CC3K - Villain

Spring 2017

tmaekawa

y2423zha

Introduction

CC3k is a rogue-like terminal game implemented with C++ using Visitor Pattern, Strategy Pattern, NVI idiom and other C++ skills.
#Basic commands:

- no, so, ea, we, ne, nw, se, sw: moves the player character one block in the appropriate cardinal direction
- u <direction>: uses the potion indicated by the direction
- a <direction>: attacks the enemy in the specified direction, if the monster is in the immediately specified block
- f: stops enemies from moving until this key is pressed again
- r: restarts the game. All starts, inventory, and gold are reset. A new race should be selected
- q: allows the player to admit defeat and exit the game

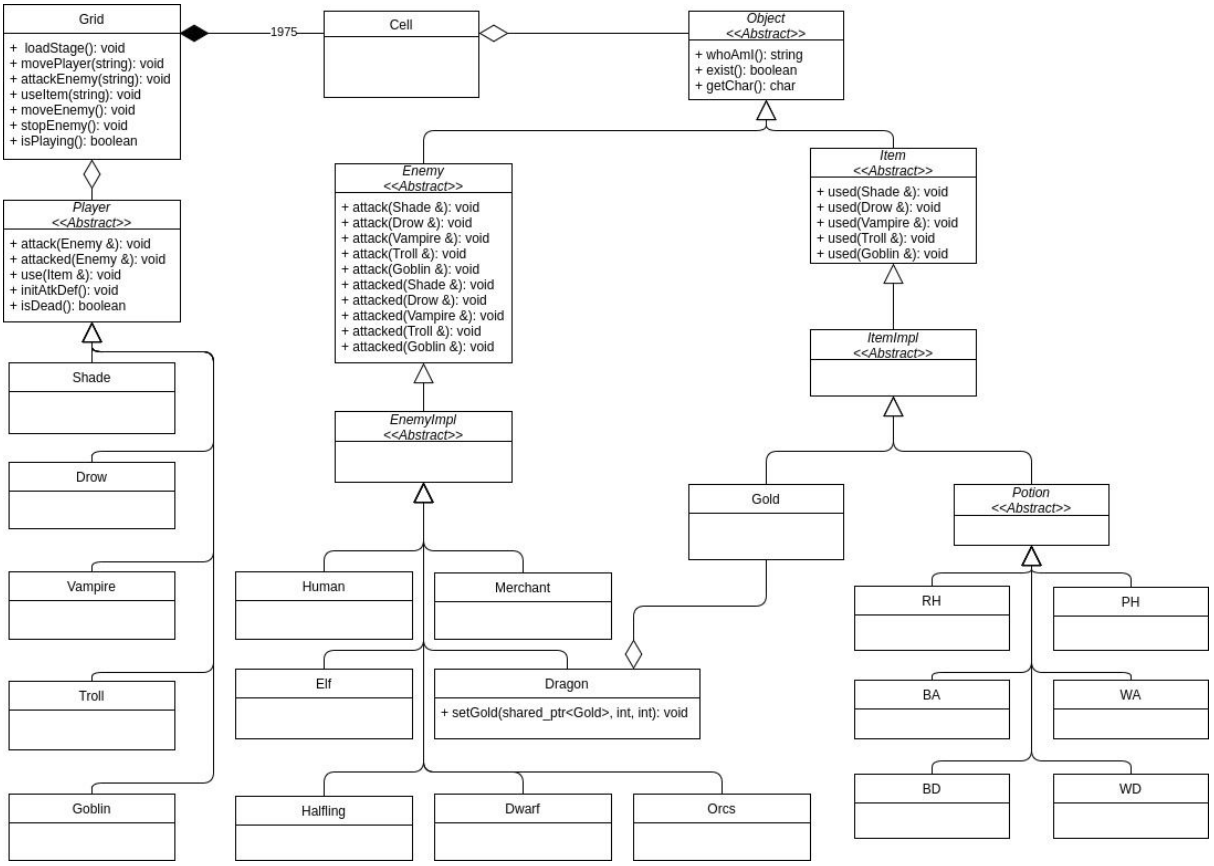
#Bonus commands:

- l: chooses the level of intelligence of enemies, ranging from 1 to 4

Overview

In our project, we use a Grid class to store the information of the whole game and respond to commands. Grid owns 79 * 25 (the size of the display) Cells, and each Cell contains information of a specific position, including x, y-coordinates of this position, a character representing this position and a shared pointer to an Object. An Object is either an Enemy or an Item, i.e., Object is an abstract class with Enemy and Item as its subclasses. For player characters, we create an abstract Player class and 5 concrete subclasses inherited from it. For enemies, we also create an abstract Enemy class like Player, but we create an EnemyImpl class between the abstract Enemy class and 7 concrete subclasses as well to reduce duplicated code. For the generation of items, we use Strategy Pattern. And similar to Enemy, we create an abstract Item class and ItemImpl subclass inherited from Item to avoid duplicating code. There are 2 subclasses inherited from ItemImpl--Gold and Potion, and Potion has 6 concrete subclasses. We use Visitor Design Pattern for Player and Enemy as well as for Player and Item. And we adopt Template Method Pattern and NVI idiom in the design of Player, Enemy and Item.

UML



The difference between the original uml and the updated uml lies in:

Enemy

In the original uml, we only pass a Player reference to Enemy's "attack" and "attacked" methods. But in the updated uml, we pass 5 concrete player race references to Enemy's "attack" and "attacked" methods. This is due to compilation dependency. Since we've already included "enemy.h" in "player.h", we cannot include "player.h" in "enemy.h". So we use forward declaration to declare Player and its 5 races. We were intended to use wrapper for methods which get Player reference as its parameter and call methods for each player race, but it requires downcasting between Player and its 5 races, also the compiler does not know the relationship between Player and race classes since forward declaration only tells the name of the class, so we cannot switch between an abstract player reference and a concrete player reference. For example, in our original design, we intended to implement "attack(Player &)", "defaultAttack(Player &)" and "attackImpl(Shade &)" in this way:

```
void Enemy::attack(Player &p) { attackImpl(p); }
void Enemy::attackImpl(Shade &s) { defaultAttack(s); }
void Enemy::defaultAttack(Player &p) { … }
```

But here is the problem. There is no automatic conversion between player and shade and we cannot explicitly downcast from Player reference to Shade reference. Due to this problem, we abandon the “attack(Player &)” method and set 5 public “attack” methods which consume concrete player references, and the same for “attacked” methods.
As the original uml, we also adopt Template Method Pattern and NVI idiom, where we have 5 protected pure virtual “attackImpl” methods and 5 protected pure virtual “attackedImpl” methods in Enemy. However, if we implement them in each concrete subclass, there must be much duplicated code. So we add a EnemyImpl class to implement all “attackImpl” and “attackedImpl” methods by calling protected pure virtual methods “defaultAttack” and “defaultAttacked”. Both of “defaultAttack” and “defaultAttacked” consume a player reference because there is no restriction for upcasting and compiler knows the relationship between “player.h” and “enemyImpl.h” at this point since EnemyImpl class includes each Player concrete subclasses. Thus in each concrete enemy subclass, we just need to override “defaultAttack” method, “defaultAttacked” method, and handle special cases in “attackImpl” or “attackedImpl” if needed, which reuses a lot of code.

Item
We adjust the implementation of Item in the same way as Enemy since Item and Enemy are both visitors of Player. We have 5 “used” methods passed by 5 player race references due to compilation dependency. And we add a subclass ItemImpl inherited from Item to avoid duplicating code.

Object
In the original uml, we set Player, Enemy and Item to be subclasses of Object. But in the updated uml, only Enemy and Item are subclasses of Object. And we put a unique pointer to the player character in Grid and also the player character’s x, y-coordinates in Grid. We make this change because in this way we can better keep track of the player character’s movement and immediately know its status.

Design

Template Method Pattern and NVI idiom
We adopt Template Method Pattern and NVI idiom for the design of Player, Enemy and Item. For example, in the abstract Player class, we have public non-virtual methods “attack”, “attacked”, “use” and corresponding protected virtual methods “attackImpl”, “attackedImpl”, “useImpl”.

Visitor Design Pattern

We use Visitor Design Pattern for Player and Enemy, as well as for Player and Item. We set Player to be acceptor, Enemy and Item to be visitors. So for player characters, “attackImpl”, “attackedImpl”, “useImpl” methods just need to call corresponding methods in Item and Enemy.

