Design rationale

In this design, we will extend the functionality of the application by adding new classes, extend existing classes and make the new classes interact with new/existing classes. This design will not be using any instanceof, casting or downcasting an object to implement the classes. We have implemented extra methods in the Actor, Item and Ground interface to prevent any downcasting references in the game.We do this without modifying the code inside the engine class. This will help to reduce dependencies between classes as it does not have to use instanceof function to check if the object is an instance of a specific type. We have created a common method inside the class which can be used multiple times within the class to make sure we do not repeat ourselves and keep it in a good design principle.

Zombie class

For the Zombie class, we add the ZombieArms and ZombieLegs which stores the zombie body parts inside an array list. It's arms and legs which can be dropped if an attack is successful on the zombie. These Arms and legs can then be picked up by the player and be crafted into the weapon by the player. We have created classes for zombie legs and zombie arms which will be a portable item so that the item can then be used to craft weapons to attack the enemy. The zombie class has a base damage defined inside the class which will then be passed down to attack behaviour class and attack action class to determine the types of attack. Furthermore, the conversation list, zombie_convo list is created inside the zombie class to store the different types of talk that can be spoken by the zombies. We created a check weapon behaviour class to determine whether the current location of the zombie contains a weapon for the Zombie to pick up. It will then call the pick up item action when the weapon exists at the location.

Therefore, the implementation of this class will fulfill the Don't Repeat Yourself (DRY) design principle because we can just use this class to keep track of the body parts for each zombie object. It does not have to create the zombie arms and legs or uses another class or methods for each zombie body part. The variables are all private and protected so that it can protect its value when the program runs. Other classes will be required to use a method to access the values of the variables. This will helps in data hiding such as not allow other classes the access the value of variables directly.

Extend Action class:

CraftAction class

A human will craft a limb from its inventory into a weapon, an arm can be crafted into a zombie club while a leg can be a zombie mace. The class will determine the damage of the newly crafted weapons which can be crafted into zombie club and zombie mace. The class will also check if a player has a craftable item in inventory and upgrade the item if the player decides to upgrade it. This class fulfils the Don’t Repeat Yourself (DRY) design principle because we don’t have to rewrite the code when we want to craft a new weapon. We can just call the class or use the methods from the class to craft the weapons. This class will create another class called ItemCapability to determine if the item is craftable before allowing the player to access a craft action class to craft a weapon. It will check if the conditions meet in the player class such as checking through the whole player’s inventory to check if the zombie is able to craft a new weapon. This will help to reduce dependency between the player class and the weapon class if the player directly checks the craft action inside the player class to create new weapons.

FarmingAction class

A farmer will decrease the time left to ripen by 10 turns. This class will implement methods to decrease the crops time to ripe inside the crop class. Therefore, this will give the FertiliseAction class a dependency to the crop class. This class fulfills the Fail Fast design principle because the system will fail immediately and visibly when the farmer class or fertilise action has an error. It will be easier to find the errors out too as we get to know which part of the crops part have errors instead of having all the functions/methods all in one crop class. This will make it harder to catch or identify the error and it might even cause the system to crash. This class also sow the crops by first checking if the actor such a farmer is allowed to perform such action inside the farming behaviour class. Another reason for creating this class is to perform two actions such as sow action and fertilise action so that it can reduce dependencies because if an extra class is needed, then it will be required to call the class separately. As only farmers sow and fertilise the crops, therefore, we have chosen to implement the class in this way.

HarvestAction class

A farmer or a player harvests the crop when standing next to a ripe crop. This class will create methods that harvest the food for the player and place it into the player’s inventory. The player will then be able to consume the food by accessing its inventory. This class will have another method to harvest the crop and drop it on the location. This class fulfils the Don’t Repeat Yourself design principles as it can reuse the code for both players and farmers to harvest the crop instead of just creating the method on both the classes.

EatAction class

This class implements the eating behaviour for the players and the humans. It is called by Eat behaviour class and if it’s a human it will heal them and if it’s a player, it will also heal them but can add the food to their inventory so that at a future time the player can consume the food. This class fulfils the Don’t Repeat Yourself design principles as it can reuse the code for both players and farmers to eat the food instead of just creating the method on both the classes. It also has a single responsibility principle which promotes code organization, less dependencies and easier maintainability.

ConvoAction class

Stores an arraylist of zombie words and prints them out with every zombie turn. It randomly chooses a slang with a 10 percent probability of screaming. We implement this class because we can reduce the use of literals inside Zombie class. This reduces the risk of data variables being modified.

