CC3K
Plan of Attack

Program Design:

1. Implementation of Cell Class:

- Cell: An abstract parent class; it has a string field of its type for printing purpose; it has a vector field that records all its neighbour Cells; every Cell is an Observer to its neighbour Cells.

- Other: Subclass of the Cell Class; players can step on Other Cell but enemies cannot be generated on Other Cell and cannot be decorated.

- Tile: Subclass of the Cell Class; players can step on Tile Cell that is not occupied by any other items or characters; enemies can be generated on Tile Cell, and can be decorated.

- DecoratorTile: Subclass of Cell Class; Decorator; decorate the Tile Cell which is chosen to have an item or enemy; abstract parent class.
    - ❏ Potion: Decorating the Tile Cell with a Potion that can be used by Player; has six subclasses and each of which represent one type of Potion with its own implementation.
    - ❏ Gold: Decorating the TIle Cell with a Gold that can be picked up by Player.

2. Construction of Floor:

We decided to model a floor as an object that is made up of Chambers, paths and doors. Inside a floor object, chambers are modelled using a vector of a Chamber pointers. Paths and doors are modelled using a vector of vector of Other pointers. The objects pointed to will be allocated on the heap. And accordingly, they will be deleted when the floor goes out of scope.

- Chambers:
  - ❏ Chambers in a floor are modelled using a vector of Chamber pointers. A Chamber will also act as a factory to generate enemies and items within it.

- Paths and Doors(vector of Other *):
  - ❏ Paths and doors are modelled using a vector of vector of heap allocated Other Cells. They are different from the cells in chamber since only a player can move onto an Other Cell.

- Items:
  - ❏ Items in a chamber are modelled as decorated Cells within that chamber.

3. Designing Characters:

- Player:
  - ❏ A Player is one of Shade, Drow, Vampire, Troll, and Goblin. A player's stats will be determined by what race is chosen.
  - ❏ A Player will be allowed to move and attack various enemies in the floor. A Player also keeps track of how much gold s/he is carrying.

- Enemies:
  - ❏ An Enemy is one of human, dwarf, elf, orcs, merchant, dragon and halfling.
  - ❏ An Enemy, with the exception of dragon, is allowed to move within the chamber it has spawned in
  - ❏ An Enemy, with the exception of merchants, will attack a player if a player passes within a one block radius of it. Merchants, if attacked, will become hostile to the player, for the remainder of the game.

4. Display:

- TextDisplay: The TextDisplay Class is used to print the current appearance of the current Floor and the status of the Player; it is the Observer of every Cell in the current Floor.

5. Driver Program:

The main driver program would create and mostly interact with a Floor object. Since any functionality implemented in a class is related to floor, the main function would only need to call methods of a Floor object, with appropriate information. The Floor will then call appropriate methods from its fields.

Division of work:

Character Classes: Luke
Cell Classes: Adil
Floor and TextDisplay: Yueyao
Driver: Collaborated

Expected completion dates:

- Concrete Character Classes: 18 July
- TextDisplay and concrete Cell Classes: 20 July
- Floor and Chamber Classes: 22 July
- Driver Program: 23 July
- Final Testing: 24 July

Answers to questions:

- **Q)** 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?
- **A:** We have an abstract class called Character, and it has subclass Player. Player is also an abstract class, each race is a subclass of Player. So, when the user choose the race that they want to be, we just generate the specific subclass of the chosen type using its constructor. And the different functionalities of that specific race were already implemented and are ready to function. Since one race is just one subclass of Player, adding additional races are actually adding subclasses to Player. The only thing we need to do is implementing methods that are unique to the new race and adding this race as an option to user; nothing else need to be changed.

- **Q)** How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?
- **A:** Generation of enemies is specific to chambers. Each chamber will have a method to generate enemies and place them inside of it, using Factory Pattern. Enemies inside a chamber will be destroyed once a player progresses to the next floor. Since we need to preserve the player, we have to generate a player in our driver program. This will make sure it is not deleted when a floor goes out of scope.

- **Q)** How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.
- **A:** We decided the way of implementing enemy characters' abilities based on how Player and Enemy are going to interact. We thought about using the visitor pattern to model any interaction between a player and an enemy but since we have 5 classes for players and 7 potential enemy classes, we would need to write too many methods. Thus, we decided to implement the various abilities for the enemy characters as the same as for the different abilities of Player and make a double dispatch. Both Enemy and Player inherit from abstract Character class and we implemented the different functionality of each character in that character's own attack method, which calculates the damage result in the combat between two characters, which is more feasible and reduce the amount of repeated code.

- **Q)** 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/disadvantages of the two patterns.
- **A:** The decorator pattern makes more sense for our game since, based on if we are decorating with gold or a potion, we are adding functionality on top of a cell that isn't decorated. During our design, we had already put checks in place to make sure there is no possibility for a second decoration. We preferred the decorator pattern since it makes intuitive sense to "add" functionality to a cell. Furthermore, the added functionality is similar. We would use a Strategy pattern if we had very different implementations of what a Cell would do if there was a different type of potion on it

- **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?

- **A:** We used Decorator Pattern for the generation of Treasure and Potions. The DecoratorTile subclass is added to the Cell class. Both Treasure class and Potion class are the subclass of the DecoratorTile Class. Thus, when generating, we first decide which one is going to be generated. Then, we use the random generation function that gives us a number that helps us choose a Cell. The last thing we do is wrap the original Cell with the matching decorator type. Hence, once we know the type of generation, the following process is exactly the same for both of the type. And they will share the same code for generation.