Plan of Attack: CC3K

Carter A, Ian G, Lucas M

Definitions:

Player Status: The player status is the collection of the players location on any given floor, the floor number that the player is on, HP, DEF and ATK and win/lose.

Turn: A turn is the process of the client interacting with the game through a move, attack or use of an item.

Tile: A Tile is a cell in the playable region for the game as defined in the project description.

Entity: An entity is a Character, Item, or a tile that the player can interact with. This includes the staircase found in some chamber on a given floor.

Design Questions:

Q1: How could you design your system so that each race could easily generated? How difficult does said solution make adding additional races?

We have designed our system such that all objects that have a result in the players status changing in some way are an Entity. Entities are an abstract object that a given Level object uses to manage the interaction of Entities with each other and individual Tiles. Entities themselves contain some information about the Entity itself:

- The location of said Entity on the floor
- The type of Entity, either a Character or an Item

Characters inherit from Entities. Character is implemented as a class for players and NPCs. A Character can move, attack, or interact with an item. Characters know their own race, HP, DEF and ATK, and ACC stats. ACC is an accuracy stat. A player has a default accuracy of 100.

Each Race inherits from a Character. Races take advantage of polymorphism allowing them to have unique attack and effect abilities applied to them without too much extra work. Using this method one simply writes a class to contain the overridden attack or potion functions. With this approach all the stats and abilities of a given can very easily be added to the game.

Adding more races is done through creating a class for it which inherits from a Character. The new race can override whatever methods are necessary in order to implement that races unique traits or abilities. Another modification would be required in the process of generating the entities for a given floor.

The player's update method will be "modular," so the standardized parts can be distributed through all races, whereas the differing can be specialized to a small virtual method, called dynamically at runtime depending on race.

Q2: How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

The same technique is applied to generating the enemies as is the character. Enemies share the same attributes and members of a player, because they both inherit from character. The difference is movement which can be overridden as part of that enemies move function in its class definition. Since enemies are randomly generated they are placed chamber by chamber according the their associated probabilities.

Dragons are only spawned around hoards, during the enemy random generation step in the Level class. Note that since hoards can only be picked up after a dragon's death, the hoard will have a shared_ptr to its specific dragon.

An enemies position will be determined by the Level class, during floor generation, randomly.

Q3: How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

All races have their own class. As such an enemy can have a non-default attack method taking advantage of polymorphism. For example the elf gets two attacks against every race except drow. This can be implemented be overriding the characters attack method to attempt to attack 2x instead of once.

Q4: 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?

We will take advantage of the ability to cast a normal potion to a knownPotion. This way we will know if the potion has been or not by deducing its type.

Q5. 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?

This is light work. Since all of these Objects are Entities we can just generate the entities as needed. Specifically in order of gold->potion->enemy because dragons need to have a gold hoard in order to spawn in the first place. Taking advantage of different input types. We always generate a position first then

give the entity attributes that are associated with said entity. Essentially we generate all the similar things to do with a specific entity then we look at specifics and apply specifics to that object.

Discussion of Play Area design concepts:

Floors:

Floors require 2 key implementation features. The first being the option to read in a floor from a file and the second being a default floor. This requirement must be handled by our level class. When a level is constructed from a file, it denotes the locations of enemies and potions. If its default constructed then entities will be randomly generated on the default floor. If a file is provided the level will construct a grid and place entities wherever necessary according the given file.

Bounds detection will be done through querying neighbouring Tiles for the type that they have been assigned. A Tile can be of multiple types: Floor, Stair, Hallway, Wall, Void, Door. An entity that moves (Character) can move to Tile types that aren't void or wall. Enemies cannot enter the Stair, Hallway, or Door Tiles. This avoids enemies entering a different chamber by randomly moving. When generating Entity locations the tile can be queried for its Type along with other data to determine if an enemy can be spawned in said location.

User Input:

One may want to play the game (or not). It is important to account for such necessities. A client can input an action for the game (use item, move, attack) or they can apply other commands like quit. All of these things will be handled cleanly by the Game class. The Game class will handle the general inputs that actual actions will be handled by the level.

In Account of Unexpected Change:

There are many ways in which the game could require changes from simple changes like adding a new Race to requiring a complete resize of the game area. The design should account for such unexpected changes. As discussed above, it is quite simple to add a new race.

Combat Changes:

The current combat system is easily modifiable to an extent, default combat can be adjust by simply adjusting the values in the update method within the character class. For a complete rework it would require the individual classes for each race being modified. If different stats were added for accomplishing feats or collecting an item, then adding them could be achieved simply by modifying a single value in the character class.

Map Changes:

The map can be made to be different sizes without much effort since we are applying vectors to save and generate it. Additionally, the movement mechanics are based on what the tiles around a current tile are, so any adjustments made to an entity are relative to the tiles around the current tile. Adding additional tile types such as traps or chests would be done by adding another type to the cells above and adding the interaction code to character in general.

Dropped Items:

In the default version of the game dropped items exist internally but not on the game board itself. If the rules where to require they exist as entities on map then it would follow that the game need only generate some entities (dropped items) within a specific bound, which is no different than generating an enemy in a specific chamber.

Overall:

The design takes advantage of the subject-observer design pattern and effective OOP practices which allow changes to implemented quickly an effectively.

Completion Goals:

This is an estimation and may not reflect actual completion times.

Monday July 21: Interfaces complete
Wednesday July 23: The game runs
Friday July 25: DLC and and report done

Estimated Coverage Of Work

Ian -> Entity, User Input

Carter -> Grid, Subject, Observer, Tile

Lucas -> Game, Level, Floor Generation