

A dark blue vertical bar on the left side of the page. A blue arrow points from the bar towards the right, containing the date 4/3/2017.

4/3/2017

CC3K Design Documentation

CS246

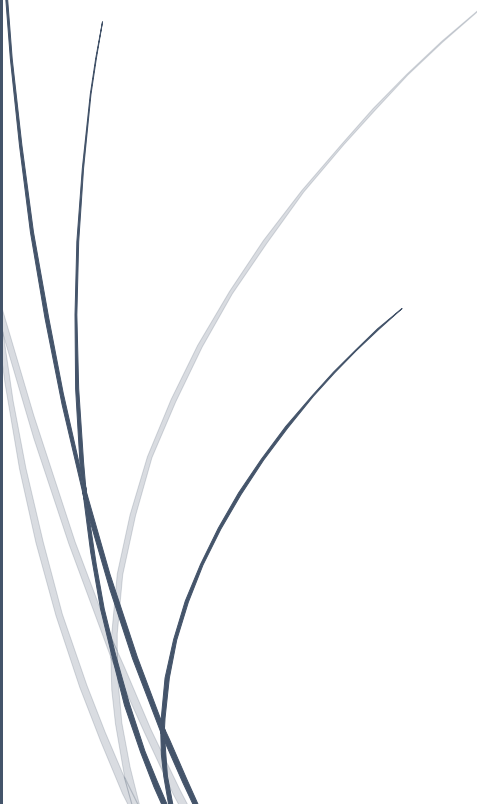
University of Waterloo

Faculty of Math

Raymond Tan

Yinong Wang

Qifan Zhu

Several thin, curved lines in dark blue and light grey originate from the bottom left corner and sweep upwards and to the right, creating a dynamic, abstract design element.

INTRODUCTION

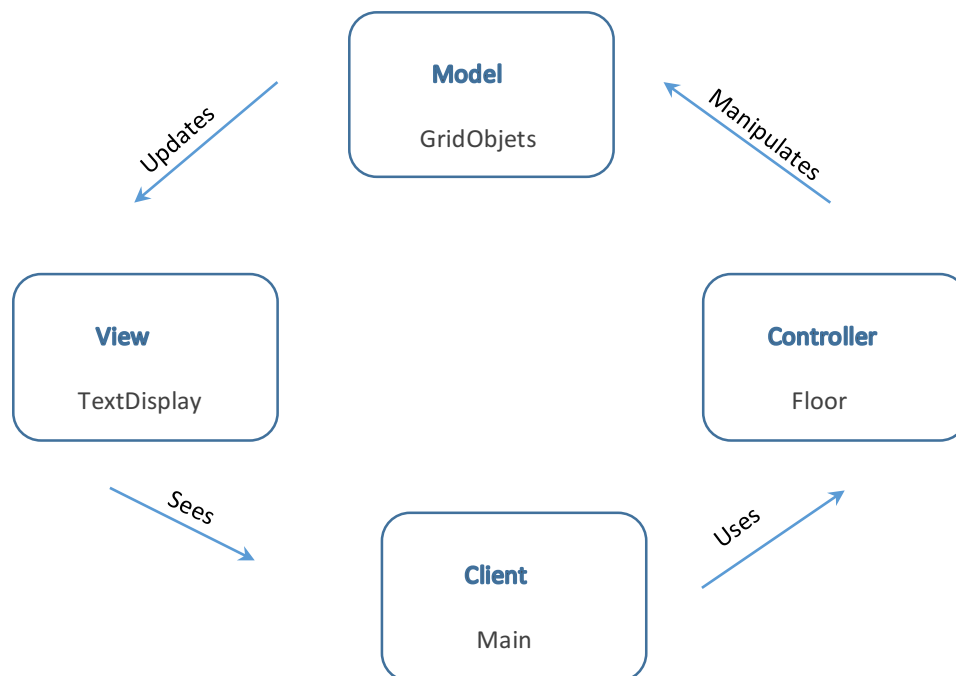
Chamber Crawler 3000 is a classic simplified rouge-like game. The only significant difference is CC3K does not update the terminal in real-time instead redraws the game board and other elements every turn.

The game has two parts of text display. The top part of the display consists current floor, chambers, player, enemies, items, stairway, passages, etc. The very bottom of the display shows player's stats such as health, attack, defense, movement and floor level. The game grid consists of a board with 79 columns, 25 rows and 5 rows to display player stats.

Player is denoted as '@' can go in 8 different directions and can collect gold when enemy is slayed. These actions work similar to the basic rouge-like game. There are 5 different floor is available to the player and the goal of the game is to reach the highest floor by going through the stairways.

OVERVIEW

According to the Model-View-Control Model, the program can be divided into 4 main components.



Clients uses commands in Floor to manipulates the actual map in GridObjects and update the actual map to TextDisplay and display the map onto the screen.

MAIN

Client executes a command and the game will perform it accordingly.

