Overview

- **Tiles, drawables**

We chose to represent our game state with Tile classes. Each Floor in the game is represented by a 2D array of Tiles, which are stationary and correspond to one of the 25x79 squares in the game. Each tile holds its own unchanging row and column within the floor. Each Tile holds a pointer to two Drawable objects (a lower and upper). These represent "things" in the game that can take the place of a tile. The lower Drawable is unchanging throughout the lifetime of the game, and it represents the underlying base of that tile. For example, the floor of a chamber, versus the empty spaces between chambers. The upper Drawable can change, and represents the object/entity that is on that Tile at the moment. For example, the PC, an enemy, a potion. Drawable objects are named as such because they are an abstract class. Every concrete Drawable implements the getChar() method, providing an interface that returns a char to be printed to the screen (ie. drawn). If there is an upper Drawable, the tile prints that char, otherwise, it prints the lower Drawable. Every object that is printed to the screen and potentially exists on a floor inherits from Drawable.

- **Creating Floors**

The data for our game is contained in the GameModel class, which stores an array of Floors representing the entire layout of the game, along with other game data. It also provides methods to generate the Floors from a file, or randomly. When generating from an array of strings (representing the floor, line by line), we iterate through the array character by character and create a corresponding Drawable object, assigning it to the respective Tile.
Drawables can be moved between Tiles, or removed entirely. We implemented the Observer pattern to control the removal of Entity(ies) from the floor (Entity inherits from Drawable and Subject). When an Entity wishes to have itself removed, eg. if an Enemy dies, or if a Treasure is collected, it calls Subject.NotifyDeathObservers(this), and passes itself. The GameLogic observes these Entities and will then look up the Entity's location in the floor and remove it.

- **Controlling the PC**

The PC is controlled by the GameLogic class that works with the Player class, which provides an interface to have the player perform actions, like attacking, gaining gold, modifying stats, special race attributes, etc. The GameLogic class has a reference to a GameModel, which always holds a reference to the player. The GameLogic reads player input, ie. a direction to move, an attack command and a direction to attack, etc. and performs the corresponding action. Because it has access to the GameModel, it can look up the enemy at the tile the player wishes to attack and call Player.useAttack(enemy) to attack the enemy. It also handles calling Tile.moveTo(new tile) to move the player to another tile, and maintains the state of GameModel, for example GameModel.currentTile, a reference to the current tile the player is on.
The primary game loop is controlled by the GameLogic class, which makes calls to GameView methods to display things to the user. For example the state of the floor after each move, and the player's stats.

- **Enemies**

GameLogic also handles when enemies can act. After the player moves, GameLogic goes through the floor and calls Enemy.act(player, playerTile). Enemy(ies) handle their own logic on what they wish to do, based on the player's position (which is retrievable via playerTile). A reference to the Player is passed in so that the Enemy can attack or perform other actions on the player. Since Enemy has reference to the tile it is on, it can independently determine where to move (has its own AI in a sense). Enemy decision making is handled by the shouldAttack(playerTile) and shouldMove() methods, which can be overridden to provide different behaviour in various Enemy subtypes. Many stats and calculations are the same between instances of players and enemies. These commonalities are reflected in the Character superclass of both Player and Enemy.

- **Potions, items**

Potions are initialized with a function callback that is a public method of Player, that the potion calls on use. Most items are also function callbacks, which provides some encapsulation as opposed to passing a reference to the entire object, since most items perform only a specific task. For example the potions given in the spec modify stats, and so they are initialized with a callback to Player's useStatPotion(statType, amount) method, which modifies the Player's stats by the provided amount. These stat modifications are modelled using the Decorator pattern, where each potion's effect is represented by a Decorator that increments the base stat by a certain amount. These Decorators can be popped and deleted when the player moves to the next floor, removing the potion's effects. The concrete Decorator simply returns the Player's base stat (which is passed as an anonymous function callback that returns the value of the player's base stat). Treasure is done similarly, with a callback to Player.collectGold(amount) instead.
The compass and staircase are initialized with callbacks to GameLogic.onCompassUsed and GameLogic.onStairsUsed(), which reveal the staircase and progress to the next floor respectively. Implementing items with a useEffect() method that calls it function callback makes it easy to introduce new items into the game as nothing else in the code needs to be changed - following the open/close principle.