Strategy Pattern
We use Strategy Pattern to model the effects of potions. We set “defaultUsed” in class ItemImpl to be pure virtual and implement it in each concrete subclass.

Use of Vectors and Smart Pointers
To avoid explicitly managing memory, we handle all memory management via vectors and smart pointers. In grid, we store the information of the whole game in a two dimensional vector of Cells. We keep track of the status of the player by a unique pointer to the player character in Grid. And in each Cell, there is shared pointer to an Object.

Resilience to Change

In our implementation of the game, we have partitioned tasks in different objects and reduced coupling as much as possible between them. All properties of objects are implemented in each object classes, the behaviour or player is implemented in item and enemy classes by visitor pattern. User interface is implemented in main.cc and all objects are controlled in Grid class. By doing this, we only need to have very small changes to add new components. The details of the expected variation to given changes required are discussed below.

Case 1: Change in Rules
If some rule changes, only classes that are related to the change should be changed. If the status of the character (either player or enemy) changes, only constructor of the character should be changed. If the ability of the character changes, some methods in EnemyImpl should be changed in case of player character, and overridden methods in each race class of Enemy should be changed in case of Enemy character. If the effect of a specific item changes, the overridden method in this item subclass should be changed. If there are other changes in rules other than the changes mentioned above, these changes can be implemented in the Grid class. Thus, only main, Grid, and few other classes will be recompiled when rules change.

Case 2: Change in input syntax
When input syntax changes, only “main.cc” and “grid.cc” should be changed. In most cases, only main.cc will be changed when input syntax changes because inputs are

interpreted in main.cc and each method is called from main.cc. However, grid.cc will also be changed when the syntax of directions (“no”, “ea”, etc.) or the symbols for races (‘s’, ‘d’, etc.) have changed since the interpretation of these features are done in Grid class. Thus, only main and Grid needs recompilation when the input syntax changes.

Case 3: New Features
To introduce new features, new class should be created and few other classes may need to be changed. For example, adding new player race class needs to create a new race class inherited from Player, and create corresponding visitor methods in Item and Enemy. Grid and main should also be changed to construct the new player character race. Thus, main, Grid, and all classes under Item and Enemy will be recompiled by data dependency. However, adding new enemy and item is much simpler. It only requires to create a new subclass and modify main and Grid to construct the object. The special abilities can be implemented by overriding some methods in the new class. Thus, only main and Grid should be recompiled in this case. Introducing a whole new component may require adding new classes, modifying main, Grid and some other existing classes. For example, introducing inventory system needs to modify Grid class, and introducing class system may need to change Player class.

Overall, the new changes only requires to change main, Grid, and few other related classes, but other existing classes remain the same.

Answers to Questions

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

Answer: Similar to our original design, we create an abstract Player class and have 5 subclasses inherited from it. Calling the default constructor of each subclass automatically sets the fields of the player. Unlike our original design, where we implement “attack”, “attacked” and “use” methods in the abstract Player class and leave blank in each concrete player subclass, we need to implement these three methods in each of the 5 subclasses in our new design due to compilation dependency(as stated in uml part). Here we adopt the Template Method Design Pattern and NVI idiom, so we need to override the protected pure virtual “attackImpl”, “attackedImpl” and “useImpl” in each subclass. In Grid class, there is a unique pointer to the player character and a private method called “setPlayer” to generate

the player character. It's not difficult to generate a new player character by following these steps:

1. Create a new subclass inherited from Player and implement methods as other races.
2. Add corresponding "attack", "attacked", "attackImpl", "attackedImpl" methods in Enemy and EnemyImpl classes.
3. Add corresponding "used" and "usedImpl" methods in Item and ItemImpl classes.
4. Slightly change "setPlayer" method in Grid (add another case).
5. Slightly change main.cc (add another input option).

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

Answer: Similar to our original design, we also create an abstract Enemy class, but we have a subclass called EnemyImpl to reduce duplicated codes. We set 7 kinds of enemies to be subclasses of EnemyImpl. Calling the default constructor of each kind of enemy automatically sets the fields of this enemy. In Grid class, there is a private "setObject" method which generates Item and Enemy. It's quite similar to generating the player character, because both player and enemies are generated when calling "loadStage". I think the difference of generating player character and enemies lies in that the type of player generated is determined by command line input while enemies are randomly generated. Also, it is much easier to add a new kind of enemy than a new player race as we just need to add a new subclass inherited from EnemyImpl and change "setObject" and "loadStage" in Grid slightly.

