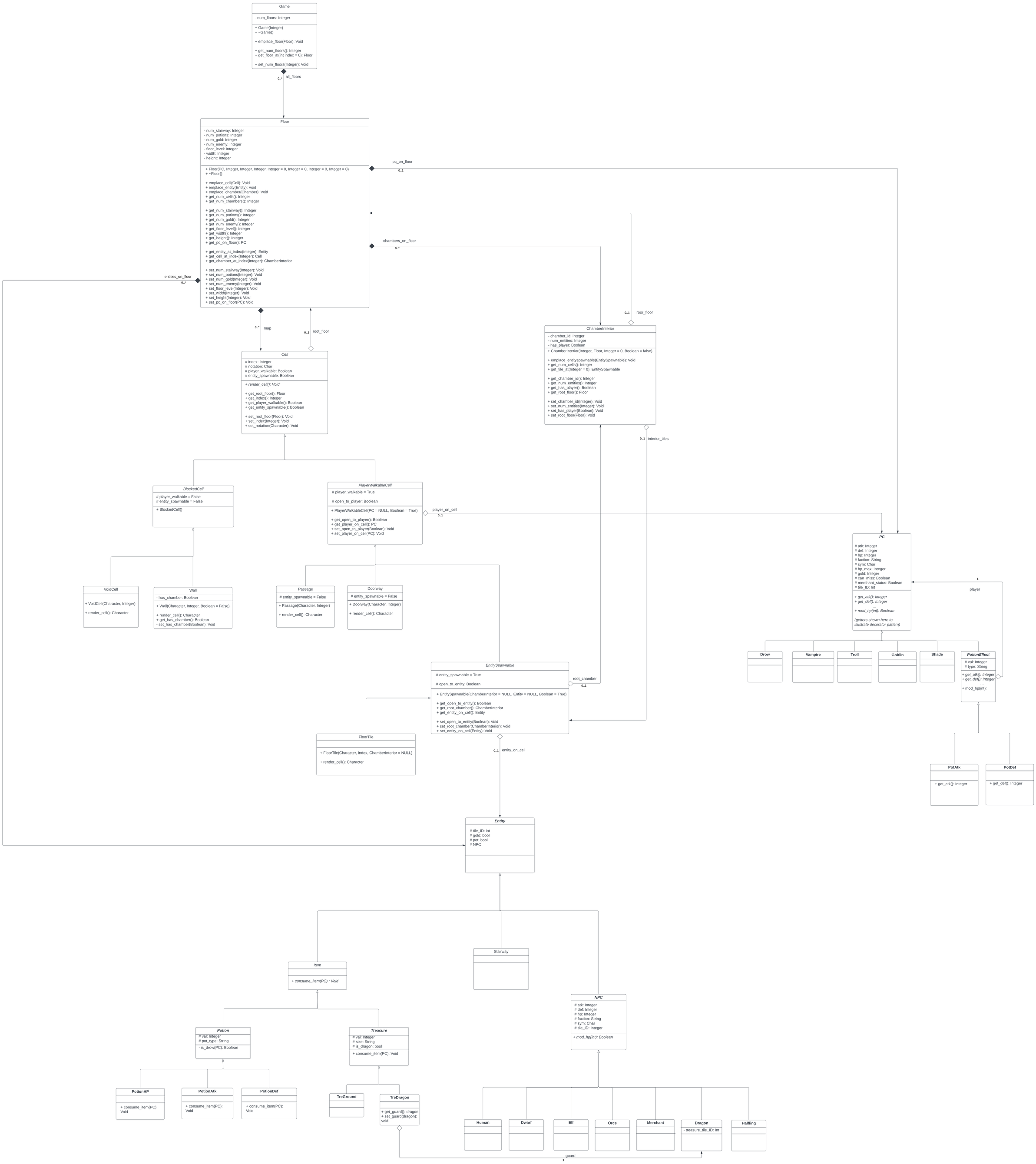
Game
- num_floors: Integer
+ Game(Integer)
+ ~Game()
+ emplace_floor(Floor): Void
+ get_num_floors(): Integer
+ get_floor_at(int index = 0): Floor
+ set_num_floors(Integer): Void
0..*
all_floors
Floor
- num_stairway: Integer
- num_potions: Integer
- num_gold: Integer
- num_enemy: Integer
- floor_level: Integer
- width: Integer
- height: Integer
+ Floor(PC, Integer, Integer, Integer, Integer = 0, Integer = 0, Integer = 0, Integer = 0)
+ ~Floor()
+ emplace_cell(Cell): Void
+ emplace_entity(Entity): Void
+ emplace_chamber(Chamber): Void
+ get_num_cells(): Integer
+ get_num_chambers(): Integer
+ get_num_stairway(): Integer
+ get_num_potions(): Integer
+ get_num_gold(): Integer
+ get_num_enemy(): Integer
+ get_floor_level(): Integer
+ get_width(): Integer
+ get_height(): Integer
+ get_pc_on_floor(): PC
+ get_entity_at_index(Integer): Entity
+ get_cell_at_index(Integer): Cell
+ get_chamber_at_index(Integer): ChamberInterior
+ set_num_stairway(Integer): Void
+ set_num_potions(Integer): Void
+ set_num_gold(Integer): Void
+ set_num_enemy(Integer): Void
+ set_floor_level(Integer): Void
+ set_width(Integer): Void
+ set_height(Integer): Void
+ set_pc_on_floor(PC): Void
pc_on_floor
0..1
chambers_on_floor
0..*
entities_on_floor
0..*
0..*
map
0..1
root_floor
0..1
roor_floor
Cell
index: Integer
notation: Char
player_walkable: Boolean
entity_spawnable: Boolean
+ render_cell(): Void
+ get_root_floor(): Floor
+ get_index(): Integer
+ get_player_walkable(): Boolean
+ get_entity_spawnable(): Boolean
+ set_root_floor(Floor): Void
+ set_index(Integer): Void
+ set_notation(Character): Void
ChamberInterior
- chamber_id: Integer
- num_entities: Integer
- has_player: Boolean
+ ChamberInterior(Integer, Floor, Integer = 0, Boolean = false)
+ emplace_entityspawnable(EntitySpawnable): Void
+ get_num_cells(): Integer
+ get_tile_at(Integer = 0): EntitySpawnable
+ get_chamber_id(): Integer
+ get_num_entities(): Integer
+ get_has_player(): Boolean
+ get_root_floor(): Floor
+ set_chamber_id(Integer): Void
+ set_num_entities(Integer): Void
+ set_has_player(Boolean): Void
+ set_root_floor(Floor): Void
0..1
interior_tiles
BlockedCell
player_walkable = False
entity_spawnable = False
