CC3K REPORT

JULY 25TH, 2018

- **Introduction**

Cc3k is a platform and puzzle game developed by three Waterloo students. If you love to explore the unknown land, if you love to slay the evil, if you love to find undiscovered treasure, this game is specially made for you! Come and find a way out by passing through all five floors. During the adventure, you may see different kinds of enemies: strong Dwarf, unpredictable merchant, sneaky halfling and so on! Is this the bad potion or the good one? Try it yourself! Sometimes it can turn your game over! Play it wisely and win the treasure!

- **Overview**

Overall:

we print the exact puzzle as required. The dimension of each chamber is counted to set the valid range of which the item (including character, potion and gold) can be produced within. Items are randomly generated with equal possibility in each chamber. Each action that the player makes will call notify on the floor. The floor will then reflect the changes in position or action of the player on the display and it will notify the enemies to make move.

By Class:

Floor:

Floor is a class containing a vector of strings (the puzzle display), a vector of Potion pointers, a vector of Enemy pointers, a vector of Treasure pointers and a Player pointer. Once the floor is created, (a player type is selected or the default race "shade" is used), the specified numbers of Potions, Enemies and Treasures are created with equal possibility in each chamber. The different types of treasures, potions and enemies are generated with respect to the given ratio. When PC's hp reaches 0, end the game and display a message with the player's score. The player is given two options, restart or quit. When PC steps on the stairs, the next floor is automatically generated with all temporary attack and defense zeroed. If the stairs in the 5th floor is reached, game is won and the player is asked if he wants to restart or quit. Each time an action is done by the PC or to the PC, a string specifying the action is appended to the string field called action in the floor. The action is then printed and cleared before the next command. If the command is restart, change floor number back to 1 and initialize the floor with a new Player. The Observer design pattern is used in Floor:

1. If PC takes actions (move, take potion or attack enemy), notify all Enemies who is not stationary to make move (either attack or move around randomly).
2. If PC kills Enemy, call notify to drop the gold or add the gold directly to player and delete that Enemy pointer in the vector. Replace the coordinate with '.'. Print the killing action.
3. If a Potion is consumed, call notify to delete the Potion pointer in vector, replace the display with '.'.
4. If gold was picked up, call notify to delete the Treasure pointer in vector, replace the display with '.'.

Item (abstract class):

Potion, Character and Treasure are derived classes of Item. We set the row and col coordinates as private and implemented public methods to get/change the value of the row/col numbers. These methods will be inherited to the derived classes: Potion, Character and Treasure.

1. **Treasure:**

   A concrete derived class of Item. We use visit design pattern for implementing "Treasure being picked up by the Player". When a player steps on a treasure, call the corresponding treasure's accept(Player*) which will then call the it to call Player's visit(Treasure*). Moreover, the value of Treasure are protected so that the value is accessible in Treasure's derived class (aka DragonHoard).

1.1 DragonHoard:

   Derived concrete class of Treasure. A DragonHoard has a value of 6 and it contains a Dragon pointer and a method that returns whether or not the pointer is a null pointer (check whether the dragon is dead to determine if it can be picked up).

2. **Potion:**

   Derived concrete class of Item. A potion is one of the 6 TypeofPotion (RH, BA, BD, PH, WA, WD). It contains a Boolean "revealed" which indicates if this type of potion has been consumed by the player. The accept(Player*) methods takes a Player pointer and call Player's visit(Potion*) and complete the different effects of different types of potions. If the particular type of potion has not been consumed before, we go through the vector potions in floor to set all potion of the type as revealed. We also add the TypeofPotion to the vector of TypeofPotions eaten in the player so when a new floor is created, all potions of types in eaten will be set as revealed. When the potion is consumed it notify the floor to make changes: update the player's stats, replace the 'P' symbol with'.', print the action.

3. Character:

Derived abstract class of Item. The derived class of Character are Player and Enemy. Since Player and Enemy both have Hp, Def, Atk. We have protected field Hp, Def and Atk and "get method" for each field so that we can set and get each character's information in Character's derived classes.

