Fall 2015 CS 246 CC3K Plan of attack

• Implement the Board class. Read from a map asset file containing the empty board and store this into a 2D array of characters that will represent the board	Thursday, November 19
• Create an abstract Character class and create Player (abstract) and Enemy (abstract) classes that inherit from it	Friday, November 20
• Create two character races (inherits from Player) and two enemy races (inherits from Enemy) and set up visitor design patter between them (for attacking).	Friday, November 20
• Create the other character and enemy races and set up the visitor design pattern	Friday, November 20
• Set up a Caption class that will keep track and print out the player's character race, hp, def, atk and action	Saturday, November 21
• Set up attacking between all characters and enemies. Calculate the damage accordingly.	Saturday, November 21
• Update the caption when a user is attacked	Saturday, November 21
• Create an abstract Item class. Create Treasure, RestoreHealth, PoisonHealth, BoostAtk, BoostDef, WoundAtk, WoundDef classes that inherit from Item	Sunday, November 22
• Set up visitor design pattern between two of the items (e.g., treasure) and the player character	Sunday, November 22

• Set up the visitor design pattern between the remaining items and player character	Sunday, November 22
• Update the player’s stats according to the effect of the item picked up. Update the caption when an item is picked up.	Sunday, November 22
• Generate the Player character, Items and Enemies randomly and place them onto the board	Monday, November 22
• Add optional command line argument to read board from a file	Thursday, November 26
• Read movement commands for Player character from stdin	Thursday, November 26
• Read attack commands for Player character from stdin	Thursday, November 26
• Read potion usage commands for Player character from stdin	Thursday, November 26
• Read Player race selection from stdin	Monday, November 30
• Allow user to restart or quit the game	Monday, November 30
• Allow user to go up a floor/level	Monday, November 30
• Go back and implement anything missing	Tuesday, December 1
• Implement bonus features	Tuesday, December 1

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?

I will create an abstract Player class and have the different races inherit from Player. This will make adding additional races easy because most methods and fields will be inherited from Player and the race (child class) can just implement or overload as needed.

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

My system generates enemies and players in the same way. There is an abstract Player class and an abstract Enemy class that both inherit from their abstract parent class Character. The Character class contains methods and fields common to Player characters and enemy characters. I chose to do it this way because enemies and player's have a lot of fields and methods in common (e.g., health, atk, def, makeMove(), etc.)

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.

I have generic attacks between players and enemies. If a player or enemy have a 'special' ability against a specific race/type, I overload their attack functions or makeMove() as required.

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/dis- advantages of the two patterns.

I decided to use the Visitor design pattern. I had a generic Item (abstract) class and created the Potion objects and Treasure objects that inherit from Item. The abstract Item class had a pure virtual pickUp(Player &player) function that the subclasses implemented. In these methods they would call pickUpItem on player and the effects of the Item will modify the player accordingly. I believe the Visitor design pattern is superior to the strategy pattern because the item can tell the player that it needs to update one of its stats. Since I decided the player and items should be a part of the Model, I think that they should be able to "talk" to each other. I think that it is better than the decorator pattern because it seems like a lot of structuring for not very much result. I thought that it was over complicating the Item Hierarchy I designed.

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?

I made an Item parent class that has the child classes Treasure and all the potions so I can have one method that generates all the items whether it's a potion or treasure. This method will be called twice. It will take in a list of Items, choose the item to create by random (it will create it by making a copy of the Item) and placing it on the board. The method will be called twice – once with a list of Treasure and once with a list of Potions since 10 of each must be created.