+ BlockedCell()
PlayerWalkableCell
player_walkable = True
open_to_player: Boolean
+ PlayerWalkableCell(PC = NULL, Boolean = True)
+ get_open_to_player(): Boolean
+ get_player_on_cell(): PC
+ set_open_to_player(Boolean): Void
+ set_player_on_cell(PC): Void
player_on_cell
0..1
PC
atk: Integer
def: Integer
hp: Integer
faction: String
sym: Char
hp_max: Integer
gold: Integer
can_miss: Boolean
merchant_status: Boolean
tile_ID: Int
+ get_atk(): Integer
+ get_def(): Integer
...
+ mod_hp(int): Boolean
(getters shown here to illustrate decorator pattern)
1
player
VoidCell
+ VoidCell(Character, Integer)
+ render_cell(): Character
Wall
- has_chamber: Boolean
+ Wall(Character, Integer, Boolean = False)
+ render_cell(): Character
+ get_has_chamber(): Boolean
- set_has_chamber(Boolean): Void
Passage
entity_spawnable = False
+ Passage(Character, Integer)
+ render_cell(): Character
Doorway
entity_spawnable = False
+ Doorway(Character, Integer)
+ render_cell(): Character
Drow
Vampire
Troll
Goblin
Shade
PotionEffect
val: Integer
type: String
+ get_atk(): Integer
+ get_def(): Integer
...
+ mod_hp(int):
EntitySpawnable
entity_spawnable = True
open_to_entity: Boolean
+ EntitySpawnable(ChamberInterior = NULL, Entity = NULL, Boolean = True)
+ get_open_to_entity(): Boolean
+ get_root_chamber(): ChamberInterior
+ get_entity_on_cell(): Entity
+ set_open_to_entity(Boolean): Void
+ set_root_chamber(ChamberInterior): Void
+ set_entity_on_cell(Entity): Void
root_chamber
0..1
FloorTile
+ FloorTile(Character, Index, ChamberInterior = NULL)
+ render_cell(): Character
0..1
entity_on_cell
PotAtk
+ get_atk(): Integer
PotDef
+ get_def(): Integer
Entity
tile_ID: int
gold: bool
pot: bool
NPC
Item
+ consume_item(PC) : Void
Stairway
NPC
atk: Integer
def: Integer
hp: Integer
faction: String
sym: Char
tile_ID: Integer
+ mod_hp(int): Boolean
Potion
val: Integer
pot_type: String
- is_drow(PC): Boolean
Treasure
val: Integer
size: String
is_dragon: bool
+ consume_item(PC): Void
PotionHP
+ consume_item(PC): Void
PotionAtk
+ consume_item(PC): Void
PotionDef
+ consume_item(PC): Void
TreGround
TreDragon
+ get_guard(): dragon
+ set_guard(dragon): void
Human
Dwarf
Elf
Orcs
Merchant
Dragon
- treasure_tile_ID: Int
Halfling
guard
1