FLOOR

Floor, as the controller, is responsible for linking all the elements of the game. It manages the lifecycle of the game and interact with the model and client. The class Floor is the basic setup and the core of the game. Floor contains a “map”, which is a two-dimension vector that contains a list of GridObjects’ shared pointers, on the current grid to keep track of all objects’ position. Map is a protected member in Floor since both player and enemy need to know where the items are and need to know the direction on the map when they take action during the game. In addition, TextDisplay is a private member in Floor class, which is a view model, when ever an action happens in the map during the game, map will update current stats through observer pattern (will be discussed later) to the display.

Some of Floor’s responsibilities include:

1. Setting up the “map” (grid).
2. Handling the randomness of spawning player, spawning enemies, spawning items and spawning cells.
3. Managing a uniform transition between levels.
4. Destroying all the objects and recreating new objects when restarting the game.

GRIDOBJECTS

The GridObjects represents the objects on the grid. It will have Player, Enemy, Item and Cell, they are all abstractly modelled by GridObjects. Each kind of races of player or enemy is implemented with a single subclass. Each subclass in Character must define its stats (MaxHP, HP, Attack, Defense). For instance, Shade set it stats to MaxHp(125), HP(125), Attack, Defense(25). Each subclass will know it's own HP, Attack and Defense. Therefore, this makes it easier to add more races to Player and Enemy. Similarly, Potion and Gold are the subclasses of Item since each kind of potion and each kind of gold has different effects to player. Items is implemented with Decorator Pattern. (See Design Section)

TEXTDISPLAY

The display of the game only has text display. TextDisplay class represents the console display of the game and it is implemented with the Observer Pattern (See Design Section). It contains a two-dimension vector of char to store each symbol in the provided file.

Some of TextDisplay's responsibilities include:

1. Client can virtually see the map on the screen by displaying the text map.
2. Notify observers and update itself when the actual map in GridObjects has changed.(eg. Spawn objects on the map and character's action during the game.)
3. Display player's current stat during the game.

DESIGN

1. COUPLING AND COHESION

We made a significant changes to our code to reduce coupling, but our design still need improvement.

- Signatures of loose coupling:
 - Modules communicate via function calls with basic parameter. (Yes, all methods in each class have just basic parameters.)
 - Modules pass arrays back and forth. (Yes, we pass an two-dimension to player and enemy since they both need a map.)(For reference, visit Character.h line 27.)
 - Modules share global data. (No global data are used.)
 - Module access to each other's implementation(Only used friend once in Level for displaying the game grid)(For reference, visit Level.h line 31.)
- Signatures of high cohesion
 - Elements manipulate start and over an object's lifetime.(Yes, open and read provided file)(For reference, visit Floor.cc line 351.)
 - Elements cooperate to preform exactly one task.(Yes, there are significant amounts of methods in Floor are working toward one gold, that is, initialization and generation)(For code reference , visit Floor.h.)

2. DESIGN PATTERN HIGHLIGHTS

We employ the following Design Pattern:

○ Observed Pattern

We can take the advantage of the Observer Pattern to implement text display and player's/enemy's move since objects on the grid and text display have an observer-subject relationship. Player is the main subject and text display are the subjects and observers of player. Once the player is created, all enemies will be attached to player's observer list. Similarly, text display will be attach to each object's observer list when an object is created.

- Used to notify enemies and text display when player's action is taken.

(For reference, visit Enemy.h)

```
void updateDamage(double damage);  
void notify(Subject &notifier) override;  
SubscriptionType getSubType() const override;  
GridObjectType getObjType() override;  
.
```

- Used to notify text display when objects is been slayed or picked up.
- We use the enum as type to distinguishes which class to notify. For instance, it will notify ALL(text display and enemies) when player's action is taken , it will notify display only when an item is picked up or when an enemy is killed.
-

○ Factory Method Pattern

In our implementation, we take the advantages of Factory Method Pattern to create grid objects and allowing us to hide information of the subclass to the client. Level is the abstract class that have all pure virtual methods of creating objects and placing objects on the grid. Used in the Level class (can think as an abstract factory class) and floor is the concrete factory to implements all the creation of objects. (For reference, visit Floor.h).

```
public:  
    virtual std::shared_ptr<Enemy> createEnemy(char *type) = 0;  
    virtual std::shared_ptr<Potion> createPotion()=0;  
    virtual std::shared_ptr<Cell> createCell(char type) =0;  
    virtual void placeEnemy(std::shared_ptr<Character> pc) = 0;  
    virtual void placePotion() = 0;  
    virtual void placeGold(std::shared_ptr<Character> pc) = 0;  
    virtual void placeStair() = 0;  
    virtual void placePlayer(std::shared_ptr<Character> pc) = 0;
```

- Visitor Pattern

The visitor pattern is helpful to construct polymorphism code. For instance, Player and Enemy are both inherited from the Character class, and when the attack method in the Enemy class is called, it will decide which method will be called in the subclass depending on the player's race. (For reference, visit Enemy.h)

```
virtual void attackIt(std::shared_ptr<Enemy> e, std::shared_ptr<Player>pc);
```

- Decorator Pattern

Decorator Pattern is very useful when attaching additional functionality to an object dynamically. Decorator pattern makes it easier when temporary potion's effects is constructed on to the player with the potion decorator and permanent effects and different kinds of gold type can just set it on to player stat. Once player reaches the next level, temporary potion effects can be removed by iterating through the decorator and eventually reach the bottom of the decorator, which is the player. (For reference, visit in PotionEffects class.)

