CS246 - Final Project : Chamber Crawler 3000

Arul Ajmani - 20641980
Aditya Maru - 20618112

Final Design Document

Introduction

This project implements a simplified rogue-like video game called ChamberCrawler3000 (CC3K). The game follows all the specifications mentioned in the instructions document and is played on 30x79 board with predefined chamber shapes and passageways. There are five floors linked by staircases, which if cleared successfully by the player, results in him/her winning the game.
Through the course of this document we will go on to explain how we exploited various object oriented techniques and design patterns to ensure our program accommodates change easily and exhibits good coding practices.

Structure

Spending adequate time on laying out a detailed structure of our program before coding it out, played a key role in helping us preempt and avoid several roadblocks, which could have made the interaction between the several components of this game complicated. We decided to divide all the major components of the game (i.e the ones which played a role in affecting the game-state) into their own classes. With the help of **abstract classes** we organized the various **concrete classes,** into groups as shown in our UML. At a high level our program follows the following structure:

1. The user interacts with a **Game class** which in turn **"has-a" Player** and **Floor**.
2. The **Floor** in turn **"has-a" GameElement** and **Chamber**.
3. The **GameElement class** splits up into a **Character** and **Item** class.
4. As suggested by the class names, **Character** is divided into **Player** and **Enemy** (containing the concrete player and enemy classes respectively), while **Item** consists of **Potions** and **Gold** (containing the concrete potion and gold classes respectively)**.**

Initially, we considered the idea of dividing each **floor item** into its own class, but as we gained a deeper understanding of how the game-state changes, we realized that these components were of a static nature and did not play an active role in affecting the way our game worked. Instead we maintained a 2D character array to store our main GameBoard and a **DefaultGrid** with only floor items (i.e. an empty board) which we can use to reset any part of the board to its original state. Our GameElement class taps on the commonality between Items and Characters, which are the coordinates, and thus

enables us to perform all the functions such as **move, attack** and **pick** using just coordinates of the particular object.

Design Patterns:

Game Display:

Our implementation of a **Model-View-Controller** design pattern helped us in keeping up the good coding practice of not merging **"cout related functionality"** and other functions in the same class. We thus created a **View class** which primarily serves two purposes.

a) It maintains a 2D array which is **constantly updated** whenever there occurs a change to the game-state, which is then displayed to the screen along with the character statistics.
b) It maintains and displays an **ostringstream** which is populated throughout a particular iteration of the game cycle, with the different messages to be printed after the board.

Player, Enemy, Gold and Potion generation:

The design pattern which we used to ensure that each race and item can be easily generated at runtime was the **Factory Pattern**. We were able to encapsulate all the **"generation"** functions in a single factory which gave us a way to create different types of objects, all derived from common base abstract classes. For example, the creation of Humans, Orcs, Dwarves and Elves all of which stem from the superclass Player.

To meet the **probability specifications** when it came to the spawning of Gold, Potions and Enemies, we maintained arrays of the names of each item (occurring n times depending on given probabilities), then randomly generated an index and utilized our factory (by passing the type as a string) to create an object of that type. We believe the array model helped us avoid duplicating a lot of code.

The second advantage of a Factory Pattern was that it offered the client with an interface which was **independent** of the implementation of the actual pattern. Thus, at the end we were able to add two additional Player types while keeping the interface unchanged.

Dragon and DragonHoard interaction:

The interaction between a Dragon and its hoard was interpreted by us as a textbook example of an **Observer Pattern.** The dragon is the subscriber while the hoard is the publisher. Whenever we spawn a dragon hoard, we consequently spawn a dragon and **attach** it to the hoard. This attachment allows us to **notify** the dragon to be hostile/not hostile, depending on the current location of the player. When the dragon is killed we **detach** it from the hoard thus allowing the player to pick up the hoard.

Use potion and pickup Gold:

The fourth pattern we implemented was the **Visitor Pattern**, which gave us the power to easily adapt each Player's reaction to a particular potion/gold depending on its unique characteristics. For example, a Dwarf doubles the gold it receives, while an elf reverses the effect a potion has on it. We implemented two **virtual functions** called pickItem and getPickedBy (can be found in Player and Item class respectively) which allow either **default or overridden** interaction between a specific Player type and an Item type.

Points to highlight:

Following are some concepts which we took the extra initiative to learn and incorporate in parts of our program:

Shared pointers: Through the course of this project we only used shared pointers. While the learning curve on how to use them was steep, it eventually paid off as it enabled us to avoid any sort off manual memory management.