Overview:

In our structure of the game, each individual elements including different types of player, different types of floor cells, different types of enemies and other things, are built and implemented in their own individual classes. For overlapping properties, we designed superclasses and used the concept of inheritance and polymorphism to reduce redundant code and implement common functionalities across similar types.

While the classes provides the special functionalities, our main function includes helpers which handles the user interaction with console input output. Moreover, the main function is also responsible for initializing the instances of the classes in order to initialize and start the game playing.

Updated UML: Please see the first page of this file

Design:

One challenging problem is that in order to randomly generate NPC and other entities on an empty map, the chamber space of each chamber would need to be determined since the probability of each entity spawning is equal in every chamber regardless of the size. In order to determine the chamber space, we developed an algorithm that first determines the top left corner of a chamber, recursively checks the 4 cells around it, assign every floor tile to the same chamber ID, and mark each wall tile as read so no duplication can occur. This was particularly difficult since you can not determine any information about a chamber while reading in from the files. More specifically, it is very hard to distinguish between

Moreover, this makes it difficult to interpret the corners as the reading only goes left to right and top to bottom and the same for the index in the vector of floors. Thus another algorithm needed to be used to identify the interior of the chambers.

Another challenges is that the file consist of 5 consecutive maps instead of 1 floor at a time. This makes it difficult to read and transfer player information since there is only 1 player character in the entirety of the game. Thus the original design of letting the game hold the player had to be reworked to floor holding the player, and each time the player enters the new floor, the client had to ensure the information are transferred correctly without any misinformation.

Moreover, we designed our system with a lot of abstract classes. Originally, we planned to use unique pointers to free us from manual memory management. However, the program involves a lot of abstract class pointers rather than concrete ones. Since the make_unique always creates an instance of the class, this can not be done through unique pointers in the short period of time given after countless attempts. Thus we had to be extremely careful with manual memory management implementation, even though the UML clearly identified the ownership. Especially with including / importing versus forward declaration involving ownerships which we had struggle quite.

Deviation from DD1 design:

NPC:

The field chamber_ID is changed to tile_ID and stores the index of cell that the NPC is on instead of the chamber that it is in for easier movement and access to surrounding cells

Every non getter/setter method for NPC other than mod_HP is removed since they would all either need to call methods of the PC pointer, which would present a cross include problem (PC.h including NPC.h and NPC.h including PC.h). Thus, every helper's responsibility is moved to I/O in a helper in main instead.

NPC->Dragon:

Dragon no longer has a treDragon (dragon hoard) pointer, but is instead replaced with a treasure_tile_ID that can be used to locate the tile of the dragon hoard. Once again, dragon does not have access to treDragon's methods, and an integer is used instead to decrease module dependency.

PC:

Every method for PC is removed for a similar reason to NPC, and responsibility is handed over to I/O.

PC->Vampire: allergy field removed, opted to check the faction of enemy attacked directly in I/O in main

PC->Troll: regeneration handled in I/O by calling mod_hp(5) every turn instead of a regen method

Entity:

Every method for entity is removed as well for a similar reason to NPC, and 3 booleans (gold, npc, pot) are added instead in order to make dynamic casting available in the main function and determine the type of entity stored.

subclasses of Potion:

each individual unique helpers are removed, and instead replaced with overriding the universal pure virtual method consume_item in their parent class item. This makes calling the function much easier without the use of dynamic casting, and it also avoids code duplication caused by writing individual unique helpers.

