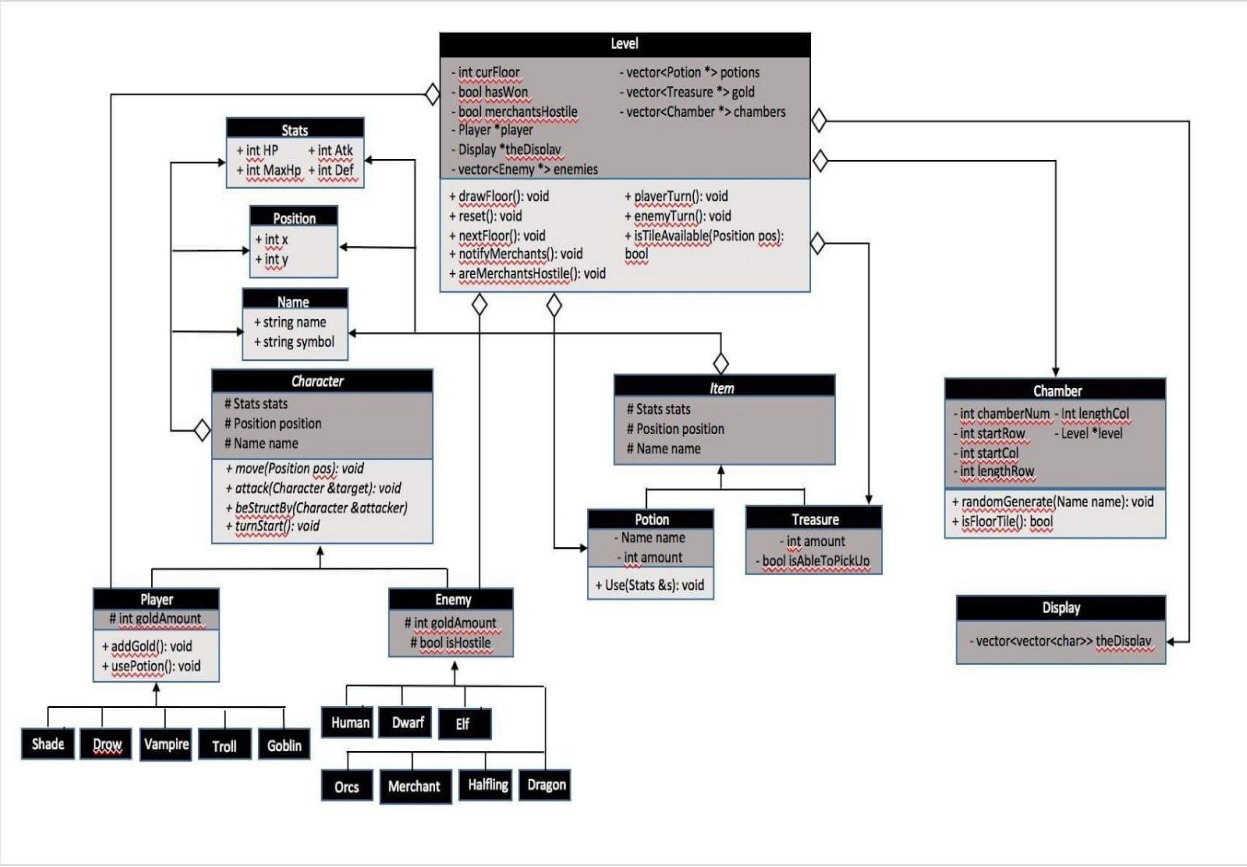
ChamberCrawler3000 - Plan of Attack

UML



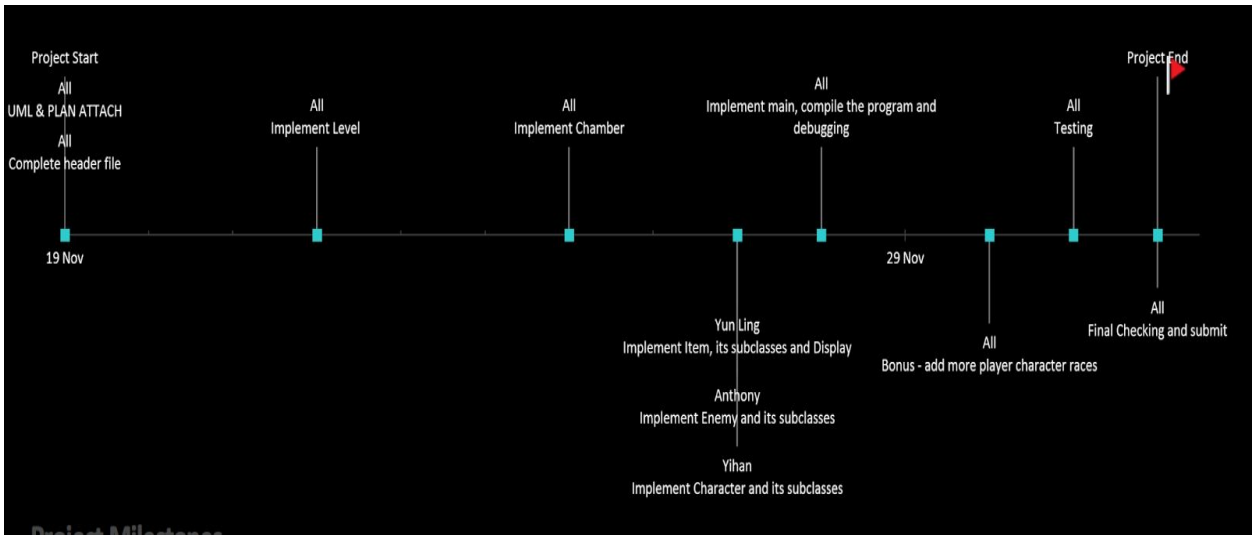
Breakdown of the Project

1) Interface
    - Structures: Stats, Position, Name
    - Classes: Level, Chamber, Character, Player, Shade, Drow, Vampire, Troll, Goblin, Enemy, Human, Dwarf, Elf, Orc, Merchant, Dragon, Halfling, Item, Potion, Treasure, Display
2) Implementation
    - Implement Level
    - Implement Chamber
    - Implement Character and its subclasses
    - Implement Enemy and its subclasses
    - Implement Item, its subclasses, and display
    - Implement main, debug
    - Bonus - add more player character races

Estimated Completion Dates

Project Milestones

Date	Breakdown
2016-11-19	Project Start
2016-11-19	UML & PLAN ATTACH
2016-11-19	Complete header file
2016-11-22	Implement Level
2016-11-25	Implement Chamber
2016-11-27	Implement Character and its subclasses
2016-11-27	Implement Enemy and its subclasses
2016-11-27	Implement Item, its subclasses and Display
2016-11-28	Implement main, compile the program and debugging
2016-11-30	Bonus - add more player character races
2016-12-01	Testing
2016-12-02	Final Checking and submit
2016-12-02	Project End



Individual's Responsibilities

UML & PLAN ATTACH	All
Complete header file	All
Implement Level	All
Implement Chamber	All
Implement Character and its subclasses	Yihan
Implement Enemy and its subclasses	Anthony
Implement Item, its subclasses and Display	Yun Ling
Implement main, compile the program and debugging	All
Bonus - add more player character races	All
Testing	All
Final Checking and submit	All

Questions & Answers:

- *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?*
    - Both the Player and Enemy classes are abstract classes of the base class, Character. We know that every character can move, attack, and do something (i.e. move/special effects) at the start of the turn, so this is why the Player and Enemy classes derive from it. For Players, the concrete classes are Shade, Drow, Vampire, Troll, and Goblin, each representing the race specified in the ChamberCrawler3000 document. We have another abstract function named, "SpecialEffect()", which allows the different races to have a strong level of control over its unique special ability. For example, the special ability for Drow is that it has a 1.5x magnifier for any potion it uses, and the special ability for Troll is that it regains 5 HP every turn. Overall, by this implementation, in order to add a new race to the game, we would simply write a class that inherits the Player class, and provide implementation for the abstract classes.

- *How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?*
    - The system for generating enemies is essentially identical to the generation of the Player. Class Enemy inherits the base class Character, and the specific types of enemies inherit the abstract class Enemy. Much like Player, this makes it easy to implement new enemies into the game: You just create a new class that inherits the Enemy class, and provide the implementation of its abstract classes. What is different about the Enemy class is mostly its fields. Enemy has a field determining whether or not it is hostile, it has a gold amount that the Player receives for slaying it, and cannot use potions, which is why it is a separate entity from the Player class. However, in general, most of the classes, Player and Enemy are identical in terms of the interface functions.

