**Design Rational for Assignment 1 & 2**

1. **ZombieAttackActionclass**

This class is created that extends the class to accommodate the need for zombie to decrease their probability of punch when they lose one or both their limb. Modifying the code at AttackAction class has a high risk of breaking it, because some major changes are needed in the execute method, making a new class shorten the code. The reason we have not created an AttackBehaviour class for zombie is that, if human class were to be able to attack in the future, a simple if and else statement to return action from different attack action class( AttackAction and ZombieAttackAction) is all that we need, hence we think that making an new class for that is unnecessary and will create a lot of redundant code.

1. **WeaponLeg and WeaponHand**

This class is created to create a weapon obtained from fallen zombie’s limb. It inherits the WeaponItem class. We created this class, to make it easier to create an instance of this weapon every time a zombie’s limb fall off. This allow us to instantiate this item onto a location of the game map easily as well as allowing any actor to pick it up as a weapon. It will be used in the in the *hurt* method in the zombie class, where every time a zombie is hurt, it will call a method from limb class to have a probability of knocking off a hand

1. **BiteAction**

A new class for bite action is created and it will be implemented inside the GetIntrinsicWeapon method to add the probability of having a bite attack. The reason it is added here instead of adding it to the weapon because biting it’s an attack from a part of the zombie (i.e. not weapon), hence its more suitable to have it in the *GetIntrinsicWeapon* method. This class will increase modularity and extensivity of the code, where if any other actor requires the bite action, we can reuse this class.

1. **Zombie class**

The zombie getIntrinsicWeapon method is overridden to return two possibilities of intrinsic weapon which is bite or punch. We decide not to create a new class for this attack action because, we think that no other future implementation of this method is required. Hence, a one line code of adding another intrinsicweapon class instance is enough for now.

The probability of the zombies saying brains is implemented in the playTurn function where in every turn it takes it will have a probability of saying ‘Brainns’. This is implemented in playTurn instead of making it an action because, in every turn an actor can only take one action. We decided that we should not waste a turn for zombie just to say ‘Brainns’, so at the start of each turn the zombie will have a 10% chance of saying ‘Brainns’.

1. **Limb class**

A class limb that keeps track of the number of limbs a zombie has left is created instead of putting a counter in the zombie class. This follows the design principle that ‘classes should be responsible for their own properties’, where in this case, the purpose of this class is to keep track of a zombie’s limb. This makes future code refactoring easier.

This class is also responsible of knocking off a zombie arm, because the number of limbs of a zombie is kept here, hence modifying it will be easier. In addition to that, a method is created to have the probability of dropping weapon the zombie is holding whenever the function to knock of zombie arm is called. Again, this follows the principle where ‘class should be responsible for their own properties’ because everything related to zombie’s broken limb a placed inside this class. This class also have a method called canMove to check if a zombie can perform any move action in this round if it has a broken leg. It keeps a counter of tick to determine how many round this zombie hasn’t move.

1. **RandomGenerator class**

This class is created to generate random output based on the probability given on each possible event. This class is modularised because it’s required by more than one class e.g. (ZombieAttackAction class and ZombieLimb class). This will reduce repetition of code.

1. **Shotgun and Sniper**

In order to implement shotgun in the game, 4 additional class were added to the project

which are the ShotgunShootAction, ShotgunMenu, ShotgunUtil and Shotgun class.

ShotgunShootAction, ShotgunMenu and ShotgunUtil are created as the utility classes for shotgun where ShotgunUtil is the function to determine the area affected by the shotgun while ShotgunMenu is created to provide a submenu for user to choose the direction of where they wanted to shot, and ShotgunShootAction implement the action based on the user choice and area calculated in ShotgunUtil.

These classes were created based on the Single Responsibility Principle where each class should have their own purpose, in this case ShotgunShootAction is to implement the action chosen by user, ShotgunUtil is to determine the affected area while the ShotgunMenu is used to create a submenu for the user. This design also implements the Open/Closed Principle because if we decided to modify the area affected by shotgun, we can simply change it in the ShotgunUtil class instead of modifying the ShotgunShootAction if the calculation to determine affected area in done in ShotgunShootAction.

