CC3K: Final Report

Roselyn Zhang, Shutang Gong

CS246

Instructor: Rob Hackman

July 26, 2022

Introduction:

The project is a recreation of the famous game, CC3K+, with simplified implementation. This game allows the user to initialize different races of the player, and by eliminating various monsters, the player can progress to the next floor. Eventually, the player wins if it can finish all five floors, fails if the player's health dropped to 0 before it wins. In addition to monsters, the game also obtains potions, gold and other features to attract users. The game is implemented by Sean Gong and Roselyn Zhang using programming language C++.

Overview:

The gameplay implementation consists of five floors with five chambers in each floor. Each floor is created using 2d vector, which contains shared pointer to "Object", which is the class name of the superclass of all entities in the game. The player, monsters, potions, golds are generated randomly in location with equal possibility at every start of the game. All of behaviors and entities, are controlled in wrapped class "Game_Board". All the other characters including player, monsters are under the superclass Object, which is an abstract class. The subclasses including Player, Enemy, Potion, the children's classes of Player / Enemy / Potion contains each race of the player / monsters / potions. The gold and other treasure are stored as fields in each race class of the player for convenience.

Updated UML:

Please refer to the separate file uml.pdf

Design:

Structures:

Character:

stores a string that records the type of the character, two integers represent the row, column location, respectively.

Status:

stores three double values, the character's hp (health), atk (attack), def (defense).

Return_type:

Stores return types for play method of the game. In this way, the return value can not only contain exit code, but certain features. (For example, when play returns 0, which means the game should go to the next level, it also returns a copy of shared pointer of player, so the information can persist between different levels.

Classes:

Game_Board

Game_Board()	Constructor for game board to initialize the game.
Bool Visible_Potion(String name)	Return whether certain potion is visible.

Void Move_round()	Move certain entities to certain direction.
Valid_Move()	Determines whether certain move is legal.
Play()	Start play the game.

**For class Player and Enemy, since the majority methods of each race and monster type is the implementation of the abstract class Player and Enemy, the usage of concrete classes are all similar. The methods that are unique to each concrete class act more as a helper function to assist the overridden virtual functions. So, the concrete classes with their unique methods will be omitted here.*

Similar private fields that occur in both Enemy and Player:

Character c;	Stores the type and the location of the current object.
Status s;	Stores the hp, atk, def of the current object.
bool isLive;	True if the current object is alive (hp > 0); false, otherwise.

Similar public methods that occur in both Enemy and Player:

Character getCharacter();	Return a Character structure that consists of the current object's type, and location
Status getStatus();	Return a Status structure that consists of the hp, atk and def of the current of object.
double getEffect(double def)	In Player: Enemy's def In Enemy: Player's def By passing the opponent's defense value to calculate and return the cause damage based on provided formula.
bool getLive();	Return true if the character is alive (hp > 0); otherwise, false.
void move(int x, int y);	Change the location of the current object in the its private field structure Character c.

Other important private fields in each race in Player:

bool hasBA;	Default value false, if the BA potion is used, change it to true. Reset to false when reaches new floor.
bool hasBD;	Default value false, if the BD potion is used, change it to true. Reset to false when reaches new floor.

bool hasWA;	Default value false, if the WA potion is used, change it to true. Reset to false when reaches new floor.
bool hasWD;	Default value false, if the WD potion is used, change it to true. Reset to false when reaches new floor.
int gold;	Record the number of golds the player has.

Other important public methods in Player:

bool takeDamage(shared_ptr<object> e)	Returns true if the attack is success, false if not success. It takes a smart pointer, the base class of e will be a type of Enemy. Use e to call getEffect method in Enemy, and deduct the player's hp based on the return.
void takeEffect(shared_ptr<Potion> p);	Change the player's Status based on the type of potion by calling Potion::getEffect().
void rmEffect()	Remove the effect of temporary potion when reaches a new floor.
void addGold(int x);	Add the number of golds in the private field.
int getGold();	Get the number of golds in the private field.
void attach(shared_ptr<Object> o); [non-virtual method]	Attach the adjacent neighbors to its private field neighbors.
void neighbors_clear();[non-virtual method]	Clear the content inside the private field neighbors.
vector<shared_ptr<Object>> getNeighbors(); [non-virtual method]	Get private field neighbors
void set_chamber(int n); [non-virtual method]	Keep track of the chamber that player is in, so the generation of stair can be in another chamber.
int getChamber();[non-virtual method]	Return the chamber player is in.

Other important private fields in each monster type in Enemy:

bool isHostile;	Record if the enemy is hostile or not.
bool compass;	Record if the enemy carries the compass or not.

Other important public methods in Enemy:

void takeDamage(shared_ptr<Player> p);	Use smart pointer p to call the getEffect function in Player, and deduct the enemy's hp based on the return.

void take_compass();	Let the current enemy carry the compass.
bool get_compass();	Return true if the current enemy carry the compass; otherwise, false.
bool access_Hostile();	Return the value of the private field isHostile.
void getHostile();	Set the private field isHostile to true.
void change_fight(bool n); [non-virtual method]	Set the private field in_fight to the parameter n
bool get_fight(); [non-virtual method]	Return the value of the private field in_fight
void not_live();	Set the current enemy to die (isLive = false)

Important public method in Potion:

Status getEffect();	Return a Status structure that consists of the effect of the potion based on the type of potion. For example, RH will return {10, 0 , 0} (hp, atk, def); WA will return {0, -5, 0} (hp, atk, def).

