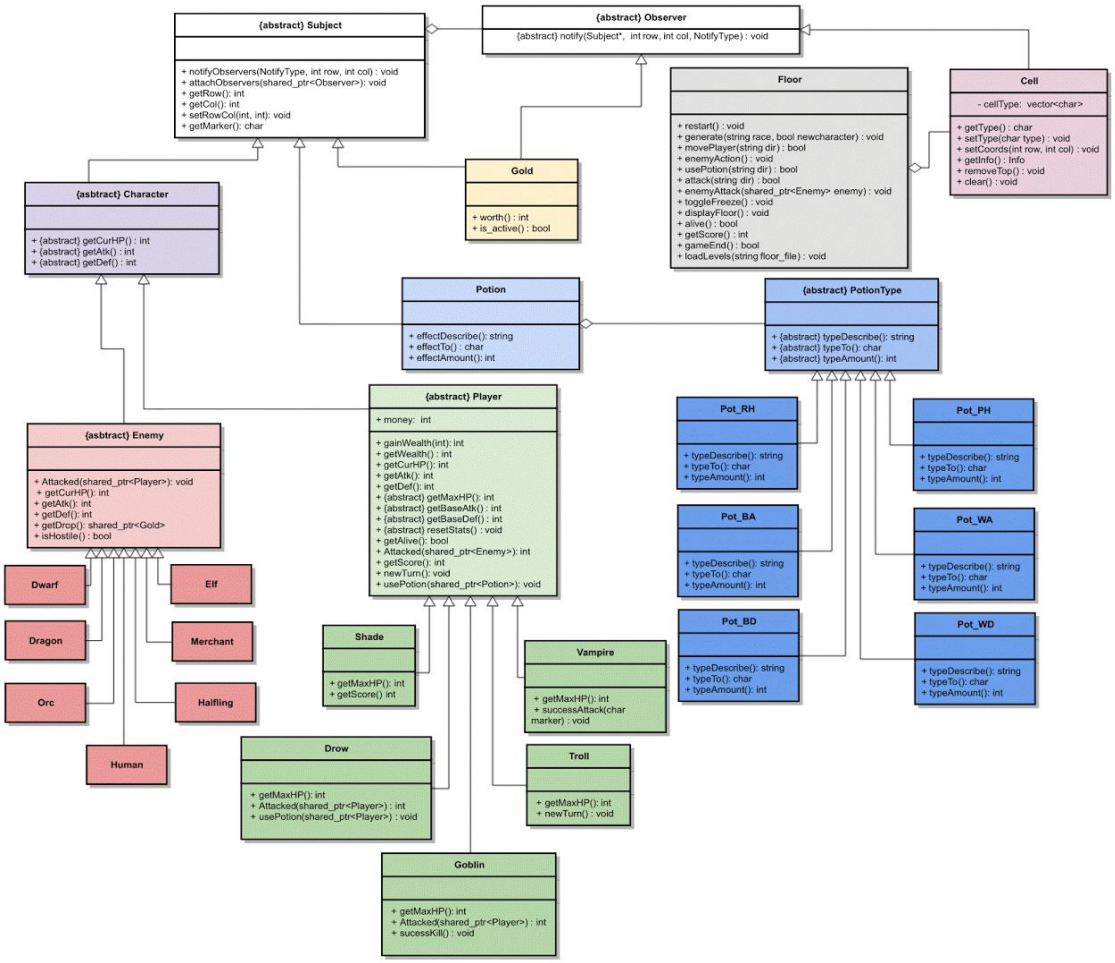
CC3K DESIGN DOCUMENT

-kkqzhou-msbukal-p35nguye-
CS 246 - Fall 2016

OVERVIEW

Kevin (kkqzhou)- Floor Implementation, Object Generation, Player Interaction
Milena (msbukal)- Abstract superclasses: Subject, Character, Player, Enemy and Potion along with all concrete subclasses and class Gold
Pascal (p35nguye)- Plan of Attack, UML Diagram, Keybinding DLC

UML DIAGRAM



DESIGN

The game is implemented through the Subject-Observer design pattern.

