CC3k+ Final Design Document

By: Frank Tao, William Zhao, Lily Cao

Introduction

Our final cc3k project consists of three main classes: character, item, and floor, along with an additional RNG class used for random generation.

Overview

Character

The character class consists of two child classes, player and enemy, which inherit from the parent class. Character Class is a generalization of characters in the game ie, Player, Merchant, Dragons, etc. Character class includes basic data fields and operations that are shared between Player and Enemy i.e. positions, health, attack damage, defense damage, attacking action function, movement functions. Functions such as Attack and Move are implemented for the intention of reusability since both player and enemy share the ability to attack and move but may differ in mechanics.

Players have different races, which all have different attributes. Other than the attributes, these races all move the same way (controlled by PC), so the best design pattern to implement the player class is with the visitor. Depending on the race, a different visitor is called, and each visitor overrides any functions necessary for its race to be implemented correctly. This way, the same method can be called during the game no matter which race the player is.

The enemy class consists of subclasses for each type of enemy. Each subclass overrides the constructor with its own attributes, so they are spawned with the proper stats. Since almost all enemies (except dragon) move and attack in the same way, none of the move and attack functions need to be overridden in the subclasses.

Item

The Item class consists of four child classes: Potion, Gold, Compass, and BarrierSuit. Since some types of items require an enemy guarding it (compass, barriersuit, some types of gold) and cannot be picked up until the enemy is dead, a pointer to the enemy guarding the item is kept. This variable is protected, which means both a getGuardingEnemy and setGuardingEnemy are required in order to access this pointer. In addition, the item class has a function to indicate what it should do when picked up by the player, called onPickup. Because all items that can be picked up must belong to one of the child classes, it is a pure virtual function.

The Potion class is a child of the item class, and inherits the parent class's attributes and operations. It consists of two additional attributes, the EffectType, which is one of HP, AtkDmg, and Def, and the effectScalar, which is the integer the potion will affect its user by. Because the potion generation has an aspect of randomization (1/6 chance of one of the types of potions), the constructor is overloaded to incorporate this. The onPickup function in the Potion class is concrete, and calls on setters from the player class to add its effects to the player's bonus stats.

The Gold class consists of one additional attribute, value, which indicates the amount of gold in the pile. Similar to the Potion class, there is randomization involved, and so the constructor is overloaded to account for this. The onPickup function works similarly to the Potion class as well, where the value of the gold object is added to the player's attributes.

The Compass class consists of the onPickup function, which allows the user to add a pointer to itself to the player object, and set the hasCompass attribute in the player to true.

The BarrierSuit class works very similar to the Compass class, with just one overridden onPickup function that adds itself to the player object and sets its (the player's) hasBarrierSuit attribute to true.

Floor

The floor class acts as a container for entities in the game, ie. floor class has a vector of enemy pointers to keep track of all enemies on a specific floor similarly with items. Functions in this class serve the purpose of returning information about the floor itself. For example, checkCoord function returns the type of the cell given coordinates X and Y (individual spots on the map are referred to Cells), popItem (returns the item pointer at a specified X and Y), checkEnemy (returns the enemy pointer at a specified X and Y). These functions are used in the handling of character interactions in classes such as player/enemy/floor. As an example, when the player is required to move, the player calls checkCoord to see whether if the move will be valid (player cannot move into walls or coordinates that doesn't exist on the map)

The floor class also implements functions such as generateEntities, moveEnemies, killEnemy. generateEntities is called during the initialization of the game, it uses the RNG class to randomly generate positions on the map for entities to spawn. moveEnemies calls enemyMove in enemy class to randomly generate a valid direction for the enemy to move to which changes their positions according to their movement speed(0 block of Dragon, 1 for others). killEnemy will delete the enemy pointer on a position on the map and will also drop their inventory item if they are holding items such as compass.

GameController

This class is used for initializing the game ie. generating the floors and setting the map for each floor to be the one from user input as well as generating the player pointer and setting the player race according to player input, loading floors from the ifstream initializing gameMap as well as roomTrack which is the same as map except the cells in each block is used to indicate the rooms (ie. The cells in each room have its own unique number identifier), listening to user input from cin and also determines the type of action (ie. move, attack, pickup), ascending floors which moves the player pointer to the next floor, and ending game which removes any left over instance of objects.

RNG
The RNG class is used primarily to create randomization in the game. It is used for enemy movement, attack success, potion, gold, and enemy generation (both type and location), etc.

The class consists of a function generateInt(), which takes in up to two parameters. If only one parameter n is used, the function will return an integer between 0 and n (inclusive). If two parameters m and n are used, the function will return an integer between n and m (inclusive).

Design

We have implemented the visitor design pattern for changing the stat of the player based on the potion effect namely PlayerVisitor as well as its subclasses. We think that this design pattern simplifies our code and promotes high cohesion. Since all player races are able to pick up potions and apply the effects to their stats (ie. increase/decrease health, attack damage, defense damage, gold…) an operation to “apply the effects” is required (setAtk for applying attack potion effect, setDef for applying defense potion effect, setHP and setGold), however, each player race can have different effects for the same potion/gold (ie. dwarf gets 2 times the amount of gold) we decided to group this operation together to the PlayerVisitor class and override any necessary functions for individual races (ie. DwarfVisitor will override setGold function to obtain two times the gold when called). This provides better organization to the overall code since we no longer need to put these operations in the player class and have a switch/if statement in all of the effect operations to determine the race of the player every time. This method also promotes high cohesion since these elements cooperate together to perform one task (applying the effects of potion/gold). It also introduces high maintainability and possibility to expansion since a race with different effect operations can be easily introduced by inheriting another class from the PlayerVisitor and overriding any necessary functions.

