Universal Definitions

«dataType»
Coords
(from Universal Definitions)

+ x: int
+ y: int
+ chamber: int

+ compare(c1: Coords, c2: Coords): bool

«dataType»
Effect
(from Universal Definitions)

+ HP
+ money
+ ATK
+ DEF
+ passiveDMG

Player

+ move(dir: string): bool
+ attack(dir: string): bool
+ receiveEntityActions(): void
+ ReceiveEffect(e: Effect)
+ isDead(): bool
+ printStatus(): void
+ getRender(): vector<char>
+ tickItems(): void
+ setFloor(f: Floor): void
+ trade(dir: string): void
+ setCoords(c: Coords): void
+ isComplete(): bool
+ quit(): void
+ tickSelf(): void
+ getGold(): int

Floor

- map: vector<char>
- stairs: Coords
- ppos: vector<Coords>

+ Floor(file: string, plain: bool)
+ getMove(c: Coords, dir: string, isAttack: bool): Coords
+ inRange(c: Coords): vector<Entity*>
+ getTarget(c: Coords, dir: string): Entity*
+ updateEntities(): void
+ complete(c: coords): bool
+ addEntity(e: Entity, cham: int): bool
+ spawnEntity(e: Entity, c: Coords): bool
+ getRender(): vector<char>
+ pickup(i: Item): Item
+ addPlayer(): Coords
+ getH(): int
+ getV(): int

Chamber

+ updateEntities(ppos: vector<Coords>): void
+ getTarget(c: Coords, dir: string, isAttack: bool): Entity*
+ inRange(c: Coords): vector<Entity*>
+ addEntity(e: Entity, cham: int, ppos: vector<Coords>): bool
+ spawnEntity(e: Entity, ppos: vector<Coords>): bool
+ getAllEntities(): vector<Entity>
+ pickup(i: Item): Item
+ getMove(c: Coords, dir: string): Coords

Shade

Drow

Vampire

Troll

Goblin

Healer

+ isHealing(): bool

Character

+ getHP(): int
+ move(dir: string): void
+ receiveDMG(int playerATK): void
+ getATK(): int
+ getDrop(): Entity
+ isSpawner(): bool

Entity

+ getType(): string
+ getLocation(): Coords
+ setLocation(int x, int y): void
+ isItem(): bool

Item

+ getDuration(): int
+ getEffect: Effect
+ getPrice(): int
+ locked(): bool
+ unlock(): void
+ tick(): void

Child
OBSERVES
Spawner

Spawner

+ addChild(c: child): void
+ notifyDeath(): void
+ getNewChild(): Child
+ getFreq(): int

Mandrake

VineHead

Root

Vine

Child

+ die(): void
+ notifySpawnerDeath(): void

Human

Elf

Dwarf

Orc

Halfling

Dragon

+ getHoard(): Item*

Spitter

Merchant

+ isHit: bool

+ getTrades(): vector<shared_ptr<Item>>

Design Overview

Universal struct definitions (yx2wang)

- Coords, coordinate system used throughout the program
  - x, y, chamber
  - Added various overridden operators as needed
- Effect, a single data structure for all item effects (including treasure)
  - HP, ATK, DEF, money, passiveDMG
  - Helper methods are added as needed in other parts of the code

Map Logic (yx2wang)

