CC3K: Design Document

Contents

1 Overview **1**

2 UML Diagram **1**

3 Design **3**
- 3.1 Entities . . . 3
  - 3.1.1 General Entity . . . 3
  - 3.1.2 Player . . . 3
  - 3.1.3 Enemy . . . 4
  - 3.1.4 Potion . . . 4
  - 3.1.5 Treasure . . . 4
- 3.2 Game control . . . 4
  - 3.2.1 Tile . . . 4
  - 3.2.2 Grid . . . 4
  - 3.2.3 Game . . . 5
  - 3.2.4 TextDisplay . . . 5

4 Resilience to Change **5**

5 Questions **6**

6 Extra Credit Features **7**

7 Final Questions **7**

1 Overview

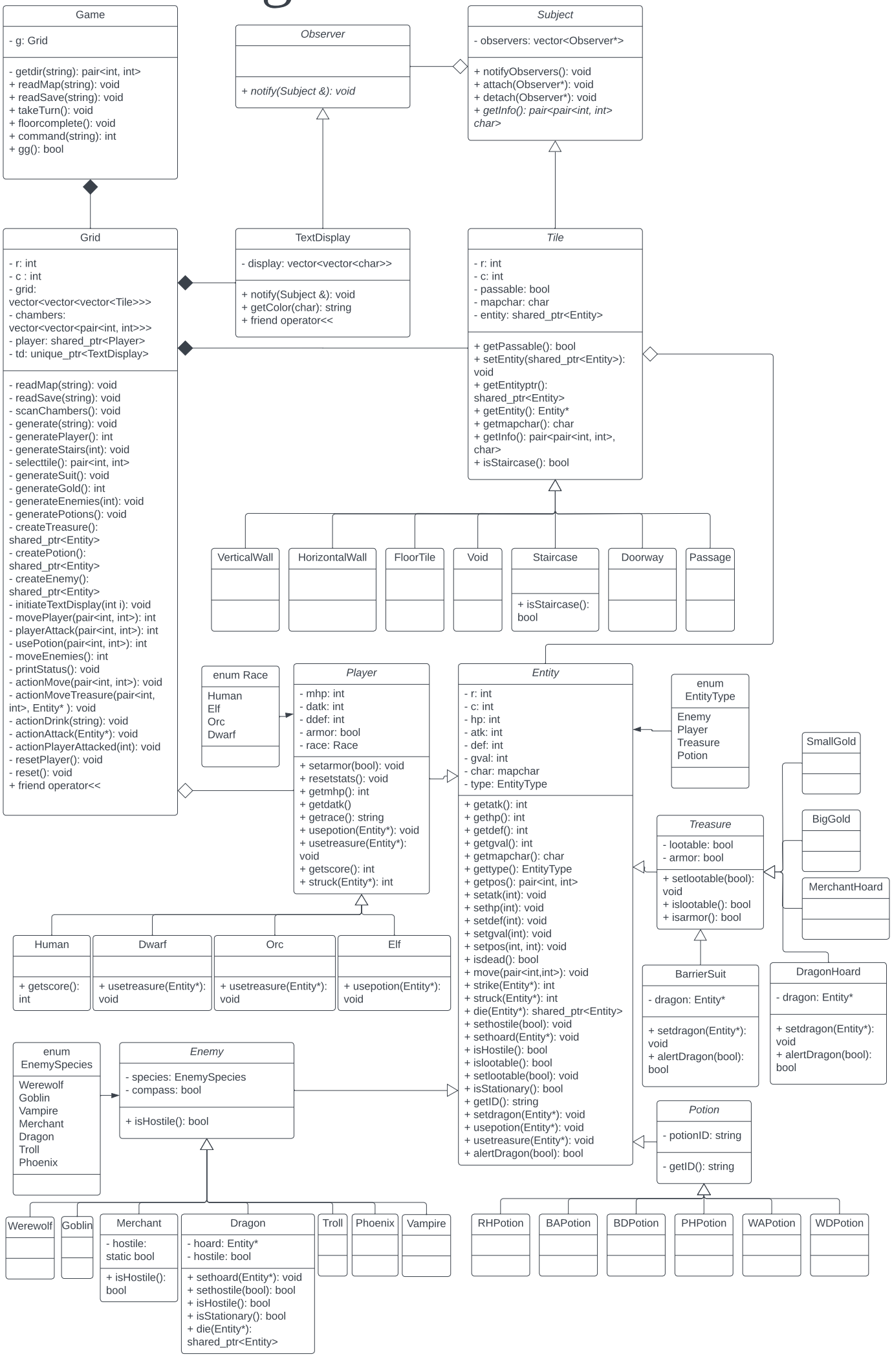
My implementation of the game CC3K is built around two major components: entities and the grid. The grid is where the state of the game board is handled: where the player is generated, moved, and so on. Entities are the objects on the game board: enemies, the player, and so on, and used to facilitate combat and item interaction.