Another way to implement the potion effects is to use a decorator design pattern since the potion effects stack on top of each other and potions are picked up at run-time, however instead we decided to use visitor pattern and indicate the potion type by an enum class EffectType and simply adding the effect to the player stats. We think that the visitor pattern is a better fit because it allows us to differentiate the effect types (as mentioned above) by the player race whereas a decorator pattern cannot.

Resilience to Change

Our project is fairly good at supporting possible changes to implementation, and the addition of new features (or the expansion of existing ones).
Our modules are very cohesive, and each class is dedicated to one type of purpose. In our case, character class is used specifically for objects that can move along the floor, and is further split into player, whose movement is controlled by the PC, and enemy, whose movement is randomly generated by the game controller. Adding additional possible PC inputs for the player will not affect the enemy, or any other object in the game.

Our project also attempts to minimize coupling as much as possible. Instead of directly accessing values in classes, any communication between the different modules are done through function calls. These functions use little to no parameters, which reduces code dependency. In addition, no global variables were used. Any attributes that affect a certain class are initialized and kept within that specific class. Since the project was written using object-oriented programming principles, each class can be used without knowing how it is implemented, which is another example of how coupling is minimized.

By maintaining code that maximizes cohesion and minimizes coupling, our program is highly resilient to change. Additional changes can be easily implemented since changing or adding code does not require large changes to already existing classes and modules (due to low coupling). On the other hand, the code is easy to read and organized, which makes maintenance and extension of features more simple (due to high cohesion).

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 classes?

We planned to use the visitor design pattern to implement each race through different getters and setters, with different default attacks and special abilities being initialized in a function of the visitor class. We did stick to this idea, so in the player class constructor, an initialization of the specific visitor class for the player (depending on race) is initialized. Depending on the race, specific functions may be overloaded and overridden for its specific implementation. By using the visitor method, the same class can be used to create multiple races, and this idea can be extended without much difficulty.

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

Like we originally planned, each type of enemy is split into a subclass, to account for their more unique abilities. This is different from how we generated the player character as the abilities are more unique and are harder to implement within a single class.

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

If we wanted to implement special abilities for different enemies, we could use template methods to apply these special abilities (including possible passive abilities) in the parent Enemy class, with the unique abilities themselves applied in the individual enemy classes.

4. 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?

Potion effects can be modeled using the decorator pattern since they are not known until the player picks a specific potion up (changes to features at runtime). The decorator pattern would also allow users to clear the effects after each floor. However, we decided not to use this pattern, and stick with adding the effects directly into the bonus stats of the player. This is because the decorator pattern results in many small objects linked together, which need to be traversed every time the player wants to calculate its current attributes. Instead, we used temporary stats, which get reset after the beginning of new floors. This increases efficiency, as the attributes can be calculated by accessing a single value in the player object, rather than adding the data for each potion separately.

5. 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 treasure does not duplicate code? How could you reuse the code used to protect both dragon hordes and the Barrier Suit?

Templates can be used to generate items, since template code for generalized "items" can be reused to generate multiple different concrete classes, one for each type of item, without copying the code each time. We did not use this idea. Instead, we chose to create a general Item parent class, which consists of a pointer to guardingEnemy (the enemy it is being guarded by), as well as getters and setters to access this pointer. This pointer prevents code duplication, as dragon hoards, barrersuit, and compass all need to be protected by an enemy, and so any item that requires an enemy protecting it can simply set its pointer to said enemy (the pointer defaults to nullptr otherwise). To generate treasure, we overrode the default constructor, and used the RNG class to randomize the treasure being spawned. The location of treasure also uses this RNG class, though the spawning is done in gameController with the generation of the other required objects.

Extra Credit

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?

Developing software in teams relies heavily on good communication. It is extremely important to ensure all members are caught up at all times to ensure no overlap of work is being done and code can be reused by everyone, to prevent unnecessary duplication. When writing large programs in teams, good planning is also crucial. Being able to create a clear architecture of what the expected final results should look like will help immensely in the process, as the plan breaks down the project into smaller, more easily digestible portions. These portions can be split between team members more effectively, while helping everyone keep track of the progress of both their individual parts as well as the project as a whole. It is important to have a timeline for major tasks; after each smaller task is complete and tested, the project still needs to be compiled and tested as a whole, and the larger the project, the more difficult this task is.

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

If we had the chance to start this project over, one of the big things we would have done differently is the planning process. Our original UML had a plan for how we thought the game should be implemented, but not enough details about class and function interactions were discussed, as many functions ended up being added or changed to accommodate major overhauls in implementation methods when we realized some of these interactions would not work with the original plan.

Conclusions

This project was a challenging but rewarding experience for all of us. It allowed us to apply everything we learned from the CS246 course in a way that forced us to truly think about and understand when to apply design patterns when developing software. There were many instances where we considered specific design patterns, or noticed the possibility of using one to accomplish a task, then realized it was unnecessary, or that the added complexity would not improve the final result. It also gave us the chance to practice good object-oriented programming principles and experience the difficulties of building a larger project from scratch.