All of Player, Enemy, Potion and Gold are part of the Subject abstract base class as it makes notifying the floor for movement simpler. The shared code limits duplicate code and has limiting coupling. The floor is an observer to each instance of the Subjects and reacts accordingly.

Character: Player & Enemy

Potion and Enemy are both subclasses of the abstract superclass Character. This is useful as it limits the shared information of Player and Enemy but still allows them access to each other's protected and public methods as necessary.

Player and Enemy are also abstract superclasses. Player and Enemy contain the shared code of all the players and enemies, respectively. The methods are virtual therefore the subclasses can override them as necessary for special features specific to them. This limits duplicate code and as a result the race subclasses are highly coupled to their abstract superclasses. However, this is unavoidable and expected.

Player and Enemy are kept as separate from each other as possible to limit coupling. They only interact through accessor methods and sometimes protected character.h methods when necessary.

Potion

The potions are implemented through the Strategy design pattern. This has already been covered in another question, but we decided it was the best method for implementing potions as we only need one wrapper instead of possibly multiple such as offered by the decorator design pattern. The Potion class has a pointer to the abstract superclass PotionType, which defined the behaviour of the potions. The potions are adaptable, with new potions being simple to add through extra PotionType subclasses and potions being simple to change by just changing the global constant in the subclass .cc files.

Gold

Gold is implemented through a simple class structure as it is a simple concept. There is not much difference in between each gold type, so we decided an abstract base class with subclasses or other design patterns wouldn’t be necessary. To accommodate for the game

feature where dragon hoards are “locked” until its guardian dragon is slain, we have declared Gold as an Observer class of Dragon. This allows the dragon hoard associated with its guardian dragon to open automatically when the dragon is slain, and also allows for future adaptations where different or multiple enemies can guard a particular treasure.

Floor

Floor is main class through which the player commands are translated into in-game interactions. The Floor stores the playing board, all objects created for a particular level (player, enemies, gold, potions), the layout for all different levels, and the messages to be printed which informs the player of events that have occurred in the game (e.g. player attacks enemy).

The playing board is represented by an array of the class Cell, which is designed by a location coordinate (row, column) and holds an internal stack of characters. The stack system is implemented to handle the possibility of multiple objects occupying the same location, such as a player standing on top of a dragon hoard when the dragon is alive, and is handled in a last in, first out (LIFO) fashion. Thus the object at the top of the stack will be printed out to the display, while the next object is revealed after the top object disappears from the cell.

Player Interaction

The game interactions work by feeding player input from the command interpreter through the appropriate Floor member functions. For example, the member functions Floor::movePlayer, Floor::attack, and Floor::usePotion takes the player input and performs the necessary actions to the player, enemies, potions, and gold in game. Information displaying is also done via Floor member functions rather than in the command interpreter, as this keeps the interpreter less cluttered and also allows us to easily modify messages if fields or classes are added or modified.

RESILIENCE TO CHANGE

The player, enemy, potion and gold classes are easy to change or add too. Most of the important shared code is done in player.h and enemy.h so most of the subclasses will be affected and will not have to be changed if say, for example, the mechanics of attacking are changed. Only subclasses where the attack methods have been overridden would have to be changed.

The add races to the player and enemies, they just require and extra race subclass which is simple. In fact, we easily added an extra player class (the Sheep) in less than five minutes and designed our own stats and special effects.

Modifying potion and gold is also simple as most of the code is shared and would just require initializing methods to be changed. To add to the potions would be straightforward by just adding an extra potion type and then adding an extra case in the potion constructor. To add to the gold just requires an extra case in the gold constructor with the proper value. Changing the specific amount that a potion effects a stat is also simple as only the global constants in the .cc files would need to be changed.

Since the playing board is implemented through an array of Cells, any structural game changes related to the playing board could easily be accommodated for by adding the desired properties in the Cell class. Also, changes in game behaviour can also be easily accommodated for since each action has its only Floor member function, so these can be called in any order and follow any logic. For example, if we wanted the monster to attack before the player, we would simply call Floor::enemyAction before Floor::attack.

ORIGINAL QUESTIONS

Q: 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?