My project is thus broken into two major components: the entity block, containing the Entity class and its subclasses, which are used to implement the player, enemies, potions and treasure. The second component is the game controller, consisting of the grid and functions which control the generation and state of said grid. Players interact with the controller to manipulate the grid state, which is output to them using a third, smaller component, the text display.

2 UML Diagram

Included on the following page.

UML Diagram


Game
- g: Grid
- getdir(string): pair<int, int>
+ readMap(string): void
+ readSave(string): void
+ takeTurn(): void
+ floorcomplete(): void
+ command(string): int
+ gg(): bool
Observer
+ notify(Subject &): void
Subject
- observers: vector<Observer*>
+ notifyObservers(): void
+ attach(Observer*): void
+ detach(Observer*): void
+ getInfo(): pair<pair<int, int> char>
Grid
- r: int
- c : int
- grid: vector<vector<vector<Tile>>>
- chambers: vector<vector<pair<int, int>>>
- player: shared_ptr<Player>
- td: unique_ptr<TextDisplay>
- readMap(string): void
- readSave(string): void
- scanChambers(): void
- generate(string): void
- generatePlayer(): int
- generateStairs(int): void
- selecttile(): pair<int, int>
- generateSuit(): void
- generateGold(): int
- generateEnemies(int): void
- generatePotions(): void
- createTreasure(): shared_ptr<Entity>
- createPotion(): shared_ptr<Entity>
- createEnemy(): shared_ptr<Entity>
- initiateTextDisplay(int i): void
- movePlayer(pair<int, int>): int
- playerAttack(pair<int, int>): int
- usePotion(pair<int, int>): int
- moveEnemies(): int
- printStatus(): void
- actionMove(pair<int, int>): void
- actionMoveTreasure(pair<int, int>, Entity*): void
- actionDrink(string): void
- actionAttack(Entity*): void
- actionPlayerAttacked(int): void
- resetPlayer(): void
- reset(): void
+ friend operator<<
TextDisplay
- display: vector<vector<char>>
+ notify(Subject &): void
+ getColor(char): string
+ friend operator<<
Tile
- r: int
- c: int
- passable: bool
- mapchar: char
- entity: shared_ptr<Entity>
+ getPassable(): bool
+ setEntity(shared_ptr<Entity>): void
+ getEntityptr(): shared_ptr<Entity>
+ getEntity(): Entity*
+ getmapchar(): char
+ getInfo(): pair<pair<int, int>, char>
+ isStaircase(): bool
VerticalWall
HorizontalWall
FloorTile
Void
Staircase
+ isStaircase(): bool
Doorway
Passage
enum Race
Human
Elf
Orc
Dwarf
Player
- mhp: int
- datk: int
- ddef: int
- armor: bool
- race: Race
+ setarmor(bool): void
+ resetstats(): void
+ getmhp(): int
+ getdatk()
+ getrace(): string
+ usepotion(Entity*): void
+ usetreasure(Entity*): void
+ getscore(): int
+ struck(Entity*): int
Entity
- r: int
- c: int
- hp: int
- atk: int
- def: int
- gval: int
- char: mapchar
- type: EntityType
+ getatk(): int
+ gethp(): int
+ getdef(): int
+ getgval(): int
+ getmapchar(): char
+ gettype(): EntityType
+ getpos(): pair<int, int>
+ setatk(int): void
+ sethp(int): void
+ setdef(int): void
+ setgval(int): void
+ setpos(int, int): void
+ isdead(): bool
+ move(pair<int,int>): void
+ strike(Entity*): int
+ struck(Entity*): int
+ die(Entity*): shared_ptr<Entity>
+ sethostile(bool): void
+ sethoard(Entity*): void
+ isHostile(): bool
+ islootable(): bool
+ setlootable(bool): void
+ isStationary(): bool
+ getID(): string
+ setdragon(Entity*): void
+ usepotion(Entity*): void
+ usetreasure(Entity*): void
+ alertDragon(bool): bool
enum EntityType
Enemy
Player
Treasure
Potion
Human
+ getscore(): int
Dwarf
+ usetreasure(Entity*): void
Orc
+ usetreasure(Entity*): void
Elf
+ usepotion(Entity*): void
Treasure
- lootable: bool
- armor: bool
+ setlootable(bool): void
+ islootable(): bool
+ isarmor(): bool
SmallGold
BigGold
MerchantHoard
BarrierSuit
- dragon: Entity*
+ setdragon(Entity*): void
+ alertDragon(bool): bool
DragonHoard
- dragon: Entity*
+ setdragon(Entity*): void
+ alertDragon(bool): bool
enum EnemySpecies
Werewolf
Goblin
Vampire
Merchant
Dragon
Troll
Phoenix
Enemy
- species: EnemySpecies
- compass: bool
+ isHostile(): bool
Potion
- potionID: string
- getID(): string
Werewolf
Goblin
Merchant
- hostile: static bool
+ isHostile(): bool
Dragon
- hoard: Entity*
- hostile: bool
+ sethoard(Entity*): void
+ sethostile(bool): bool
+ isHostile(): bool
+ isStationary(): bool
+ die(Entity*): shared_ptr<Entity>
Troll
Phoenix
Vampire
RHPotion
BAPotion
BDPotion
PHPotion
WAPotion
WDPotion

