviours to objects without affecting the behaviours of other objects from the same class. The advantages of the decorator pattern include its flexible alternative to subclassing for extending functionality, and its ability to modify behaviours at runtime. The disadvantage of it includes that it can result in many small objects in the design which can be complicated and sometimes harder to maintain.

The strategy pattern allows individuals to define a family of algorithms, which enables using their objects interchangeably at runtime. The advantage of strategy patterns includes eliminating the use of conditional statements, and algorithms can be easily changed at runtime. The disadvantage of it includes that the application must know the existence of different strategies, and it increases the number of objects.

If we have to choose from the two patterns, considering the characteristics of the decorator pattern and strategy pattern, we choose to use the decorator pattern to model the effect of the potion. This is because in this case we only need one algorithm, thus there is no need of using a strategy pattern. The potion is used to decorate players. When the potion is used, we would use the decorator to create a new instance of the character with their fields changed according to the potion. When the character arrives at the next level, the initialize function removes the effects of the potion that changed attack and defence and keeps the change of health points.

However, we ended up just implementing the potion using inheritance. When a potion is consumed, we directly call the modifier function of the player character in the map module to change its properties. By doing this we do not have to write individual subclasses for each type of potion, making the implementation process more time efficient.

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?

The generation of treasure and potions will be generated in a module called map, whose purpose is to generate all items and characters on the map at the beginning of each floor. There would be a generator function in the map class, which is a generic function that takes parameters of what is being generated, and how many are being generated. Then it will go through the process of choosing from the chambers with equal probability, choosing an empty cell, and calling individual functions to put the items on the map. Also, because treasure and potions have similar fields, we plan to write an abstract class called items, where both of the classes for gold and potion use it. Thus, the implementation can also be optimized by reusing code as much as possible.

Extra Credit Features

1. Trophies

We included trophies at the end of the game to reward special accomplishments players achieve in each round of the game. The trophies are only rewarded if the player successfully completes the game. We have three trophies:

- HP defender: the player finished the game without losing HP
- HP Protector: the player finished the game with more than half HP
- HP WasterL the player finished the game with less than half HP
- Poor kid: the player finished the game with 0 gold
- Demon slayer: the player killed 50 enemies
- Coward: the player killed 0 enemies

The first four trophies are simply determined by accessing the fields of the player character when the game is finished. The last two trophies are determined by keeping track of the number of enemies killed during the game. The DLC particularly enhances the playability of the game, so in addition to completing the game, the player has more goals to work towards.

2. Radar

The radar detects all elements on the map that are within 1 block radius of the player character. The information about the direction and type of element is then displayed in the console as text. As a part of the requirement, after the player has consumed a certain type of potion, the potion

becomes visible to the player for the rest of the game. This requirement is completed through the radar. When the player character is around one block radius of the potion, if the type of potion has been consumed, their name would be outputted to the console. Otherwise the potion would be outputted as an unknown potion. The feature is done by getting cell information of all cells around the player character and output accordingly. This function makes it easier for players to know what is around them, and what action to take next.

3. Sound and Visual Effects

We have added different sound effects to some of the game moves. Some examples are the welcome page, when a player gets attacked by enemies, and when an invalid command is entered. This DLC was included because sometimes it is hard to determine what happened in the game, and reading texts in the console can be time consuming. By adding sound effects to the game, the player can quickly learn what has happened. This also makes the game more interesting as the game is not only visually appealing, but also auditorily. The feature is made possible by using the <cstdlib> and <thread>. Using multi-thread allows us to play music while the programming is running. Coloured visual effects in the console were also added to increase the readability of the program.

4. Teleport

The teleport function allows player characters to spend 15 golds to teleport themselves to another position on the same floor of the map. This function can be used by command line argument “t [x] [y]”, which [x] [y] is the coordinate of the point the player wants to move to. The command is done by moving the player character’s pointer to a different cell.

5. User’s Manual

The game included a user’s manual by inputting “m” on the command line. The feature is useful to inform the player of possible actions they can take during the game, so people who have not read the game outline can also enjoy the game.

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?

One important lesson from the project is to plan out the structure of the program first, then implement it. One way we planned the structure is by drawing the UML. Other than the UML, we also have a shared document that contains the summary of every requirement we need to implement from the pdf. Then we divided up the work using the UML and the summary. This way of planning makes the overall structure of the program organized and allows us to make minimal changes during our implementation. Also, we learned that it is important to have a plan, but should also be aware of the possible changes to a plan and leave time for those potential

changes. In the beginning, we planned to have only one day to debug our program, but it took way longer than we thought. Debugging the program involves running the program hundreds of times, which can be time-consuming. After catching a bug, we also have to take time to locate and fix the bug in our code. Thus, the debugging process ended up taking about 5 days for us to complete. Luckily our group finished the programming of the required features a week before the due date, so we were left with enough time to debug the program. However, we had to cut some of our DLC in order to finish everything on time. Another important lesson is the communication between members. We found it quite hard to just assign work and let everyone work on themselves. We found that when everyone gets together either in person or through video meetings, the process of coding can be way more efficient because we get to discuss our ideas and solve challenges collectively.

2. What would you have done differently if you had the chance to start over?
If we get to start over, we would definitely leave more time for debugging by finishing the required features a few days earlier, in order to leave more time for the implementation of DLC. In addition, we would plan the overall structure better before we start to implement the code. For the first three days, we had to change the design of the structure quite a few times. Thus, we had to change the implementation as well. So next time before we start the implementation, we would like to spend more time discussing the feasibility of the structure, so we would not have to make as many revisions as we made this time. Moreover, we would pay more attention to details in the requirements. Even though we wrote a summary of requirements before we started, there were still points missed during the implementation and spotted during the debug period. Some revisions were hard to make, which caused us to have a longer-than-expected debug period. So next time we would try to pay more attention to those details during our implementation.

Conclusion

The ChamberCrawler3000 (CC3k) game resulted in a well-structured and thoughtfully tested game. By employing various techniques such as shared pointers, polymorphism, inheritance, and the Model-View-Controller (MVC) pattern, we ensured a modular and maintainable program. The game's flexibility in generating items and characters using inheritance allowed easy changes to the program. Throughout the project, valuable lessons were learned about effective teamwork, communication, and structuring the design process. Also, our knowledge gained on object-oriented programming throughout the semester was effectively reflected in the project. These knowledge and experiences can be especially helpful in future projects, and even career paths.

CC3K UML

Shadowmaze
-j238wang-k4zhu-t2hua-