SHADOWMAZE

Project CC3K

July 19, 2023

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

Project Summary

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.

Project Breakdown

NUMBER	TASK TITLE	TASK OWNER	START DATE	DUE DATE	DURATION	STATUS
1	Summary of Problem	Justin, Kevin, Tracy	2023-07-13	2023-07-13	1	Done
2	UML	Justin	2023-07-14	2023-07-15	2	Done
3	Plan of Attack	Tracy, Kevin	2023-07-14	2023-07-16	3	Done
4	Interface	Justin	2023-07-15	2023-07-16	2	Done
5	Implementation	Justin, Kevin, Tracy	2023-07-17	2023-07-22	6	In Progress
6	Debug	Kevin	2023-07-23	2023-07-23	1	Not Started
7	DLC	Justin, Kevin, Tracy	2023-07-24	2023-07-27	4	Not Started
8	Documentation	Tracy, Kevin	2023-07-28	2023-07-29	2	Not Started
9	Check / Test	Justin, Kevin, Tracy	2023-07-29	2023-07-31	3	Not Started

Timeline

	WEEK 1				WEEK 2							WEEK 3							WEEK 4	
	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T
Date	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	1
Summary of Problem	🟦																			
UML		🟦	🟦																	
Plan of Attack		🟦	🟦	🟦																
Interface			🟦	🟦																
Implementation					🟦	🟦	🟦	🟦	🟦	🟦										
Debug											🟦									
DLC												🟦	🟦	🟦	🟦					
Documentation																🟦	🟦			
Check / Test																	🟦	🟦	🟦	

Q&A

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 a template method. The template method allows the skeleton of the algorithm to be defined in the superclass. We can choose to override necessary fields and methods in the subclass without changing the structure. This matches the definition of players as they have similar fields including but not limited to their health points, attack, defence, and some additional abilities. 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 abstract class, 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.

Considering the characteristics of the decorator pattern and strategy pattern, to model the effect of the potion, we choose to use the decorator pattern. 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 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 potion that changed attack and defence, and keeps the change of health points.

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 class 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.