CC3K Final Design Document

Christina Liu (c379liu)
Dani Castro (rcastroj)
Helen Yang (sq2yang)
July 24th, 2018

Introduction

This report is going to walk through the plan, process and implementation of the game CC3K using object-oriented programming C++. The outline will cover the overview of the structure of this project, our updated UML with a description of change since the first submission, the design techniques we chose and how our design support the possibility of changes to the program specifications.

Overview

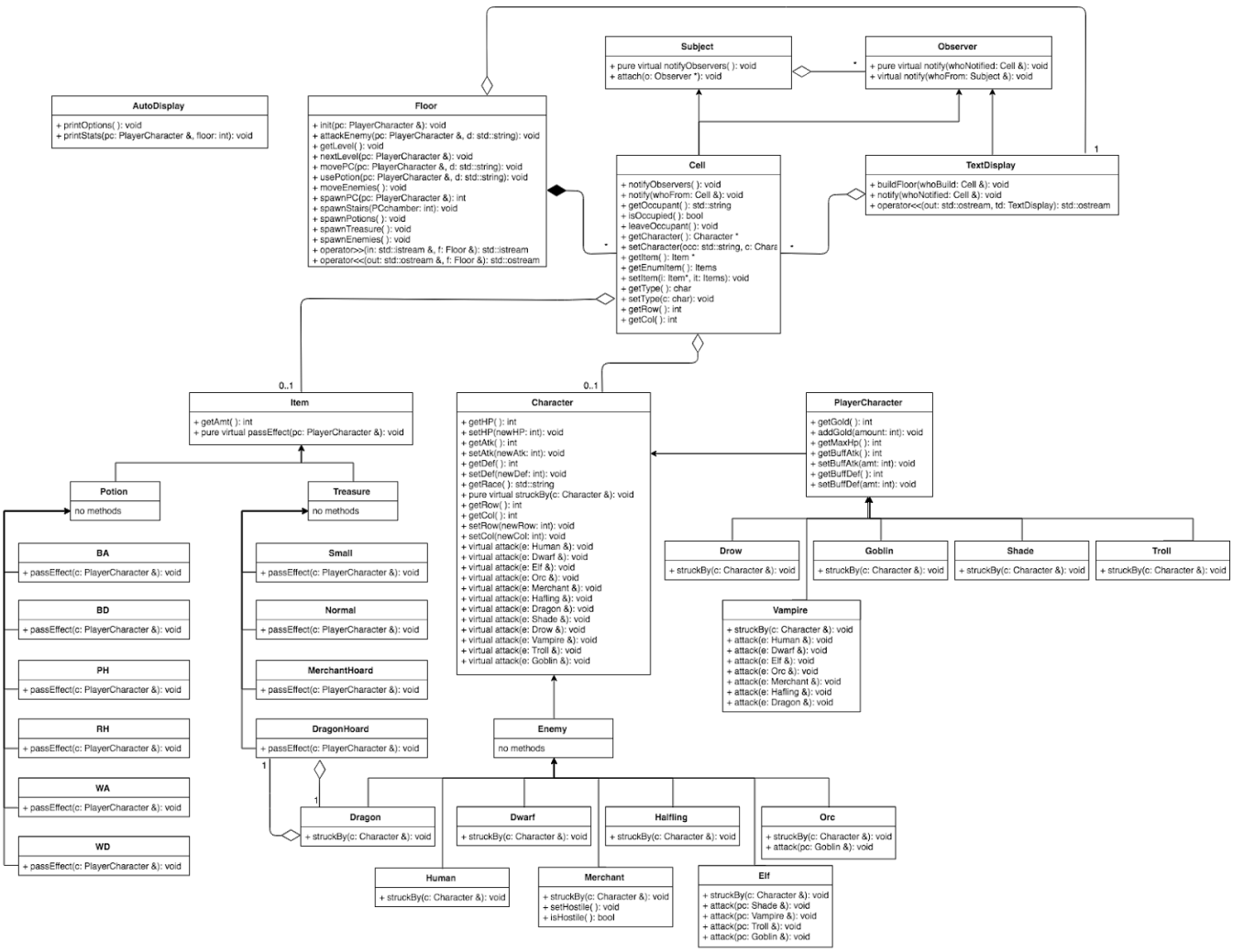
Our implementation of CC3K embraces many features of object-oriented programming. This includes taking advantage of encapsulation, polymorphism, inheritance between classes, customizable constructors, multiple design patterns and smart pointers.

To maintain single responsibility principle, we implemented all classes with separate modules. The core classes that make up our CC3K game are floor.cc and character.cc. The floor module controls the floor and all actions that take place as the game progresses. It displays the dungeon, all the characters and items and constantly makes updates. A Floor class is initialized in the main function, and from there all user commands call upon different methods implemented in floor. The character class stores the core methods of both player character and enemies. Almost all surrounding classes inherit or support the Character and Floor class.

