**Design Docs Assignment 3**

**Going to town:**

* A new method called addTown() was added to the Application class, which adds a new GameMap to the world (NewWorld). This method is simply used to draw a new GameMap and add items/Actors into it. It was implemented in this way to prevent the main method from doing everything. This also means that the town map could be extended and modified in order to add more interesting changes.
* The random generation of adding humans into the map has also been moved into a separate method called addHumans. This prevents the repetition of code, as we decided to add equal numbers of Humans to both maps. Hence, applying the Open-Closed Principle.
* Both Maps now have 1 tile allocated to a Vehicle (Ground). If the player enters this tile, they get removed from their current GameMap and get moved to another GameMap (map to town map or Vice Versa), costing them a turn. The Vehicle was designed this way in order to maintain as much simplicity as possible. (Liskov Substitution Principle)
* Since the player cannot stand on a Vehicle tile and remain on their current map, the Vehicle will teleport the player 1 tile next to the Vehicle in the other GameMap (radius of 1 tile). This prevents the program from looping infinitely and crashing. The tile that the player teleports will be chosen randomly, given that there are no other Actors on it. If all 8 tiles surrounding the Vehicle contain an Actor, the player will get teleported to the next available tile starting from the coordinates (0,0). The teleporting mechanics were implemented this way, as the probability of 8 tiles around the Vehicle containing an Actor each is very unlikely.
* These features were added and modified in a new class called NewWorld. This Class is an extension of World and is required to add features that rely on the actions that take place every turn. This class overrides and extends some of the features that run every turn of the game. (More information about this class is added below, for more relevant/specific features)
* The vehicle features were added through an extension of the ProcessActorTurn method in the World class. As if the player matched the conditions of teleporting, he would be moved to the new GameMap instantly, and his next action will be DoNothingAction. (Liskov Substitution Principle)
* When the game starts, both maps are active and running, as the Run method of the World Class ensures that all GameMaps ‘tick’. (Open-Closed Principle)

**New weapons: shotgun and sniper rifle**

* In order to make the shotgun and sniper rifle, a new RangedWeapon class was created. This class is a subclass of PortableItem and is specifically used to create weapons that can be used at a distance. This was done to follow the Open/Closed Principle so that we can add more ranged weapons, but its inherent properties do not change. This class holds two variables and four public methods.
  + Variables:
    - ammunition: holds the amount of bullets the ranged weapon currently has
    - fullCap: dictates the maximum number of bullets a ranged weapon can have
  + Methods:
    - hasAmmo: returns a boolean based on whether ammo is empty or not
    - reduceAmmo: reduces the ammunition by 1
    - getAmmo: returns an integer representing how many bullets are left
    - reload: resets the ammunition to full capacity
* These methods and variables were created to:
  + Check if the player can use their weapon at a given turn
  + Reduce the ammunition after a player shoots
  + Allow the weapons to be reloaded
* This in turn leads to the Ammo class. The Ammo class is a subclass of PortableItem and is associated with the ReloadAction. ReloadAction is a subclass of Action that allows the player to reload their ranged weapons (i.e. set their ranged weapon ammunition to the maximum capacity) given that they have an ammo item in their inventory. Once the ReloadAction is executed, the ammo item is removed from the player’s inventory. If the player has more than one ammo item, it will be able to reload proportional to the number of ammo items it has in its inventory.
* The SniperRifle class is a subclass of RangedWeapon. It can shoot any zombie within a 5 x 5 area, where the centre of this area is the current player’s position. If there are no Zombies nearby, the player can only aim. If their concentration level is full, it cannot use the sniper rifle. If the player shoots a zombie, the sniper rifle’s ammo is decremented by one and their concentration level is reset. Based on the player’s current concentration level, the sniper rifle will do more damage:
  + Concentration 0/2: 75% chance to deal 40 damage
  + Concentration 1/2: 90% chance to deal 80 damage
  + Concentration 2/2: 100% chance to instantly kill target

The SniperRifle holds one variable and three public methods.

