# Design Rationale

Jiten Verma and Ayesha Ali

## Zombie attacks

| Feature Number | Feature description |
|---|---|
| 1 | Zombies should be able to bite. Give the Zombie a bite attack as well, with a 50% probability of using this instead of their normal attack. The bite attack should have a lower chance of hitting than the punch attack, but do more damage – experiment with combinations of hit probability and damage that makes the game fun and challenging. (You can experiment with the bite probability too, if you like.) |
| 2 | A successful bite attack restores 5 health points to the Zombie |
| 3 | If there is a weapon at the Zombie's location when its turn starts, the Zombie should pick it up. This means that the Zombie will use that weapon instead of its intrinsic punch attack (e.g. it might "slash" or "hit" depending on the weapon) |
| 4 | Every turn, each Zombie should have a 10% chance of saying "Braaaaains" (or something similarly Zombie-like) |

### Feature 1

The bite feature will be implemented by creating a new class which extends IntrinsicWeapon with the name "Bite" and damage of 15. The getInstrinsicWeapon method in Zombies class will be modified to have a 50% chance of selecting punch and 50% chance of selecting bite. These probabilities will be stored as private integers.
Furthermore, the AttactAction class would be modified to identify if an attack is a Bite attack. If a bite attack is identified then the actor would have a 40% chance to hit.
The reason for this design choice is to produce subclasses for Intrinsic weapons called Punch and Bite which will be used quite often for the Zombie class. This would allow us to identify Bite by finding the instance of the weapon in AttackAction rather than getting a string

of its verb. This would help reduce indirect dependencies which would be otherwise difficult to identify.

## Feature 2

In AttackAction, identify if weapon is an instance of bite. If so, the actor using the attack would heal 5 hit points when the weapon lands a successful hit.
Actor already has an existing method called heal which we can call in AttackAction to heal the attack user.
The reason for this design is because we cannot modify the Weapon class to create an attribute called heal which would allow us to centralise the healing ability of using a weapon. However, by creating a new class for Bite, we have been able to easily identify if a weapon is an instance of Bite. The healing abilities have instead been centralised within the AttackAction, and only appears once in the entire code to reduce any indirect dependencies.

## Feature 3

Create a new method in the Zombie class called pickUpWeapons which requires an ArrayList of actions and the map to be passed through. This method iterates through each action within the array list and checks if the action is an instance of PickUpItemAction. If a zombie is standing on a weapon, then it will have an action of this instance. The action is simply executed to place the weapon in the inventory of the zombie and this will not count as the zombie's turn.
This method will be called within the playTurn which is contained in the Zombie class.
The getWeapon method in Actor is already optimised to give weapon selection priority over IntrinsicWeapon.

The reason behind this design choice is to simplify the IntrinsicWeapon classes so that each weapon is clearly identifiable.

## Feature 4

Create an array called zombieNoise which stores a few noises that zombies make.
Modify the playAction method in the Zombie class so that it produces a random integer between 1-100. If the integer is between 1-10 inclusive, then produce a random integer between 0 to the size of ZombieNoise to select a random noise for the zombie to make. This can be printed onto the game.

This design choice allows for multiple zombie noises to be stored so that there is variation and the gamer experience is enhanced. Additionally, by implementing this design we reduce any dependencies which could be present between the array of ZombieNoise and playTurn as the execution depends completely on the size of the array. Therefore, to modify the sounds zombie can make, we must only modify the zombieNoise array.

# Beating up the zombies

| Feature Number | Feature Description |
|---|---|
| 1 | Any attack on a Zombie that causes damage has a chance to knock at least one of its limbs off (I suggest 25% but feel free to experiment with the probabilities to make it more fun) |
| 2 | On creation, a Zombie has two arms and two legs. It cannot lose more than these. |
| 3 | If a Zombie loses one arm, its probability of punching (rather than biting) is halved and it has a 50% chance of dropping any weapon it is holding. If it loses both arms, it definitely drops any weapon it was holding. |
| 4 | If it loses one leg, its movement speed is halved – that is, it can only move every second turn, although it can still perform other actions such as biting and punching (assuming it's still got at least one arm) |
| 5 | If it loses both legs, it cannot move at all, although it can still bite and punch |
| 6 | Lost limbs drop to the ground, either at the Zombie's location or at an adjacent location (whichever you feel is more fun and interesting) |
| 7 | Cast-off Zombie limbs can be wielded as simple clubs – you decide on the amount of damage they can do |

