Design Rationale

ZOMBIE ATTACKS

ZOMBIES SHOULD BE ABLE TO BITE

Next we need to create a new class ZombieAttackBehaviour. This implements Behaviour interface, but stores a list of AttackTypeBehaviours. It will be used for picking which AttackTypeBehaviour the Zombie will perform.

Next we need to create the BiteBehaviour class that extends AttackTypeBehaviour and will be stored in ZombieAttackBehaviour. BiteBehaviour will create instances of another new class, BiteAction that will get a Zombie's BiteWeapon, another new class that inherits from IntrinsicWeapon, and perform a "bite".

Now instead of Zombie having an AttackBehaviour, it will have a ZombieAttackBehaviour that will pick which attack to use. In ZombieAttackBehaviour, for now it will randomly select one of its AttackTypeBehaviours (for which there are only two. One for punch and one for bite.) and call its getAction method. It will return the result of this method call.

We chose this method of implementation as it allows us to add other attacks in the future easily. It also keeps the Behaviour classes returning the Action which simplifies the class responsibilities. We can also add additional functions to the Bite Attack easily (e.g add a critical strike chance).

(see Z1.ZombieBite)

ZOMBIE BITES SHOULD RESTORE HEALTH

To implement this feature we simply need to add a healAmount attribute to the BiteWeapon class. Now in the BiteAction's execute method, if the attack lands, after we have hurt the target Actor we can query the BiteWeapon for the healAmount, and heal the Zombie the according amount.

We chose this method as it was the simplest, and introduced no new dependencies.

ZOMBIES CAN PICK UP WEAPONS

To implement this feature we need to create a new class PickUpWeaponBehaviour. This will inherit from Behaviour and be responsible for creating PickUpItemActions for Weapons. It will be dependent on location.

To implement this feature in the Zombie's playTurn method we first query PickUpWeaponBehaviour for an action. If a Zombie can pick up a weapon, it will return a PickUpItemAction that will pick up the Weapon when executed.

We chose this method to implement, as it keeps the Behaviour classes as the only ones returning Actions, and we can easily slot the PickUpWeaponBehaviour into the Zombie's behaviours.

(see Z3.ZombiePickUp)

ZOMBIES USE PICKED UP WEAPONS

Our previous tasks have already implemented this. If an AttackAction is returned by Zombie's playTurn, when it is executed it will query Zombie for getWeapon and it will loop through the Zombie's inventory, and if it has a Weapon, it will return that.

ZOMBIES SAY "BRAAAIIINNNSSS"

To implement this, in Zombie's playTurn method we need to use a random number generator to execute a sayBrains method 10% of times it is executed. This sayBrains will simply print "BRAAAIIINNNSSS."

We chose this implementation technique as we couldn't use the Display class as Zombie doesn't have access to it, but the Displays print method just uses System.out.prinln() which we can use. This implementation also introduces no new dependencies and is easy to code.

BEATING UP THE ZOMBIES

ZOMBIES LIMBS GET KNOCKED OFF

> ** assumption**
> Zombies cannot travel between GameMaps

To implement this feature we need to create a class DropLimbAction that extends Action. We also need to create a LimbFallOffBehaviour class that implements Behaviour.

The LimbFallOffBehaviour will create a DropLimbAction, depending on which limb is going to drop. The DropLimbAction will drop the limb to the ground.

Finally, in the Zombie's hurt method, if a Zombie is hurt, getAction is called on LimbFallOffBehaviour, which will return a DropLimbAction, which will then be executed.

To call getAction on LimbFallOffBehaviour, we need the GameMap of the Zombie, which means we need an association between Zombie and GameMap.

This implementation isn't great as now we have an extra association, but we couldn't think of a better solution. Aside from the new association, we have again kept Behaviour classes as the only ones returning Actions.

(see B1.ZombieLimbs)

INITIAL ZOMBIE LIMBS

We need to give Zombies two extra attributes: legCount and armCount. When a Zombie is constructed they are initially set both to two. We also need to add looseArm and looseLeg methods that check that we are not going into negative limb counts (e.g. legCount = -3 is not allowed).

We chose to store the limb counts as ints as it is more space efficient than having a list of Limb objects stored in the Zombie. This implementation also introduces no new dependencies and will be easy to code so we went with it.

PUNCH PROBABILITY

To implement this, when getAction is called on ZombieAttackBehaviour, we need to query Zombie for its armCount. If armCount == 2, return AttackAction 50% of the time, if armCount == 1, return AttackAction 25% of the time, and finally if armCount == 0, never return AttackAction.