* + Variable:
    - concentration: holds the current player’s concentration level. As mentioned in the assignment rubric, concentration allows to deal bonus damage based on its current level.
  + Methods:
    - canExecute: returns a boolean based on whether a player can use the sniper rifle this turn
    - setCon: set the player’s current concentration level
    - getCon: returns the current player’s concentration level

These methods were essentially used to:

* Check if the player can use the sniper rifle at a given turn
* Reset the concentration level when the player does anything other than aim or reselect their target
* Check if the player can aim
* The Shotgun class is also a subclass of RangedWeapon. The shotgun can be shot in any compass direction up to 3 blocks away. Doing so deals an area of effect damage in the shape of a cone in that direction. Any zombies hit with the shotgun have a 75% chance of being hurt for 35 damage. It holds a single public method and no variables.
  + Method:
    - canExecute: returns a boolean based on whether a player can use the shotgun this turn

This method was used to check if the player can use the shotgun at a given turn.

* Alongside these new weapons, corresponding actions for each weapon were added, namely the SniperAction and ShotgunAction. The purpose of these classes was to allow the player to use the weapons if they could be used (which is determined by the canExecute method for each weapon).
* The SniperAction allows the player to do (at most) one of three actions:
  + Select/change targets: choose a target that’s in the sniper’s range
  + Aim: increases the player’s concentration by one
  + Shoot: a chance to damage the selected target

If the player has not selected a target, then the game will only give them the option to either select a target or aim. Since all the zombies look the same on the game map, I have given relative coordinates to distinguish which zombie the player will choose as their target. For instance, a player may lock onto a zombie that is 3 units to the right and 2 units up of their position. If it has selected a target, the player can choose to change their target or shoot. If their concentration bar is full, then they can only reselect a target or shoot. If the player shoots, their concentration is reset and they lose a bullet.

* The ShotgunAction allows the player to shoot (given that they have enough ammo) to shoot regardless of whether a zombie is nearby or not. Doing so will potentially damage any zombies inside a cone-like shape of up to 3 blocks away. The player can choose to shoot in any compass direction they wish.
* Both the SniperAction and ShotgunAction have private methods in its inner workings for the actual program to function. These methods were set to private to add a layer of abstraction so that when someone actually plays the game, they do not need to worry about how the player locks onto the zombie, how the player shoots the zombie or whether the player can use its weapon at a given turn or not. Furthermore, these private methods were made to ensure that I did not repeat my code. In doing so, it makes it a lot easier to fix bugs when a problem comes up since you only need to edit the private methods, as opposed to editing every instance of when that block of code was called.
* All the variables in the ShotgunAction and SniperAction were made private to ensure that there were no privacy leaks.

**Mambo Marie:**

* The spawning/disappearing chance of Mambo Marie was implemented in the NewWorld class, by extending the StillRunning() method from the World Class. We added counters that keep track of the number of ticks and turns that relate to the spawn chance (5%) and the despawn turn. We implemented it this way instead of overriding the Run() method, as it shows more extendability to the World class.
* A variable was also used to ensure that the Mambo Marie respawning after 30 turns, is the same Mambo Marie (has the same health). We also prevented Mambo Marie from respawning after she is killed by the player by only letting her have a chance to respawn if she is still conscious.
* The MamboMarie class uses the summoningAction class in order to spawn zombies. This Action is called every 10 turns, given that she is on the GameMap. This method of implementation follows the Single Responsibility Principle. The class also follows the Open-Closed Principle as in future this action can be called by different Actors and is not limited to Mambo Marie.
* Zombies will spawn around Mambo Marie in a radius of 9 tiles. This is done to ensure that the game does not get too annoying/difficult from Zombie spawning in different corners of the Map. The addZombie method in summoningAction also returns a Boolean depending on whether the spawning of a zombie was successful. The spawning process is repeated until 5 zombies are spawned.  This maintains the Liskov Substitution Principle and Open-Closed Principle (extendability of the method/Class).

**Ending the game:**

* We added a new quitGame action for the player, which he can use anytime he chooses the option. This turn prints a relevant message and removes the player of the current GameMap (from NewWorld). Hence, this would stop the game. (Single Responsibility Principle). Although we could have overridden the Run() method in World, that would not show any extendibility to the World class and hence make it pretty useless.
* In the StillRunning() method of NewWorld we also added variables that keeps track of the amount of peaceful and hostile actors. If these variables got to 0 or 1 (as Player is Human), the game would end while printing a relevant message based on the win condition. Implementing this feature in the method ensures that the game will immediately detect if the game will end, as Still Running is called many times in the Run() method. (Open-Closed Principle and Single Responsibility Principle)
* The method to display the endgame message (in World) has also been overridden, in order to suit the condition in which the game ends (It should not say “Game Over!”, if the player wins). This implementation is based on the Open-Closed and Liskov Substitution Principle.