Resilience to Change:

- Add more monster types / player races

Each character has its own class and has been placed in well-named files with its header files and implementation file, which increases the readability of the whole program. Since the monster types and player races have their own classes and has a common superclass, it provides the possibility of convenient implementation if there is need in having more types or races in the future. As the programmer only needs to add the new monster type / player race class in the separated module. As the class is defined, the programmer then can simply call the responsible functions in the gameplay function to establish the change.

- Add / Change properties to the enemy or player

As all the functions that return the corresponding character' properties are returning either a Character or Status structure. Thus, the property of enemy or player can be implemented by changing the two structures with property related functions in each class such as constructor. In addition, as each class has its constructor, if the initializing property needs to be changed, modifying the constructor allows to use minimum steps to achieve the goal.

- Add / Change skills of enemy or player

This can be achieved by adding more functions under each monster type / player race class and apply the corresponding function in the gameplay function.

Answers to Questions:

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

All the characters in the game including player are under a superclass, Object. Player class is a direct subclass of Object, and each race is a subclass of Player. In this way, when constructing a specific race of player, the gameplay function can initialize a list of Object pointers and store the race under Object. Moreover, the particular skills of each race, such as elves have the ability to negate negative potions to positive, are implemented under each race class. With each race's constructor, the default Status and Character will be initialized. This solution provides ease in adding additional classes, since adding new subclass under Player and adding new race to the list of Object in the gameplay function allow flexible changes in possible additional classes in the future.

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

All the entities in the game inherits from the same superclass" Object. In this case, for the generation of each entity, they all utilize the same generation function "random_gen" to prevent repetition. Since each monster type is under class Enemy, and Enemy is a child of the superclass Object, assigning new monster to the specific location by calling each monster type's constructor, the monster can be created with the initial property including Character and Status. The constructor takes two integers, number of rows and columns as parameters. It is similar to generating player character, except the player only has one in each game. The high similarity is also because Player is also a subclass under Object, and each race is a child class of class Player.

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

Special abilities can be implemented as methods in each corresponding character's class. For example, the elves' ability of changing negative potion effects to positive is implemented as a function under class Elves. Since each of the race has the takeEffect function, the implementation will occur under its own takeEffect. Moreover, as Player is an abstract class with a pure virtual function of takeEffect, this special ability does not require anything special in the gameplay game. For stealing for gobbling, health regenerations, and health stealing, it can be achieved either by modifying the existing methods or adding new methods under the related class.

Question: 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?

After careful consideration, no design pattern was applied to model the effects. Instead, there is corresponding boolean variables for each type of temporary potion, when the potion is applied, the variable becomes true under takeEffect method under each race class. There is also a rmEffect (stands for remove effects) method under the class. When the game enters a new floor, the rmEffect will be called to remove all the temporary effects that have become true on the

previous floor. We could have used observer pattern to avoid explicitly track which potions the player has consumed on each floor. When it observes a new floor, the program could set the player to the status before taking the temporary potion. However, we decided to not apply the design patter, since we believe explicitly tracking which potion would be easier to understand and implement in this scenario, as there are not much additional skills applied on potions.

Question: 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 a potion and then generation of treasure does not duplicate code? How could you reuse the code used to protect both dragon hordes and the Barrier Suit?

Firstly, in terms of the generation. All the entities, including player and enemies, use the same generation algorithm and same function (random_gen). The only extra functions required is to generate the correct sub class according to the type of entities. (the function takes type as an parameter, and return the correct type of shared pointer of subclass). As for dragon hordes and barrier suit, they both have a private bool field named "pickable", which implies whether they can be pick based on the dragon status. When dragon is slayed, their pickable status is changed to true and they behave like normal items so the player can easily pick them up.

Extra Credit Features:

Throughout the program, shared pointer is used. Raw pointers are only used when there isn't ownership. As a result, no memory had leaked.
Different colour is also implemented to distinguish between enemies and player.

Final Questions:

Question: 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?

Sean: The most important thing I learned about developing software collectively is to communicate before the software start building. An hour discussing the separation of work and the tool/idea used to develop the program can easily save hours of work when combining different member's code. Especially the class relationship, as dynamic casting cannot solve all problems arise from complex class relations.
Roselyn: I would say the most important thing would be taking responsibility and communication when working in teams. Taking responsibility of the assigned part of the project and being responsible to complete them before deadline are vital for the whole team to successfully produce a product. Communication is also as important as taking responsibility. We have encountered the problem that something me and Sean did were overlapped, or neither of us wrote some important codes. This reflects that we need more communication during collaboration to avoid wasting time and efforts. If I worked alone, I would try to split one large problem into small parts and develop those with careful design. This enables me to test the

modules separately to ensure each small parts functioning, so that the testing of the large problem would be simplified.

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

1. Communicate with more details of each person's task to avoid overlap.
2. Design more carefully on how to write each class.
3. Consider applying design patterns to assign more flexibility to the game.
4. Manage time better by building a daily plan before we started.
5. Break down the floor and chamber into modules to test.

Conclusion:

From this opportunity, we accumulate the experience of collaboration and obtain a better sense how team works when developing a large project in real life. What is more, developing the game CC3K allows us to facilitate our knowledge of object-oriented programming and many syntaxes in C++. The project also practices our creativity, problem-solving, communication skills which we can utilize in our career.