Because of our hard work earlier, this function is very easy to implement. We chose this method as it introduces no new dependencies, and it will be easy to code.

WEAPON DROPPING

** assumption **

If a Zombie drops a Weapon but still has one arm left, it can pick it up on its next turn.

To implement the feature of a Zombie sometimes dropping a weapon, we need to change the DropLimbAction. As the DropLimbAction stores the limb being dropped, if the LimbDropAction is going to remove an arm, we need to add a 50% chance that the Zombie drops the weapon. If the LimbDropAction is going to remove a Zombie's last arm, we need to drop the Zombie's Weapon 100% of the time.

Implementing the feature like this means we are introducing no new dependencies.

ZOMBIE MOVEMENT

In the Zombie's playTurn method, before we return an Action, we check if it is a MoveActorAction. If it is, we check how many legs a Zombie has. If it has two, we return the MoveActorAction. If it has one, we only return the MoveActorAction if its last move wasn't a MoveActorAction. If a Zombie has zero legs, we never return a MoveActorAction.

Zombie is now dependent on MoveActorAction.

DROP LIMBS

We have already implemented this in Zombies Limbs get knocked off.

ZOMBIE LIMB WEAPONS

When dropped, we will create new ZombieLimb objects. ZombieLimb will inherit from WeaponItem and will have damage and verb.

CRAFTING WEAPONS

ZOMBIECLUB AND ZOMBIEMACE

A Player now has the capability to craft weapons when holding a zombie limb. To achieve this, we are adding CraftAction to the ZombieLimb's list of allowable actions. Now if a Player is in possession of a ZombieLimb, using getAllowableActions they have the option to execute CraftAction as their turn.

The CraftAction class is inherited from Action. This removes the instance of ZombieLimb and creates an instance of the ZombieWeapon class and adds it in the Player's possession.

CraftAction can create two different instances of ZombieWeapon depending on the instance of ZombieLimb the Player is holding. ZombieClub is crafted when the Player is holding ZombieArm and ZombieMace is crafted when the Player is holding ZombieLeg. They are inherited from WeaponItem. ZombieClub has damage points of 30, ZombieMace has damage points of 40.

This implementation was deemed most appropriate. ZombieClub and ZombieMace share the same attributes so it is appropriate that they are instantiated from a single class we have titled ZombieWeapon. Since only the Player has the capability to craft weapons the creation of an additional behaviour was unnecessary.

FARMERS AND FOOD

CREATING THE FARMER

The Farmer character is an extension of a Human with the additional capability to sow crops, fertilize crops and harvest crops for food.
To achieve this, we've:

- Created a class Farmer that extends Human
- Created the FarmerBehaviour class which the Farmer has access to
- Created classes FertilizeAction, HarvestAction and SowingAction allowing Farmers to execute these capabilities
-

To action the Farmer's capabilities, we need to know if they are in range of dirt, crops, or food. The FarmerBehaviour class implements the Behaviour interface and is dependent on the Location class. This allows the Farmer to detect if it can execute the appropriate actions based on its Location.

To interact with crops and food we needed to create classes that the Farmer can interact with.

CREATING THE CROP AND FOOD CLASS

The Crop class is instantiated when a Farmer successfully sows a crop on a patch of dirt. Left alone it will ripen in 20 turns or a Farmer can fertilize it to speed up the aging progress by 10 turns. Crop has an age attribute to track how many turns since it was instantiated, a tick method to increment its age and a displayChar attribute that is dependent on if the Crop is deemed ripe or unripe.

Crop is an extension of the Ground class, just like Dirt and Tree. This is appropriate as it cannot be picked up like an Item, it needs to be displayed on the map and interacted with by Farmers.

When a Crop is considered ripe it can be harvested by a Farmer, creating an instance of the Food class. Food can be picked up by Players and thus is an extension of the Item class. It will have an attribute displayChar ‘F’ so that Players can recognise it on the map.

Now that these classes exist, we can create actions so that the Farmers and Players can interact with them.

SOWING ACTION

When standing next to a patch of dirt, a Farmer has a 33% chance of sowing a crop in it. The FarmerBehaviour class detects if the Player is in range of an instance of Dirt, and if this is true executes SowingAction.

