# **Design Rationale For OOD Assignment 1**

## Documentation of Each Task And How It Is Implemented Into The System

### Zombie Attacks

**Zombies should be able to bite**

The Actor Class has a getWeapon() method which iterates through the Actor's inventory and returns the first weaponItem in the inventory if it exists, if no weaponItem exists, it then returns an intrinsic weapon from getIntrinsicWeapon() method. The getIntrinsicWeapon() method will be overwritten in the Zombie class to have a 50% probability to return a bite instead of a punch.

**A successful bite attack restores 5 health points to the Zombie**

Actors have a heal() function that increases the actor's hitPoints by a given integer. The Zombie class will need to know when the heal() method can be used.

Currently AttackAction only has a single 'target' attribute of class Actor. The 'target' attribute only denotes the target of the attack, so a new 'caster' attribute of class Actor needs to be added so that AttackAction knows which actor is casting the attack. This way the AttackAction can call methods of both the caster and target of an attack.

Then the execute() method can be modified with additional logic: when the attack caster is a Zombie, when the attack is bite, when there is a successful attack, then call caster.heal().

**If there is a weapon at the Zombie's location when its turn starts, the Zombie should pick it up.**

Since Zombie extends ZombieActor which extends Actor, Zombie has an inventory attribute and the method addItemToInventory(). However, Zombie has no ability to pick up a WeaponItem and place it in its inventory. Hence a new PickUpBehaviour() class that implements behaviour must be made so that the when playTurn() in Zombie is called, the Zombie can call call the addItemToInventory() method when a WeaponItem and the Zombie is at the same location.

**The Zombie will use the weapon instead of its intrinsic punch attack.**

The Actor class already has a getWeapon() method that iterates the inventory list and returns the first instance of a Weapon if it exists, else it returns the Actor's intrinsic weapon. Since getWeapon() is already called in AttackBehaviour, the Zombie will automatically use the weapon if it has been picked up.

**Every turn, each Zombie should have a 10% chance of saying "Braaaaains"**
A new Behaviour called ZombieBehaviour() can be implemented, and added to the Zombie's behaviours attribute. ZombieBehaviour() will be the first in the behaviours list and will  have a 90% chance of returning a null. The 10% when ZombieBehaviour() does not return null, will return a String that will make the Zombie sound like it is doing something Zombie-like

## Beating Up The Zombies

**On creation, a Zombie has two arms and two legs**
We define two six Enum types to denote the number of arms and legs that a Zombie has; BothArm, OneArm, NoArm, BothLeg, OneLeg, NoLeg. Each Zombie will be initialised with BothArm and BothLeg Capabilities. When the Zombie loses an arm or a leg, the respective Capability will update: removing capability BothArm and adding capability OneArm. Adding and removing capabilities will be done by using the Capabilities class addCapability() and removeCapability() methods

**Any attack on a Zombie that causes damage has a chance to knock at least one of its limbs off.**
On a successful attack, the AttackAction will call the hurt() method of a target to reduce its hitPoints. The hurt() method of the Zombie class will be overwritten to include the ability for a Zombie to lose its limbs. Everytime the hurt() method is called, if the arms and legs are not 0, then  there will be a 25% chance for a Zombie to lose either of its limbs. When a limb is lost, the respective lost limb capability will be updated as mentioned previously with one of the six enum types.

**Effects of Zombie losing an arm**
When a Zombie loses its arm its probability of punching instead of biting is halved. For this functionality, the getIntrinsicWeapon() method must be overwritten to add this additional logic for punching, biting and checking with ArmCapability is as such: if it has BothArm capability then punch 50% and bite 50%; if it had OneArm capability then punch 25% and bite 75%; if NoArm capability then punch 0% and bite 100%.

When the Zombie is holding a weapon it has a 50% chance of dropping it when it loses an arm, and 100% chance of dropping it when it loses both arms. This logic will be implemented in the hurt() method when the Zombie loses an arm.

**Effects of Zombie losing a leg**
When a Zombie loses a leg, it can only move every second turn and when it loses both legs, the Zombie would not be able to move at all. This logic can be implemented in the Zombie's playTurn() method. It will have a condition that will pass on any behaviour related to moving when it has NoLeg capability in LegCapability. The Zombie needs an additional counter attribute that will start incrementing when it has OneLeg capability, the playTurn method will

then have to check if OneLeg capability is true  and counter%2 = 0 in order to pass on any moving behaviour.

**Lost limbs drop to the ground and can be wielded as simple clubs.**
The hurt() method will have to be modified to create a new limb as a weaponItem() object, then call the object's getDropAction() method to place the limb on the map for the player to pick up.

## Crafting Weapons

For our design, we would create a new class called CraftAction class that would extend from Action class and override its Execute() method. Also, we add a new capability enum called CraftableCapability which has Capability CRAFTABLE. For Item objects in the game that can be used as resources for crafting, they will have a CRAFTABLE capability. The Execute() method would get the player's inventory (getInventory()), and then iterate through the inventory to check if the player has any items with CRAFTABLE capability. When a CRAFTABLE item is found, Execute() will return an action that allows users to remove the CRAFTABLE item from the inventory, and replace it with a WeaponItem. In this case, CraftAction will be looking for ZombieArm and ZombieLeg. If a ZombieArm is found, users can choose to craft it into a ZombieClub. When a ZombieLeg is found, users can choose to craft it into a ZombieMace.

