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.~~ Now, ZombieAttackPicker will store one main attack, and a list of special attacks. This allows us to add more "special attacks" without affecting the chances of a normal attack occurring.

Now instead of Zombie having an AttackBehaviour, it will have a ZombieAttackPicker that will pick which attack to use. In ZombieAttackPicker, for now it will ~~randomly select one of its AttackTypeBehaviours (for which now, 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.~~ ZombieAttackPicker will now randomly select whether to perform a main attack, or a special attack. If a main attack is chosen, it will return the Action for the main attack as there is only one. If a special attack is chosen, we will choose at random one of the special attacks from the list.

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, as Actions are already dependent on actors.

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.~~

Instead, we decided to remove the LimbFallOffBehaviour and DropLimbActions action as they need to be performed on actors, but these only occur on Zombies so moved the functionality in the Zombie's hurt method, so now whenever a zombie is hurt, 35% of the time the zombie will loose a limb it it has one to loose. We decided to do this as it removes unnecessary downcasting, and moves limb loss functionality into the only place it will be needed. (unless we add some other kind of monster, which we can then refactor out the limb loss functionality).

(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 ZombieAttackPicker, 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.~~

Instead, first we see if a Zombie has a proper weapon, and if it does we return the mainAttack. Then we see how many arms the zombie has. If it has two, there is 50% chance of performing a mainAttack, and 50% chance of performing a specialAttack. If it has one arm, there is a 25% chance of performing a mainAttack, and a 75% chance of performing a specialAttack. Finally, if a zombie has zero arms, it will always perform a specialAttack. If specialAttack is performed, it selects a random specialAttack within the list of specialAttacks.

We changed it to this so that if a Zombie has a weapon, it will use it.

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.~~

Instead, if a Zombie just lost an arm, we check how many arms the Zombie has. If it has one left, there is 50% chance of dropping its weapon. If it has zero arms left, it has a 100% chance of dropping its weapon. To drop a zombie's weapon, we create a DropItemAction, and execute it.

We implemented it like this instead, as we moved the functionality into the Zombie class.

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

ZOMBIELIMB AND ZOMBIEWEAPONCAPABILITY

Zombie limbs can currently take two forms; an arm or a leg. Now using class CraftWeaponAction a Player can spend their turn crafting a zombie limb into a more dangerous weapon; a club or a mace.

Arms, legs, clubs and maces are all instantiated from a single class ZombieLimb as they all share the same attributes. Zombielimb will have an attribute “potential” that can store what each limb can be crafted into. The potential of an arm will be set to a club, and the leg set to mace. There is currently no future potential of crafting clubs and maces so their potential will be set to null.

All features of each possible ZombieLimb will be stored In the ZombieWeaponCapability class. The ZombieWeaponCapability is a special enum class that appropriately stores corresponding variables for each zombie limb iteration. This includes the name, damage, display character, display verb and potential. There will also be corresponding methods to retrieve all this information during construction of a zombie limb.

Using enum was deemed most appropriate as it eliminated repeated code, is easily extendable and reduced dependencies.

The zombie limb’s potential will determine whether the player has the capability to craft or not. To give the Player the capability to craft weapons when holding a zombie limb, we are adding CraftWeaponAction to the ZombieLimb’s list of allowable actions. Only when the zombie limb’s potential is not null will the player be able to access this action.

The CraftWeaponAction class is inherited from Action. It removes the original ZombieLimb object from the player's inventory and creates a new instance of the ZombieWeapon class defined by the original potential and stores it in the Player's possession.

Since only the Player has the capability to craft weapons the creation of an additional behaviour was unnecessary. Instead we just needed to add a new capabilityCraftWeaponAction to the appropriate zombie limbs.

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 or crops. 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.

Actors will be able to pass over Crops so that they can use functionalities such as FertilizeAction.

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 and interacted with by actors and thus is an extension of the PortableItem class. It will have an attribute displayChar ‘f’ so that Players can recognise it on the map. Now that an item can be eaten, we will include a feature in the ItemInterface to check if an item is edible.

Now that these classes exist, we can create actions so that the Humans 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 farmer is in range of an instance of Dirt, and if this is true can execute SowingAction as their turn.

SowingAction is inherited from the Action class. When successful a Crop class is instantiated, and the ground is changed from Dirt to Crop. To achieve this SowingAction uses a random number generator to implement the probability and the Location method setGround to change the ground extension.

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 an additional method will be added to the Crop class that FertilizeAction class can call to increment the age by 10. An attribute is also included so that FarmerBehaviour can call and check if the Crop in the Farmer’s range is unripe.

