# CS 246 Assignment 5
# Final Design Documentation

ChamberCrawler3000

Fall 2015

Jiancheng Chen
Nian Liu

# Changes In Program Design

## Actual v.s. Plan

The final design UML is not much different from the formal plan UML. We add more methods and fields to some classes.

Note that in order to implement decorators for buffs that are added to `Player`, we change all methods of all superclasses of `Buff` class to virtual.

Also, for `Potion` class, we decided to remove its subclasses since we implement potion effects with `Player` decorators; thus we found subclasses are not as useful as we excepted, we can simply use a string to indicate a potion's effect and then apply to `Player`.

In `Game` class we add several `generate[something]()` methods to make initialization of every floor more clear and easier to manage. Also, for `Init()` methods, we overload it with a string parameter so that when we are loading a specific floor plan, it will generate floor according to the plan instead of generate random enemies and items. Similarly, in `Controller` class, we overload `newFloor()` methods with a string parameter for loading files.

In Enemy class, we add a protected bool field act_flag in order to prevent enemies do both attack and move in one turn.

# Project Approach

## Ideas and Designs

At the very start of the project, we found the structure of each floor is very similar to Assignment 4 question 2 Flood-it, that each floor contains [row * column] cells and MVC pattern is the best choice to deal with receive input, manipulate game data then output result. Therefore we came up with the first three classes: TextDisplay, Controller and Game, and Game class will have a field that to store every Cell in the floor. Then we realized that, we can store type of a cell in Cell class as a char so that when we implement movement, we can use the char to indicate if the Cell can be accessed. In addition, we planned to use a pointer to indicate whether the Cell is occupied by any object.

We classified that there are three major objects for this project: items( golds and potions), characters(player and enemies) and floor structures(stairs for now). Thus we decided to create an abstract Object class that contains three subclasses Item, Character (both abstract) and Stair(Concrete). and in Object class, we need something to indicate which Object it actually is, thus we add string filed type to store type information. What's more, we need to indicate where

the Object is located, so we add another pointer that points to the Cell it is on. Note that Object pointer in Cell can point to NULL if no Object is on the Cell, but Cell pointer in Object must pointer to a valid Cell.

We will describe detail design of every class in Classes Breakdown.

## Classes Breakdown

### Object: (object.h & object.cc)

Object Class is the superclass of all game elements, characters, items, stair. It has two fields that one indicates the char will displayed on the floor txt and the other one indicates the type of the Object, it can be a “player”, “enemy”, “stair” or a “item”. There is also a Cell pointer that to indicate the position of the Object.

### Character: (character.h & character.cc)

Character class is a abstract class that handles all game races including player races and enemy races since all races have HP(health point), Atk(Attack point) and Def(Defence point) and all game races are able to move (except Dragon) and attack.

Here are some important methods of Character class need to mention:

- Character(int Ark, int Def, int HP, string race): line 18*
  This constructor is suppose to called by it subclass Player and Enemy’s constructor to initialize a specific race with their race stat. string race will be use to indicate character race and apply special effect s.

- string move(int d): line 82
  This method is called when a Character needs to move. parameter int d indicates the direction of movement. Upon calling this method, we need to firstly covert direction which should be originally string to int (e.g. no -> 0, ne -> 6). Note that for Enemy, they only move 4 straight direction, no, we, so, ea corresponding int is 0, 1, 2, 3 and there are some restriction for Enemy, e.g. they can’t enter passageway and walk over golds. By checking Cell’s char and Object pointer to implement this feature. It returns a string that represents the movement message.

- string Damage(Character* defender): line 132
  This method is used for a Character to dest damage to another Character. We use this equation *dmg = ceil((100 / (100 + Def)) * Atk)* to calculate damage. Additionally, we also



*: all line numbers are referred to .cc files

apply attack effects here for example Vampire's heal, Orc's increased damage on Goblin. We also found out that we can actually override this method in each race class to simplify code and make it much easier to add new races with other attack effects in the future. it returns a string that represents the attack result message to Controller.