- Floor class, overall wrapper for all features inside a dungeon floor
  - Stores overall floor map, stairs location, player locations (note that this is NOT the entire player object, because it's the only thing the map needs to know)
  - ctor: reads a (potentially populated) map from file and prepares itself as well as its chambers (which have their own constructor)
  - addPlayer: adds player position to track, returns valid position to start with
  - getMove: returns the result of moving in a given direction, potentially staying put if the cell is occupied
  - getTarget: returns the relevant entity (character or item, depending on parameters) occupying the cell in that direction
  - inRange: returns all entities in range to attack (or apply AOE effects) to player
  - updateEntities: deals with movement, removal of dead entities etc. (management of entities found within the map)
  - complete: checks if the coords given are stairs (and thus the player has finished the level)
  - add/spawnEntity: puts an entity in the relevant chamber, spawn lets the entity set coordinates while add randomly puts the entity in that chamber
  - getRender: returns a character render minus the player sprites
  - pickup: transfers ownership of the item pointed to the caller (i.e. the player), who can then apply its status effects
- Chamber class, representing each chamber inside a floor (entities can only move inside their own chamber)
  - Stores chamber limits, entities
  - All floor methods mentioned above except complete, addPlayer: since entities are stored per chamber, the floor will call the chamber(s) needed for executing the methods
    - Note that this is from an optimization point of view: since entities cannot move between chambers, player interactions can only happen with entities in the SAME chamber, so checking entities that are not in that chamber is pointless

Overall, these two classes serve to abstract away the problem of finding the relevant entity or entities for the player. This means the player can now just ask the floor class for what it needs to worry about, and get straight to processing. Note that there is no destructor due to the use of smart pointers.

Enemy and Item Logic (hbmahida)

- Entity is an abstract class which contains the coordinates of items and enemies.
  - Has getters and setters to access and mutate the location of enemies and items after each turn.
  - Also contains virtual methods getType() and isItem() which are overridden in the item and character subclasses for spawn and player logic.
  - getType() returns the type of potion, treasure, or enemy and isItem() checks if an entity is an item or not.
- Item is a subclass of entity.
  - Has coordinates, type, isLocked, duration, and price as its fields.
  - isLocked is set to true only for dragon hoard as the hoard is locked until the dragon is slain. Once dragon dies, the unlock() method is called to allow the player to get the treasure by setting isLocked to false.
  - price of potions is required for merchant to sell his wares (DLC).

    - duration is for the heal effect of passive heal potion and Healer player class (DLC) as all base game potions are single use only (duration = 1)..
    - type indicates the nature of the potion or treasure.
    - getType(), getPrice(), getDuration(), and getEffect() are getters for player logic.
    - tick() is a method called after every turn to reduce the duration of item by 1 for heal logic.
    - isItem() is always true here.
- Character is an abstract subclass of entity. It is a wrapper for various enemy types.
    - Has health, attack, defense, and type.
    - getHP() and getATK() are getters for health and attack respectively.
    - isItem() and isSpawner() are such that they both return false unless the enemy type is Spawer when isSpawner() returns true.
    - getDrop() returns a shared_ptr of type Item. It gets called when an enemy dies to spawn the gold in its place (or SpitBall when Spitter dies).
    - move(string dir) is a virtual method which changes the coordinates of enemy to move in the direction dir. Overridden in dragon class which is stationary.
    - receiveDMG(int playerATK) is a virtual method that calculates the damage to be done to a character and changes its health accordingly. Overridden in halfling who has a chance of making the player miss (get 0 damage).
- The base game has the following enemy classes: Human, Elf, Dwarf, Orc, Halfling, Merchant and Dragon.
    - Human, Merchant, and Dragon override the getDrop() in Character as Human drops 2 piles of gold (4 gold), Merchant drops a Merchant hoard and Dragon simply unlocks the dragon hoard (as discussed above) when it dies.
    - The Merchant is a special type as it acts just like the other enemies in the base game but is capable of selling potions in the DLC:
        - Has a static inline bool called hit set to false when a new game starts. Also contains a vector of shared_ptrs called trades which is a vector of the potions player can buy.
        - isHit() returns true if any Merchant has been hit by the player as it would cause all merchants in that game to become hostile and not trade with the player.
        - getTrades() returns the trades field of the merchant so the player can buy potions (DLC). If a merchant is hostile, getTrades() returns an empty vector indicating that it is hostile and wouldn't trade.
        - getATK() would return 0 if the merchant is not hostile and receiveDMG(int playerATK) would set hit to true if playerATK > 0.