## Rising From The Dead

For this part, since the portability of corpses of humans and zombies is true in Item class, this means that the Actor is able to pick up and drop them. For our design, we created a Corpse Class that extends Item. We also modify corpse Item in the Execute method of AttackAction class from PortableItem object class to of Corpse object class.Also, we assume for our design that even when the player is holding the corpse of a human actor for multiple turns, the human corpse would still experience the passage of time. So from that, We made a new method called checkAgeofCorpse() in Corpse class to check whether the item is between 5 to 10 turns, and if it is false then we increment the age of the corpse. If this is true, then we have a variable be a random number between 1 and 0. If the random number > 0.5 , then we would change the portability of the PortableItem to false, call getDropAction(), call execute method in DropItemAction to make the player drop the corpse and change the DisplayChar of corpse to DisplayChar of zombie.

## Farmers and Food

**when standing next to a patch of dirt, a Farmer has a 33% probability of Sow a crop on it**

For our design, we would create a new class in the game package called Farmers that inherits methods from Humans (extend Human), a Crop class that inherits from Ground class, a SowBehaviour class that implements Behaviour and a SowAction class that inherits from Action.

The crop class would have a constructor with its displayChar().

The SowBehaviour class would have a getAction() method which would use method getExits() from GameMap (map.locationOf(actor).getExits()) in order to get all the adjacent cells around the farmer. Then for each exit, we would use method getDestination() in Exit class in order to get the location for each of them and use method canActorEnter() from Location class to check if the terrain is passable by overriding the canActorEnter() method in Ground. If the canActorEnter() method in Farmer class returns True, then the farmer takes the first exit from destinations to have a 33% chance of Sow a crop on the dirt. If a farmer does Sow the crop (33% is true), then the method would return a new SowAction() class. Else, it would return null.

The Farmer class would have the same methods as Human class except that Farmers have a variable which is an array of Behaviours of WanderBehaviour and SowBehaviour.

We also created a SowAction Class that inherits from Action class which would have an overriding Execute() method that would change the displayChar() of the dirt into the displayChar() of the crop.

**left alone, a crop will ripen in 20 turns**

We created a new CropCapability enum class with a state, RIPE and the Crop class has a tick() method that counts the number of turns until it turns ripe, which is 20 turns.

**when standing on an unripe crop, a Farmer can fertilize it, decreasing the time left to ripen by 10 turns**

For our design, We created 2 new classes called FertilizeBehaviour and FertilizeAction in the game package. We also created a new method called getAgeOfCrop() in Crop class.

In FertilizeBehaviour class, we have an override method getAction() which uses method locationOf() from GameMap (map.locationOf(actor).) in order to get the location that the farmer is standing on. For the current location of where the farmer is standing on, we would use method getGround() from Location class and then use method hasCapability() from Ground class to check whether the ground that the farmer currently at is a unripe crop or not. If the hasCapability() for RIPE is false, then the getAction() method returns a new FertilizeAction(Actor actor, GameMap map, Location cropLocation). Else, it would return null.

For the FertilizeAction class, we created an Execute override method that calls the getAgeOfCrop() from the Crop class in order to get the age of the crop. With that, we added 10 to the age of the crop since fertilizing the crop would reduce the time to ripen by 10 turns. Then, we check whether the age of the crop is >= 20. If this is True, then we change the DisplayChar() of unripe crop to DisplayChar() or ripe crop. The method also returned a String stating that the crop is fertilized.

**<u>when standing on or next to a ripe crop, a Farmer (or the player) can harvest it for food. If a Farmer harvests the food, it is dropped to the ground. If the player harvests the food, it is placed in the player's inventory</u>**

We created 3 classes called HarvestAction class that inherits from Action class, HarvestBehaviour class that implements Behaviour and Food class that inherits from PortableItem class.

For Food class, we extended from Item class, would have a constructor which would have parameters of name, displayChar and int of how many health points in recovers.

For HarvestBehaviour class, we made an override getAction() that would use the method getExits() from GameMap (map.locationOf(actor).getExits()) in order to get all the adjacent cells around the farmer. Then for each exit, we would use method getDestination() in Exit class in order to get the location for each of them. After that, we checked whether the ripe crop is at adjacent or on the location where the farmer/ player is standing. If either are true, then the method would return a new HarvestAction() method. Else, it would return null.

For HarvestAction class, we made an override Execute method that would removeCapability of a ripe crop and use addItem() method in Location class to drop the food at harvested location and return a String stating that the crop has been harvested.

**<u>Food can be eaten by the player, or by damaged humans, to recover some health points</u>**

We made a UseAction class that inherits from Action class and a IsHurtBehaviour class that implements a Behaviour interface. We also added a new method in Food class called asFood() that would go through the Actor's inventory and check whether the Item is of type Food or not. If it is of type Food, then we compare the health points of the Food with the current Health of the Actor. If both when added up equals to 100, then we return the Food.

