CC3K Project Documentation

Introduction

Rogue has been a popular video game which all of our group members have tried and enjoyed in the past. We are excited to design and implement a similar game, ChamberCrawler3000 for our final project. From the process of designing and implementing this game, we expanded our understanding of the core principles of Object-oriented programming. Upon completion, we are confident enough to say that our design and implementation perfectly demonstrates polymorphism, abstraction, and encapsulation.

Overview

The followings are the main classes included in this project:

Floor and TextDisplay:

The class Floor acts as a subject in the Observer pattern which handles and processes all core functional elements, including processing and allocating user input commands, locating and organizing in-game enemies, potions, and treasures, as well as notifying the class TextDisplay. The class TextDisplay has a role of a observer in the Observer pattern. TD's main responsiblilities include reading in and processing a map from a .txt file, modify the display as notified, and printing out the map, PC's status, and actions to the stdout or displaying them in a window according to the user's demand.

Base:

The class Base is the most general abstract class that every class that shows on the map inherits to, including all characters and item. Its usage is to store, modify, and provide the location of the character or item to the floor when needed. It is worthwhile to mention that Base class is extremely useful with an enum class called Type. Type specifies what Type the object is, such a Vampire, a Dwarf, a MerchantHoard, etc. In this case, the developer can easily modify a PC, enemy or an item's behavior based on what type it is or its conter part is. One example is vampire's attack method, vampire can detect if the victim it's attacking is a Dwarf and react differently.

Character:

The class Character is an abstract superclass that all player character and enemies inherits from. It stores fields HP, Atk, and Def that both Character and Enemy have. One advantage of having this inheritance hierarchy is when passing parameters to methods that are suitable for both PC and enemies, such as move, attack, modifyHP, etc.

PC:

The class PC is an abstract superclass that all races of PC inherits to. This class implements methods that all or most races have in common, while allowing its subclasses to override due to

different features. One example is that modifying most races' HP needs to consider max HP, while Vampire needs to override this method because it does not have an upper HP limit.

Enemy:
The class Enemy is a similar class to PC in terms of its usage and responsibility. It is a superclass of all kinds of enemies, containing implemented methods that all or most enemies have in common, while allowing various forms of override. One advantage of the Enemy class is its capability to implement the Factory pattern – when a random enemy is generated, the method calls it can generalize the kind of pointer it receives as an Enemy pointer. It is worthwhile to mention that a subclass of it, Dragon, has a field named myHoard. It is used to modify its corresponding DragonHoard to be not guarded upon its own death. Addition to that, another subclass, Merchant, has a static field named isHostile, which is set to false by default and changed when any of the merchants on map is attacked or slain. If it is true, all merchant will become hostile and attack the PC when they are nearby.

Item:
Item is an abstract class that both Potion and Treasure inherits from. It does not have any fields but have a common method that both Potion and Treasure need – useMe(), the feature that both Potion and Treasure have in common.

Potion:
Potion is an abstract class that all potions, both temporary and permanent, inherits from. Potion is an implementation of Strategy pattern, which has different algorithms beneath it (temporary and permanent), while showing the same interface, which is useMe. One advantage of the Potion class is its ability to apply the Factory pattern – when a random Potion is generated, the method calls it can generalize the kind of pointer it receives as a Potion pointer. It is worthwhile to highlight that temporary and permanent potions have essentially different implementation: when temporary potions are used, they are collected in an array of tmpPotions that the Floor keeps track of and remove when a new level is reached; on the other hand, permanent potions directly modify the PC's field without being tracked.

Treasure:
Treasure is an abstract class that all treasures inherits from. One advantage of the Treasure class is its capability to realize the Factory pattern – when a random Treasure is generated the method calls it can generalize the kind of pointer it receives as a Treasure pointer. It is worthwhile to mention that it has a field named isGuarded and an Enemy pointer. It is implemented to facilitate DragonHoard to track if the dragon guarding it has been slain, to decide if it should be used.