SowingAction is inherited from the Action class. This occurs on every Farmer’s turn. When successful a Crop class is instantiated, and the Location of the Dirt is instead set to Crop. To achieve this SowingAction has a setGround(Crop) method

*** Assumption: Farmer has a 33% chance of sowing a crop onto every dirt instance it is next to per turn

SowingAction was decided not to be used as the Farmer’s action in playTurn(). This is because the majority of the GroundMap’s Ground is Dirt and it is likely that the Farmer will be surrounded by up to 8 instances of Dirt for most of the game. This would mean their movement around the map and interaction with Zombies would be limited as it is likely every turn, they would sow a Crop. Instead FarmerBehaviour loops through all the locations of Dirt in range and executes SowingAction on each. The Farmer is then able to use their playTurn() to move around, attack Zombies and/or pick up Items.

FERTILIZE ACTION

To fertilize an unripe crop the class FertilizeAction was created. Now when the Farmer is standing on an unripe crop (Crop with age attribute < 20) it can fertilize it, decreasing the time left to ripen by 10 turns (increase Crop age by 10).

To achieve this FertilizeAcion has a fertilize method that increase the Crop’s age attribute by 10.

FertilizeAction is an extension of the Action class in the Engine package. To check if a Farmer can execute FertilizeAction the FarmerBehaviour class loops through the surrounded locations and if it is method setGround returns Crop, FertilizeAction can be executed.

HARVEST ACTION

To harvest a ripe crop the class HarvestAction was created. Now when standing on or next to a ripe crop (Crop age => 20, a Farmer can harvest it for food. If a Farmer harvests the food, it is dropped to the ground.

Similarly, to FertilizeAction, the FarmerBehaviour class loops through the surrounding Farmers location using the getGround method. If a Crop with age => 20 returns, then a Farmer can execute HarvestAction.

HarvestAction is an extension of the Action class. To achieve this there is method to create a Food instance at the Crops location. HarvestAction also uses the setGround method to remove the instance of Crop and reset the ground to Dirt.

*** Assumption: We have interpreted this feature as:

- If a Farmer harvests a ripe crop, the ripe crop is replaced with food and is dropped to the ground
- A player can harvest food, placing it in the player's inventory
- A player cannot harvest a ripe crop
- We were unsure if ripe crop = food, so we have said they are not the same and implemented the above

*** Assumption: After crop is harvested, ground returns to dirt

FARMER BEHAVIOUR

In summary FarmerBehaviour loops through the Farmers location and surrounding locations, checking for instances of Dirt and Crops. Dependent on the results the FarmerBehaviour creates certain actions as explained above in SowingAction, FertilizeAction and HarvestAction.

FarmerBehaviour inherits the Behaviour interface and is dependent on Location.

PLAYERS, HUMANS AND FOOD

*** Assumption: Features state that only a Player can harvest food and place it inventory, but next feature mentions food can be eaten by damaged humans. We have assumed Humans are thus able to pick up food and store in inventory (therefore also farmers and players).
*** Assumption: A Human can harvest food (pick up) if standing on or next to food.

Humans can harvest food and as a result store the item in their inventory. No additional classes, associations or dependencies are needed to include this. Humans can use existing methods including but not limited to getPickUpAction, addItemToInventory and removeItem.

To eat food, the human must be damaged and have food in their inventory. To determine if this criterion is met the EatFoodBehaviour class was created. EatFoodBehaviour retrieves the Humans hitPoints and checks if it is below what Humans are initialised as (50). It also retrieves the list of Items in the Humans inventory using the getInventory method.

If the criteria are met, the Human can eat the food by executing the EatFoodAction, a class inherited from Action. Eating food recovers the human's health by 20 points and thus EatFoodAction has a method that increases the Humans hitPoints by 20. EatingFoodAction removes the Food instance using the removeItem method.

EatFoodBehaviour implements the Behaviour interface.

Implementing this feature using a behaviour that creates an action was a way to limit dependencies and associations whilst following the rules of design. Adding the behaviour to the Human class meant that Farmers also had these capabilities.

RISING FROM THE DEAD

A dead Human can now be reincarnated as a Zombie after 5-10 turns. To achieve this, we have added some attributes to the Human class so we can track how many turns have passed after their death. These attributes are corpseAge which tracks its age since death (initialised as 0) and reincarnationAge which is set to a random number between 5 and 10.

When a Human is killed it's ZombieCapability is set to UNDEAD. When this happens the only Behaviour they can implement on playTurn() is DeadBehaviour. DeadBehaviour implements the Behaviour interface and its role is to retrieve the Humans corpseAge and reincarnationAge and dependent on these numbers use the getAction method to execute either RotCorpseAction or RiseFromDeadAction.

If corpseAge < reincarnationAge the Human is not ready to be reincarnated yet.
RotCorpseAction is executed and a method is used to increment the Humans corpseAge.
If corpseAge = reincarnationAge the Human is ready to rise. RiseFromDeadAction is executed and removes the current Human and instantiates a new Zombie at the same location.