- DLC enemy classes are: Spitter, Mandrake, Root, VineHead, and Vine.
    - Spitter overrides getDrop() as it drops a SpitBall when it dies which deals passiveDMG to players in a 1 block radius but it can be picked up by the player for a significant amount of damage and gold.
    - Spawner is an abstract class that acts as a wrapper for VineHead and Mandrake. Has a vector of Child pointers.
    - getNewChild() is a virtual method that returns a shared_ptr of type Child for spawning. This is overridden in both Mandrake and VineHead. VinHead spawns vines in a random direction from the top-left corner of room while Mandrake spawns Roots at a random spot on the chamber.
    - deleteChild(Child *child) removes the child from vector children.
    - notifyDeath() notifies the children that the spawner has died and removes all of them from the vector children.
    - getFreq() is overridden in both VineHead and Mandrake as the rate at which they spawn children is different.
    - Child is an abstract class that acts as a wrapper for Vine and Head.
    - die() method is called when the child has 0 health and it notifies the spawner to remove the child from vector children.
    - notifySpawnerDeath() method reduces child's health to 0 when the spawner dies so it also dies.

Player and Hero Logic (s4khanna)

- Player is an abstract class which contains the health, attack, defense, money, coordinates, floor and the pointers to all the items picked up by the player.
    - Provides methods for player movement, attacking enemies, and receiving effects from nearby entities (enemies or items).
    - Subclasses include different player races like Shade, Drow, Vampire, Troll, and Goblin.
    - Each subclass may have unique behavior or passive abilities affecting the game.
    - Player has a pure virtual method RaceType() that prints out the race of the player.
    - Player also has setter methods for setting the coordinates and floor everytime these fields change.
- Shade is a subclass of Player. It overrides the Racetype() method. It is the default hero.
- Drow is a subclass of Player. It overrides the receiveEntityActions() and RaceType() methods.
    - The overridden receiveEntityActions() method provides a magnified effect of all Items by 1.5 factor.

- Vampire is a subclass of Player. It overrides the attack() and RaceType() methods.
    - The overridden attack() method provides a HP buff of 5 for each successful hit and reduces the HP by 5 if the enemy is a Dwarf, as Vampires are allergic to Dwarves.
- Troll is a subclass of Player. It overrides the receiveEntityActions(), attack() and RaceType() methods.
    - The overridden tickItems() method increases the HP of Troll by 5.
- Goblin is a subclass of Player. It overrides the attack(), receiveEntityActions() and RaceType() methods.
    - The overridden attack() method provides an increase in gold everytime an enemy is killed.
    - The overridden receiveEntityActions() method provides a 50% increase in the damage that is received by Goblin from an enemy which is an Orc.

Design Philosophy

GENERAL

- When returning vectors, the vector is passed by reference inside the method call instead of being specified in the return statement
    - It is still recorded as the “return type” in UML owing to it being edited as the “return” part of the method
    - Though this is a C-style convention, it is the easiest way to set the elements of the vector to what we want
- shared_ptr were used for this entire project instead of unique_ptr due to problems with the compiler and passing the pointers among various sections of the codebase having different responsibilities.
- The rand() and srand() function were used throughout to generate random numbers using the % operator so we would always get 0 or positive integers. Its application ranges from spawning mechanics to combat (enemies missing attacks).

