

## **Zombie Attacks**

**1 new class was implemented and 2 existing classes were changed.**

### **Zombie**

#### **INSTANCE VARIABLE**

1. A ZombieLimbs class was added. This class represents the limbs that the zombie has.
2. A new behavior called pickUpItemBehaviour will be added to the behaviors list.

#### **METHODS**

1. public IntrinsicWeapon getIntrinsicWeapon()

There'll a 50 percent chance that the zombie will bite. If it doesn't bite, it will punch instead.

2. public String zombieLoseLimbs()

This method calls the ZombieLimbs method called loseLimbs(). zombieLoseLimbs will return the String of the limbs lost.

3. public Action playTurn(Actions actions, Action lastAction, GameMap map, Display display)

A random double, "say" was created. If "say" <= 0.1 (10%), then Braaaaain will be printed out. The rest of the method remains the same. If an action can be carried out, that action will be carried out.

### **AttackAction**

#### **METHODS**

1. public String execute(Actor actor, GameMap map)

The actor will first get the weapon they have. If the actor is a zombie and the zombie's next attack is a bite, the zombie only has a 25% chance of a successful bite. If a successful bite occurs, then the zombie will gain 5 hitpoints. If the actor is a zombie and its next attack is punch, we will first check the number of arms the zombie has. If the zombie has 2 arms, the chances of success remain the same which is 50%. If the zombie has 1 arm, the chances of hitting are halved, which means the chances of failure is 75%, If the zombie has no arms, it is confirmed that the zombie has failed in attacking. If the actor is a human, there's a 50% chance that they will hit. If the hit was a success, there is a 25% chance that the zombie will lose its limbs. It will call lossLimbs() in the zombie class and that method will randomly make the zombie lose one of its limbs. Whichever limb the zombie loses, the limb will drop at the zombie's location. If the zombie loses its arms, we will first check if the zombie has any weapon. If the zombie has a weapon and the zombie has 1 arm, there'll be a 50% chance of the zombie. If the zombie has a weapon and has 2 arms, it will definitely drop its weapon. If the zombie loses a leg, then the zombie can move once every two moves, but can still attack as usual. If the zombie loses both legs, the zombie cannot move at all, although it can still attack. Whichever limb the zombie loses, the zombie limb will now be a **FallenZombiePart** object.

The rest of the method remains the same. If there is a successful hit, their victims will lose hit points depending on the attack on them. If the target's hitpoint is less than 0, the target will turn into a corpse and all their items are dropped.

### **PickUpItemBehaviour**

Inherits from Behaviour and will have a target class. Used by Zombie to pick up WeaponItems and used by Humans to pick up Food (Re-use of code follows Do Not Repeat Yourself Principle).

#### ATTRIBUTES

**1. Class<?> targetClass**

The class of the item actor should pick up.

#### METHODS

**1. PickUpItemBehaviour(Class<?> class)**

Initialize targetClass with class.

**2. @Override**

**getAction(Actor actor, GameMap map)**

Checks if location actor is standing on has instance of the target class. If it does, a PickUpItemAction will be returned, otherwise, null.

## **Beating up the Zombies**

**2 new classes were implemented**

### **ZombieLimbs**

This class represents the limbs that the zombie has. By default, the zombie has 2 hands and 2 legs.

#### **INSTANCE VARIABLE**

1. `private ArrayList<String> zombieLimbs= setZombieLimbs();`

This represents the list containing all the limbs that the zombie has.

2. `private int noOfHands=2;`

This represents the number of hands the zombie has.

3. `private int noOfLegs=2;`

This represents the number of legs the zombie has.

4. `private int noOfLimbs=4;`

This represents the total number of limbs the zombie has.

#### **METHODS**

1. `public ArrayList<String> setZombieLimbs()`

This method will initialize the zombieLimbs ArrayList.

2. `public String loseLimbs()`

This method will randomly cause the zombie to lose its limb. The number of hands, legs, and total limbs will be updated. This method will return the limb that the zombie lost.

3. `public int getNoOfHands()`

This method returns the number of hands the zombie has.

`public int getNoOfLegs()`

This method returns the number of legs the zombie currently has.

4. `public int getNoOfLimbs()`

This method returns the number of legs the zombie currently has.

### **FallenZombiePart**

Inherits PortableItem class

Instance Variable

1. `String typeLimb ( type zombie limb(hand/leg))`

#### **Constructor**

```
Public FallenZombiePart (String name,char displayChar,String type){  
    super(name,displayChar,true)  
    this.zombieLimb=type  
}
```

## METHOD

1. `getZombieLimbType()`{  
    Return "either hand or leg"}
2. Overrides the previous `getAllowableActions()`. It adds a new `craftAction` into `getAllowableActions`.

## **Crafting weapons**

**3 new classes were implemented**

### **CraftAction**

This is a class that inherits from action class. This class allows the player to craft weapon from zombie limbs.

#### **METHODS**

**1. public String execute(Actor actor, GameMap map)**

The player will first find zombie limb in the inventory. It will then clear the zombie limb from inventory. If the zombie limb is a hand, a zombieClub is created. Else a zombieMace is created. This weaponItem is then added back into the player's inventory. Menu description is then returned.