3 Design

I will expand further on the components mentioned in the overview.

3.1 Entities

I chose to implement all players, enemies, and so on as entities due to the majority of them sharing various data fields: HP, attack, defense, gold value, and so on. Moreover, the Entity superclass could be passed around to various functions to facilitate the interaction between different entities without needing to explicitly know exactly what type of entity was being passed through.

3.1.1 General Entity

The general entity has the `type` field which keeps track of what entity type it is (player, enemy, potion, or treasure), integer row and column fields to keep track of its position on the board, and the common fields mentioned earlier (health, attack, defense, gold value) as integers. It also has a character field which holds how it should be represented on the board. All data fields are private.

For each data field, there is an accessor and mutator method to prevent unwanted tampering with the data fields.

For each entity, a `move` function taking in a pair of integers is implemented which allows the entity to update its position after movement. Virtual `strike` and `struck` methods are implemented for each entity as well, which implement what should happen when an entity strikes another entity, or when an entity is struck. These functions handle combat. These both take an entity pointer and return the amount of damage taken. By default, the struck entity takes damage equal to `(ceil (100/(100.0+getdef())) * striker->getatk()`.

Lastly, a virtual `die` method is implemented, taking in the entity's killer and returning a `shared_ptr<Entity>` which represents the remains an entity leaves behind when it ceases existing (is killed or consumed). By default, `die` gives gold to the killer and returns a null pointer. More specifically, `usetreasure` is called on the killer with the killed entity, which allows the killer to interact with the entity's gold value.

The above methods are virtual so that side effects that may occur when they are called can be implemented.

The remaining functions implemented for entities are virtual dummy functions which effectively do nothing, but which are overriden by the respective subclass when necessary.

3.1.2 Player

Players are given a few extra data fields: their maximum HP, default attack and defense stat, along with a boolean armor field which keeps track of whether or not the player has picked up the Barrier Suit. They also have a race field keeping track of their race (Human, Elf, Dwarf, Orc).

Implemented for players are default potion and treasure consumption functions, which add the corresponding data fields to the player (health, atk, def for potions and gval for treasure). Lastly, a `getscore()` function is implemented which retrieves the player's score for end of game scoring purposes. An overridden `struck` function is in place to apply the barrier suit's damage reduction.

Each race is implemented as a subclass which call's the Player's constructor with different stat fields, and with overriden functions which handle class specific functions. For example, the Elf has an overridden `usepotion` which implements its potion ability.