DATA STRUCTURES (yx2wang)

- These serve as the main conduits of data between objects aside from primitive types
- As a result, they are built using structs instead of classes, because they are just a few fields strung together with no need for getters, setters, and fancy methods
- One alternative would be to use a fixed size array, however the code would be much messier and harder to read
- The two data structures, Coords and Effect, represent the two “units” of data that cannot be easily represented by a single primitive type (for example, HP, ATK, and DEF can be represented by ints, but Coords must have x, y, and chamber values at the same time)
- The choice was also made to include it as a separate “header only” section: it is used in every other file, and any methods that may be defined are helpers that need to meet the requirements elsewhere (thus they are put into the header instead of just declared)
    - This means that we do #include “defs.h” in every other header file

MAP LOGIC (yx2wang)

- The reasoning behind this section stemmed from it being the “link” between the Player and the Entity classes
- The player needs to sometimes only know what entities are around it, sometimes a render of the whole map is needed
- The Floor and Chamber classes are mainly based on the wrapper design pattern
    - The design ensures as much of the “dirty work” is hidden from other classes as possible
    - Players are provided with a pointer (or ownership of the shared_ptr, in the case of item pickup) for any relevant entity at every point in the game
    - They do not need to check where anything nearby is, or even know if they are “in bounds”, as there is a function to ask the map for these pieces of information
    - Entity spawning is entirely done by the map, as many checks before spawning must be done with knowledge of the map layout and anything present in the spawn cell
- The interactions between Player/Map, Player/Entity, Map/Entity are kept relatively simple as a result, since a lot less checking needed to be done in the other sections
    - This provided a very logical distribution of work for us, with one person on Player/subclasses, one person on Entity/subclasses, and one person for Floor and Chamber
    - The lack of interplay between the 3 sections of code meant that our group could work asynchronously and generally not come into conflict, very important since we had exams while working on it
- Another beneficial result is the minimal movement of memory: the game map and associated helper maps (see gamemap.h) are not present in either Player or Entity (which may have many instances), but rather are handled in the one Floor instance loaded at any given time, entity spawning and deletion are also handled efficiently since the map has a better idea of where everything is
    - One notable exception is the passing of player position to Chamber: because we don’t realistically see more than 4 people playing at a time, that vector of Coords will not get large at all, and our current way is the easiest to implement
- The methods used in the map classes are divided based on possible actions in the game: move, attack, get attacked
- Note that the map DOES NOT carry full player instances: because Player logic is pretty much unrelated to the functions of the game map (and instead mostly for acting directly on Entity), the issue of multiplayer position tracking was solved by keeping track of only each player’s coordinates, which are a much smaller data structure

ENEMY AND ITEM LOGIC (hbmahida)

- The Entity class works as a high-level wrapper class for all the entities spawned on the map apart from player as all of these have a type and coordinates.

ITEM:

- The Item class is a subclass of Entity that handles the effects of various kinds of potions using the Effect struct.
    - This design choice was made as it would be much easier for us to incorporate passive damage and slow healing into the game which requires us to rely on a tick count to ensure that the effects stop after a set duration.
    - We noted that the items applied a flat amount to the player, which is accomplished with simple addition/subtraction. Therefore, the added hassle of things such as decorators is not justified. Our method is probably the simplest way to implement what we need..
- The purpose of the getters in this class is to allow the player to know how much they have been healed or damaged by depending on potion type as the logic is handled by the player class.
- For convenience, we opted for treasure to also be an item and thus when a player steps on the treasure, they just receive the Effect struct with the money field equal to the treasure amount.

CHARACTER:

- The Character class is a subclass of Entity that has a wrapper class design and is supposed to encapsulate all the enemy types as all of them have the same fields: health, attack, and defense. They also require a position coordinate and a type and so they are under Entity class.
- Most of the methods in Character and its constructor are used to construct all the different enemy types. Some of the virtual functions are overridden by enemy classes such as Merchant and Dragon which have certain special requirements.
- Hence, as most of the code can easily be reused by all the enemy classes, keeping them all under one class made a lot of sense from a design perspective.
- Moreover, if we would like to add new enemy types with basic attack, defense, and health, it could be done very easily as we could just call the constructor and methods of Character and it would work exactly as intended unless the enemy has certain special actions which would require overriding the respective functions.
- The Dragon class has a pointer to the hoard it is associated with so that upon its death, unlock() method of the hoard (which is an item) is called so that it can be collected by the player.
- Merchant's logic was created primarily by keeping the DLC in mind which allows the merchant to sell the potions and hence he has a vector of shared_ptr of type item. Moreover, as all the merchants turn hostile upon hitting a single merchant, the field hit is a static inline bool.

