Overview:

Monday/Tuesday:
Meet up to create framework (classes, non-design pattern fields and methods)

Wednesday:
While we work on the Characters and Items we will also start working on the Decorator Design pattern to get the potion features working and have them be able to place effects on the playable characters

Thursday:
Once classes and the abstract classes have all been created and implemented we will start working on the Map class which handles a bulk of keeping track of each item, enemy and player on the map. The init() function on the map is what initializes the map and spawns in the enemies, items and characters so we will work on it first.

Friday and Weekend:
Then, once the map implementation we will build the Observer Design Pattern since the Observer Design Pattern handles all of the change of states of the Merchants and having characters. Lastly, the display will be worked on and we will try to get the game fully running with all functionality.

Any remaining time will be spent adding DLC, debugging, and preparing our demo to present to the TA.

The work will be split in the following way:

- Catherine:Map
- Vincent:Item
- Jethro:Character

Design:

Implementing the Playable Characters and Enemies:

In our cc3k+ project we implement an abstract Character class to encapsulate the shared attributes between playable and non-playable characters.

These attributes are mainly, health, attack and defense, which we can consider as parameters for a function in which each character is an element defined by these properties.

Other than the shared attributes between characters we also provide 3 methods which every character has in common which are the move(), attack(), and getInfo(). The first two dictate actions which all characters can do, and the getInfo() class is used by the TextDisplay to determine what char to display on the screen.

Note that we have made the Character, PC, and Enemy abstract base classes as we should only be able to create instances of its concrete subclasses, which have different behaviours.

- Common data fields among characters:
  > Health, Attack, Defense
- Common methods among characters:
  > move(), attack()

Within the PC class two new methods are defined, these methods are :

  - useLoot(Loot l)
  - IncGold(Gold g).

Both of the newly defined functions do not return any value; they simply mutate the playable characters' base stats or gold count depending on which loot or how much gold was used or picked up.

There are no new methods within the Enemy class but two new attributes are defined for them. These attributes are

- Bool wasMoved
- Bool hasCompass

These two new attributes allow for the map to see which enemies have been previously moved to account for enemies which may have moved to an unprocessed row of the map. This will ensure that each enemy moves once per turn

The hasCompass attribute simply gives us the information whether an instance of an Enemy has a compass which it should drop upon its death or not.

Once we have split the Character class into 2 smaller classes we will also implement each respective character with their own class that inherits from the PC and Enemy class depending on their character type.

The classes that inherit from the PC class are:

- Human
- Elf
- Dwarf
- Orc

Each PC daughter class will implement their own version of incGold(Gold g) and useLoot(Loot l) since some characters have special abilities such as the Dwarf's double gold or the Elf's positive potion effects even when affected by a negative potion.

Having the PC class holding the virtual methods allows for unique implementations within each respective race of the playable characters.

Now, for the Enemy classes we have the following

- Werewolf
- Vampire
- Goblin
- Dragon
- Merchant
- Troll
- Phoenix

Each enemy except the dragon and merchant only differs in terms of the value of the three attributes health, attack and defence, the two remaining classes dragon and merchant have special behaviours.

The added attribute in the Merchant class is:

- Bool isHostile

This attribute keeps track of whether or not the player attacked a merchant which would turn all subsequent merchants across the dungeon hostile towards the player.

The only unique feature of the dragon enemy class is that it is completely stationary, it spawns near a dragon's hoard and it also has a chance of dropping the Barrier Suit item this is represented by the attribute:

- Bool hasBS

We have also implemented an Observer Design Pattern to keep track of what the player does, for instance, enemies will check adjacent cells on the map for enemies and notify the map accordingly. It also keeps track of the text display, which is updated based on the GameObject which inherits from the subject class. This means that the PC is a subject of TextDisplay, while Enemy is both a subject of TextDisplay and an observer of PC, to handle hostility when PC is in a 1-block radius of an enemy.

Implementing the Items and Loot:

In our implementation of the items and loot within the game we create an abstract class Items that encapsulates the shared attributes between items such as random generation and their usage for playable characters.