- Template Function

We take the advantage of using Template function to reduce code rather than using the Template Method Pattern since potion and gold have similar code. (For reference, visit Floor.cc)

```
void placePlayer(std::shared_ptr<Character> pc) override,  
template <typename T>  
std::pair<int, int> spawnItem(T itemType, char type);
```

RESILIENCE TO CHANGE

Additional Characters (Student and Nomair)

Adding additional character races is not a significant change in our implementation. With Factory Design Pattern, we can just add another subclass for enemy/player and modify the method in the factory(Floor). Similarly, the attack action for new character race can be implemented with the Visitor Pattern by modifying the attack method in Enemy.h/Player.h

An example of the flexibility of Factory Pattern,

```
else{//Nomair
    spawnEnemy = make_shared<Nomair>();
    *type = 'N';
```

Nomair is created. (For reference, visit Floor.cc)

QUESTIONS:

Q1: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?

Since all races have a common make () method, we can use the factory method pattern to generate the races and let the factory to decide which races is needed to be created.

Q2: How does your system handle generating different enemies? Is it different from how you generate the player character? Why or Why not?

In our final UML, you can see that Character is a superclass of both Player class and Enemy class since they have common fields. The way to spawn player and enemy is quite similar. Player character is generated when player has selected a race in GameGrid. In comparison generating enemies, we spawn enemies in the Level class. By using the Factory Method Pattern, Level itself is an abstract factory and Floor is the concrete factory. Whenever the createEnemies() method is invoked in the Floor class to spawn, each created enemy is attached to the player's observers list.

Q3:

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

We can use the visitor pattern for double dispatching. Implement overloading methods for each different combination of attack effects for player. Special attack abilities of enemies and player are implemented as virtual function.

Q4:

The Decorator and Strategy pattern are possible candidates to model the effects of potion, so we do not need to explicitly track with which potions the player character has consumed on any particular floor. In your opinion, which pattern would work better? Explain detail, by weighing the advantages/disadvantages of the two pattern.

In our opinion, Decorator Pattern is the best fit for adding potion effects to the player. Decorator Pattern allows you to add functionality to your object. In this case adding temporary potion's effect to player. When a player uses the potion, potion can be a decorator and wrap it onto the player. When the player goes to the next floor, those temporary potions' effects are gone. We could just simply unwarp that potion. According to the definition of Strategy Pattern in Wikipedia, Strategy pattern defines a family of algorithms and encapsulates each algorithms and allow algorithms to be interchanged at runtime. The disadvantage of using Strategy Pattern is we might have to add extract fields to the potions and we might have to explicitly track the potions that the player consumed.

Q5:

How could you generate items so the generation of Treasure and Potion reuses as much code as possible? That is, How would you structure your system so that the generation of a potion and the generation of treasure does not duplicate code?

Before Due Date 1, you can see that we use the Template Method pattern for code reusability on generating treasures and potions since the Item class is made up of Potion class and Item class. We use the template function instead of template method. As you can see in the Floor Class Line 51. The spawnItem can spawn both potion and gold by passing the them in itemType.

```
51     template <typename T>
52     std::pair<int, int> spawnItem(T itemType, char type);
53     //returns the position of the item when item spawns
```


EXTRA CREDIT FEATURES:

WASD Control

We wanted our client to have best experience on CC3K. We added WASD control feature on this game. First of all, thanks to X.Li with the provided library(<http://hughm.cs.ukzn.ac.za/~murrellh/os/notes/ncurses.html>). With the WASD input, there is no need to type command followed by hitting an enter key. Client can enjoy the game with a real RPG experience. In addition, client can switch between legit control and WASD control by hitting the 'c' button. This feature allows us to switch control dynamically. You can visit our code in Main.cc.

Shared Pointers

CC3k is written without using explicitly memory allocation and deallocation.

AI for Enemy

Enemies now have the ability to pursue player with in the range of 3 block radius.

This AI algorithm is implemented in Player::move.

FINAL QUESTIONS:

Q1:

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 project is enjoyable and it taught us a lot, including good communication skill and how to collaborate with teammate . This project allows us to brain-storm the best solutions ,share ideas with teammate and have a good plan on how the project work before the implantation part start. Also, good planning and good comments on can really increase the pace of the project. Good comments can help teammate to understand the code when they want to use you code and good planning can help teammate to know what he/she is planned to do and due dates. Finally, this projects help us to build a high-trust relation between teammates. For instance, you can assume

your teammate's code are done correctly when your code require other teammate's code. If there something wrong with the code, debug and figure out together.

If we all worked alone, we probably won't able to make it on due date. This project requires a significant amount of details in coding. In comparison with debugging alone, debugging with teammates is more faster and more efficiency.

Q2:

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

If we had the chance to start over, we would put more time on designing then rushing in code development. We want our code to be as flexible as possible.