Updated UML

Our final UML demonstrates the actual structure of our program. We have made some changes to our initial UML and they are listed below:

i. The most significant change in our structure is that we abandoned the idea to construct a map of Bases, which contain all characters, items, and floorDesign elements, and accepted a modified Observer pattern and included a textdisplay. The reason why we did this is that allocating resourses to and printing out an entire map of Bases is not efficient since most Bases will not change, such as tiles and walls. As a result, we came up with the modified Observer pattern that only updates the bases that are modified using the textdisplay and then print out a map of chars. In practice, this method is indeed efficient and saves us a lot of computing time.

ii. In addition, we added some methods that a subclass implements to its superclass as a pure virtual method. The reason behind this is that since we use the Factory pattern, all kinds of enemies, potions, and treasures are tracked by an Enemy, Potion, or Treasure smart pointer. As a result, it is nearly impossible to access a subclass's method or field without declaring a pure virtual method in their superclass.

iii. Finally, we added two more enum classes, Direction and GameStatus, to deliberately eliminate the possibility of bug due to typo. On top of that, we overrided several operators to facilitate the conversion between a user input, a string, and an enum class, as Direction. GameStatus is designed to be thrown, thus clearly indicating what to catch and what to print out or display, while eliminating the chance of not catching due to typos.

Design

Design Pattern and Technics

With thoughtful planning and detailed implementation, we deliberately pursue a good design that uses encapsulation, abstraction, inheritance and polymorphism. Indeed, we have used the following patterns.

1. Observer Pattern

Observer Pattern defines a one-to-many (in this case one-to-one) relationship that when one subject changes state, the observer is immediately notified and automatically updated. This pattern is used twice in this project for TextDisplay and Floor, Dragon and DragonHoard classes. In TextDisplay and Floor case, Floor is the subject and TextDisplay is the observer. In the class TextDisplay, *notifyDead()*, *notifyMove()*, *notifyPCMove()*, *notifySpawn()*, *notifyPCSpawn()* are implemented to respond to corresponding methods in Floor such as *spawnPC()*, *spawnEnemy()*, *PCmove()*, etc and modifies the corresponding char in the TD object and thus the display on the

screen. This pattern improves efficiency by not printing out a map of Bases, reducing method calls when printing to the stdout.
In Dragon and DragonHoard case, it uses the same concept in a modified form. Once a DragonHoard object is spawned, a Dragon ctor will be called(i.e., notified) and generates a dragon around the hoard to protect it. Also, when a Dragon is slain, its corresponding DragonHoard will be notified and thus modify its guarded or guarded status.

2. Strategy Pattern
Strategy Pattern defines a family of algorithms, encapsulates each one and makes them interchangeable. The strategy implementations can vary according to clients calling it. This is a good practice of encapsulation, abstraction, and polymorphism as the client won't know what the algorithm is implemented, an abstract class is called, and the parameter is allowed to be multiple classes. This pattern is used in the PC class. In the PC case, one example is the method *attack(victim)*, which attacks the victim and calls victim's beAttack(attacker), all victim's beAttack(attacker) will be called, but if the victim is a Halfling, the chance of Halfling's beAttack(attacker) be called is only 50%, instead of 100% for other kinds of enemies. An advantage of using it is that no additional methods need to be implemented, reducing coupling.

3. Factory Pattern
Factory Pattern defines an interface for generating objects and allows child classes to decide which class to instantiate. This pattern is used for randomly generating an Enemy, a Potion, or a Treasure with assigned possibilities. It is a good implementation of abstraction and polymorphism since the method calls the random generation can store different kinds of Enemies, Potions, or Treasures in one array or vector.

4. Template Method Pattern
Template Method Pattern defines the skeleton of an alogrithm in an operation but defers some internal steps to client child classes. This is a good practice of abstraction and inheritance since the specific method being called by a superclass's method can be varied for each subclass, such as max HP in the following case.
For the method PC::modHP(), it checks if the modified HP value exceeds the current player character's maxHP by calling this->getMaxHP(). Since different races have different max HP, PC::modHP() calls different getMaxHP() from different races, to realize abstraction and inheritance in this case. This is reflected in *PC.cc line8-18*.