- **MVC**

As already apparent from the above descriptions, we use the MVC pattern for the main game classes. GameModel holds the game state and some functions related to retrieving or modifying the state. GameView handles displaying the game to the user, including printing the current state of the floor, the player's stats, gold, race, and the current floor, the player and enemies' actions as a bonus. The GameLogic class acts as the controller, and handles the majority of the game logic at a high level. It also manages communication between the model and the view, denoting when and what to output to the user. It handles the majority of player input during the main game loop.

Updated UML

The UML stayed pretty similar between the first draft and final version.
Changes:

- Subject has two notify methods, notifyDeathObservers, notifyActionObservers instead of just one, and Observer has two notify methods, notifyDeath and notifyAction (to handle these two things separately). It would also be possible to define two different observer classes to handle each type, we chose not to for simplicity since GameLogic does both in our implementation
- GameModel handles floor generation instead of GameLogic
- GameModel stores a pointer to each staircase and each startTile in each floor, Floor no longer holds staircaseLocation
- Floor contains some helper methods to check for certain Drawable types in its tiles
- No need to store compass holder in Floor since it becomes an Enemy's loot
- Tile contains some methods to check if it is valid for the player or an enemy to move to it, and a helper function to return the char that should printed at that tile (based on its lower and upper Drawable)
- We now show where drawableToReplace is implemented
- Added more enumerations (PlayerRace, StatType, TreasureType) for const values
- Updated decorator to handle player stat changes, including the StatMultiplier class to handle multiplicative stat changes such as damage taken (used for Barrier Suit)
- Certain character methods now return strings which we use to display actions they perform after a turn
- Character now has onReceiveAttack following template method pattern
- Character now has didKill(Enemy), didKill(Player) following visitor pattern
- Enemy now has getDeathRewardGold
- In Player, replaced usePotion with useStatPotion since that's the only type of potion currently implemented
- Player.useAttack is public (following NVI)
- Player now has modifyStat()
- Added StatPotion as a subclass of Potion, made Potion abstract
- Updated Merchant to override onReceiveAttack and shouldAttack based on isAggro
- Dragon overrides shouldMove and shouldAttack

Design

A central design challenge we first encountered was how to represent a floor, and all the things that can appear in it. Essentially, how should we model the things that are displayed to a user every turn? We had a number of initial ideas, some being simply having a 2D array of "objects", equivalent to the Drawables in our current design, stored in a Floor class. The problem with this design is that relations between neighbouring objects are not well-known by each other. An enemy wouldn't know on its own which of its neighbouring tiles are valid places to move, whether they are occupied or not. An easy solution would have been to define various utility functions in Floor, such as checkValid(row,col), however this would require giving every entity that needed such information a reference to the Floor class, which seemed to involve a high

degree of coupling. Additionally, since the player can walk on passageways, doors, and floor tiles, we would need a way to model that as well, which might require two 2D arrays of Drawables. It seemed to make more sense to provide an additional layer of abstraction by creating the Tile class, which can have a lower and upper Drawable. Drawables that need positional information can be given a reference to their own Tile, but don't get access to the whole floor.

