**DESIGN RATIONALE**

**ZOMBIE ATTACKS**

Implementing Zombie Bite

All features described in this segment are depicted in Class Diagram – Zombie.

The Zombie Bite feature may be added by modifying the Zombie Class method getIntrinsicWeapon(). A probability check can be added that, if successful, will return a new IntrinsicWeapon(15, “bites”) instead of the default IntrinsicWeapon already written.

The damage number is subject to change, but the verb “bites” will be used to identify the weapon as a bite, and so will be unique to this type of Zombie attack.

The probability check used to determine the type of attack must be modifiable based on how many arms the Zombie has. This can be done by checking the value of the zombieArms variable in the Zombie object making the attack. zombieArms is described in detail later in this document.

The AttackAction class will be modified, to implement characteristics specific to the Zombie’s bite attack. It will do this by checking the verb associated with the weapon when an attack is made; if IntrinsicWeapon.verb() == “bites”, a lower hit chance will be factored into the attack roll, via if/else switch.

The same IntrinsicWeapon.verb() == “bites” check will also determine whether a successfully landed attack restores health. If the attack lands and the verb check is successful, the bite will execute actor.heal(5) to heal the zombie.

Implementing Zombie Weapon Pickup

All features described in this segment are depicted in Class Diagram – Zombie and Sequence Diagram – Zombie Pick Up Weapon.

To accommodate Zombies picking up weapons, a new behavior class named PickUpBehavior will be added to the Zombie class behaviors ArrayList. This behavior will be lower priority than AttackBehavior, but higher than HuntBehavior or WanderBehavior.

PickUpBehavior will first check that the Zombie has arms, by checking the zombieArms variable in the Zombie object making the attack.

The presence of a weapon on the ground that is eligible to be picked up must then be checked. This can be done by using map.LocationOf(zombie) to get the zombie’s current map location, then calling getItems() on that location to fetch all items on the ground at that location, then calling item.asWeapon() to check the item’s status as a weapon.  
  
If there is a valid Weapon on the ground in the same location as the zombie, PickUpBehavior will return a pickUpItemAction for that Weapon, which Zombie will then execute.

Implementing Zombie Moan

Every time the playTurn method in Zombie is called, before the program loops over the Behavior ArrayList, a random probability will return a 10% chance that the system prints a zombie moan.

**BEATING UP THE ZOMBIES**

Implementing Zombie dismemberment

All features described in this segment are depicted in Class Diagram – Zombie and Sequence Diagram – Zombie Lose Limbs.

Each Zombie object will have two int variables keeping track of their limbs; zombieArms and zombieLegs. Each of these variables will start at 2 when the object is instantiated, and will be reduced to 1 then 0 as limbs are lost.

The attackAction class will be modified to check if the target of any given attack is a zombie, and if so, call a new Zombie class method limbLoss(). This method will be called at line 61 of attackAction, after the death check and damage calculation, to ensure that it is not needlessly applied to an already deceased Zombie.

If the Zombie targeted has no remaining arms or legs, limbLoss will do nothing. If the zombie has at least one arm or leg, represented by zombieArms + zombieLegs >= 1, the method will roll a 25% probability check, and do nothing on failure.

On success of that probability check, if zombieArms == 0, the zombie will lose a leg. If zombieLegs == 0, the zombie will lose an arm. If neither is 0, the method will do a random Boolean check to decide which limb type should be lost.

When a limb is lost, a representation of the limb will be instantiated as an object of a new class, which are ZombieArm and ZombieLeg respectively. Both of these classes will be extensions of the WeaponItem parent class, in the same manner as the Plank class already in the game. The new limb, as a WeaponItem, will be added to the Zombie’s inventory. getDropAction() will then be called on that item, dropping it at the zombie’s feet.

Damage numbers for the limbs are currently set at 10 for ZombieArm and 15 for ZombieLeg, but this is subject to change as the game is balanced.

If an arm is lost, the resulting accuracy loss will already be factored in as described in “Implementing Zombie Bite”. The limbLoss method will also call the method fumbleWeapons(), which will iterate over the items in the zombie’s inventory and check if each item is a weapon. If the item is a weapon and the Zombie has 1 arm remaining, a random Boolean check will decide if the weapon is dropped. If the Zombie has 0 arms remaining, all weapons will be dropped.