RotCorpseAction and RiseFromDeadAction are both inherited from the Action class.

Implementing a behaviour that is only active when a human is dead was an efficient way to introduce this capability. Dependencies and associations were limited by adding the attributes only to the Human class and creating external classes that had methods to interact with these attributes.

Class Responsibilities

Human

Included additional behaviours:

- FoodBehaviour
- DeadBehaviour

Included additional attributes:

- corpseAge initialised at 0
- reincarnationAge initialized at random number between 5 and 10

Farmer

Extends Human
Includes additional behaviour:

- FarmerBehaviour

CraftAction

Extends Action
Creates instance of ZombieWeapon if holding a ZombieLimb
If ZombieLimb = ZombieArm

- create instance of ZombieWeapon titled ZombieClub

If ZombieLimb = ZombieLeg

- create instance of ZombieWeapon titled ZombieMace

Removes instance of ZombieLimb
Add ZombieWeapon to inventory

ZombieWeapon

Extends WeaponItem
ZombieClub, damage = 30, verb = "clubs"
ZombieMace, damage = 40, verb - "maces"

FarmerBehaviour

Implements Behaviour Interface
Accessible by Farmer
Dependent on Location
Generates a

- SowingAction if the current Farmer is standing next to a patch of dirt
- FertlizeAction if the current Farmer is standing on unripe crop
- HarvestAction if current Farmer is standing on or next to a ripe crop

Loops through Farmers location and surrounding locations using getGround to check if the above actions are executable

FertilizeAction

Extends Action
When standing on an unripe crop, a Farmer can fertilize it, decreasing the time left to ripen by 10 turns
method to increment Crop.age by 10
method for menu description

- (name of Farmer) fertilized unripe crop

SowingAction

Extends Action

- When standing next to a patch of dirt, a Farmer has a 33% probability of sowing a crop on it
- happens every move
- 1/3 chance per turn its successful (use random number generator)
- if successful, creates instance of Crop class on location in map
- uses setGround to achieve this
- method for menu description
    - (name of Farmer) sowed a crop

HarvestAction

Extends Action

- When standing on or next to a ripe crop, a Farmer can harvest it for food
- Food is placed on the ground
- Create instance of Food
- Remove instance of Crop

Crop

Extends Ground

- age attribute
- tick method
- if age <20
    - Unripe
    - display Char 'c'
- if age >= 20
    - ripe
    - displayChar 'C'

Food

Extends Item

- attribute recoverHealth
- displayChar 'F'
- Food can be eaten by the player, or by damaged humans, to recover some health points
- Actor is the current player/human
- 20 health points

FoodBehaviour

Dependant on Behaviour Interface

- Accessible by Human
- Generates an EatFoodAction if the current Player/(damaged) Human holds food in their inventory

EatFoodAction

Extends Action

- Actor is current human/player
- Recover health points based on recoverHealth attribute of Food
- Removes Food instance
- method for menu description
    - (name of Player/human) ate food and recovered 20 health points

DeadBehaviour

Dependant on Behaviour Interface

- Accessible by Human when Zombie.Capability.UNDEAD

RotCorpseAction

Extends Action

- method to increment corpseAge
- method for menu description
    - (Human)'s corpse rots

RiseFromDeadAction

Extends Action

- Creates instance of ZombieActor
- removes instance of Human
- both at current location
- actioned when corpseAge = reincarnationAge
- method for menu description
    - a Zombie (name) rises from the dead!

ZombieAttackBehaviour

This class is responsible for returning an AttackTypeAction, for a specific Zombie
It will implement Behaviour.
It will store a list of AttackTypeBehaviours.

BiteBehaviour

This class is responsible for creating BiteActions.
It extends AttackTypeBehaviour.

BiteAction

This class is responsible for executing bite attacks.
It extends AttackTypeAction.

AttackTypeBehaviour

This abstract class is used for organising Behaviours that involve attacking, and reduces repeated code.
It implements Behaviour.

AttackTypeAction

This abstract class is used for organising Actions that involve attacking, and reduces repeated code.
It extends Action.

PickUpWeaponBehaviour

This class is responsible for creating PickUpItemActions for nearby Weapons.
It implements Behaviour.

DropLimbAction

This class is responsible for removing a Zombie’s limbs, and sometimes dropping it's weapon.
It extends Action.
I will store the limb that is being removed.

LimbFallOffBehaviour

This class is responsible for creating actions that remove a zombie’s limb, and maybe drop a weapon that the zombie is holding.