The idea of chambers was also another implementation problem. As an abstract concept (chambers are quite obvious visually in the game, however that is simply due to the floor layout), chambers are only really important in a few parts of the spec. Namely that enemies cannot leave the chamber they are in, the player and stairs cannot spawn in the same chamber, and that enemies/treasure/potions have an equal probability of being spawned in any given chamber (regardless of the number of tiles within the chamber). We felt it made more sense to model a Floor as the "source of truth" for the objects in the game, since each floor is printed in its entirety to the player, rather than by chamber. Thus we decided to model chambers simply as a 5-element array of vectors of Tiles, representing the tiles in the chamber. During generation, we randomly pick a chamber by choosing a random vector within the array, then choose a random tile in the chamber by choosing a random Tile in the vector (and removing it if something has been generated to occupy it). We prevent enemies from leaving their own chamber by constricting valid moves for enemy to only Tiles that contain no upper Drawable (which would represent an already occupied Tile) and a lower Drawable that has to be a FloorTile (the '.' char, though it's represented with its own class in our design). In this way, since each chamber is bounded by walls and doors, the enemy cannot leave. This seemed to be a standard way to delineate a chamber within a floor, and would be unlikely to change in the spec. (Otherwise, if two chambers could be separated by floor tiles, how would you know if such an arrangement was two different chambers with a hallway connecting them, versus a single strangely shaped chamber?)

Enemy actions were another challenge. We considered having GameLogic have greater control of the enemies, for example it could directly call Enemy.shouldAttack(), and if it should attack, call Enemy.attack(player) for example. However we felt this was too high a level of coupling, as the enemy's action control flow is essentially handled by GameLogic. We instead wanted the enemy to act "autonomously," and have it handle its own decision-making logic and actions. We created the act() method following the template method pattern. The enemy can either attack or move, which is dictated by Enemy.shouldAttack() and shouldMove(), which are virtual functions. This allows subclasses to have custom decision-making logic, which exists entirely outside GameLogic. GameLogic simply needs to call Enemy.act() on each enemy, and doesn't need to know how the Enemy handles its actions.

Another interesting problem was having enemies drop the compass (and also having Merchants drop a Merchant Hoard). There were many potential approaches to handling this, many of which were tied together with a specific implementation of enemy death and removal of Drawables from the map as a whole. We thought it made the most sense for Enemies to *have*

(and technically any Entity, although we don't use the same term) loot, which is a Drawable they drop on death, that replaces the tile they were on. Loot could be anything, including a compass, or Merchant Hoard, which made this design versatile. However we also needed a mechanism to handle replacing an Enemy with their dropped loot. Technically it was possible for the enemy to do it, since it has a reference to its own tile. However we wanted the Enemy to modify its own Tile as little as possible, with the exception of moving itself from one tile to another. Also, not every Entity has a reference to its own Tile, as that seemed like needless coupling. Instead we employed the Observer pattern, where each Entity inherits from Subject and Drawable. When it "dies," or needs to be removed from the board, it calls notifyDeathObservers(*this). GameLogic observes these entities, and when it is notified of an Entity's death, it will remove the Entity from the floor, and also call the Entity's drawableToReplace() method, which returns a unique pointer to the drawable that will replace the Entity that just died. GameLogic then handles placing the new Drawable in the same position.

Most of the races' special effects were fairly easy to implement, but the elf presented a unique challenge in handling new types of potions not given in the spec. Every potion in the spec is simply a numeric increment/decrement, so taking the absolute value is fairly simple. However not every potion is necessarily additive or even numeric. Some potions might not even have a negative counterpart at all, so how should those be handled? We considered creating a dictionary to map elements of the PotionType enum to their respective negative counterparts. However this meant that the implementation of the potion's effect would need to be entirely based on the value of the potionType (ie. Player.usePotion(potionType) is called, and elf implements a virtual method that preprocesses the potionType, remapping negative types to their positive counterparts, and Player.usePotion handles the effect based on the type. This seemed somewhat viable but also left the majority of the implementation of the potion's effect in Player, or some other helper class owned or possessed by the Player. This also meant that the negative to positive potion mapping needed to be updated for every new potion added. Although this wasn't necessarily a bad implementation, we instead chose to implement Potions in types. For example, every potion in the spec is an additive numeric potion that modifies a single stat. So we defined a type called StatPotion, whose effect calls Player.useStatPotion(statType, amount), which then calls player.ModifyStat(statType, amount), and the player can handle the modification of each stat. In this way, elf can be defined with an override for useStatPotion, to take the absolute value of the decrement/increment. New potion types can be defined freely with their own implementations, and Elf's implementation can override where needed, following the template method pattern.