We then break down Items into three main categories which are as follows:

- Loot
- PermPotion
- TempPotion

We have decided to split the potions since the health potions have permanent effects while the remaining 4 potions change the attack and defensive capabilities of the playable character class but these effects only last through the floor that they have been activated on and are removed after the player finds the staircase and leaves that current floor. For temporary potions, the Decorator design pattern has been implemented to enable potion effects to wrap arround the player class, modifying its stats accessor functions.

The Loot class adds a new method which allows the player to pick up the loot from the ground by walking over it:

- onPickup()

The Loot class it then broken up into the following classes that inherit from Loot:

- Gold
- BarrierSuit
- Compass

Each item has in common the fact that they are to be picked up by the player by being walked on and then it enters the players inventory.

The gold item represents the hoards of gold that can be found throughout the dungeon. There are 4 different values of gold hoards which are 1, 2, 4 and 6. The Gold class passes on the value attribute to the following classes:

- DHoard
- MHoard
- Hoard

The DHoard class is the class for the dragon hoard which when taken has a value of 6 gold. It is only available to be taken by the player once the dragon guarding it has died. A new attribute added on the DHoard class is:

- Bool isDead

This added attribute checks if the dragon is dead or alive, if it is still alive the value is false and true otherwise. It also has a one to one relationship with the dragon class since each dragon hoard instance always comes with a dragon

The MHoard class has a one to one relationship with the merchant class since each merchant drops an MHoard on death.

The Barrier Suit allows the player to take only half damage from enemy attacks. The Barrier Suit is dropped by one dragon throughout the whole game's runtime and once collected the Observer class activates the half damage output from enemies.

Lastly, the Compass simply toggles the display to allow the player to view the staircase within the floor they are currently in.

For the implementation of the potions by separating them into two categories of permanent and temporary effects we were able to create a Decorator Design Pattern class to keep track of the temporary potion effects.

The classes that inherit from the PermPotion class are:

- RH
- PH

These potions once within a 1 block radius will affect the player's health by either adding or subtracting once again this implementation is handled by Observer Design Pattern. The Observer checks if the player is within a 1 block radius and applies the according effect on them.

Now for the TempPotion class here are the following classes that inherit from it:

- BA
- WA
- BD
- WD

Now here the effects are applied using a Decorator Design Pattern so that it can be properly toggled off once the player has left the current floor.
The Decorator Design Pattern has the following methods that are passed onto TempPotion and PC class:

- getAtk()

- getDef()

It uses these functions to see the current Attack and Defense the player has and uses the TempPotion classes to apply the corresponding temporary effect for the remainder of the floor.

Implementing the Map:

The Map handles all the current positions of all the items, npc, and playable characters. We will initialize the map as a vector of vectors containing characters. The Map class will have the following methods:

- tick()
- init()
- quit()
- reset()

The tick() method is used to ensure all enemies move one step for each step the character takes and it will move the enemies in a row by row check as mentioned within the project guidelines.

The init() method will initialize the map, generate the 2D vector of GameObjects, and set everything in place for each floor. This includes the generation of items, enemies, and stairs.

The quit() method allows for the player to leave the game before it ends and ensures the game ends as well.

Lastly, the reset() method will reset the game's map and randomize all items and enemies once again once the game either ended, the player died or the player quit and replayed.

The Map class has a one to one relationship with the Game Object which is the class that handles the running of the dungeon crawler itself hence, having one instance of the game will create one instance of the displayed map.

The Map also has a one to one relationship with the TextDisplay since it is called each time the TextDisplay needs to output the current state of the game.

Implementing the display:

We implemented the display very similarly to how it was done in A4 for Conways Game of Life. Our TextDisplay has the following data fields:

-theDisplay
-Atk,Hp,Def,Race,Action
-displayStairs

-potionSeen

We created a class TextDisplay which holds a 2-D vector of Chars which each corresponds to a GameObject on the map. Since theDisplay is an observer of each game object, if anything is modified the map will notify the TextDisplay and trigger a change in the vector. For the HUD elements, since the map contains the PC, all information such as player stats and gold can be forwarded to TextDisplay through an accessor function. The displayStairs field is a boolean value dependent on whether the PC has the compass, and can see the stairs or not. Finally, the potionSeen field is a vector of Booleans to indicate which potions have been seen and which have not to determine whether their effects should be broadcasted to the player

----

Resilience To Change:

Using the Decorator Design Pattern makes it easy to expand the game without having to dig through and rewrite a bunch of existing code. For example, if we want to add new Potions with different effects, we can just "decorate" existing potion classes with new behaviors instead of rewriting the whole thing. This keeps things clean and modular, and it follows good design principles. Basically our code is open to extension but closed to modification.