3.1 Player:

Derived class of Character. Each Player has a name "race" and its own Hp, Atk, Def, MaxHp, gold, tAtk and tDef (for when it consumes potions to temporarily change its def or atk). Whenever the player completes an action, notify the floor to reflect the changes on the display. The player can only walk on '.', '#', 'G' that is not a guarded dragon hoard or '+'. If he walks on gold, the gold is picked up. If he walks on the stairs, nextlevel() is called to initialize the new floor:

1. When a Potion is consumed, visit(Potions) is called on the PC. An overridden function visit(Potion*) is implemented for Drow since potions have enhanced effects on it.
2. When he hits the Enemy, do the damage to Enemy and notify floor.
3. When he picks up the gold, add gold to its "gold" field and notify floor.

3.2 Enemy:

Derived class of Character. Each Enemy has its own Hp, Atk, Def and MaxHp. Merchant is set to non-hostile until it is attacked by PC. When PC attacks a merchant for the first time, the atkMerchant field becomes true and set all the merchant in enemies to hostile and all the merchants that are created in the floors after will be set to hostile. Dragon is set to immobile and it attacks the PC when PC gets within one radius of the treasure or itself. When a dragon is slain, the Dragon pointer in DragonHoard becomes null and makes it available to be picked up. When an Enemy hit the Player, call visit( ) method and do the given effect on each specific Player. When an Enemy is hit by a Player, call Player's Visit( ) method.

- **<u>Updated UML</u>**

Since we took a long time to think through the construction of the program, the general structure of the updated UML and the old UML were very similar. During implementation, we added some public methods and fields that are needed to achieve certain functionality.

We tried to achieve the different abilities of PC and enemy using virtual overloading functions but ended up with a lot of code duplication. So we decide to give a string race to Player and a name to Enemy to identify their types and their special abilities. We were able to implement all the abilities of Players and Enemies in one function that takes a parameter player* or enemy* instead of their derived classes e.g. Drow*, Shade*, Dwarf*, Halfling*.

- **<u>Design (describe the specific techniques you used to solve the various design challenges in the project)</u>**

Four main design pattern was used :

Abstract Factory design pattern was used through our whole project since it's quiet useful and easy to understand. For example, the abstract class "Item" and its derived classes "Potion", "Treasure" and "Character", each Item has a coordinate (x,y), we implement get coordinate method in "Item" class so that every derived classes of it can access this method, reduce the duplicate code. Each sub class are related to the "Item" but they are also dependent classes from "Item" since each class can also implement their own methods.

Observer is another important design pattern we used. Whenever PC makes an action, we notify the floor, then the floor will notify all the enemies to make move. They will either move randomly or attack the PC. Then they will again notify floor for their actions or change in position and it will be reflected on the display. It supports the principles of loose coupling between floor and items that interact with each other. Moreover, it allows sending information to each other without any change in objects or observer class.

Visit pattern is used when Player interact with Enemy, Treasure and Potion. When consumed Potion, picked up gold or hit Enemy, different kinds of items have different effects on different Player, thus we use the visitor pattern so that when drow takes a potion the effect is different. Using visit design pattern allows us to deal with new classes without changing everything. For example, when we implement the bonus question, we add a new Player class called “Turtle” which is immune to the adverse effects of bad potions. We only needed to add its features in visit methods. Adding this new class does not affect other classes at all.

Strategy design pattern is also used in our code for maximizing the reuse of the code. We encapsulate the algorithm that each time generating an Item in a class hierarchy will have a coordinate in the floor. In the base class -- Item class holds the floor pointer, in this way, each Item can have their own features implemented in their own class and also have a common Item-use coordinate.

- **<u>Resilience to Change (describe how your design supports the possibility of various changes to the program specification)</u>**

For example, when we add a new class of an Item, the visit pattern allow us to add its features in visit methods, adding this new class does not effect other classes at all. Low coupling and high cohesion are shown through our project. The resistance of each class is necessary and logically. Normally each class will be include 3 or 4 times, this indicates the high cohesion of our project. For example, we do not have calling triangle model function(a calls for b, b calls for c, c calls for a again).