Our random floor generation makes use of our generation from a text file, by essentially building a valid map as an array of 5 2D vectors of chars, each representing the contents of a floor. Building the map is relatively simple, and occurs in GameModel, which has an array of 5 Floor objects, which contain our representation of chambers as described above. During generation, we randomly choose a chamber in the floor, then choose a random tile in that chamber (via helper methods in Floor), and place the character corresponding to a given

Drawable at that location in the char map. We handle spawn weights via a dictionary (std::map) of chars to spawn weight (as an int, for example if vampire has a 3/18 chance, its corresponding weight is 3, and the sum of all weights in the dictionary should be 18, to give a 3/18 chance). We have a utility function that generates a random number from 0 to the sum of the weights, then converts that to the corresponding char (based on the sum of the weights of previous elements KVPs in the dictionary). Thus, to generate a random enemy for example, we simply use the weighted generation to generate a random char from the enemy dictionary, and place that in the char map. The from-file generation method handles instantiating the actual objects. Dragon hoards/barrier suits and Dragons get special treatment to ensure only one Dragon is within range of a ProtectedTreasure (superclass of barrier suit and dragon hoard). For the barrier suit, a random floor index is generated at the start. When generating that floor, the barrier suit is randomly placed on the floor.

Generating the compass is handled in the from-file generation method. We iterate by floor. We first count all the enemies in the floor since it can be arbitrary (assumed >0) when generating from file. Then we generate a random number between 0 and the number of enemies -1, which represents the enemy to hold the compass. We then iterate through each char in the char map and generate a corresponding Drawable to be placed in the Floor at the same row and column. When a Dragon is encountered, its neighbouring chars are scanned for a ProtectedTreasure, whose location (along with the location of the Dragon) is stored, so that it can be generated at the end (since Dragons take a reference to a ProtectedTreasure in its constructor).

To handle the gold the player gains on killing an enemy, we implemented the visitor pattern. This is because the attack methods are implemented in Character, so the player wouldn't be able to get a reference to the Enemy it killed without the visitor pattern (it would only be a Character, which might not have death gold). The player can then call Enemy.getDeathRewardGold() to find the amount of gold the enemy dropped.

Resilience To Change

The systems we have designed are built to be flexible to change, and able to accommodate complex new features.

- **Enemy (new behaviours, special effects)**

The Enemy class is designed to be very extendable and easy to customize in subclasses. You can define custom decision-making for when to attack and when to move by overriding shouldAttack() and shouldMove(). When an enemy attacks, a virtual method onAttack(target, damageDealt) is called. This can be overridden to allow for special effects of subclass enemy types. For example the Vampire stealing health, or the Goblin stealing gold. It would also be easy to create an enemy that never attacks, and Dragons are already an example of an enemy that never moves. It would also be possible to override DetermineMoveTile, which is called if shouldMove() is true in act(), to move in a different pattern (default is random valid tile within 1 block). It could for example, favour going in a certain direction, towards the player, etc.

Our Observer pattern with Entities also allows, for example, the Phoenix to hold an egg, or another Phoenix, as the Drawable that it gets replaced by when it dies, allowing it to be reborn. The act() function can contain a method like onAct() that is run regardless of the Enemies action to allow for invariant effects like the Troll regenerating health.

- **Player (stats, permanent/temp effects, races)**

Trivial changes to the spec such as a race's stats or the damage calculation formula are easy to change, as these are defined only once and re-used throughout the program (eg. the race's default stats are passed into the Player superclass' constructor for each respective stat, so simply changing the values passed in would change the race's stats, and no calls to the race's constructor needs to change).