5. Accessing Fields
Since fields are either private or protected, getters and setters are used to access and modify the fields. This is an excellent demonstration of encapsulation since the designer has control over what the fields shoud be, ensuring invariant. And the clients cannot directly access the fields.

6. Operator Overloading
Some operators(*, !, ==) are overloaded in this project to simplify the implementation. (*Type.cc, Direction.cc, Info.cc*)

Cohesion and Coupling

We have achieved high cohesion in our modules. For each class, .h, or .cc file, we only define related methods and standalone functions in that class, .h, or .cc file. For example, PC only has methods that either take effect on a player or used by a player. On top of that, standalone functions such as TextDisplay's operator<< override is declared in TextDisplay.h and defined in TextDisplay.cc.

Low coupling is achieved through separation of upper and lower genres, such as separating Floor, Base, TextDisplay, as well as separating different kinds of Characters and Items into separate final classes. Therfore, if the a new PC race, an Enemy race, or any different Item is introduced to our game, we do not need to change any of our codes except for the enum class and random generation chance. It is almost effortless since these changes are minor and do not affect the original implementation at all.

Resilience to Change

We have designed the program to accommodate various changes due to the implementation of inheritance, polymorphism, and operator overrides. If we wish to add a new character race, a new class can be simply added and inherits to the Character class with all implemented methods, private fields can be added to support any special requirements. The user can simply add new classes under PC and Enemy class and implement the override method if the new race has any different abilities. In the case that some changes in the abilities of some pc or enemies, it requires only a few changes for the specific methods. This also works for Treasure and Potion.

Answers to Questions

1. How could your design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional races?
We designed our character generating system based on the Template Method Pattern. For fields and methods that all Player Characters (PC) have in common or similar, we design general fields or methods in the superclass named PC. Then we allow its subclasses, which are the different races, to inherit them from the PC. If any race's fields or methods are at variance with the general one, we design a specific field or method for that race class to accommodate. For example, the race Vampire's class has an override implementation of attack to facilitate its ability to gain or lose health from each successful attack.
Using this approach, it is fairly simple to add additional races. All the designer needs to do is to add a subclass in PC, and implement any fields or methods that are different from the general case, without considering incapability with the original codes.

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

We used the Factory Method Pattern to generate all Enemies. First, we randomly choose 1 out of 5 chambers. Second, we randomly select one of the floor tiles in the chamber. In the end, we randomly pick the race of the enemy using the Factory Method Pattern. This method is substantially different from how we generate the PC since we do not randomly pick the race of the PC. The player has the right to choose which race to start with.

We use different approaches because the differences in how the PC and enemies need to function. For PC, we create a PC character in the main function stack and then keep tracks on the PC's position and fields (gold, HP, Atk, Def) throughout the game. In contrast, we generate enemies after a floor map is read in and remove all of them when we go to the next floor.

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

Yes, we do use the same techniques as for the player character races. For fields and methods that all enemies have in common or similar, we design general fields or methods in the superclass named Enemy. Then we allow its subclasses, which are the different races, to inherit them from the Enemy. If any race's fields or methods are at variance with the general one, we design a specific field or method for that race class to accommodate. For example, the race Human's class has an override implementation of dropGold() to facilitate its ability to drop gold in a different fashion.

4. The Decorator and Strategy patterns are possible candidates to model 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.

There are two different duration effects in the game, permanent effect and temporary effect (the effect will be removed when going to the next level). For the design of permanent potions, the method is the same as which of treasures, the PC uses the potion or picks up the treasure, and the HP or gold field of the PC gets modified. Moving to the next floor does not remove these effects.