- **<u>Answers to Questions (the ones in your project specification)</u>**

<u>2.1 How could your design your system so that each race could be easily generated? Addition- ally, how difficult does such a solution make adding additional races?</u>
Abstract class were applied for this question. Since this pattern has the advantages to enable add new classes without changing everything. Also in our code, separate classes were designed for each race. When generating a player of a specific race, an object of that specific race would be generated. Our system was designed with low coupling. To add an additional race, a class of that race would be attached as a derived class to the base class Player. And then, several overloading functions will be added so that different enemies react differently to the additional race. In this way each race could be easily generated.

<u>2.2 How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?</u>
Not much difference but when some special features applied to some combats of Enemy and Race, we use strategy pattern for that. First we specify the type of the

race, then applied visit pattern to see if Enemy combat with this race has special events. The same general strategy was used when generating different enemies as compared to generating a player of different race. Each Enemy is a separate class as well.

<u>How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.</u>
Since each enemy's ability is only targeted towards some specific race of Player, we decided to use visitor pattern. Virtual and overloading methods are used so that each enemy behaves differently against different race of Player. Similarly, with Player, we created virtual and overloading methods so that different races of Player reacts differently to different races of Enemy.

<u>2.3 The Decorator and Strategy are possible candidates to modified the effects of potions, so that we do not need to explicitly track which potions the player character has consumed on any particular floor. In your opinion, which pattern would work better? Explain in detail, by weighing the advantages/disadvantages of the two patterns.</u>
We chose to use strategy pattern. First of all, strategy pattern is designed for classes have similar algorithms, which good and bad potion have similar patterns. Moreover, we do not need to add more responsibilities to a type of potion so decorator pattern is a bit extra. Defining different types of potion as one class allows for alteration of application behaviors without changing its architecture.

<u>2.3.2 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?</u>
An abstract factory design pattern was used such that when a pile of gold is picked up or potion is consumed; in other words, the item is obtained; these kind of similar functions can be implemented in the parent class. Due to the hierarchy of this pattern, both child classes, gold and potion can access to it and use it. Also the feature "gold can be walked over" (which potion does not have) can be implemented in gold's own class. Moreover the overridden functions of each child's own class will not have an effect on others.

- **<u>Extra Credit Features (what you did, why they were challenging, how you solved them—if necessary)</u>**

1.Adding a new Player called “turtle” with ability to neglect effects of poisonous potions.

2. An Enemy called “mascot” which is non-hostile and does not take damage. Every time the player hits mascot, he drops a piece of gold where it was before.

It’s very convenient for us to add these two classes. As I mentioned previously, visitor pattern allows us to add the features very easily without having to make changes to other classes.

3.When Player walks towards a direction, Player only see one thing in that direction (if any), we implemented a “turn around” method so that if Player wishes to see the other direction, he is able to do that. For example, if the player faces north and would like to see what kind of potion is in the south within 1 block, just turn south and the action will print what you see.

- **Final Questions (the last two questions in this document).**

1. It will be very difficult to implement a large software alone as there are many components and many cases that need to be considered and it’s almost impossible to think of all of them by yourself. Working as a group, we were able to implement different components of the program individually and then putting them together later. It is a lot easier as you can implement you component of the program trusting that the other person can implement their parts correctly so you can use the methods that the other group member is to implement. If working alone, it is important to think thoroughly of the structure of the program to avoid logic errors. Sketch out a UML diagram and the .h files. Then implement the functions in the .h files. During that process you might realize that there are extra fields or methods that are needed. Finally leave half the time to compile the code and debug. There may be a lot of errors such as syntax errors or circular include errors that prevents the program for compiling. Once the program compiles, it’s time to test the program. Write out a test case and see if your program does what it’s supposed to do. And start debugging!
2. We managed our time very well and divided the work between the group members very well so that each person has equal participation on the project. We started two weeks before the due date and finished most of the coding part in less than a week. Then we tried compiling our program. As we were very cautious when writing the program there were not that many errors preventing a successful compilation even though there were more than 20 .o files. It took one night to fix all the circular include problems and do forward declaration in the correct places. Then the rest of the time were just testing to see if we have considered all the possible cases and fixing the problems along the way. If we had a second chance, we would like to include more bonus features in our

program as we only included two this time. We did really great overall and every member were extremely motivated and responsible. We were always on the same page. We don’t think there would be many things that we would have done differently if we started over as we did not encounter many detours during our implementation of the program, we managed time wisely and every team member contributed a lot.