Breaking the game down into smaller, focused classes also makes it a lot easier to manage. When each class handles a specific responsibility like enemies, player races, or items it becomes much more straightforward to add new content. Let's say we want to introduce a new enemy type or a playable race; we can just create new classes for them without having to worry about accidentally breaking something elsewhere.

We also make use of the Observer Design Pattern, which helps the game world react to the player's actions in a smooth and consistent way. For instance, when the player moves or changes state, other parts of the game like enemies or the UI can "observe" those changes and update themselves accordingly. This keeps everything connected without tightly coupling different parts of the system, which is super useful when you start adding new features.

Finally, by keeping the Character and Item classes separate and well organized, we make it much easier to add new characters, weapons, or gear in the future. It's all about building a solid foundation now so we can keep expanding the game without getting stuck in a web of messy code.

Questions In Document:

Question. How could you design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?

We designed our system so that each race is a concrete subclass of the PC ABC. The PC ABC has 2 related virtual methods that will handle the unique skill associated with each class. While the constructor of each concrete subclass will define the pre-determined stats of each class such as Atk, HP, and Def, the useLoot(Item I) and the incGold(Gold g) methods, will be implemented differently depending on the class's unique perk.

For example: the dwarf class will implement incGold by increasing player gold by 2 * g, and the orc class will increase that same gold pile by ½ g. This can be done by overriding the default useLoot() and incGold() functions.

By Implementing our races with this inheritance hierarchy, we ensure that adding additional races with special skills related to the gold and loot, or different starting stats is as simple as creating a new concrete subclass of PC, with its own implementation of a particular virtual method(s).

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

how to generate a player: first randomly decide which chamber, then randomly decide which floor tile in that chamber

- don't have to check for stairs since stairs spawn after character
- don't have to check for any other objects since pc is the first thing to be spawned

For enemies we have to consider what type of enemy is being spawned, each type has a different probability. In total, we generate 20 enemies

- before enemies, we spawn gold
    - every time a dragon's hoard is generated, spawn a dragon and decrease number of enemies that still need to be spawned
- to spawn non-dragons:
    - randomly select chamber
        - check if the chamber is full (is it possible for a chamber to be full?)
            - if so then can't spawn enemy in that chamber
    - randomly select floor tile within the chamber
    - check if anything else is already on that floor tile
        - if not, randomly decide what type of enemy to spawn based on given probabilities
        - if so, continue randomly selecting floor tile until we get something that's empty
    - continue spawning until we have 20 enemies on the floor
- overall there are a variety of differences due to the order of being spawned, different types of enemies, number of enemies, etc.

Question. How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?

Using the same design concept as the PC, enemies could also have special abilities by overriding certain methods of the Character class such as attack() and move(). Since each enemy has a concrete subclass, the attack() method could also add health to a vampire's own HP field during the attack() method, and similarly subtract gold from the PC for a goblin.

Question. What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?
decorator design pattern -> use to add/subtract from the atk/def of the pc. We implemented this design pattern into our project to mitigate the need for tracking each potion on any given floor. Since our temporary potions inherit from the Decorator, different statuses could also be added at run-time such as adding/subtracting gold, or any combination of Atk, Def, and health modifiers.

Question. How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? That is for example, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code? How could you reuse the code used to protect both dragon hoards and the Barrier Suit?

We could use the factory design pattern so that the generation of each item is taken care of by a “virtual constructor” which would return a unique pointer of type Item. This generation would be dependent on an enumerated class which identifies what kind of item is to be generated. However, this would conflict with our implementation since we have a GameObject class which Item and Character both inherit from, which is the type stored inside of our Map class. Thus, this would have to also define a virtual create method for all of our PCs and enemies, which would require an entire restructuring of our class design.

<u>Final Questions:</u>

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?

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