**Updates/changes made to other classes**

**Zombie limbs:**

* From previous gamplays, it was found that the weapon damage for the zombie arm, zombie leg, zombie club and zombie mace was lackluster given that the zombie has 100 starting HP, whilst the player only has 50. To help balance the gameplay, we decided to implement the following changes:
  + Zombie arm damage: 12 → 15
  + Zombie leg damage: 14 → 18
  + Zombie club damage: 14 → 25
  + Zombie mace damage: 17 → 30

Having such changes being implemented ensures that the player actually deals enough damage to the zombie before it dies.

**Intrinsic Weapons:**

* Bite and Punch are now subclasses of IntrinsicWeapon, rather than instantiating intrinsic weapons with such names. Doing this once again ensures that we do not repeat ourselves.

**Improvements to Engine Class**

**Positives**

* The IntrinsicWeapon and WeaponItem class had good design because they followed the Liskov Substitution Principle. Despite having very similar code, they did not use inheritance for the sake of reusing code. Furthermore, these classes also follow the Dependency Inversion Principle by having IntrinsicWeapon and WeaponItem implement a Weapon interface.
* The Actor’s inventory is designed well, such that the user can add or remove items from it, without getting access to it. That is there is good privacy protection in this area, as the getInventory() method returns an unmodifiable List, instead of the original. Hence, correctly following the Open-Closed Principle.

**Negatives**

* There were multiple cases of privacy leaks in some aspects of the Actor class. (*Privacy Leaks in Actor Class*)
* There are limitations to extending the Run() method in World Class. (*Extendability of the World Class*)
* The Item class should have made its variables private to help reduce privacy leaks. Furthermore, the returned values for methods such as getDisplayChar() and toString() should be returning copies of the private variables, not the private variables themselves. On that note it’s also important that additional methods are added so that these variables can still be utilised. For instance, an addAction could be added to this class so that it’s allowableActions can be extended. This is better as opposed to doing something like: “super.allowableActions.add(newAction)”, as it adds a layer of abstraction.

**Extendability of the World Class:**

* Currently there is no method in the World Class that is called exactly once every iteration of the loop in the Run() method (while (StillRunning())). This adds restrictions while extending the game.
* Since calling super.Run() will not be of much use, as the game ends once the Run() method is called, there needs to be an easy way to extend the Run() method to add additional features into the game that requires an extension of the World Class.
* An example of this issue is: In order to implement a way to keep track of Mambo Marie’s spawn rate, we had to add additional counters while extending StillRunning() method to ensure that her chance to spawn is only checked once per iteration (tick). This has to be done to keep her 5% spawn rate consistent. This same issue will apply while adding other Actors that have similar properties as MamboMarie.
* Although this means that the Run() method is extendable, it is definitely annoying and not ideal. It also does not entirely follow the Liskov’s Substitution Principle and Open-Closed Principle.
* This issue can lead users to naively override the Run() method in an extended class. Hence, cause them to repeat a lot of unnecessary code, as most of the Run() method cannot be replaced in order to maintain the functionality of the program.
* One solution could be to simply add a new method that has no initial functionality given. That is to create an empty method in the World class and add a line in the while loop (while StillRunning()) of the Run() method, to call this method. The advantage of this method would be its very open extendability, while also preventing any direct modifications to the Run() method (Open-Closed Principle). The main disadvantage of implementing a method like this, would be it’s uselessness before being extended. Hence, it would be an unnecessary method unless the Run() method needs to be specifically extended.
* The optimal solution would be to move some code in the current while loop of the Run() method into another new method, and add a line to simply call this method. This would fix the previous issue about having an unnecessary method, while also following the Open-Closed Principle.

**Privacy Leaks in Actor Class:**

* There are few variables in the Actor class that are prone to privacy leaks. The getDisplayChar() and toString() methods return protected variables. This enabled the user to modify these values externals.
* In order to avoid any privacy leaks, the majority of the protected variables in the Actor class, needs to be set to private.
* On top of this, methods such as getDisplayChar() and toString() should be returning a copy of these variables to once again reduce privacy leaks.
* The disadvantage of this would of course be the lack of extendability of variables such as the Actor’s max health, which is required for most classes that extend Actor. This issue can be solved by leaving the variables as protected and simply returning deep copies of these variables in getter methods.