CHAMBER CRAWLER 3000 : PLAN OF ATTACK

Arul Ajmani - 20641980
Aditya Maru - 20618112

OUTLINE

While making the initial UML, we decided to make the corresponding .h files of all the classes, alongside. This allowed us to step through the ideas we had, to implement certain characteristics and attain a high-level understanding of how our code could potentially be implemented. What we found necessary was to establish the control flow of how all the the different classes would interact with one another. Having figured out most of the structure of our implementation, we have a template to work with which can be modified and reevaluated as the need arises.

We were extremely vigilant when it came to incorporating Design Patterns to aid the interaction between classes, and tried to model our solutions using the 5-6 patterns taught in class. At the same time, whenever we thought that a pattern would unnecessarily complicate the code, we searched for a simpler solution. We decided to use BitBucket to create our private repository, so we could work on our separate branches and organize our code better. Exploiting the functionality of working on multiple ideas simultaneously, while preserving the sanctity of our original project, offered great flexibility while we structured our code. We are currently looking at building the project from base up, starting from the character class. Being a two person group, one person will look to implement the enemy side classes, and the other will implement the player side classes. We will try to implement harder/trickier methods and design patterns together instead of working on them alone. We hope to save time by catching mistakes as and when they come. Finally, we plan to compile each and every class separately after writing it, so that we find and debug all errors as we go along.

TIMELINE

A general timeline which we are looking to follow is to utilise this weekend to get a very good stronghold on the basic functionalities such move, attack, spawn etc. Once we have that figured out, we believe it should take us up to next Friday to code and compile all the classes successfully. Following that, we plan to use the next weekend for enhancements, efficiency tweaks and documenting of our code. This takes us to the submission date, on which we are confident we will have a fully functioning crawler game, to submit for evaluation.

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?

The design pattern which would ensure that each race can be easily generated at runtime is the Factory Pattern. This pattern offers a very handy solution to create different types of objects, all derived from the same subclass. In our case this would be the creation of Humans, Orcs, Dwarves and Elves all of which stem from the superclass Player. Each of the player options which the client has, overrides the method written in the Factory class to return a Player pointer to a new instance of the chosen object. In the case of our game the Factory will make this decision on the basis of a string read from user input.

The second advantage of a Factory Pattern is that it offers the client with an interface which is independent of the implementation of the actual pattern. Thus, if any change has to be made to a current subclass, or an additional class has to be added, the change will only occur in the implementation class while the interface remains unchanged. The code using the factory would pass a new string to the factory, and allow the factory to handle the new types independently.

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”).
It is similar to how players are generated because enemies are made as per different probabilities, which are computed at runtime. This computation at runtime requires us to generate enemies as per the factory pattern, passing in a string to determine particular enemy types.

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

There is no no need for a visitor pattern unless different player characters have different reactions to enemy special abilities (Eg. immunity, reversal etc.). A simple specialAbility(Player &p) method in the enemy class, declared pure virtual would suffice. This would be overridden in every concrete enemy, which will have a particular effect on the player reference given to it. So say a goblin’s specialAbility method is called, it will extract all the gold from the player.

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 to solve this problem, but we felt that this was overkill. 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) while 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 realised 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. Our current structure handles this random number generation in the Floor class which then uses a Factory to actually return pointers to either generated Gold or generated Potions. These are then sent along with valid coordinates to any of the five chambers, chosen at random, to be placed on the board. The duplication of code will thus be significantly reduced because we can achieve two goals through a single round of random generation.