DLC CHARACTERS:

- The Spitter has attributes as explained in the design overview. We wanted to create a character that would force one of the team players to take a loss by picking up the SpitBall left behind and lose a huge chunk of their health or let their teammates lose health by passive damage.
- The Spitter's SpitBall is the reason Effects struct was created the way it was so that the passive damage could be easily incorporated into the game together with normal damage.
- Spawner and Child are wrapper classes as explained in the design overview. Mandrake is a spawner which spawns Root on any square on the chamber it is in. VineHead is a spawner that spawns on the top-left corner of the chamber and then grows Vine in a random direction which is capable of blocking doorways out of the chamber.
- The purpose of the spawners was to give the players a bigger challenge as there would be more than one playable character on the floor. getNewChild() is overridden by both Mandrake and VineHead as VineHead randomly chooses one direction to grow its next vine in and Mandrake simply returns an invalid coordinate to instruct the spawn mechanics that the entity to be spawned is a Root.
- The frequency of spawning is different for both Mandrake and VineHead and thus both override the getFreq() virtual method. VineHead spawns a new Vine after a set number of turns while Mandrake has a probability of spawning a new Root after every turn.
- Based on the UML diagram, the two Spawners and Child subclasses: VineHead and Vine, Mandrake and Root follow a sort of observer structure with the design slightly differing from the actual design pattern due to the Root requiring a pointer to its Mandrake and Mandrake also having a Root pointer. The same applies to Vine and VineHead too. This design is required so that the Root enemies would die once the RootHead is killed.

PLAYER AND HERO LOGIC (s4khanna)

- The idea for making Player an abstract class was to encapsulate the inner working of the game from our main.cc file. The Player object in main.cc is responsible for controlling the flow of the game.
- Player class owns the Floor and has an Item. This makes it easier to produce the effects caused and update the duration of the Items.
- The Floor pointer gives us access to Entity class, as Floor has methods to access it. We get the pointer to Entity through the public methods of Floor. We know whether an Entity is an Item or a Character by calling the isItem() method defined in Entity class, which
- We update the health of all the Characters attacked through the player. When an enemy tries to damage the player, the player gets the information of all the entities within its range and reduces its

own health. This way we decreased coupling by not providing access to the Player class to the Character class.
- The fields of Player class are all protected to reduce code redundancy, as all the children classes will need to update the same fields. Hence it was not sensible to write getters and setters for all the fields. The encapsulation is still intact as the user can not change these fields manually.
- For printing the status of the player we have created a printStatus method, which prints out the playerActions and entityActions fields.

HEROES:

- The Heroes are a set of subclasses of Player, i.e Shade, Drow, Vampire, Troll, Goblin. They have their own details pertaining to attack, receiveEntityActions, tickItems and RaceType methods.
- RaceType is a pure virtual function that prints the race of the player.
- The playerActions and entityActions fields are reset after we print them out. We set these fields in the method calls for move and attack.
- Drow is a subclass of Player. It overrides the receiveEntityActions() and RaceType() methods.
    - The overridden receiveEntityActions() method provides a magnified effect of all Items by 1.5 factor.

- Vampire overrides the attack() and RaceType() methods.
    - The overridden attack() method provides a HP buff of 5 for each successful hit and reduces the HP by 5 if the enemy is a Dwarf, as Vampires are allergic to Dwarves.
- Troll is a subclass of Player. It overrides the receiveEntityActions(), attack() and RaceType() methods.
    - The overridden tickItems() method increases the HP of Troll by 5.
- Goblin is a subclass of Player. It overrides the attack(), receiveEntityActions() and RaceType() methods.
    - The overridden attack() method provides an increase in gold everytime an enemy is killed.
    - The overridden receiveEntityActions() method provides a 50% increase in the damage that is received by Goblin from an enemy which is an Orc.