- void heal(int n): line 234
  This method was designed before the lecture of c++ casting. In Character class, it does nothing and it will be override in Player class. This method can actually be removed since we can cast the player Character* to Player* and call heal method from Player class.

## Player: (player.h & player.cc)

Player class is derived abstract class from Character class. Player class has addition field such as MAX_HP and gold which indicates player's maximum hp and golds held. It has 5 subclasses that specify each race of player character. We override heal method here that is used to increase player character's health point.

Here are some important methods of Player class need to mention:

- Player(…): line 14
  The constructor is supposed to be called by it's subclasses constructors. It will then call constructor for Character to fill fields of Character class.

- Player* getPlayer(): line 39
  This method is designed for the implementation of decorator. It simply return "this". we will discuss this in Buff section.

- void heal(int n): line 47
  We override heal(int n) method here. If HP exceeds the MAX_HP after healing, HP will be set to character's MAX_HP. Note that we will override it again in Vampire class since Vampire has no MAX_HP and drain health from enemy upon every successful attack.

Race subclasses of Player:
they are all similar that they initialize a player character with race start stats.

As we mentioned before, we can add more methods to implement race effects which will be virtual in their superclass such as attackEffect(), uponKillEffect(), turnEffect(), etc. to improve program efficiency and clarity and reduce code in some methods
e.g. Character::Damage.

## Buff: (buff.h & buff.cc & buffdeco.h && buffdeco.cc)

Buff class is an abstract class and its subclasses are implemented as decorators for buffs/ debuffs that effects player character's stats.
In Buff class, we have to re-implement all methods from it's superclasses. It is not complicated that we will simply call the same method on the Player pointer field player. Note that only difference is on the method getPlayer(): we will delete this then return player so that we are able to clean all buffs/debuffs when player enter a new floor or start a new game.

## Enemy: (enemy.h & enemy.cc)

Enemy class is designed for enemy characters. It is mostly similar to Player class and subclasses though a little bit simpler. However there are some methods and fields are complex.

- void dropGold(): line 32
  This method will be called when enemy character are killed. It will drop a random pile of golds on the cell the character stands before killed. Note that this method is override in Human, Merchant and Dragon classes.
  Upon death, Human will drop two normal hoard which worth 2 golds instead of one pile of random gold. We will drop one on the Cell human is on and the other one on a random neighbour or the human. If all neighbour cells are occupied, then the gold will not be generate.
  For Merchant, a 4-gold-hoard will drop instead of a random value gold, and Dragon don't drop gold since it will guard dragon hoard.

- bool Merchant::hostile: line 84
  This static field is default false so that Merchants are not hostile to player; however If player ever attacks any Merchant, all Merchants will be hostile to player. Thus we decided to create a static field so that we don't need to modify every Merchant's hostility.

- Dragon() and Dragon(DragonHoard* hoard): line 102
  Note that we have two constructor for Dragon. This is because Dragons are not count in the 20 enemies that every floor generates. Dragon will be created whenever a DragonHoard is constructed. When a DragonHoard is constructed, it will also call Dragon(DragonHoard* hoard) to assign the hoard to Dragon.
  The zero parameter constructor is used for load files since the floor plan is not random and it is possible that a Dragon is created before its hoard is create ( since we read in floor plan from left and top to right and bottom.

## Stair: (stair.h & stair.cc)

Stair class is very simple and have no any extra fields and methods from it's parent class Object. We just want to create a class to indicate stair which no enemy character can step on and when player character walks to a Stair, we will move to a new floor or display score( if player finish all 5 floors).

## Item: (item.h & item.cc)
## Potion: (potion.h & potion.cc)
## Treasure: (treasure.h & treasure.cc)

These three classes are created for game items: golds and potions. Potion and Treasure are derived from abstract class Item. We have discussed the Potion class at the very beginning. There are only one subclass, DragonHoard, of Treasure need some detail specification:

When a DragonHoard is created, we need to also create a Dragon that guards the hoard. In addition, DragonHoard can't be taken unless the guarding Dragon is killed. Thus DragonHoard class is a little more complicate then other classes.