For IsHurtBehaviour class, we used the getAction() override method that checks through Actor's inventory and using asFood() method to check whether the item is of type Food or not. If it is of type Food, then we compare the health points of the Food with the current Health of the Actor. If both when added up equals to 100, then we return a new UseAction. Else, it would return null.

For the UseAction class, we used the Execute override method that deletes the Food item from the Actor's inventory and increase the Actor's health back to 100 Hit points, it would return a String telling that the Actor has regained to full health.

# Reasons Of Why We Chose To Implement The Designs For These Tasks

## Zombie Attacks

To change how Zombie Attacks we must know the process of how an attack is done, and know it affects other classes. As such, we do not need to create any new methods or classes and only need to modify the logic in Zombie Class and AttackAction. There are already methods that have been made for our intended purpose, but not with our intended logic. Hence, methods such as getIntrinsicWeapon() and Execute() must be overwritten, the heal() method needs to be used, and the initializer in AttackAction needs to be modified.

## Beating Up The Zombies

To add additional functionality to the Zombies, we use the given codes that have already been provided in the engine, Capabilities class. Capability class acts like a flag for Item objects and Actor objects, which makes it suitable for this implementation. By creating two new enum types for Zombie, we have given it 2 new flags that we can use to define new logic for it. We can decide on which Capability is true by using the Actor's hasCapability() function and create the logic of what will happen to the Zombie when it loses its limbs based on that.

When Zombies lose their limbs, the limbs should drop onto the map for the player to pick up and use. Because limbs are wieldable weapons by the player, we instantiate new limb objects as WeaponItem() and add the item on the map.

## Crafting Weapons

For our Crafting Weapons task, we chose to create a new class called CraftAction class and a new capability enum called CraftableCapability. The reason we implement the enum is to allow us to efficiently check whether the item/weapon that the actor currently has is craftable or not by using as a flag without needing to go through the extra steps of checking such as checking if the item is of type weapon or if its name is has word "Zombie" in it. With this implementation, we can reduce the line of code required so that it is easier for other programmers to understand.

Other than that, the reason we chose to implement CraftAction class is because it would allow the player to craft the item and end its turn after crafting that item. From this, we also assume that whenever the player chooses to craft an item, it would consume a turn/ use an action. Thus, the reason we chose to create and call it CraftAction.

## Rising From The Dead

For our Rising From The Dead task, we chose to implement some designs which are creating a Corpse Class that extends Item class, make a new method called getAgeOfCorpse() in Corpse class and modify the Execute method of AttackAction class. The reason we chose to create a new Corpse class that extends Item instead of PortableItem is because we want to be able to change its portability depending on whether it has risen from the dead or not. Other than that, we also created this class so that we can more easily keep track of the age of the corpse without needing to go through added effort of checking it from Item class.

Moving on from that, we also created the method getAgeOfCorpse is to be able to simulate to the player that the corpse is rising from the dead by changing certain parameters of its Item type to show it, also the reason we did it in this way is because we are not able to add or make any changes to the classes in the engine package.

Lastly, the reason we modified the Execute method of AttackAction class to change its corpse item from PortableItem object class to Corpse object class is so that we can more efficiently change its portability and displayChar if need be, rather than going through PortableItem class because PortableItem class would always set its portability to true.

## Farmers and Food

For our Farmers and Food task, the designs that we have implemented designs for the game package include,

- Multiple classes that implements Behaviour Interface such as SowBehaviour, FertilizeBehaviour, HarvestBehaviour and IsHurtBehaviour.
- Multiple classes that inherit from Action class such as SowAction, FertilizeAction, HarvestAction and UseAction classes.
- Classes that inherit from different classes such as the Crop class that inherits from Ground class and Food class that inherits from PortableItem class.
- New methods tick() and getAgeOfCrop() in Crop class and asFood() in Food class.
- New enum called CropCapability

The reason we created these Behaviour classes is so that Farmers are able to simulate these behaviours within the game because Farmers or Actors in general have to be coded/designed to behave in a certain way in order to simulate an actual human that is alive within the game world.

Other than that, the reason we created Action classes is because the behaviours for Actors, more often than not, would result in them having to take actions to complete that task, so we have to create these classes to execute these actions which would also consume a turn for the Actors because every action an actor takes would use up a turn.

Moving on from that, we also created Crop and Food class that inherits from Ground and PortableItem class respectively, with the intention of helping the programmers or readers to be able to distinctly identify subtypes of the Ground and PortableItem class which would in turn make it easier for them to understand the code.

Aside from that, the reason for creating the new methods tick() and getAgeOfCrop() in Crop class was to help keep track of the age of the crop to ripe and make it easier for FertilizeAction class to get the reduced time taken to ripen the crop after fertilizing it. Moreover, the asFood() method in Food class was also created to help the UseAction class be able to compare the health points of food with the hit points of the actor to correctly determine when to use the food to regain the actor’s health.

The new CropCapability enum class was also created to help make the checking of whether the crop is ripe or not be more easier because the checking of displaychar for each crop would prove to be more difficult to implement than making an enum class.