All the player races are subclasses of an abstract base class Player. Therefore, to generate the player according to user input, you only need to create a Player pointer to the correct subclass. This is useful and makes generation easy because it makes the rest of the code reusable as all operations are performed on a Player pointer. Adding additional races is simple as it just requires an extra case in user input to create the proper object and since the common methods are defined in Player, writing the subclass is also simple as only special effects would require overriding the necessary methods.

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

Generating enemies is similar to player as they follow the same design pattern, and just require creating pointers to the abstract base class Enemy according to the required enemy subclass race. Player character generation differs from the enemy generation process only in that the player picks the starting race (so it's not random). The process of picking a random room and a random floor tile in that room remains the same.

The different types enemies are generated by first passing a list of names and their probabilities through a random number generator and then creating the enemy associated with the random

name. Then a random room is picked, and a random floor tile in that room is picked to spawn the monster on.

Q: 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?

Since the generation of Treasure and Potions follows the same process - pick a random name, a random room, and then a random tile in that room, we could actually generate a Treasure and a Potion simultaneously in each iteration to avoid duplicating code. In this case, we would just pick two random names, two random rooms, and then two random tiles associated with each of the rooms, and then simply assign assign the Treasure to the first and the Potion to the second.

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

Both the player and enemy special effects were implemented similarly. This was done by overriding virtual methods in the abstract superclass (Enemy.h and Player.h) to behave differently for certain subclasses. For example, the halfling when attacked has a 50% chance that the player will miss, and in Halfling.h and Halfling.cc the method Enemy::Attacked(shared_ptr<Player>) is overriden to take this into account.

Q: Decorator and Strategy are good patterns for the potion effects so that we do not need to explicitly track which potions the player has consumed. In your opinion, which pattern would work better? Explain in detail by weighing advantages/disadvantages.

We decided to use the Strategy design pattern as the decorator is more useful if you are required to wrap multiple objects within each other. However, the potions only would need one level of wrapping for the stat they affect and the amount they affect. Therefore, a decorator design pattern would open up to confusion if a potion was wrapped twice, leading to undefined behaviour. Therefore, for simplicity and understandability, we decided to use the Strategy design pattern as we only need one level of wrapping.

EXTRA CREDIT FEATURES

SHEEP DLC:
An extra player race was added as a fun addition. The Sheep race, with 250 HP, 20 ATK and 100 DEF. The sheep is immune to negative potions. Whenever the sheep successfully kills something, it gains 5 attack and 1 defence for only the floor.
To use the sheep character, select the 'h' option as the player race at the beginning of the game.

KEYBIND DLC:
We decided that typing in commands for movement and actions was far too inconvenient for users. As such, we decided to implement keyboard controls using getch, removing the need to press "enter" every time the user wants to perform an action in-game. Looking into various methods for keybinding, we initially opted to use the ncurses library; however, this proved to be incompatible with our method of rendering the map using cout. As such, we used a custom getch function using getchar and termios to turn echo off.

The commands that change are:

Command	**Key**
Start Game/Restart	Enter
Exit Game	Esc
Freeze Monsters (for testing)	`
Move (no, ne, ea, se, so, sw, we, nw)	W, E, D, C, X, Z, A, Q
Attack	O + [direction]
Use Potion	P + [direction]
Choose player race (s, d, v, g, t, h = Sheep)	1, 2, 3, 4, 5, 6

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?

We learned that coordinating with a team is harder than expected. Certain parts of the program need to be completed for other parts to work, which lead to a lot of confusion and

waiting for other members to complete the work. Milena, who wrote the code for Player, Enemy, Potion, and Gold, was unable to test her code without the running game while the others needed the code to actually run the board. It showed to us the importance of low coupling. We also saw how abstraction is useful to make everything simpler to understand as one member didn't need to know how the Enemy and Player characters were implemented to use them to run the game.

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

We would have planned better with each other we didn't think how to write our code through well enough and therefore had to do a lot of last minute changes. The members who had the base code to work would have started earlier, giving the members writing the floor and command interpreter more time to accommodate to changes and to work with the final result. This would have also given us more time for testing.

CONCLUSION

This assignment was a challenging yet fun experience to actually implement a full game, where our motive was clear throughout the assignment as it was more personal and we were working towards a larger goal compared to short assignments. We managed to successfully apply design concepts and styles that were taught in class by ourselves.