In the DragonHoard constructor, we need to create a Dragon as well and let a field pointer points to the Dragon.

We also need a method to delete the Dragon when the Dragon is killed. ( deleteDragon() )

## Cell: (cell.h & cell.cc)

Each floor is actually a 2D array of 79 * 25 Cells. When we initialize the game, we also initialize every Cell according to the floor.txt. Thus every Cell has their own coordination and char that is the original symbol of a Cell. Also, the Object pointer will point to an Object if the Cell is occupied by any game Object, or points to NULL.

- void setObject(Object* obj) : line 71
  This method is used to modify the Cell occupant. When this method is called, it will set object filed to obj and also calls notify in older to modify the TextDisplay as well.

- void notify(int row, int column, char ch): line 102
  the notify method is used to notify TextDisplay to change the char on the coordination to the original Cell char or the occupant's char.

It is mostly called when setObject(…) is called since when object on a cell changes, it needs to change the TextDisplay.

- void setNeighbou() and Cell* neighbours[8]

  We decide to create this field and method in order to increase efficiency. For instance, when a Character needs to move to another cell or attack other Character, we don't need query the Cell from Game class, but getting it from neighbours. It is also useful to check enemies around the player character.

### Game: (game.h & game.cc)

Game class deals with the most computation. We create a 2D Cell array to store the floor Cell in game. We also create a enemy array which has length 20 so that during game in each floor, we can keep track on all enemy character's. We introduce 4 methods that are used to generate player, enemies, treasures, potions and stair in a game floor. As we mentioned before, there are two initFloor methods for new game and load floor data game.

In order to prevent memory leak, after players enter next floor, restart game or go to main menu, we go through every Cell in game and delete their object except player. That's why we have a player field so that we can store player pointer there. Also, we will call player->getPlayer() to remove all decorators that buff/debuff player character.

### Controller: (controller.h & controller.cc)

Controller class deals with player input and then manipulate module and finally modify the TextDisplay. We also create devMode so that we can use cheats to test our code. Additionally, method mainManu and printOldMan are used to display some ASCII UI and messenger to display texts.

## Design Specification Questions

**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?**

Create Character class with  two subclasses Player and Enemy to separate player characters and enemy characters. then create subclasses for both classes for races. It is not difficult to add additional races since we can simply create new subclasses with race data.

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

We decided to use random number. Each floor will generate 20 enemies that we will create a Enemy array with length 20 in Game to keep track of them. use rand() % 18 we will randomly generate number between 0 to 17 and then we can use simply if-statements to generate enemies with specific chance.
Generating enemies is pretty much similar to generate player character but with less fields, for example we don't need MAX_HP for enemies.

**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 decided to create virtual methods in Character Class and default implementation will be simply a return, however in specific Player subclasses and Enemy subclasses, we will override some those methods if the race has some abilities. It works for both player character races and enemy races

**The Decorator and Strategy [8] 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/dis- advantages of the two patterns.**

We decided to use decorator pattern for the effects of potion as we discussed in Classed Breakdown : Buff section. The disadvantage will be that we have to set all superclasses' methods virtual.

**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 Potions are both subclasses of Item class but with some different fields. For generating potions, we add field string type and for generating treasure, we add field int value. Note that for Dragon Hoard, we need an addition pointer field Dragon* dragon that points to Dragon : Enemy and addition method spawnDragon so that Dragon will be generated as Dragon Hoard is created.

## Conclusion

### Lessons Learnt:

This project is a great opportunity of practicing object-oriented programming skill and implementing lecture knowledge into real life application. It is important and necessary to have a detail plan and well-designed model before start coding. We have wasted some time on writing codes that we later realized are not useful.

### Improvements/ Changes:

We found that some parts of our code are not low coupling. Also, as we mentioned before, Damage method need a better design so that we don't need to squeeze all if-statements and race effects in one Damage method, we override the method in every race class instead. Moreover, the MVC pattern is not totally correctly used as well. We should let TextDisplay deal with all text out put and put less logic computation in Controller class.