For the design of potions with temporary effects, there are two possible candidates. The advantage of using Strategy Pattern is the users can make a Strategy Parent class, make the strategy of potions its children class, and PC can pick a particular potion strategy to use. The disadvantages are that we have to make an explicit list to record the temporary potions that the PC has taken on the floor and remove the effect by removing it from the list. One advantage of Decorator Pattern is that we can easily remove the temporary effect by trace back the decorated PC and remove the decorators when going to the next floor. However, the disadvantage of using Decorator pattern is we have to make every potion its own class to be the children class of the Decorator parent class. Initially, we designed to use Decorator Pattern on the effects of potion, and implement each potion as a concrete decorator. However, we found out that calling a

specific Potion's ctor needs to call both PC's ctor and Potion's ctor, both of them are the subclass of the class Base, we worried about implementation errors so we abandoned our initial plan. Eventually, we use a modified Strategy Pattern because there is no cross effect among different temporary potions and there is no difference which temporary potion is used first. We used a map to store all temporary potions with its quantity. In this case, we can just clear the map when the PC reaches the next level.

5. 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?
Treasure and Potion share very similar structure in terms of fields and method functionalities, so we have a parent class Item setting up the skeleton of each class, and some private fields are added to each child class based on the unique properties of the class as required. Then we implement two very similar functions createPotion() and createTreasure() for generation purpose. This is slightly different from original design since we planned to only have one createItem() function because it can improve efficiency. It is because that the Floor can track Potions and Treasures in two different vectors, thus unnecessary loops can be prevented.

Extra Credit Features

1. Random Chamber

We realized the capability to read in maps that have chambers randomly allocated and connected. Our idea is to read in the original map first, and allocate each valid char (such as '\', '@', '.', etc) into the same Chamber as the one to its north, south, east, and west if they are valid as well. Our implementation is to read through the map and find an unallocated char, generate a Chamber for it, then expand from that point to allocate all char beside them into the same chamber, until no other valid char is beside. Then we read through the map again and find another Chamber. This loop breaks when all valid chars are allocated to a Chamber. Eventually, we can generate a map with all chambers separated. To randomly generate, we use the number of Chambers and number of valid tiles in that Chamber. The challenging part is how to find all valid chars and correctly separate them into chambers without leaving any valid char unallocated. We overcame it with several while and for loops, with helper function to indicate if a char is valid, and a map of bools to track if a valid char has been allocated.

2. Intelligent Enemy

We realized the ability to allow enemies to chase the PC when the PC is in the same chamber as it. The implementation is to find all available spots around the enemy, and sort them based on their distance from the PC, using the distance formula.

3. WASD

We realized the ability to allow users to use UP, DOWN, LEFT, RIGHT keys to move instead of typing in a direction everytime. We implemented it by adding a window to the TextDisplay class and print in the window if needed.

4. Smart Pointer

We used all smart pointers by using shared_ptr<Class> for a “own” relationship and weak_ptr<Class> for a “has” relationship. Indicating and separating ownage can successfully prevent memory leak or dangling pointer.

5. Extra PC race: SuperMan Character with teleport

We implemented a SuperMan race and teleport to facilitate beginners to enjoy the game without worrying death. The teleport is implemented by taking in an (x, y) position and teleport to that location if that location is not occupied (except for Treasure).

6. Random race generation

We implemented random race generation by randomly select a PC race per level. It is implemented using rand().

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?

This is a middle size project, so we believe that the initial design is of great importance to developing such a program. It is essential for all group members to actively participate in the design discussion and understand the design thoroughly. This ensures the implementation goes smoothly and everyone is on the same page when any changes are made.
Breaking down the entire project into smaller pieces helps make the thinking process for each piece easier and clearer. Then consider the internal relationship between the classes and connects the pieces together completes the entire design.

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

We would probably spend more time separating tasks, designing UML, and going through which parts should be called for each user command. We thought we haved clearly separated our tasks, but upon implementation we realized that there are still many scenarios that we did not consider, such as how to display messages to screen. If we had spent more time separating more detailed tasks, we would spend less time combining all parts from everyone.