3.1.3 Enemy

Enemies have only one extra function: `isHostile` which returns whether or not the enemy is hostile to the player. By default, this returns true.

Each enemy type is implemented with its own stats. The Merchant is implemented with a static boolean which keeps track of whether or not Merchants are hostile, along with a corresponding overidden `isHostile` function, and an overidden `struck` function, which sets the boolean to true (when the player strikes a merchant, they become hostile).

The dragon was more complicated to implement. It has an extra boolean which controls hostility. I gave it a pointer to another Entity, its hoard. When the dragon dies, the dragon sets the hoard to lootable (the player can then pick it up) in its overridden `die` function. The hoard itself is responsible for toggling the Dragon's hostility. The dragon also has an overriden `isStationary` which keeps it stationary on the board.

3.1.4 Potion

Potions are basically just normal entities.

3.1.5 Treasure

Treasures are normal entities, with the exception of the Barrier suit and Dragon hoard, which have pointers to their dragon guards as well as functions which notify them when the player is near.

3.2 Game control

3.2.1 Tile

Tiles are a class with a boolean field `passable` which keeps track of whether or not entities can move onto them as well as a `shared_ptr<Entity>` which keeps track of what entity is on the tile.

Tiles inherit from Subject, and are observed by the text display. They thus also have row and column fields to keep track of their position on the board for notification purposes. When the tile contains no entity, it sends the tile's character to the Observer. When an entity is on top of the tile, it calls the entity's `getmapchar()` instead to return the entity's character to the text display.

3.2.2 Grid

Grids are implemented as a 2D dynamic array of tiles. As they handle much of the game's logic, I will expand on each portion below:

Chamber layout: The grid is able to read in either blank map layouts or save states (with entities and so on already generated). This is done by passing in a filename, which is then read from.

Chamber generation: In the case of a blank map, the grid will then generate the layout of each floor. The `generate` function is passed the player's race, and then generation proceeds as follows: for each floor, the chambers of the floor are scanned and placed in a vector. Then, for each object which needs generation (player, stair, enemy, etc.), a random chamber is chosen. Once the chamber is chosen, a random tile in the chamber is chosen, and if that tile is valid (has no entity on it) then the game places an entity on that tile.

As desired, generation proceeds in the order of player, stair, potions, treasure and enemy.

The chamber that the player spawns in is passed as an argument to the stair generation function, so that they do not spawn in the same chamber.

The number of dragons spawned during treasure generation is passed to the enemy generation function so that the correct number of enemies is spawned. For spawning entities such as enemies and potions, I employed the factory method pattern which randomly selects an entity of the desired type with the according probabilities.

Movement and combat: When an entity is moved, the respective tiles are updated (the tiles then point to different entities). When an entity dies, the tile it is on points at its death entity instead (returned by `die`). The player movement is handled based on a passed in direction, whereas enemy movement is random: if the enemy is not stationary (not dragon) and not next to the player or not hostile to the player, it will randomly move in a valid direction. Otherwise, it will attack the player.

Ascending floors: When the player reaches the staircase, the player is sent to the next floor and its stats are reset (for attack and defense). It does this by maintaining a pointer to the player (this is also used for player movement).

3.2.3 Game

The game class handles input from the player and passes it to the grid as necessary. For example, it will translate `"no"` into the pair (-1, 0) and send a movement command to the grid so that the player moves in the correct direction on the grid.

3.2.4 TextDisplay

The textdisplay observes the grid's tiles to accordingly update a `char` array. Output corresponding to the player's health and actions occuring is handled by the grid.

I also decided to output two grids per turn: one after the player takes its action and one after the monsters take action. This let's the player see exactly what is happening on the board at the cost of more cluttered output.

4 Resilience to Change

My design is able to accomodate change relatively easily due to little coupling and high cohesion of the entity classes. If the behaviour of an entity in combat is to be changed, then that is easily done by changing a corresponding `strike` or `struck` option, for example.

Sweeping changes to classes is also easily handled since they all inherit from the same class: if enemies started dropping their gold in piles instead of depositing it to the player, then this is easily handled by updating the `enemy` superclass.