## Feature 1

Create a new method in the Zombie class named tryToDropLimbs which requires the map to be passed through. If the Zombie has no arms or legs then this method simply returns an empty ArrayList called limbsDropped. If the zombie does have arms or legs, then we can get a random integer between 1-100. If the integer is between 1 - 25, then a we can proceed to drop limbs. A random integer will be generated between 1 - 20. If the number is between 1-10, only one limb will be knocked off. If the number is between 11-17, then 2 limbs will be knocked off. If the number is between 18-19, then 3 limbs will be knocked out. Finally, if the number is 20 then all 4 limbs will be knocked out.
The dropLimb method will also be implemented in the Zombie class. If the number of limbs a zombie has is less than the number of limbs determined to be lost, then the dropLimb method will be called repetitively until all limbs have been dropped. Alternatively, the dropLimb method will be called as many times as was randomly selected.
The dropLimb method will randomly select either an arm or leg, depending on which body parts the zombie still has.

The reason for this design choice is that it allows us to uphold the single responsibility principle as we ensure that all methods associated with zombies are all centralised within the zombie class.

The reason for this design choice is that it allows for a unique hurt method in zombie, which will affect zombie actors only and allows us to program unique features for zombies.

## Feature 2

Two new attributes are introduced to the Zombie class; Arms and Legs. Upon initialisation these have an integer value of 2 to represent 2 arms and 2 legs.
Another attribute is also introduced; minLimbs which will be initialised to 0. This will help ensure that the number of limbs never falls below zero.
The reason for this design choice is that it allows for a unique hurt method in zombie, which will affect zombie actors only and allows us to program unique features for zombies.

## Feature 3

A method in the Zombie class called accountForLostArm will be called to make changes to the zombie instance if it loses any arms. The new attribute called chancePunch was created to hold the probability of punching rather than biting which was initialised to 50. If one arm is lost, then this attribute is modified to show a 25%. Losing the arm would give a 10% chance to drop a weapon. If both arms are lost then chance of punching becomes 0% and the zombie drops all items they are holding.
The reason for this design choice is that it allows for a unique hurt method in zombie, which will affect zombie actors only and allows us to program unique features for zombies.

### Feature 4

An attribute will be added to Zombie which stores a boolean value called canMove. This will be initialised as True. If this value is true, then the zombie will be able to perform WanderBehaviour actions (moving around). If this value is False, then the zombie would be unable to move around essentially.

A method will be added to the Zombie class called accountFotLostLeg. If the number of legs is equal to 1. Then the value of canMove will be inverted (False now becomes True, and vice versa). This means that canMove will be True every second turn. If the number of legs is 2 then ensure canMove is True.

The reason for this design choice is that it allows for a unique hurt method in zombie, which will affect zombie actors only and allows us to program unique features for zombies as each class should be responsible for one purpose only.

### Feature 5

If the number of legs is 0 then ensure canMove is False using the modifyMovementSpeed method.

### Feature 6 and 7

The dropLimb method created previously initialises a new arm or leg as a weapon and drops this onto the map where the zombie is standing. New classes will be created called ZombieArm and ZombieLeg which extends Weapon. These weapons can be picked up and used as simple clubs. The damage an arm does is 10 and the damage a leg does is 15.

The reason for this design choice is that it allows for a unique hurt method in zombie, which will affect zombie actors only and allows us to program unique features for zombies. Additionally, we are trying to simplify the weapons class by creating subclasses of weapons to be able to have clear dependencies.

## Crafting weapons

| Feature Number | Feature Description |
|---|---|
| 1 | If the player is holding a Zombie arm, they can craft it into a Zombie club, which does significantly more damage. |
| 2 | If the player is holding a Zombie leg, they can craft it into a Zombie mace, which does even more damage. |

| | |
|---|---|

### Feature 1

Create a new class called ZombieClub which extends Weapon. We need to create a class called CraftZombieClubAction which extends Action. If the player has a zombie arm as a weapon in their inventory then then the action is added to their list of actions in playTurn in Player. If they select this action, then we create a new weapon object in CraftZombieClubAction called ZombieClub with 25 damage and remove the zombie arm from the player's inventory without dropping it to the map.

### Feature 2

Create a new class called ZombieMace which extends Weapon. We need to create a class called CraftZombieMaceAction which extends Action. If the player has a zombie leg as a weapon in their inventory then the action is added to their list of actions in playTurn in Player. If they select this action, then we make a weapon object in CraftZombieMaceAction called ZombieMace with 35 damage and remove the zombie leg from the player's inventory without dropping it to the map.