Cell:

The cell_type field is taken out from the class during implementation as the client is able to identify the different types of cells through casting or through render_cell() method and comparing the returned character. The removal of this field also leads to the removal of subsequent getters and setters.

The pos_row and pos_col fields for the cells are also removed. This is because since the Cells are placed in a vector of the floor, their index / position in that vector is unique. Our implementation choose to use this index as the unique identifier of the cell. Moreover, since the floor is assumed to have a constant width and height, the row and colume positions of the cell can be easily found using % operator on the width and height. The removal of these fields also lead to the removal of subsequent getters and setters.

The most significant change to the Cells is that the pure virtual function, render_cell(), used to output the map layout, now returns a character instead of doing direct printing within the classes. This improves reusability and flexibility of the classes and to accommodate various needs and formatting for the client.

Wall:

The Wall class now includes an additional parameter, has_chamber(), which is used by the client while distribution and identifying the different chambers in the floor.

CmdInterpreter:

This class is removed entirely, as the client / main function should have total control over what they choose to operate with the classes. Moreover, the existence of this class previously would require it to include all modules offered by the classes, including coupling and more prone to errors.

The purpose of this class before was to interpret the various inputs of the users and construct the game. For the current implementations, all these are done by the client using helper functions.

Floor:

All floor's spawning functions, render_map, and random_chamber functions are taken out, and instead are being handled by the client using helper functions.

With the previous design, the floor would also need access to all subclasses involved in generation and other functionalities, for instance, the constructor and methods for Enemy generation. This design is deemed to be bad since this significantly increase coupling between modules. Moreover, the floor would be taking way too many responsibilities over the other classes which will significantly reduce reusability of individual classes.

Therefore, we choose to instead implement these on the client side with helper functions, offering flexibility to the client on how they choose to generate objects or conduct other operations

An additional change following this, is that the enemy_movable field is also taken out and handled by the client for similar reasons, along with the associated getters and setters.

A significant change to Floor is that instead of "having a" player, where the floor has not responsibility nor ownership over, it is now implemented to "own a" player instead. This change means that floor has complete ownership of the player and must manage its memory and other resources accordingly.

The reason for this change is that we realised during implementation that a single map file consists of multiple floors and possibly each floor has a player symbol already implace. With the previous design, this means that the game would have to have multiple players in a vector that is corresponding the each player on each floor. This increases complexity and makes the implementation more prone to errors such as mismatching the player index with the floor index for the client.

With that reason, we choose to change the implementation of floors and leave room for the client on whether they want to generate multiple floors at the same time, or generate one floor at a time. This change means that the Floor must free the space occupied by its player when it is freed itself.

Subsequent change to Game is also made and mentioned below.

Game:

Fields of the Game such as num_players, curr_floor, game_on, pc, are taken out and instead being handled by the client. With the same reason mentioned above, the game

would have way too many responsibilities with the previous design where it keeps track of multiple things. The previous design is therefore prone to error, leading to a change in implementation of the class, and leaving the freedom to the client.

Functions of Game such as generate_floor and load_floor is also take out to be part of the client's options with helper functions. As mentioned before, the Game would have to have full access to other classes' implementations, increasing coupling and decreasing reusability. The Game would also have much more responsibility with the original implementation. Overall, the original implementation leads to a fixed way of using the class which inhibits reusability which is the key of OOP.

Resilience to Change:

The spawn rate for various entities are set as constants at the beginning of files. This allows modifying the spawning rate of different types of enemies, gold, potions etc. For instance, the client can choose to spawn only dragon hoards and dragons and nothing else, or choose to take out the change of spawning negative effect potions. This also makes adding new entities easier where the chances do not have to be recalculated everytime.

Similar concept is used for characters in the maps representing different entities, both from reading pre-defined maps and also rendering out the game map. The client has the ability to change those to any character they would like, increasing flexibility and reusability.

Similar concept also appies colors of the text, values of gold and so on. Moreover, this also applies to the dimensions of the map.

Answers to 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?