Implementing Slowed Movement

All features described in this segment are depicted in Class Diagram – Zombie.

A new behavior collection named leglessBehavior[] will be added to the Zombie class, containing only the Behaviors that do not involve movement. A Boolean variable called slowWalk will also be added to Zombie, to act as a toggle.

At the start of each zombie playTurn() call, the method will check the number of attached legs the Zombie has. If zombieLegs == 2, the method will iterate through the regular behavior[] collection as normal. If zombieLegs == 1, the method will iterate through behavior[] if slowWalk is false, or leglessBehavior if slowWalk is true, then flip the slowWalk toggle to the opposite value. If zombieLegs == 0, the method will only iterate through leglessBehavior[].

**CRAFTING WEAPONS**

All features described in this segment are depicted in Class Diagram – Human.

A CraftAction class was created as a child class of Action in order to handle the crafting of a weapon. This way, the parent Action class will handle the user pressing a button to perform an Action, similarly to how AttackAction works. Since ZombieLeg and ZombieArm are both child classes of WeaponItem and WeaponItem is a child class of Item, this means that the Item class will handle the Human picking up the item using the method getPickUpAction(). The CraftAction class constructor will take in a Zombie arm or leg, and a ZombieLimbWeapon that the player wants to craft the limb to. The ZombieLimbWeapon enumeration contains the 2 possible weapons that can be crafted (mace or club). These values will tell the CraftAction for which type of ZombieLimbWeapon it is updating the damage and name of.

It was decided to create a ZombieLeg and ZombieArm that are child classes of WeaponItem. This means that we do not have to rewrite the contents of the PickupItemAction class, as WeaponItem is a child of Item, this means that zombie limbs qualify as Item objects. Hence, the PickupItemAction class will handle picking up a ZombieLeg or ZombieArm.

Also, by making a “CraftAction” class that is a child of Action, we inherit the methods we need to create the CraftAction which means that the possibility of redundant code is reduced.

**RISING FROM THE DEAD**

All features described in this segment are depicted in Class Diagram – Human.

The AttackAction class currently creates a corpse object as an Item. We will change this so that the Corpse object is a child of the PortableItem class. The Corpse object inherits the method tick() from the Item class. The tick() method is run for every turn in the game so we will override this method in the child Corpse class, with a class integer variable of “turns” which is incremented each time tick() is called. Now that we can track time, we can see when 5-10 turns occur. When it does occur, AttackAction will simply remove the Corpse item using removeItem() from the Location class and place a Human in the position of the corpse.

By making Corpse a child class of PortableItem rather than alternatively a child of Item, we reduce the need of having to pass the boolean value “true” into the constructor of Corpse, as PortableItem passes “true” into the Item constructor for us already, hence reducing repetition of code. Doing this also means that the Corpse item will inherit the tick() method that is implemented in the Item class, which corpse will override and count the amount of days that it has been a Corpse, as explained above. Therefore by making a Corpse a child of PortableItem, we also reduce the need to create a new tick() method and somehow ensure it is called on each turn of the game, and therefore once again repetition of code is reduced.

**FARMERS AND FOOD**

All features described in this segment are depicted in Class Diagram – Human.

Since the Assignment spec has asked for Farmers to shares the same characteristics and abilities as a Human, we make “Farmer” a child class of Human. In doing so, we prevent the need to copy and paste methods that have already been implemented for us in Human as they will all be inherited by the child class “Farmer.” In this Farmer class, the plan is to have an array of behaviours in a similar way Zombie has an array of behaviours. This array will contain 3 behaviours: FertilizeBehaviour, HarvestBehaviour and SowBehaviour. By creating a collection of behaviours, we are making it easier to add more behaviours in the future to Farmer, as we can simply add this behaviour to the array.

Each of these behaviours represent three objectives of a Farmer and returns an Action in the getAction() method of the behaviour. Each of these behaviours implements the Behaviour interface

The Food class has been created to represent a Food PortableItem. This is done as previously stated so that we do not have to worry about setting the boolean parameter “portable” for Food, as PortableItem’s constructor sets the portable to true by default. In order to be able to be eaten, we must attach an EatAction to the Food. To allow for this to occur, we can add a method to Food named addAction which adds the EatAction to the Iterable Actions allowableActions as defined in parent class Item.