The complexity of adding new races depends on the complexity of their race-specific effects, however many effects can be easily implemented through the use of the template method pattern by defining a new virtual method once in Player with a default implementation (for example in collectGold, call virtual method onGoldGained(amount), which does nothing by default). The subclass can then override the virtual method to perform its special ability (eg. heal for 2x the amount of gold gained), while no other subclass needs to be changed.

We modelled temporary stat modifications due to potions with the Decorator pattern. The stat modifications could be used for more than just potions (and in fact we also use it for the barrier suit as an example). Truly permanent changes would not be difficult, as we could simply modify the base value passed into the decorator (which is a variable). Adding effects to other stats is also not difficult. For example, adding a temporary potion that doubled gold gain could be done by applying the same decorator pattern to the goldModifier field of the Player (we chose not to implement this for every stat due to time constraints and because it adds unnecessary complexity in the given spec, but it is not a difficult change).

- **Potions (different effects, temp/permanent)**

Adding new potions with similar effects as the temporary ones given in the spec is trivial. It would be as simple as defining a new entry in the PotionType enum and changing the amount field of StatPotion. We also defined a Potion superclass that handles consumption in a generic way for all potions, but its effect can be overridden in various ways, which allows for more creative potions as well. The elf race's effect, that negative potions have positive effects, is somewhat vague. With that in mind, we allow elf to override useStatPotion to take the absolute value of its effect. If for example we wished to create a type of potion with a multiplicative effect, rather than additive, it seems sensible to define a different useMultiplicativePotion in player, that provides a default implementation for other races, and can be overridden by elf to take the reciprocal of the value when less than 1 (for example). Although this results in some additional implementation work if adding a new potion type, it leaves the majority of the existing implementation intact. (Regardless there is already a fair amount of implementation work required to create a new potion type).

- **Generation (# of enemy/potion/treasure, spawn rates, new enemies/potions)**

Small changes to the number of enemies, potions, treasure generated per floor, or the spawn weights of enemies (including new ones) is incredibly easy, these are all single-value changes. (# things generated is simply the bound in a for loop, spawn rates are defined by a dictionary from char to weight as described in the Design section, altering rate is as simple as changing the weight). Handling new enemy subtypes, potions, or effectObjects (like treasure) is slightly less trivial, as a case using the new Drawable's char needs to be added in the generation method, which handles calling the new class' constructor. For random generation, enemies/treasure/potions, they will need to be added to the respective spawn weight dictionary. New types of effectObjects will need to be handled separately (maybe a new dictionary, or if they are unique, simply choosing a random tile to place a char in, for which there are helper functions).

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?
Original answer (unchanged):

We would have a Player superclass (abstract), that has HP, ATK, Def, etc. and other functions that the player needs for any race. The superclass will have a constructor that takes in values for each of its stats. Each race inherits from this superclass. Its default constructor will call the superclass constructor with that race's default stats. We also introduce various fields for simple stat modification that can be set differently for subclasses, eg. gold multiplier for orc and dwarf, score multiplier for human. In the superclass, there will be functions for many of the basic game mechanics such as attack, getGold, usePotion. If we have subclasses with distinct behaviour for these mechanics, we can introduce virtual methods (preferably private ones to be called within these public methods).

Subclasses can override these methods to implement race-specific features/characteristics, which allows the game logic that calls these methods to remain the same, while the outcome changes. Eg. getGold will be called in the same way with a base value from a slain monster, but if a race wanted to, say heal hp upon gaining gold, we can add a virtual onGoldGained method, call it in getGold, and override it for the race subclass. A similar approach would work for the elf and reversing negative potions. (If all potions are strictly numeric, with negative potions having the opposite sign, we could introduce a virtual method called calculatePotionValue, pass in the initial value, and for elf, take the absolute value).

In order to add additional classes, a new subclass that inherits from the Player superclass would have to be created with a default constructor setting the race's default stats. In order to implement a special attribute, set the stat modification field if applicable or override the appropriate character function in the class definition. If it does not exist yet, the default behaviour for all other classes will have to be defined in the superclass (and if it doesn't exist yet it would also need to be added to external 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?
Original Answer:

The system should be similar but not the same for generating enemies. The player character and enemies share some stats, HP, ATK, DEF, that allows them to have a common base superclass (let's call it *Character*) with those fields. In addition, the player character and enemies all attack and take damage using similar formulas, which can be re-used by including them in *Character*.