Question: 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: Similar to Player, we also adopt Template Method Design Pattern and NVI idiom for Enemy. There are 5 public non-virtual "attack" methods, 5 public non-virtual "attacked" methods and 10 corresponding protected pure virtual "attackImpl" and "attackedImpl" methods in the abstract Enemy class.
However, since player is acceptor and enemy is visitor, we need to explicitly implement behavior in enemy class. And unlike Player, we do not implement all "attackImpl" and "attackedImpl" methods in each subclasses. To avoid duplicating code, we create an "EnemyImpl" class inherited from Enemy and implement all "attackImpl" and "attackedImpl" methods by calling "defaultAttack" and "defaultAttacked", where "defaultAttack" and "defaultAttacked" are protected pure virtual and consume a player reference. So each concrete class just needs to override the "defaultAttack" and "defaultAttacked" methods. Also, if there are some

special cases for certain enemy and certain player, we just override the “attackImpl” or “attackedImpl” in this enemy class.

Question: 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.

Answer: The same as our original design, we use Strategy Pattern to model the effects of potions. Here is our analysis. Using Decorator Pattern can keep the initial status in the basic class and change status when propagating through decorators, but growing decorators will result in growing stack when accessing to basic class’s fields. Using Strategy Pattern can directly change the status and will not grow like decorator, but the initial status is not stored. So storing the initial status (atk and def) and creating methods which reset the current status to the initial status together with Strategy Pattern work perfectly to implement the effects of potion.

Question: 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 create an ItemImpl subclass inherited from the abstract Item class to reuse code. Then we set Gold and Potion to be subclasses of ItemImpl. In the abstract Item class, we adopt Template Method Pattern and NVI idiom. We set 5 public non-virtual “used” methods and 5 protected pure virtual “usedImpl” methods in Item. In ItemImpl, we implement 5 protected pure virtual “usedImpl” methods by calling “defaultUsed”, where “defaultUsed” is a protected pure virtual method. Then we implement “defaultUsed” in each concrete subclass. If a potion has special effect on a particular race, then the “usedImpl” method consuming this race will be overridden in this potion class. By creating ItemImpl class, much code are reused since we just need to provide default behavior of each item on all races and override methods which need to be implemented specially for certain race and certain item.

Extra Credit Features

1. Use of vectors and smart pointers
In our project, we never explicitly manage the memory. Instead, we handle all memory management via vectors and smart pointers. (as stated in Design)

2. Enemy Intelligence
We give enemies intelligence with level from 1 to 4. Typing “l” into the command line can select the intelligence level. Enemies move towards player when player is within lv (level) blocks radius of them and move randomly as usual when the player is out of range. The default value is 1 which enemies does not chase player and move randomly.

3. Two more player characters
We add another two player characters called Phantom and Zombie. Their status and characteristics are in the following chart:

Race	HP	Atk	Def	Characteristics
Phantom	1	25	25	100% ability to cause enemies miss in combat Cannot use potions or gold
Zombie	100	10	10	Atk increases by 1 as HP decreases by 1

4. One more kind of enemy
We add another kind of enemy called Alien(100HP, 15Atk, 40Def). Its special ability is that it decreases the player’s Def by 1 per successful attack. User will be asked to choose whether to add this enemy or not after selecting a player character.

Final Questions

Q1. What lessons did this project teach you about developing software in teams?
There are only two members in our team, so it’s very convenient for us to contact with each other. We have good time management, reasonable division of work and smooth cooperation. Both of us are very active and we communicate with each other in time. So everything goes well in our project.

Q2. What would you have done differently if you had the chance to start over?
We have made some changes from the original UML because what we have assumed did not work as expected, such as the problem due to compilation dependency. If we can start over, we would like to make neater original design of the project by using the knowledge we’ve gained from making this project.

Conclusion

The CC3k project is successfully completed by our joint effort. We learned how to design a program, enhanced our programming skills in C++ and are more familiar with various design patterns such as Visitor Design Pattern, Strategy Pattern and Template Method Pattern. We also learned how to use vectors and smart pointers to better manage our memory without leaking. Moreover, we feel very lucky to have each other as our teammate. We both expect the next cooperation.