Casting: We saw the need to use casting in two particular places in our code. We used a **static_pointer_cast** when casting a shared pointer of one type to that off another, and used a **static_cast** to cast integers to doubles, to accommodate for the orc getting only 0.5 times the gold he it picks up.

Recursive chamber generation: A particular algorithm which we would like to highlight, is the one used in generating each chamber. Every **Chamber class** holds a vector of its coordinates which is populated recursively. We provide the recursive function with a seed, which then expands in all directions and emplaces the surrounding coordinates up until it hits a wall. In this way we span the entire area of the chamber and register all of its valid coordinates.

Merchant hostility: Initially we thought of this as an Observer Pattern but while implementing it realized that that approach would be overkill, because the merchants only have to be **notified once**, after which their hostility permanently changes throughout the game. Thus, we decided to use a **static variable** “hostile” which is flipped if any one Merchant is attacked, thus activating the attacking capabilities of all other Merchants.

Bonus enhancements:

After implementing the required game functionalities we decided to go the extra mile by implementing the following three enhancements:

Shared pointers: As mentioned before we have only used shared pointers throughout this program.

NCURSES WASD Controls: As an added DLC we decided to allow the player to make an in-game decision of whether to move with the normal controls or WASD controls. The initial infrastructure of our code made it easy to incorporate this enhancement, however it did take us a while to learn how to interpret keystrokes on the keyboard. The player can now attack and use items simply by pressing the WASD key in that particular direction, significantly improving gameplay.

Additional Special Characters: Keeping with the current craze for Pokemon Go, we decided to add a **Charizard** special character. This character is a DLC which performs a highly damaging special attack every four normal attacks. Following the special attack however, it loses the ability to attack for one move as it must recharge its powers.

The second additional player is a **Slayer** which if chosen by the player, gives the player the option to choose one enemy he/she would like to “slay” (kill in one hit) throughout the levels of the game. Once again the highly **accommodative** nature of our initial code made these additions very easy.

Questions and Answers

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?

This has been answered in the design pattern section where we have vividly described how our program goes about generating players, enemies and items.

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

The generating of all characters and items is done through one game factory. The same factory defines different functions to return items, enemies, or a player character (or more broadly a “game element”) depending on the string it receives.

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

Approach is the same as mentioned in the first Plan of Attack document.

What design pattern could you use to model the effects of temporary potions so that you do not need to explicitly track which potions the player character has consumer on a particular floor?

While drafting out the plan of attack for this project we spent a lot of time trying to model a design pattern, particularly a **Decorator Pattern** to solve this problem, but we felt that it was over complicating the concept unnecessarily. In our opinion, the simplest way to handle the issue of temporary and permanent potions without actually keeping track of which one’s are consumed on a particular level is by maintaining two different types of fields, a levelAtk and an actual Atk and similarly a levelDef and an actual Def. Whenever a temporary potion is encountered, the value will be added or subtracted to the level variable and the actual variable. At the end of every floor however we adjust the actual variable by nullifying the change which was brought about to it by the temporary potions. Each potion class will have 3 fields (HPChange, ATKChange, DEFChange) which will reflect the above changes in the Player.

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 the generation of treasure does not duplicate code?

While modelling our solution we realized that since the number of potions and gold to be generated at the start of every floor is the same, we could reuse a lot of the code which would be randomly generating numbers between 0 and 9, to help us choose what item to create and put on the board. As mentioned before we maintain an array of item names (occurring n times depending on the specified probability) and then send the string to the factory, ensuring we get back a shared pointer of the type we require. The duplication of code has thus been significantly reduced because we can achieve two goals through a single round of random generation.

Conclusion

What lessons did this project teach you about developing software in teams?

This project was our first real experience of developing software in a team, and was fortunately a very pleasant experience. Some of the key skills we developed by working together was maintaining style consistency, discussing every concept and ensuring the other teammate was on the same page at all times. Further we established a shift system in which one person would code while the other would put in significant effort towards thinking of the algorithms, catching small errors and typos and constantly observing to avoid dealing with excessive bugs at compilation time. A good chemistry was necessary for this project to succeed and the determination to make full use of this learning experience helped us work together towards this final product.

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

We did not extensively document our code at the time we writing it as it kept hindering our thought process while we wrote the code. This however led to us investing significant time afterwards, to get back into the code and document it well. This is a double sided sword, as occasionally it takes a while to catch the train of thought we were following at that time and thus if we could start over, we would probably do both side by side.