Resilience to Change

DATA STRUCTURES (yx2wang)

- By design a data structure should only be included if it is widely used enough to warrant it
- Therefore, changes tend to be very rare unless there is some sort of large game mechanics change
- This means that this part of the program should survive unchanged through most game expansions

MAP LOGIC (yx2wang)

- Because of organization of methods based on actions, mechanics affecting those actions are only going to change the method that does that action
- Player/Entity interactions are handled by passing pointers from that entity back to the player, which means that the map has almost nothing to do with changes to either Player or Entity
- The only exception is spawning, which is handled by the map anyways (since there are certain caps on the amount of each entity in place, space constraints, as well as spawn chances per floor)
    - For spawning at the start, changes need to be made only in the constructor of Floor
    - For entity-based spawning (i.e. with the Spawner class), edits need to be made to the updateEntities() method in Chamber only
- Any changes related directly to the map (i.e. bottomless pits, quicksand etc.) can be handled in gamemap.cc and gamemap.h due to the file loading
    - Inside these files, it is only needed to define the effects on spawning and Player/Entity actions for each new tile
    - Maps are loaded from file, so where the tiles are located is changed by editing the map data itself
    - It is even possible to encode that information about tile effects directly into the map file: this is already done via the populated map testing system (which lets the user define entity spawning)

ITEM AND CHARACTER LOGIC (hbmahida)

- Due to the potions simply being Effect structs which are passed to player to be handled by its own logic, it is very easy to create a new type of potion whether single use or having an effect lasting multiple turns due to the design decisions made while creating the Item class.
- The Character design also ensures that any new type of enemy subclass can be created which would inherit all the methods from character such as getATK(). getDrop(), etc. This would allow us to create a basic enemy type with different stats quite easily. Even certain special attacks or new drop types upon death could easily be implemented by overriding the getDrop() and getATK() commands.
- Another strength of this is its easy integration into the spawn mechanics of Child class.

PLAYER AND HERO LOGIC (s4khanna)

- It is possible to add more races of Players, due to its abstract nature. We just need the attack, defense, starting health and the maximum health the race can achieve, to define a new race.
- The methods that are defined for the Player class perform the basic functionalities irrespective of the details that are specific to the race.
- If we want to add extra detail about health, attack or defense buff/nerf when interacting with an Entity, we just need to override our basic methods and add the details.

NOTE: There isn't a special section of the design document dedicated to the extra features as a huge part of the base game was done with the DLC in mind.

Questions:

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

Ans:

As there are only a certain number of interactions which every race has with its surroundings and as each race only has a special pertaining to one of its stats (health regen, effects buff) each race is very easily generated by defining a specific behavior for these interactions. Moreover, the 4 main attributes (HP, ATK, DEF, Gold) are the same for all races, meaning they are simply set to different values by the constructor upon character creation. Adding new races is extremely simple if they only change these 4 attributes, and it is a bit more complex if new abilities are introduced as new methods would be written for that character or existing methods would need to be overridden. Player being an abstract class makes it very easy to add a new race as all the attributes are inherited by the new race with only the relevant methods to be mutated as per requirement. Here, we are assuming that the basic movement and attack mechanics for these abilities are not going to be changed for game balance and implementation reasons (i.e. no moving two tiles at a time or ranged attacks).

2. How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not? How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

Ans:

We believe that the design philosophy together with the design overview provides a very detailed answer to the above question so I would save the reader some time by not reiterating the same points again for the generation of a new enemy type and its abilities.
The way a player character is created is similar to the way an enemy is created, i.e Player is an abstract class and the races are subclasses in the same way as Character is the abstract class and the enemy types are the subclasses. However, aside from their construction, everything else differs due to the completely different set of methods required by the player and the characters due to the design decision made to have the combat logic be handled solely by the player.