An important part of building CC3K consisted of building the proper design. A core design pattern we used that allows CC3K to run properly is the observer pattern. This allows the floor of the dungeon to be constantly updated as the user input new commands. Another key design pattern we used is the visitor pattern. This allowed us to customize the attack for specific characters that had special abilities with cleaner code and more productive efficiency. Other design patterns that played a key role in the design process of CC3K are the decorator pattern and factory pattern.

By using polymorphism, we are were able to manage different classes with minimal code duplication while maintaining meaningful method. Our additional use of smart pointers allowed us to deal with memory management and assure no memory leaks.

Updated UML



Design

Top Level Implementations

We centered our implementation on the class Floor. The Floor acts as a controller in our program, communicating commands from the user to update our model and then the visualization of the data/view by using our TextDisplay class. The Floor is initialized by reading the game board from floor.txt and is given a Player Character, which is chosen by the race given from user. Different updates depending on what the commands are provided by the user (i.e. use Potion, attack Enemy, move, restart and quit). All these functionalities are controlled by methods implemented in Floor.

The Floor owns a vector of Cells which have Character (Player Character or Enemy) or an Item (Potion or Treasure) or neither depending on where the Character and Items are. Using the **observer pattern,** all cells are Subjects and Observers to each other which allows notification to their neighbours whenever a new command is prompted by the user. Since TextDisplay is also an Observer, it will manage the view when a new action is made from the command. The floor also has a vector of positions tracking the position of the

occupied cells and a vector of Enemy shared pointers. Having updated our Characters the AutoDisplay will then be called to display the new HP, Atk, Def and Gold of our Character.

```
class Floor{
    const int maxRows = 25;
    const int maxCols = 79;
    std::vector<std::pair<int,int>> taken;
    std::vector<std::shared_ptr<Enemy>> enemiesPtrs;
    std::unique_ptr<TextDisplay> td;
    int level = 1;
    bool validType(Cell &c);
    bool validTypeE(Cell &c);
    int genRandCoord(int &r, int &c);
protected:
    std::vector<std::vector<Cell>> theFloor;

public:
    ~Floor();
    void init(PlayerCharacter &pc, std::stringstream &out);
    void attackEnemy(PlayerCharacter &pc, std::string dir,
std::stringstream &out);
    int getLevel();
    void nextLevel(PlayerCharacter &pc, std::stringstream &out);
    void movePC(PlayerCharacter &pc, std::string d, std::stringstream
&out);
    void usePotion(PlayerCharacter &pc, std::string d, std::stringstream
&out);
    void moveEnemies(std::stringstream &out);
    int spawnPC(PlayerCharacter &pc, std::stringstream &out);
    void spawnStairs(int PCchamber, std::stringstream &out);
    void spawnPotions(std::stringstream &out);
    void spawnTreasure(std::stringstream &out);
    void spawnEnemies(std::stringstream &out);
    friend std::istream &operator>>(std::istream &in, Floor &f);
    friend std::ostream &operator<<(std::ostream &out, const Floor &f);
```

```
};
```

Character/Item and stairway Generation

When we initialize our Floor we spawn in the order of : player character, stairway, potion, treasure and enemies. We position our Player Character in a Cell generating a random coordinate within the boundaries of the five chambers given by the txt file. The Cell then points to this occupant and notifies its observers. We then consider the chamber the Player Character was placed in order to spawn our stairway which will get the Player to the next level. The two will not be in the same chamber. For the generation of our 20 Enemies, we take into consideration the probabilities of each of the Enemies, minus the Dragon. Similarly, when generating the 10 Potions and 10 treasures the provided probabilities from the guideline are put into effect. With this implementation it is relatively easy to add on more Potions, Gold piles, and Enemies while considering their probabilities as well as changing their probabilities to implement an even harder game.

Attack/Def

For the attacks between the Player Character and the Enemies we used the **visitor pattern design** since there are many unique interactions between the different types of Characters. This way the compiler chooses the correct implementation of our "attack" function between the races by overloading to have separate implementations of strike for every race. This also allows us to easily modify the attack for a specific character if we wish to include new features in the future. Adding new characters will also not affect the implementation of current characters using the visitor pattern.

```
virtual void attack(Human &e);
virtual void attack(Dwarf &e);
virtual void attack(Elf &e);
virtual void attack(Orc &e);
virtual void attack(Merchant &e);
virtual void attack(Dragon &e);
virtual void attack(Halfling &e);
virtual void attack(Shade &pc);
virtual void attack(Drow &pc);
virtual void attack(Vampire &pc);
virtual void attack(Troll &pc);
virtual void attack(Goblin &pc);
```