For the sniper rifle, 5 classes are also added for sniper rifle which are SniperUtil, SniperRifle, SniperMenu and SniperShootAction and AimAction. SniperUtil are created as utility class to help calculating range of the range as well as calculating a straight line between target and the user. AimAction action is performed if player choose to take a round to aim at the target. SniperMenu is used for interaction with users to allow user to choose different action based on the number of rounds used for aiming and SniperShootAction are used to implement the shoot action when player fires the shot.

Using these 4 classes we can achieve delegation of one sniper shooting action into several subclasses of sniper shooting action using the Single Responsibility principles where each subclass is tasked to perform a subclass for the sniper shoot action. This allows us to perform testing on each subtask more easily as each class works independently from each other.

I have also used the dependency injection technique to allow me to change the aiming status of the player in the player class from the SniperShootAction and SniperMenu class. This allows us to change the aiming status of the player effectively from other class while performing other action (e.g. shooting, aiming).

1. **Travelling between map**

To implement travelling between maps four additional classes are added which is the

Town, Compound, Vehicle and TravelAction as well as an abstract class Level.

Level are implemented to apply the dependency inversion principle where every additional map added to the game in the futures can inherit from this class and only change the loadLevel method within to design different maps. This also implement the open close principle where additional map in the future won’t have to change any codes within to add a level but simply create a class inheriting Level to design the level.

Vehicle and TravelAction are implemented to allow user to choose the action to travel to other maps. The vehicle is placed automatically on the map based on the index of the map in the world. Vehicle are placed on the upper left of the map if the current map is the first map and conversely, vehicle are placed on the upper right if the current map is the first map.

Vehicle are placed both on the upper left and right of the map if the current map is a map in the middle of the world.

1. **OnDead**

This class is refactored from the execute method of AttackAction that determines what happened when an actor died. The reason this part is refactored is to reduce repetition when different type of attack kills an actor (e.g. from Shotgun from Sniper). This also somewhat implemented the dependency inversion principle, that says higher level module shouldn’t depend on abstraction. In this case although OnDead is not an abstract class, having this class allows me to change what will happens to an actor in this one place instead in all the attack classes that can kill an actor.

1. **Corpse (Rising from the dead)**

A corpse object from the Corpse class is created when an AttackAction from the zombie class is acted upon Human classes from its playTurn action. A Corpse class is extended from PortableAction class because from the current game design and forum discussions, it has been clarified that a corpse object should be portable. After 10-20 rounds based on some probability calculations, it will create a Zombie object. The reason for having a Corpse class is to allow reusability of multiple Corpse object to have its own individual ticks. By implementing changes in AttackAction, classes that extends AttackAction will create Corpse object when the actor is unconscious, this reduces repetition of code (DrY).

1. **CraftAction, ZombieClub, ZombieMace (Crafting weapons)**

From the current source files, we found out that many of the class objects were created as a result of actions such as corpse object being created from AttackActions class. To justify our design reasonings, we have decided to create a new class CraftAction for extensibility purposes. If we decide to implement other weapon that can be crafted into other weapons, then we can simple reuse this class or in other word, we reduce code repetition.

ZombieClub and ZombieMace are extended from WeaponItem class. The creation of these class allow us to instantiate a new instance anywhere on the map easily and we decided not to implement it in the zombie class, to adhere to the delegation principles to avoid having a long zombie class.

1. **Farmer, FarmBehaviour(SowAction, FertilizeAction, HarvestAction), Crop, Food, HealAction (Farmers and food)**

We have decided to create several new actions for the Farmer and Player class. SowAction, FertilizeAction and HarvestAction are extended from the Action class and is used to interact with Location of Ground object to produce Crop objects. We have allowed only the Farmer class to have FarmBehaviour which consists of SowAction, FertilizeAction and HarvestAction instead of implementing multiple behaviour classes to reduce dependencies (ReD). Player will have access to HarvestAction which can be retrieved from Crop.allowableActions(). This allows reduced code repetition (DrY) for checking as allowableActions does the checking of crop riping, it also reduces dependency (ReD) because allowableActions creates the new HarvestAction. If there were to be multiple Farmer objects, they would reuse this behaviour and I will not have to repeat any codes (DrY).

