CS246 Design Document

ChamberCrawler3000+

Justin Zhao

Design Overview: CC3k+

1. The Interaction Interface & Display

In this project, I've used a subject-observer design pattern to implement the Display of the game. The main subject is the state or board of Tiles contained within the level.h class. The main observer, in this case, is the display.h class, which reads the board and parses information from the level in order to form a cohesive display for the user.

The level can provide this board of Tiles as the general state, the level can also return information about the player so that it can be displayed.

Within my display class, it is isolated to just formatting of the information contained in Level. Because the Level is structured to be observed in this way, not only can multiple displays be added with 0 change to the code for Level, the display has freedom to display anything it wants with the information it has. A new display could easily parse the Level, and convert the ascii characters to images or sprites. The displays only update when the Level prompts it to update.

Next, the interaction interface, or in my program, it is textCommands.h.

TextCommands contains a continuous text command system. The primary purpose is to parse player input, and determine what the game must do after being given that player input. The way in which is prompts the game is by very simplified functions which can be used on an instantiated GameManager. This maintains encapsulation, and it would be very easy to implement an alternative TextCommands system, (and even have 2 which point to the same GameManager), since parsing the input is the main job, and the logic involved for what to do after each input is a simplified choice.

The GameManager is an instantiated version of the game, which an instance of Level and Display (or multiple displays). It also instantiates the RNG of the game. By instantiating the GameManager, and then Instantiating a TextCommand/command system, which will point to the GameManager, then the game is ready to be played. That is done in the main.cc of my program, which also passes along some command line arguments such as loading from a file.

2. The Tile/Chamber System

As previously mentioned, the Level of the game is what contains information on all the game tiles and the entire state of the game. This is how that is specifically structured:

A Level contains an array of <Tile> (smart ptr in this instance). Along with information on the dimensions, this represents the information of the entire floor of the dungeon.

Each Tile object is an object contains the underlying symbol for the ground tile, as well as a pointer to an Object (object.h). A tile can only have one object on it at once.

When trying to parse information about the tile, for example, getTile() will return the character that is supposed to be represented on the current tile. This will either be the underlying tile character or the symbol for the Object which is “on top” of the tile. Objects can be moved on and off tiles.

This object.h class is the parent class of all Items and Characters. So, any Item or Character can be “standing on” a tile. However, this is one area where I question the design of my program. Because of this, the object.h class has a lot of “unused” virtual methods which indicate to me that it could be possible to separate the Character and Item classes.

In the current implementation, most methods in object.h are pure virtual, and require children to override them. For example, items would override and (nullify) the unused methods, Characters would override methods such as move() and attack().

The Chamber system of my program currently implemented like so:

The Level will contain an array of Chambers (which I manually set up, due to time constraints, which means that “floors” of dungeons must always have the same layout)

The chambers are responsible for tracking actions, moving the player/enemy, attacks(based on location), enemy scanning. The chambers each contain a vector of pointers to tiles, each of which map out a widthxheight rectangle of the tiles in Level. So the chamber contains a small piece of the whole level.

When an enemy spawned, the game keeps track of which chamber they are spawned in/which chamber they are in. When players/enemies attack, they tell the chamber they are in to make the attack, in a visitor-like pattern.

Because a chamber only contains a small amount of the level, performing actions strictly within a chamber is efficient because it only considers the objects within that level.

Although random spawning is not correctly implemented, this is something which could easily be coded into such a chamber system. With the level randomly selecting a chamber, and the chamber randomly selecting a spot within the chamber to spawn the object.

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?

I worked alone. I learned that it is far more important to focus on a good design, rather than beginning to code. Through understanding of design and a good, planned out design from the start, implementing the code becomes far easier. Of course, this is far easier said than done, since problems in design only start to arise once you start to get into the details, aka implementation. This is a sign that experience with building large projects and code is highly valuable if you want to start off with a good design. For

me, there are plenty of design lessons which I've learned but otherwise would never have known had I not had to actually implement every part of my plan.

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

The main flaws with my program are the chamber system, and the objects not knowing their location, and (potentially), the over-generalization of the object.h class. The chambers should not be moving the objects around, instead it should be the level, but additionally, if objects knew their own location, then they could perform actions more directly on the objects around them. This leads to easier implementation of anything that comes from visitor design pattern.

How could your design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?

In the initial plan, answered that I would use inheritance for player objects such that many similar player objects could be generated. In my implementation, I used an ObjectCreator class, which can be prompted to generate enemies as well as players based on their icon '@'. The ObjectCreator will allocate memory and return a pointer to a created player class based on race. Each player race inherits from the player class, such that any race can easily be generated and used. This follows a factory design pattern, and it makes it easy generate additional classes, and adding new player classes should be easy if they are similar to the basic Player class.

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

Different enemies are generated also using the ObjectCreator class, and enemies also inherit from each other, from the common class of *Enemy*. *Player* and *Enemy* are both children of the *Character* class. In this way, it is very easy to generate enemies, since a tile can contain any type of Enemy, and enemies can share common behaviour and override specific behaviour. It is not very different from generating the player character, since the Player character and Enemy are both child classes of Character.

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

In the plan, I specified that I would create a "visitor" design for enemies triggering special abilities. Actually, the universal "attack" action of all characters follow a visitor pattern, where the attacker also calls the beAttacked() of the other character. In this way, triggering special abilities for enemies is simple. The specific enemy class will override the default behaviour when attacking or attacked, so that, for example, when a troll is beAttacked(), it can regenerate it's own health after the damage exchange.

These are all special abilities tied to the attack() action, however more would be needed to implement specific new abilities which aren't attacks (spells and such).

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?

The temporary effects of potions, (as to alleviate the need to specifically track stat debuffs) can be modeled using a decorator design pattern. A potion's status effects would be a decorator to the stats of a player. It would be efficient for each character (player) to have an individual Stats object. Then, a potion's status effect can wrap around a player's stats object (itself being a stats object).

When a player's health need to be accessed, for example, it would be stats->getHealth(), but stats itself may point to a wrapping potion which affects the stats. The potion would implement something like

getHealth() { return stats->getHealth() – 10}

and the player's stats will effectively be temporarily altered. Then, such potion status effects should be easily removed, since the original stats of the player is still retained as an object somewhere in the decorator link.

This design is not currently implemented in my program.

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 hoards and the Barrier Suit?

The generation of items, similar to the generation of Player and Enemy, can have re-used code by using inheritance for item objects. In my program, item objects all have shared methods such as beUsed(), which is a visitor design pattern for players using items. Common logic, such as items being immovable on the floor, and being able to be used, and being unchanging after instantiated, indicates that the inheritance relationship can also lead to an efficient generation/spawning of items. In my program, an ObjectCreator can also generate all items, so that they can be placed on a tile.

As much as these items inherit, there is a lot of uniqueness for specific item use. Potions and treasure are the most similar, since they are essentially the same object up until they are used by a player. Then, the beUsed(player*) method can implement some specific effects onto the player which is using it.

The barrier suit and the dragon hoard protection/linkage system can be a separate object inheriting from the same base class as gold and potions, and then the dragon hoard and barrier suit are specified instances of the linked item type. This will minimize duplicate code.