Potion use and adding Gold

For our potion use we were not able to follow our original plan of using the Decorator pattern since some potions are permanent while other are temporal for each level the PC is on. After an analysis, we concluded that implementing the Decorator pattern will create more complicated relationships and result in more work

than simply adding a buffer. As design patterns are meant to simplify work, we decided to pass and not use decorator pattern for this situation. We implemented BA, BD, WA and WD as subclasses to Item where "passEffect" allows PlayerCharacter to use the potion, and the buffer will track the difference in enhanced attack/defense compared to the PlayerCharacter's default values. For Gold, we implemented each type of gold as subclass of Item (just as Potions) which will also "passEffect" (add Gold) to the PC through the Cell. This will enforce our single responsibility principle and allow us to easily add new Potions to make the game better.

Auto Display

For our AutoDisplay, it will be regenerated for every new action/command prompted by the user displaying the new HP, Atk, Def and Gold of the PlayerCharacter. However, for the Action since it depends on the notifications from the Cell (when an Enemy is adjacent to the PC), when a Potion is being used or a Gold picked up by the Character, when the PC attacks the Enemy, or just when it moves. We created a string stream that will hold the output and is being passed through the methods and notifications and then is finally printed out for the user view.

Resilience to Change

Characters

We have set up a characters such that the Player Character and Enemies are subclasses of abstract class Character. This allows us to aggregate common fields (HP, Atk, Def, Row, Col), reduce data redundancy and increase reusability. Furthermore, this setup also allows us to well organize the different types of player characters available and various enemies. To add any new character we simply create a new subclass, call the character constructor, and add any customized functions as needed. Our robust genRandCoord function can easily spawn new coordinates and accommodate a desired probability when the character is spawned. Additionally, our implementation of the visitor pattern makes it especially easy to add extra combat effects that a race requires (simply by overriding its struckby and attach functions).

Items (Potions and Treasures)

We have organized our numerous types and potions and treasures using an abstract class Item and further abstract classification Potion and Treasure. If we wish to add an additional potion and/or treasure, simply use the abstract constructor to set the amount (effect) of the new item. If the new item is to have a permanent effect on the player character throughout the game, simply define its passEffect function to directly modify the player's stats. On the other hand, if the new item only wishes to impact the player for a given floor or for a specified period of time, make use of our buffer field (see PH - Poison Health for an example) :). The buffer keeps track of the net change during the floor, allowing for easy reinitialization when the player moves on to the next level.

Strike

By using the visitor pattern for our player character/enemy attack we can easily accommodate a new race's customized combat abilities. Method of combat may be directly inherited from the Character superclass or

simply overridden to any specification so that is includes any special interaction with any other race. Furthermore, we have swiftly integrated the effects of our potions directly to the characters HP, creating ease when calculating the amount of damage done and received during combat.

Floor
Within the floor class we have a highly reusable function genRandCoord that generates a new chamber coordinate when called. The conditional statements within this function intelligently handles a chamber's borders and checks to see it is creating a new coordinate each time. This allows us to easily modify the structure of the floor and accommodate chambers of all different shapes and sizes simply by modifying a few conditional statement ranges.

Observers
Our integration of the observer pattern allows our game to be very adaptable to new features. One such example could be the addition of a graphics display for our game. Similar to the addition we learned from Assignment 4 - Reversi; we would create a GraphicsDisplay class that uses functions from XWindows and attach it as an observer to the floor.

Answers to Questions

Q: The Decorator and Strategy patterns are possible candidates to model the effects of potions. In your opinion, which pattern would work better?

Answer:
We believe the Decorator pattern would work better to apply the effects of potions to avoid explicitly tracking which potions the player character has consumed. We can flexibly apply multiple decorators (ie. PC can use multiple potions) up to specified limits; and also get back to our base component once the PC moves to a different floor (ie. attack/defense potions expire)

Decorator

Pros	Cons
- Able to add (decorate) object functionality at runtime - Can add functionality one at a time (rather than trying to foresee all possible features)	- Can complicate process of instantiating the component (and wrap it in a number of decorators) - Difficult to look back into multiple layers

Strategy

Pros	Cons
- New algorithms can be easily introduced	- Application must be aware of all the

- Client can choose required algorithm	strategies to select the right one

Update:
We did not implement the Decorator pattern, instead we added a member in PlayerCharacter that acted as a buffer to keep track of how much the player's Attack or Defense increased or decreased so each time new level is reached, the Attack and Defence could be reset using that. The challenge with implementing the Decorator pattern was the fact that the four classes (BA, BD, WA, WD) would both inherit from Items and Potions as well as inherit from the Decorator. The method we used to overload all the potions would be inconsistent for these for classes as it would be for the remainder of the Items subclasses. In the end, as implementing the buffer required minimal work and not a much additional effort to resetting the attacks and defense, we decided to discard using Decorator pattern and it complicated the design a lot more.