The way the various abilities for the enemies and player races are created are through the declaration of a virtual method in the abstract class which has a default implementation for the normal enemies or player races and then it is overridden in the specific races/enemy types having a different ability.
Any ability can easily be created as long as it has something to do with the health, attack, or defense of the player or character. It would be very hard to implement something like a ranged attack due to the sheer number of changes to be made to game logic.

3. 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 discussing the advantages/disadvantages of the two patterns.

Ans:

Since our design philosophy doesn't work with either of the two patterns, we do not have a new answer for this question and hence it remains unchanged from the one we submitted on DD1.

A decorator pattern would work here in the following way: Potions would be the abstract decorator class which would be a subclass of Player class and have the various types of potions as subclasses. The potions class would then have a pointer to the player class and depending upon the potion stepped on by the player (random if spawned or a specific one if purchased from merchant), it would have that effect on the player.
A strategy pattern would work here in the following way: Player class would have a pointer to potion class which is the strategy interface. This interface would then be implemented by the various subclasses of Potion which would be the different types of potions inflicting different buffs or nerfs to the player's 5 main attributes. So, the moment the PC walks onto a cell with a Potion, the program would decide on runtime, given the random potion which it would determine to use based on the random number generator and run the code under that potion type. We believe that strategy pattern would work better here because the decision to run the code of a particular strategy (potion) is taken at runtime.
Our implementation is kind of similar to the strategy pattern but we have extended it over the treasure as well, and implemented the whole thing with a datatype. Essentially, there is a data type with 5 modifiers (ATK, DEF, HP, Gold, Passive Damage) that is contained in every potion (similar to the different returns in strategy), with unused effects having the value of 0. This is put inside an item which also contains the duration. If the duration is 1, the item is applied immediately (i.e. the values are added/subtracted as needed inside Player). Otherwise, the item is used once, its lifespan decreased by one, then it may be again used during the next game tick for the same effect. During this time, the item pointer is stored within the Player class.
Our structure ensures that not storing every used potion is achieved, but also ensures that potions that have a time-based effect can also be implemented.

4. 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

Ans:

The main thing we learned was that teamwork was good to get more work done in less amount of time. We could write more code, have more inputs while debugging, and ask questions freely without the fear of getting Policy 71.
It was a completely new experience working with a team and it wasn't all that good. One thing which we had to always make sure of was to communicate any changes we made to the repo so that the others pull it before working on it and don't have incorrect methods or data types. However, this was a lesson learned through pain and losing time.

Moreover, it felt easier to work on this project as we knew what exactly we had to do and had discussed in advance, the role of each team member and the sections they are responsible for. It felt more structured and organized than solo work.
There were several problems too such as our inability to use GitHub properly, leading to working on a branch that is a detached HEAD or pushing at the wrong time and creating conflicts or working on the wrong branch and the list goes on!
One of the most important things we learned was how much the structure of a large project changes over time as people find new methods which they require from other classes and this leads to a lot of communication between different team members about the ways we can bridge the difference in the code vs expectation of the team. This helps to make the project as robust and streamlined as possible.

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

Ans:

- If we had to start over, we would actually try to learn git and GitHub properly before embarking on this project due to the sheer amount of time lost figuring out the intricacies of git.
- We would also start with more time in hand as it took longer for us to figure out the parts of the project and the role assignment took even longer, which led to lesser time coding.
- Asking for help early is also something we would have done. A lot of time was spent deciding on stuff ourselves which was later solved by TAs in the office hours in an instance.
- Understanding the recently taught content in more depth is also something we could have done. Due to slight errors/gaps in our understanding, it led to a lot of small errors happening in different sections of the project and thus having a better understanding would have helped us save a lot of time and unnecessary effort. However, we aren’t stressing it much as we learned through the experience.