Enemies have some game logic that players do not, in that they act after the player moves, and their choices are dictated by the position of the player. So we create an *Enemy* abstract class that inherits from *Character*, that all enemy subclasses will inherit from. Similar to player races, the *Enemy* subtypes will handle setting the correct base stats in their constructors.

In the constructor for *Enemy*, we pass in a reference to the tile it is on, since this is a single player game, so there will only ever be one player. This allows the Enemy to perform checks on whether to attack or move.

Note that the enemy manages its own current tile (which is a pointer, not a reference) when it moves (i.e. it calls Tile.moveTo(newTile) on its current tile, which moves the pointer to the enemy from the old tile to the new one. It then reassigns its current tile to the new tile).

Enemies have a limited number of choices, and have to act after the player. So we create an Act() method in the *Enemy* superclass that models the template method pattern that can be called from external game logic after the player moves. We pass in a reference to the Player and the Player's coordinates, so that the enemy can use that information when deciding how to Act. By default, this should be similar for all enemy types, in that if the player is not within a 1-block range, it will move, and if the player is in a 1-block range, it automatically attacks.

We can define virtual functions DetermineMoveTile() and Attack() in Enemy as well. (Attack() may not need to be virtual if all enemies attack in the same way, but needs to be for the following question). DetermineMoveTile() by default calls a method on Tile that gets the valid neighbors to move to, with restrictions to prevent the enemy from leaving the chamber (cannot move to a doorway or wall tile) and this logic differs for Dragons and specialized enemies. DetermineMoveTile does not exist in *Player*, since the user input determines where the player will move, so *Enemy* differs from *Player* in this way. These can be overridden in the different enemy sub classes.

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.?

Original Answer:

Since enemies always perform an action after the player makes an action, the Act() virtual function will always be called for all enemies after a player action. The Act() virtual function will include DetermineMoveTile() and Attack(), with preconditions shouldMove() and shouldAttack() respectively (virtual functions). These will have default implementations in *Enemy*, (enemy should attack if the player is within 1 block, otherwise move by default), but could be overridden for special abilities (eg. enemy can attack if the player is within 2 tiles).

We can define a virtual function OnStartOfTurn() to handle invariant actions that always occur at the start of the turn, like health regeneration for example, and call it in Act().

Health stealing could occur in the Attack() function. If the Attack() function returns the amount of damage dealt, it would be easy to override it to call the superclass Attack() function, then increase the HP of the vampire by the amount of damage dealt.

If gold steal occurs during an attack, it can also be done by overriding Attack() and calling a loseGold() method on *Player*. To make it a separate action, we could override Attack() and define a new function stealGold(), with separate conditions. Then make shouldAttack() the OR of the base attack condition and steal condition, then decide on which to do in Attack() itself.

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?

Original Answer:

We would use a decorator pattern to model the effects of temporary potions, which will take the base value and modify it, increasing or decreasing the value. This allows potions to be stacked, as each decorator just takes the previous value and applies its modifier. The concrete component will include the attack implementation with the default player base stat, while the potion effects are concrete decorators.

To reset the potions, all the decorators except the concrete one would be deleted, and the pointer in Player would point back to only the concrete decorator, which just returns the base stat value.

The player holds a unique pointer to the “top” decorator. We implemented deletion of decorators by having a decorator return its “next” unique pointer, and then assigning that to the player’s “top” decorator, thereby deleting the top one. The concrete StatViewer (non decorator class) returns nullptr, so we perform a null check and leave the pointer pointing to that non decorator after removing all the decorators.

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?
Original Answer:
Since every item exists in the map, they are all printable. We can create an abstract superclass *Drawable* that all *Character*s and *EffectObjects* inherit from, that just contains a field for the character representing that object (and an accessor method maybe).

