// Group Members

Harshith Depa (hrdepa)
Louis Bantoto (lcbantot)
Raphael Koh (rcykoh)

// Project Breakdown

July 13
- Finish Design Questions (ALL)
- Start UML diagram (ALL)

July 15
- Finish UML diagram (ALL)

July 16
- Finish Project Breakdown (ALL)

July 17 (DD1: 5pm)
- Implement Grid class (rcykoh)
- Implement TextDisplay class (rcykoh)
- Implement Entity class (rcykoh)
- Implement reading layout into floor (lcbantot)
- Finish command interpreter (hrdepa)
- Implement Character and all derived classes (hrdepa)

July 18
- Implement Item and its derived classes (lcbantot)
- Finish combat system (hrdepa)
- Implement Spawner and Factory class (rcykoh)

July 19
- Implement PotionSpawner (lcbantot)
- Implement TreasureSpawner (lcbantot)
- Implement player movement (hrdepa)
- Implement enemy movement (hrdepa)

July 20
- Implement Dragon placement (lcbantot)
- Design Bosses (lcbantot)

July 21
- Testing (ALL)

July 22
- Start implementing Inventory (rcykoh)
- Implement Boss base class (lcbantot)

- Start implementing Boss floors (lcbantot)

July 23
- Add soundtrack (rcykoh)
- Implement graphics display (hrdepa)
- WASD movement (hrdepa)
- Start implementing extra Boss subclasses (lcbantot)

July 24
- Test bonus content (ALL)
- Finish final report (ALL)
- Update UML (ALL)

July 25 (DD2: 11:59 PM)
- Finish testing bonus content (ALL)

// 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?

We use a base player class derived itself from a character class and have a derived class for each race. So we overload the attack method and constructor to give each race different stats. This makes adding new races very simple as we just have to add a derived race class.

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

We use a base spawner class to implement character placement because it is identical between enemies and the player. Our system generates 20 different enemies using the factory method pattern. This will be implemented via an EnemySpawner class. The player character is not randomly selected; the player chooses which race they'd like to play. This will be implemented via a PlayerSpawner class.

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

We use the same strategy as the player class. Use a base enemy class derived from the character class and then overload the individual attack methods for each enemy type. As with the derived player classes, use the constructor for each enemy type to create mobs with their stats. We use a base character class to implement enemies and players because they share movement/attack/stat properties.

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 weighing the advantages/disadvantages of the two patterns.

The strategy pattern allows us to change the implementation of a class at runtime whereas the decorator pattern allows us to augment or add to existing functionality at runtime. The key difference is that the strategy pattern *changes* the whole implementation while the decorator pattern *augments* the existing implementation.
The Decorator pattern would work better, because if the player picks up multiple potions, we can just augment the new potion onto the existing functionality. The Strategy pattern would not work in this case because it does not allow “stacking” of functionalities and we cannot predict the potions the user will pick up ahead of time, thus we cannot define appropriate “strategies”.

Pattern	**Advantages**	**Disadvantages**
Decorator	● Dynamically adds more functionality to an object at run-time	● ConcreteObject is unaware of additional functionalities
Strategy	● ConcreteObject is aware that multiple “strategies” exist ● Minimizes coupling	● Does not allow “stacking” of functionalities

How 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?

We will use the same factory pattern that we used to generate enemies in order to generate items. Items is the base class from which the classes gold and potion are derived. Further, we have individual classes for each potion and gold type derived from their base class respectively. So all we have to do is overload the probabilities for each class.