FertilizeAction is an extension of the Action class in the Engine package. To check if a Farmer can execute FertilizeAction the FarmerBehaviour class checks if the Farmer is located at the same location as an unripe Crop instance. If this criterion is met FarmerBehaviour will get FertilizeAction.

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 SowingAction, 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, we create a new Food item, add it to the location 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
- After crop is harvested, ground returns to dirt
- Humans can eat food if food is in their inventory, however there's currently no requirements for humans to pick up items. Therefore humans cannot eat food.

FARMERBEHAVIOUR

In summary FarmerBehaviour loops through the Farmers location and surrounding locations using methods from Exit, checking for instances of Dirt and Crops. Depending 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. Using a single FarmerBehaviour class to retrieve all the farmers actions allows for extendability and eliminates repeated code.

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 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. 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 uses a method from the Food class to get its recovery amount (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 and Farmer classes allow them to utilise this feature. So that the Player can eat food an allowable action will be added to the Food class, allowing the Player to eat food if in their inventory.

RISING FROM THE DEAD

A dead Human can now be reincarnated as a Zombie after 5-10 turns. Currently corpses are initialised as a PortableItem class. Zombies will continue to do this but Humans when killed will now be an instance of the Corpse class. This allows different applications on the zombie corpses and human corpses.

The Corpse class is an extension of the PortableItem class but will have additional attributes to track the corpses age, a reincarnation age (random number between 5-10 turns) and necessary methods to tick the corps and retrieve information.

When the corpse's age reaches the random reincarnation age, Corpse will execute RiseFromDead. RiseFromDead checks a number of elements before allowing reincarnation:

- Is the corpse on the ground or in an actor's inventory
- Is there an actor standing on the location of the corpse
- Is there an adjacent free location that the corpse can be reborn at

A new Zombie object is created at a location depending on the above criteria and the Corpse object removed. If it is not possible for the Zombie to reincarnate at this turn due to obstacles like being surrounded by actors or terrain that an actor cannot pass, RiseFromDead will try again on the next turn.

Class Responsibilities

Human

Included additional behaviour FoodBehaviour so that damaged humans can eat food to recover

Farmer

Extends Human
Created a farmer (extension of Human). Includes behaviours

- FarmerBehaviour so they can sow, fertilize and harvest
- FoodBehaviour so that when damaged they can eat food
- WanderBehaviour so they can maneuver the map

FarmerBehaviour

Implements Behaviour Interface
Dependent on Location
Accessible by Farmer
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 Exit and Location 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
Calls method in crop to increase age by 10

SowingAction

Extends Action

- When standing next to a patch of dirt, a Farmer has a 33% probability of sowing a crop on it
- if successful, creates instance of Crop class on location in map
- uses setGround to achieve this

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
- Resets ground to Dirt

Crop

Extends Ground
Has an age attribute, tick method and method to return if the crop is ripe (age >=20)

Food

Extends PortableItem

- attribute recoverHealth (20 points)
- Food can be eaten by the player, or by damaged humans, to recover some health points
- Adds an allowable action EatFoodAction for players

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

CraftWeaponAction

Extends Action
Creates a club from an arm

Creates a mace from a leg
Creates and stores a new ZombieLimb object and removes the original from inventory

ZombieLimb

Extends WeaponItem
Creates ZombieLimb objects arm, leg, mace & club
Attribute that stores what the current instance can be crafted into

ZombieWeaponCapability

Enum class
Stores all appropriate values for zombie arm, leg, mace & club
Used in CraftWeaponAction and ZombieLimb to create a new ZombieLimb object

Corpse

Extends PortableItem class.
Class used for Human corpses only, allows them to rise from the dead using RiseFromDead
Has an age attribute and a randomly generated reincarnation age.
Age ticks every turn until the reincarnation age is met and RiseFromDead is executed

RiseFromDead

RiseFromDead checks a number of elements before allowing reincarnation:

- Is the corpse on the ground or in an actor's inventory
- Is there an actor standing on the location of the corpse
- Is there an adjacent free location that the corpse can be reborn at

Creates an instance of Zombie at an appropriate location.
Removes corpse item.
If a corpse is unable to rise from the dead because of not meeting above criterion, RiseFromDead is actioned next turn by incrementing the reincarnation age of the Corpse.

ZombieAttackPicker

This class is responsible for returning an AttackTypeAction, for a specific Zombie
It will implement Behaviour.
It will store a list of AttackTypeBehaviours special attacks, and one main attack.

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.~~