A SowAction will have a 33% chance of creating a Crop object where the farmer will sow a seed onto nearby dirt that is unoccupied by Crop object. A SowAction will have an attribute of Location to determine the location of Sowing of new Crop. This allows me to utilize Fail Fast design if a Crop object should not be sown where the instance of ground is not Dirt.

FertilizeAction class is created to allow Farmers to fertilize the crops and reducing the time left to ripe by 10 turns. This class has no attributes or dependencies as it is not required. It will only increase the duration time of the planted crop.

HarvestAction class is used to act upon Crop objects that are ripe and will return a new Food object which can be used by HealAction class to heal an actor. The attribute is the Location class of the Crop object to determine where it should be harvested. The dependencies of the class is unable to be reduced as new Dirt and Food objects has to be created as a result of harvesting ripe crop.

The Crop class is extended from Ground class as it contains many of the required methods to determine whether the crops is ripe from the tick method. Other methods such as canActorEnter() and getDisplayChar() is useful for UI purposes. A crop has dependency of returning a new HarvestAction when the Crop object is ripe.

Food class objects are created from HarvestAction by Farmer and Player classes and is used for healing by HealAction. They are extended from PortableItem class as the Player should be able to carry the Food object in its inventory. Each instance of Food object will create a HealAction when it is called using getHealAction(). A HealAction is only created when the Human class calls getHealAction().

To allow Humans and its extended class to be able to pick up Food and heal, we have created 2 new class methods in Human class which checks the location of Human and whether they could call HealAction. By implementing the 2 new methods, we are able to reduce repetition of conditional checking in the playTurn method for Player and Farmers.

We have decided to include many new classes to allow further extensibility to the game engine such as having the possibility of having multiple new Crop objects implementation which produces different kinds of Food object that heals for a different amount. Attributes are assigned to the newly introduced actions to allow Fail Fast design implementation.

These designs also reduce dependencies as much as possible, for example, method that are used by zombie only are implemented inside zombie class. We also try keep our code simple and clean to increase readability by splitting the longer method into smaller sub – method whenever possible.

1. MamboMarie

To implement MamboMarie into the game, we have initially decided to remove and add her every time it hits the 5% counter. As we proceed with that idea, we found out that it was impossible to process her counter within playTurn as world.run() was an infinite loop which processes every actor’s turn based on the actors on the map. Hence, we chose to implement MamboMarie as a disguisable character. She would be spawned into the world but disguised as a random ground towards the edge of the map. By implementing it this way, we were able to implement some custom dialogues If she doesn’t reveal herself (failed 5%). There is a con to this design as a random spot on the map would be unable to be travelled by the player as the game rule specified that a ground can only be occupied by an actor.

Otber design choices while implementing MamboMarie included a behaviour for her at first as she would have many actions based on whether she would reveal herself, chanting, etc.

However, this would be unnecessary and would create needless dependency which is a bad design choice. Hence, we would use conditional statements to decide on the actions that she would perform based on the game situation.

Finally, to reduce repeated code for her spawning and relocating at every 30th turn, we chose to implement a new class that acts as a utility class for MamboMarie. The new class is called MamboSpawn. Since we would need to get the x and y locations for mambo to spawn on (edge of the map), we would also need to check if there are any actors at the provided location. This would cause repeated codes every time mambo is needed to spawn or relocate on the map. By doing the calculations internally within a method by creating an MamboSpawn object, we could easily spawn and relocate mambo without any repeated codes.

1. **Quitting game and Player win/lose conditions**

After MamboMarie, we realized that we could have refactored the world class and re-design it to implement Mambo which would allow removing and re-adding of mambo. Refactoring world class is needed as the current world class is an engine file and it would be a bad choice to change code within the engine file. The reasoning to refactoring it is to perform checking if all humans or zombies and mambo is dead to determine the player’s win condition. By refactoring world class into NewWorld and implementing minor tweaks to the method which process actor, we were able to perform checks on the actors to see if all zombies and mambo is dead or all humans are dead. Refactoring also allowed us to check every action that was performed by the player which would allow me to determine if the player had quitted. To allow the player to quit, we have implemented quit action with hotkey ‘q’ which breaks the entire while loop causing the game to end. There were not many design choices to make as most of the implementation only required conditional statement.