AttackAction class

This class is used to perform different types of attack actions for zombies such as bite attack action or normal punch attack action. The bite attack action has a lower probability of hitting the player compared to the normal punch attack. We have implemented the weapon inside the class to get the weapon damage. We have also implemented a few methods inside the class to be called in the execute function so that it will not repeat the code. Therefore, this proves that it fulfils the Don't Repeat Yourself designing principles as it does not repeat the code to do similar actions. The reason why we decided to use methods to perform some functions within the class is because firstly it can help with reducing repetition and next is because the code has similar actions other than the probability of the attack. The class has a method for dropping zombie parts too to drop the zombie legs and arms and it can also reduce repetition. In order to split the zombie legs and arms up, we will be doing it in two classes, attack action class and zombie class. If the zombie leg is amputated, it will add a leg counter in the zombie class and make the zombie lose a turn during its next turn. This will be done by allowing the first two behaviours in the behaviours lists to be runned and the other two behaviours such as hunt behaviour and wander behaviour will not be performed. We drop the arms and legs at the player's location so that the player can pick it up. The reason for this is to allow the player to pick up the zombie arms and legs to craft it into a weapon because it takes 2 turns for the player to craft and get the weapon. Therefore, we think that it's more fair to just drop the legs and arms at the player's location.

Extend Ground class:

Crop class

Farmers will grow crops on dirt when the farmer is standing next to a patch of dirt and the crops can ripe when the farmer is standing on them, they also can be harvested for food by the farmer or player. The farmer can sow and fertilise the crop while both farmer and player will fertilise it. Crops extend from the ground class and inherit the features of the grounds implementing abstraction. To create new crops we do not have to rewrite the code such as creating a new method in its class hence the design principle is Don't Repeat Yourself.

Extend the Item class:

Food class

After the crops are harvested by the farmer or the player the food will be created on the ground. Food when consumed by a player or human will increase health. This helps to reduce the dependency on crops class from human and player as they rely on food for health. Design principle is don't repeat yourself and it will create a new food object if this event is encountered again.

Extend Behaviour class:

FarmingBehaviour class

A class that generates a FarmingAction if the Farmer is standing on an unripe crop or near a patch of dirt. Farming Behaviour is used by farmers to sow and fertilise the crops. Design principle fails fast as the FarmingAction will not get called if this event returns a negative response encountered again.

HarvestBehaviour

HarvestBehaviour class generates harvest action and checks if the farmer or the player is standing next to a ripe crop and if it then it calls the harvest action. Design principle is don't repeat yourself and it will call the harvest action if this event is encountered again. Another reason why this is having Don't Repeat Yourself design principle as is it allows both player and farmer to harvest which is better than implementing the harvest method on both classes and required to implement it twice causing repeating the code.

EatBehaviour

A class that will help the player or the human to identify the food on the ground and will call the EatAction to consume the food. This class fulfils the Don’t Repeat Yourself design principles as it can reuse the code for both players and humans to eat the food and also reduces dependencies between player and food.

ZombieAttackBehaviour

AttackBehaviour class will decide whether to do the intrinsic attack or the bite attack. It generates a BiteAttackAction which will do a bite attack on the human. Design principle is don’t repeat yourself and it will call the BiteAttackAction if this event is encountered again. This class also fulfils the Fail Fast design principle because the system will fail immediately and visibly when the Zombie Attack has an error. It will be easier to find out the errors as we have two different classes for zombie bite attack and zombie normal attack.

Extend Item class

Corpse class

After a human is killed a new corpse object is created which waits for 5-10 turns before turning into a zombie. It helps to reduce the dependency between human and corpse. Corpse extends from ZombieActor hence it is an actor which just waits for a few turns. After it’s waiting period is over it would create a new zombie object in it’s method. Although there will be a dependency from corpse class to zombie class, it reduces the dependency from human class to corpse class compared to implementing just human class to zombie class to determine the human turning into a zombie. This proves that it fulfills the reduced dependency design principles as it has reduced the dependency from human to corpse class.

Extend Human class:

Farmer

The farmer extends from humans as both player and farmer can harvest the food, it incorporates don't repeat yourself the design rationale. The farmer can use the methods implemented from its parent class which allows the class to not repeat itself as it can reuse the code from its parent class instead of rewriting the same method. The farmer also has the ability to sow and fertilise the crop. The farmer is extended from the Human class hence it reduces dependencies from the classes of engine package.

Extend the WeaponItems class

ZombieMace and Zombie clubs class

ZombieMace and Zombie clubs are extended from the weapons item and can be used as weapons against zombies by humans. They are crafted from Zombie Legs and Zombie Arms class using CraftAction class. It follows the Liskov Substitution principle as the subclasses conform to the super class and are true subtypes.

ItemCapability class

This class is created to see the type of the Item such as letting the Item to be craftable or edible. When humans or players try to get an item, instead of depending on other classes to determine what kind of item there is on the ground, they just get the item capability. If the item capability is edible, humans and players can determine it’s edible and increase its health points. Similarly for the craftable item, if a player gets hold of a zombie arm or zombie legs it can determine if the item is craftable and can then make a weapon out of it. Therefore it reduces the dependency between the classes.