Q: 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?

Answer:
We will use an abstract base class item, subclasses treasure and potion, and further subclasses for specific types. This allows us to define common functions in the base classes and simply inherit in subclass. In this structure, subclasses need only to additionally define functions that as specific to that subclass.

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

Answer:
Player characters and enemies share many similar characteristics. This includes having health points, attack levels and defense levels, the ability to attack an opponent and moving around the dungeon. Using object-oriented programming, duplication in code can be reduced by allowing all characters to inherit from an abstract "Character" class. However, player characters and enemies also have many differences. Player characters have abilities such as using potions, picking up gold and moving through doorways and passages that enemies don't have. Thus subclasses of base class "Character" can be branched into "Player Character" abstract class and "Enemy" abstract class. These classes will contain methods unique to either player characters or enemies. Finally, all 5 player character types will be defined as its own class and inherit from Player Character class and all 6 types of enemies will be defined as its own class and inherit from Enemy class. The necessity for each character to have its own class it due to the special ability that comes with some of the characters. Object-oriented programming allows player characters and enemies to share methods in which they are similar but also implement methods unique to the two different types.

Update:

Enemies and PlayerCharacter are spawned similarly. Enemy types are randomly generated with probability as indicated in the cc3k guideline documentation. Then randomly spawned into a chamber. PlayerCharacter type is chosen by the user, and then randomly spawned into a chamber as well.

Q: How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

Answer:
Using Factory method design pattern will allow all enemy subclasses with special abilities to decide which classes to instantiate. However, the newly created object is still referred through a common interface. An example is the elf, who gets two attacks against every race except drow. Its attack method can be customized while all enemies with no special abilities can still benefit from the base class with specifies the standard and generic behaviour of the attack method. Player character would not be required to be implemented using this design because only one player character will exist at any point in the game.

Update:
Factory method design pattern was not implemented, however we used inspiration from Factory method to randomly spawn Enemies in our Floor class. We did end up implementing Visitor pattern to accomodate for all the unique types of attacks that some of the player characters and enemies have. Visitor patterns allows the customization of attacked, and when a character gets attacked, the passive character called “StruckBy” which will pass it the attack function with the specific type of character as the parameter, thus allowing the compiler to know right away the Type of the defender.

Q: 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?

Answer:
For our items generation we create a superclass Item for both Treasure and the Potions, since both "enhance" the PC when the PC takes the item. From this superclass we created abstract parent classes for Potion and Treasure since Potion is different in that it enhances mainly the HP, Atk and Def of the PC while the treasure will only add to the Gold of the PC. Creating the abstract class Item will help us in our code duplication in that we will use the method of “take” of the parent class whenever the PC takes the item, still since Treasure and Potion have a different effect we will also have customized method for each as well.

Extra Credit Features

We completed our entire CC3K project by handling all memory via vectors and smart pointers; thus preventing all memory leaks without explicitly managing memory. Specifically, we used unique pointers for TextDisplay and AutoDisplay which require only a sole pointer. For our player character, enemies, potions, and treasures we used shared pointers as we wanted to allow multiple pointers to access the same resource.

Final Questions

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

Answer:

This project genuinely taught us so much about developing software in teams. From learning the very basics of Git, to familiarizing debugging platforms (gdb, lldb) we were each pushed to pick up on a number of tremendously useful skills in a short span of time. This experience also taught us about different people's different coding styles and the importance of documentation and reception of new ideas. We struggled when initially trying to integrate sections that individual members worked on, as a result of different understandings of components. However, after taking adequate time and patience to work through our parts, we were able to eventually compile our code!

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

Regarding our compilation process, we started with the idea of creating first one instance for each class so we could compile as soon as possible however since we wanted to enforce single responsibility in our project we needed to know more on what each subclass needed to provide. So we end up implementing most of our methods and by the time we compiled we had a bunch of errors which we fixed but it took us a long time. Thus we think we should have used a different method for out compilation process.

Conclusion

In conclusion, implementing this project allowed us to maximize the use of object oriented programming and take advantage of all of its unique features. The existence of classes allowed us to create encapsulate methods and members of each different feature of the game. Using inheritance, polymorphism and design patterns allowed us to avoid duplicate code in multiple scenario and produce a much more efficient script than an otherwise non-object-oriented program would allow. Overall, our group feels that implementing CC3K has not only been the biggest challenge, but also the best learning opportunity we experienced.