In order to implement and generate each race easily, they would be made as derived classes of the abstract parent class PC (playable character). This way, any of the character generated can be accessed using a PC* pointer, and there would not be any code duplication for the shared fields and methods (which turns out to be almost all the fields and methods).

Additionally, it would also make it very easy to add more races as another derived class by simply plugging in the stats into the PC constructor and add unique abilities at the main function I/O.

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

The class structure for different race of NPCs is very similar to PC, where there is an abstract parent class "NPC" with derived classes of NPC races. Each enemy would be generated based off their spawn probability or by scanned inputs and stored in a vector of entities (entities_on_floor) owned by the floor, whereas the player race is generated based off the user's input/choice.

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

The NPC's unique abilities is done at the process_NPC_action helper, where if the NPC were to attack a player, the helper would check the race of the NPC and the race of the PC and determine if there is any special ability to be used. For example, if the NPC is a halfling and it is attacking the player, there is a 50% chance of player's can_miss field is set to true and results in the player's next attack missing. Similar idea is being applied to the player's unique abilities, such as calling the mod_hp() method every loop (turn) if the player race is troll, or to determine the the attacked NPC's race and add/lose HP accordingly if the player race is vampire.

The Decorator and Strategy patterns are possible candidates to model the effects of potions; which pattern would work better? Explain in detail, by discussing the advantages/disadvantages of the two patterns.

We think that both patterns could be used for this question however, considering multiple factors, we decided to implement using the Decorator pattern.

Using the Decorator pattern, our PC class does not need to have an additional field which will be required by the Strategy pattern to contain the actual interface object. Moreover, the significant advantage with Decorator pattern is that Decorators do not affect or change the basis concrete object. In this context, this means that the effect of potion will not directly change the PC's attack and defence field, but rather, provides additional information on them. This means that upon reaching a new level, the effects of potions can be easily removed as we simply detach and clear the decorators. The downside of using the Decorator pattern would be that it cause a tricky situation with potions that deal with health (RH / PH) in which their effects will be permanent, moreover, we would have to override all the methods of PC so that when a decorator is called upon, we can correctly retrieve the basic component's methods.

The Strategy pattern will work almost exactly opposite, where all the concrete strategy could end up modifying the field of the PC. This will have it hard to undo the effect of attack and defence potion upon reaching another level. However, it will make it easier to add other permanent effect potions and make the relationships and implementation a lot more straightforward WITHOUT the step of undoing the effect, since it can be determined at runtime.

Thus considering the advantages and disadvantages of the two patterns, we have decided that using the Decorator pattern would be better as it is almost impossible to undo a change which is the hardship with the Stategy pattern. We think that separating permanent effect from the temporary Decorators will be more straightforward than working out a way to undo a change.

H**ow 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?**

In order to avoid code duplication, both Treasure and Potion are derived classes of the abstract parent class "item". Since they share the same parent class, they can both be stored in one vector of entities, which facilitates the memory management. They also share all share the same method consume_item, meaning the code for the player to interact with them would be very similar (with the exception of treDragon, which has its own additional fields to determine whether the dragon guarding it is killed or not).

Extra Credit Features:

A detailed action output that specifies the action of the turn. Player's movement, attack, and potion use are all being outputted, and the exact source of damage from.

Final Question:

1. One of the most important things that we learned from this project is effective communication. In the early stage of the project, we sometimes tunnel-visioned to the UML and the plan of attack that we made and implemented our classes and methods strictly to it only to realize that we were making unrealistic assumptions of our teammate's modules. If we require other people's implementations to function in a specific way, we should ask them in advance and make compromises and adaptations if necessary.

On the more technical side, this project taught us the importance of branching, merging and version control. Our program is very I/O heavy and there were many occasions that we were forced to work on main.cc at the same time, and we were having a hard time combining the changes that we made to the same file. Although we mainly used verbal communication to sort out situations like this, we are definitely interested in learning some of the version control tools that will help us collaborating much more easily in future coding group projects.

2. One thing that we would to differently is to do more research ahead of time and plan out the overall structure of the project better. During our planning phase, we were neglecting some of the things that would later become very troublesome to solve. For example, we were not aware of the need for dynamic casting and only solved this by adding fields to superclasses once we realize the issue. Although we did have everything figured out in the end, we definitely would have saved ourselves some time if we planned ahead better.