Each floor will be a 2D vector of tiles, which each has a reference to up to 2 drawables, a bottom layer (floor, doorway, passageway, walls), and an upper layer (entities like walls, characters, items etc.), which determines what to draw at that location (upper layer takes priority). So when we print the board, we can just iterate through the vector and print each *Drawable*’s character if it is there (no drawable means print a space).

Our Floor class contains a vector of 5 vectors of Tiles that represents the open tiles (nothing on the upper layer) in each of our Chambers. Then to generate a random tile within that room, we generate a number from 0 to the length of the Vector<Tile> representing that chamber minus one, and the element at that index yields a valid tile. We can create a function to choose a random chamber and random tile within that chamber which can be re-used for each type of item. When we place a drawable at that tile, we remove it from the Vector<Tile> representing that chamber.

We can have separate functions to generate a random type of item, for each item category (eg. a function to generate a random type of potion, or random type of treasure) according to the probabilities of each type.
Instead of using a function, we simply get a char from the Dictionary using a helper function to use the weighted probability.
Then we can store that generated item as a Drawable pointer, and find a randomly generated tile within a random chamber to place it in. This can be repeated X times in a for loop, for the amount of that item category we want. Eg.

```
fun generatePotions(int X, vector<Tile> chamber):
For (i = 0 to X):
        Drawable* d = generatePotion()
        x,y = randCoord(chamber)
```

Dragon hoards and Barrier Suits will inherit from a superclass ProtectedTreasure. When a ProtectedTreasure *Drawable* is generated, a Dragon *Drawable* is generated immediately after. Dragon will be constructed with a reference to the ProtectedTreasure, and its coordinates. It needs the coordinates to override attackCondition() appropriately, to only be aggressive when the player is within 1 block of the ProtectedTreasure (and also within 1 block of the Dragon). Furthermore, it can call a ProtectedTreasure method unlockTreasure() when the dragon dies, which will allow the Player to pick up the treasure.
An additional change is that for random generation, we generate a char map similar to what would be present in file input, then use the from-file generation method to actually instantiate the objects.

Extra Credit Features

- Smart pointer memory management (no new/delete) - RAII

- Coloured output
- Seeded generation
- Action string + enemy string - Displays the player's actions and any interesting actions done by enemies
    - Also works with all the next bonus features
- Displaying potions surrounding player (within 1 tile), known/unknown potions
    - Once a potion is used once, it is known and its type will be displayed instead of "unknown" when the player is within 1 tile.
- Enemy special effects
    - Vampire steals life when attacking (100% of damage dealt, does not go over max)
    - Goblin steals gold when attacking (0.5 per successful attack, gold is added to Goblin's death bounty)
    - Troll regenerates hp every turn (does not go over max)

Final Questions

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?

It's really important for everyone to be on the same page in regard to the spec and our implementation, this prevents a lot of headaches and miscommunication down the road. It's also important to divide up work into manageable but distinct tasks that are segregated from one another to minimize merge conflicts and make sure things are still compatible when everyone merges. Communication is key. There were many times where one of us developed a helper/utility function to perform something that the other didn't know about, and tried to reimplement. Having documentation would also be helpful (though maybe beyond the scope of this project). One main reason our project went unexpectedly roadblock free was our extensive planning. Rather than just answering the given questions and mocking up our UML, we also had a separate 5 page document detailing how we would implement every game mechanic, following the flow of the game (race selection -> game generation -> etc.).

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

Obviously ideally we would have chosen our current implementation straight away instead of toiling and thinking through many others before narrowing it down. It would also have been nice to start earlier to have more time for bonus features. A minor inconvenience was that after we implemented a bunch of classes, it was getting harder to work since we had no file management. It would have been good to separate files into folders and configure our Make command to compile code into a separate Build folder. This way we also wouldn't have to run "make clean". Overall though our execution of this project was fairly fun and successful so there isn't that much to change.