Similarly, changes to other types such as potions, treasure or the player is very easy due to their modularized nature.

Adding types is similarly easy as I would just have to write a new class inheriting from its respective type.

My grid class has less cohesion, but still little coupling with other classes, and as such my design is also easily able to handle changes to the overarching game board such as layout, size and so on because I have separated the generation algorithm into several steps within my grid class.

If the board size were changed for example, I would just have to modify some constants in my grid class to read in the new board size. I have also already implemented an algorithm which can respond to changes in board layout using some basic graph theory, which I will expand on later.

If generation was changed such as enemy probabilities, I would be able to handle it easily as I have separated enemy generation into its own factory method, and would thus only have to tweak that method.

Overall I think that unless you were to ask me to implement an entirely new feature such as a magic system, I would only need to change a couple dozen lines of code.

5 Questions

Question. How could your design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?
Answer. My system is designed so that each race is a subclass of a `Player` superclass, with some special overriden functions to handle its special abilities. This makes generation of new races incredibly easy, as I do not have to make changes to the game other than adding a subclass to `Player`. Within the game, the player is referred to as a player pointer anyway, so introducing a new subclass does not affect any of the game logic.

Question. How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?
Answer. Generation of enemies is done randomly through a factory method: `createEnemy()`. This is different to how the player is generated since I have to know what race the player is before generation. Other than that, however, they are similar in that both are done through the generation of a subclasses.

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.?

Answer. My system accomodates changes to special abilities fairly easily since each enemy/race is a subclass of a superclass, which means that adding abilities is mainly done through introducing subclasses.

For some concrete examples, I will consider the abilities mentioned within the question. For gold stealing and health stealing for goblins and vampires, respectively, I can add these with less than 10 lines of changed code by adding an overriden `strike` function to the corresponding enemy type. For example, I can simply have the goblin steal 1 gold from the entity it strikes whenever `strike` is called.

The temporal ability of the troll would require a bit more change, as I have not implemented any turn based abilities. However, this is still easily added by adding a turn counter to each entity and calling `activateAbilities` on each entity every turn. For most entities, this won't do anything, but I can override this function to restore health for the troll.

A hard ability to implement would be something like a merchant shop, since this would require changes to the game's display and some of the game logic, so I would have to change the grid a little while also adding different functions to the merchant, player and display.

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?
Answer. I can use the Decorator pattern, so that when the player consumes a potion a decoration is

added to the player. Decorations can then be removed at the end of each floor to reset potion effects.

In practice, I found that this was too much hassle for simply shifting the values of some numbers, so I chose to handle resetting of player stats within the player object itself.

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 hordes and the Barrier Suit?

Answer. Generation of treasure, potions and major items all use the same tile selection system, so I wrote a function which selects a tile from the grid to generate the item on. However, since each type of item has different probabilities, I still had to write different factory methods for each.

Protecting the dragon and barrier suit can be done using a function as well since I am just generating a dragon and tying it to the treasure entity (the protection of the treasure is handled by the dragon entity itself).

6 Extra Credit Features

I added two small extra credit features. I wrote all my code using smart pointers, so I have 0 new or `delete` calls within my code. I did pass some raw pointers around in functions, but these could basically all be replaced by smart pointers (I didn't understand them fully when I first write them).

Secondly, my program is able to handle different map layouts, provided they are 25 by 79. This was done by using a BFS algorithm on each floor of the layout to scan for the chambers of that floor, rather than just hardcoding the coordinates of each chamber.

7 Final Questions

Question. 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?

I learned quite a lot regarding writing large programs. I had basically written only 1 file programs with everything in main before, so just figuring out how to compile and incorporate all the files together was a challenge.

I also learned the importance of planning my classes out beforehand. My initial UML was only half fleshed out, and so much of my code became less organized as I found various features that I needed to add in.

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

I would design my program slightly differently, such as adding more classes to handle game control so that I don't have a grid class which does a little too much.

I would also have started working a bit sooner, as this project took a lot longer than I anticipated, which led to me not being able to implement certain parts of the program such as potion visibility or the compass. I also spent less time than I would have liked on this document, and so I'm not sure how clear the ideas I have presented are, but I hope they are clear enough to have a general idea of how my program functions.

Thanks for reading.