- *How could you implement the various abilities for enemy characters? Do you use the same techniques as for the player character races?*
    - Every race is a separate class that inherits from the Enemy class (which inherits from the Character class) In the character class, there are abstract functions that every concrete enemy must implement. That is: Move(), Attack(), TurnStart(), and SpecialEffect() (The special effect class is just for organization). This allows for us to dynamically design the special abilities to fit the requirements for any race by overriding the implementation of the base classes. It is implemented similarly for Players. For example, if we wanted to implement the Goblin's special ability to steal 5 gold from every slain enemy we would be able to without too much trouble. In the implementation of the attack function, we would attack the enemy as normal, determine whether or not the enemy has been, and if it has, call the SpecialEffects() function, which increases the Player's HP by 5. We think that this is an effective way of using special abilities because it allows for control and organization.

- *The Decorator and Strategy patterns are possible canidates 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 bet*ter? *Explain in detail, by weighing the advantages/disadvantages of the two patterns.*
    - The strategy pattern allows for us to augment the implementation of and object at runtime. In contrast, in the decorator pattern, we are adding to the implementation of an object at runtime. Therefore, the more appropriate design pattern to choose for ChamberCrawler3000 would be the Strategy pattern, because you will never have two potions mixed together (though that would be great DLC) In other words, you there won't be a time (according to the specifications) where we have

both a healing effect and a boost attack effect, therefore the Strategy pattern would be the more appropriate implementation. However, for the purposes of our project, we simply created a Potion class that inherits from an Item class and added a use () function that modifies the player's Stats (HP, Atk, Def). We believe that a heavy duty implementation of the Decorator or Strategy pattern is not necessary, and actually less intuitive for the scope of this project, which is why we did this. Still, we do not believe that it is that difficult to add a new kind of potion, as you simply need to pick a name for the potion, and implement it into the use() function.

- *How would 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?*
    - Both gold and potions are derived from a base class called Items. The Items class has fields that contain its position and its name and symbol. Since both gold and potions inherit from the base class item, we can easily associate the two objects through an item pointer *, and possibly implement an iterator to traverse through a list of items if necessary. This association by inheritance allows for us to reuse code and associate similar classes together for any potential DLC.