# Rising from the dead

| Feature Number | Feature Description |
|---|---|
| 1 | As everybody knows, if you're killed by a Zombie, you become a Zombie yourself. After a Human is killed, and its corpse should rise from the dead as a Zombie 5-10 turns later. |

### Feature 1

A new class called Corpse will be created within the game package which extends PortableItem. The existing Corpse Item will be refactored so that it is produced by this class instead. The Corpse class will have an additional attribute called revivalTime which stores the number of turns remaining until the Corpse can be revived. We can create an override tick method in corpse to reduce the number of turns each turn. The object of Corpse class will be deleted when revivalTime has reached and an object Zombie has been made.In AttackAction, an object of Corpse will be made.

Design Principles: The D.R.Y Rule is being maintained as Corpse which is a subclass of PortableItem shares similar functionalities and attributes compared to PortableItem and we are also adding our own methods and attributes to implement the functionality. It also follows Open/Closed Principle as the base class Item is not being modified but it is being extended by adding subclasses like PortableItem and Corpse. It also follows the SRP principle as a single class called Corpse has been created to implement a feature related to corpse rather than creating an object of Item or PortableItem.

# Farmers and food

| Feature Number | Feature Description |
|---|---|
| 1 | When standing next to a patch of dirt, a Farmer has a 33% probability of sowing a crop on it |
| 2 | Left alone, a crop will ripen in 20 turns |
| 3 | When standing on an unripe crop, a Farmer can fertilize it, decreasing the time left to ripen by 10 turns |
| 4 | 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 |
| 5 | Food can be eaten by the player, or by damaged humans, to recover some health points |

## Feature 1

Create a new class UnripeCrop represented by character U which is an extension of Item and a new class called Farmer which is an extension of Human represented by F. Make a new class called sowAction which is an extension of Action and sowBehaviour which implements Behaviour.In sowBehaviour we determine possible locations in which a crop can be sown and generate a random number from 1 to 100. If the number is between 1 and 33 and we have a possible location an object of sowAction is initialised.In sowAction a new object of unripe crop is placed in any of the possible locations . In Farmer, we make an object of sowBehaviour and we override playTurn in Action which will return an object of sowAction if the conditions meet.

## Feature 2

Make a new class called RipeCrop which extends PortableItem and represented by ‘R’.UnripeCrop has a variable called turns.To keep track of turns we override tick method in location which passes location as a parameter.The value of turns is decremented at each turn and when turns becomes less than or equal to 0, object of RipeCrop is added and current object of UnripeCrop is removed.

## Feature 3

Add a new class called FertiliseNewBehaviour which implements Behaviour which will check if the location of crop and farmer match.Add a new class called FertiliseNewAction which extends Action for the Farmer class whose object will be initiated in FertiliseNewBehaviour if the condition matches. This action will call a method in the UnripeCrop class called executeFertilize, which decreases the number of turns remaining by 10 rather than 1.
In Farmer, we make an object of FertiliseNewBehaviour and we override playTurn in Action which will return an object of FertiliseNewAction if the conditions meet.

## Feature 4

Create a new class called HarvestedCrop which extends PortableItem represented by character ‘H’. Create a new class called HarvestBehaviour which implements Behaviour.It checks if there are nearby crops which can be harvested.If there are then it initialises an object of HarvestAction.Create a new class called HarvestAction which extends Action for Farmer, which will remove an object of RipeCrop. If the actor is a farmer, an object of HarvestedCrop will be added and if the actor is a player, an object of PickUpItemAction will be made so that the HarvestedCrop is in Players inventory.In Farmer and Human, we make an object of HarvestBehaviour and we override playTurn in Action which will return an object of HarvestActionif the conditions meet.

## Feature 5

Create a new method in Human called pickUpFood which will make an instance of PickUpItemAction will be made to pick all items in location.If one of them is an instance of HarvestedCrop then heal method from Actor is called and it is removed from humans inventory.In PlayTurn of Human, this method is only called when human has less hitpoints.

Design Principles (for the whole task): The D.R.Y Rule is being maintained as Farmer which is a subclass of Human shares similar functionalities and attributes compared to Human and we are also adding our own methods and attributes to implement the functionality .Similarly UnripeCrop is being extended from Item and RipeCrop and HarvestedCrop from PortableItem It also follows Open/Closed Principle as the base classes are not being modified but it is being extended by adding subclasses like Farmer and UnripeCrop . It also follows the SRP principle as a single class has been created wherever possible for each functionality so that these classes can be extended for Assignment 3 functionality. It also

follows the Interface Segregation Principle as each functionality uses the needed interface only like all Behaviours are implemented by the Behaviour interface and all Actions are based on the ActionInterface.