**2. Public string menuDescription(Actor actor)**

Returns the actor and the weapon name.

### **ZombieClubs**

Inherits from weaponItem

### **ZombieMace**

Inherits from weaponItem

## Rising from the dead

1 new class was implemented and 1 existing class was changed.

Inheritance was also used here to adhere to DRY design principle.

### **AttackAction**

#### METHODS

1. **@Override**

**public String execute(Actor actor, GameMap map)**

Instead of creating a new PortableItem for corpse, a Corpse object is instantiated to represent the corpse. This was changed because the corpse should turn into a zombie in 5-10 turns so we needed to keep track of the corpse's age using tick to do so.

### **HumanCorpse**

This class was created because unlike other portable items, a human corpse can turn into a zombie which should be encapsulated within its own class.

Inherits from PortableItem because it can be picked up by the player. A bit of random chance when the corpse will turn into a Zombie but within 5-10 turns.

#### ATTRIBUTES

1. **age**

A counter for how many turns it has been a corpse

2. **chance**

Chances (in decimal) of the corpse becoming a zombie. It starts at 0, and after five turns the chances increase by 20% meaning by 10 turns, the corpse will definitely have turned into a zombie.

3. **rand**

Random double generator.

4. **map**

The GameMap the corpse is on.

#### METHODS

1. **HumanCorpse(String name, GameMap map)**

Simple constructor. Calls super.

2. **@Override**

**tick(Location currentLocation, Actor actor)**

Checks exits around currentLocation and calls becomingUndead() for all exits and the corpse should be dropped from the actor's inventory. If no empty location is found, the game will be over because the player would be trapped with a zombie.

3. **@Override**

**tick(Location currentLocation)**

Calls becomingUndead(currentLocation) because if the corpse was on the ground, the zombie should be generated at that same coordinate. Corpse is removed from the ground.

4. **private void becomingUndead(Location location)**

Increments age. If age is more than 5, chance will increase by 0.2. If the random number generated is within the range, a Zombie object is created at the location specified. **Exception handling** if Zombie cannot be created at location because another actor might be in the way.

This method is declared **private** because it should only be used within this class.

## **Farmers and Food**

12 new classes were implemented and 3 existing classes were changed.

With respect to the Don't Repeat Yourself (DRY) Principle, inheritance was used throughout the implementation of this functionality.

### **Application**

Instantiate 4 farmers in main.

### **Human**

Add FindFoodBehaviour and PickUpItemBehaviour.

Change playTurn method so that it calls HuntBehaviour(Food.class, 5) when Human is at 50% health and then iterates through the collection of Behaviours.

### **Farmer**

Inherits from the Human class because a Farmer object should have the same characteristics as any other Human but is able to plant, fertilize and harvest crops.

#### **ATTRIBUTES**

##### **1. Behaviour[] behaviours**

A collection of Behaviour objects that allow the farmer to sow and fertilize. It will contain SowBehaviour, FertilizeBehaviour and HarvestBehaviour.

#### **METHODS**

##### **1. Farmer(String name)**

Constructor to make a Farmer object. Calls super(name, 'F', 80). The displayChar is chosen to be 'F' so that one can tell the difference between an ordinary human and a farmer. It'll be good for the player to know if there are any farmers left in the game. The hitPoints we chose is higher than an ordinary Human because farmers shouldn't be so easily killed as they are essential actors that provide the opportunity to heal, but it is still lower than the player's to make it challenging.

##### **2. playTurn(GameMap map)**

Calls super to inherit other Human behaviours. Should iterate through collection of Behaviours to return an Action for that turn.

### **GrowableGround**

This abstract class was implemented to reduce duplicated code as Crop and Tree had similar implementations.

#### **ATTRIBUTES**

##### **1. int age**

Number representing the number of turns since the GrowableGround object has been instantiated. Initialised to 0.

#### **METHODS**

##### **1. GrowableGround(char displayChar)**

Calls super(displayChar).

##### **2. @Override**

tick(Location location, char midDisplay, char olderDisplay)



Calls `super.tick(location)`. `displayChar` is changed to `midDisplay` when the age has reached 10. `displayChar` is changed to `olderDisplay` when the age has reached 20.

## Tree

Now inherits from `GrowableGround` instead.

## Crop

Inherits from `GrowableGround` because it should not be able to be picked up by player. Its three display characters as it ages are static and final to reduce excessive use of literals.

### ATTRIBUTES

1. **static final char YOUNG\_DISPLAY**  
When first planted, `displayChar` starts off as 'v'
2. **static final char MID\_DISPLAY**  
When its age reaches 10, `displayChar` changes to 'V'
3. **static final char OLD\_DISPLAY**  
When its age reaches 20, the crop is ripe and its `displayChar` changes to 'Y'

### METHODS

1. **Crop()**  
Calls `super(YOUNG_DISPLAY)`
2. **@Override**  
**tick(Location location)**  
Implemented to change its `displayChar` as the crop ripens. We wanted to use characters that were similar to each other but can also be told apart.
3. **@Override**  
**allowableActions(Actor actor, Location location, String direction)**  
If age of crop is not ripe and actor is Farmer, then allowableActions include `FertilizeAction`. If the age of the crop is more than 20 and the actor is Farmer or Player, then allowableActions include `HarvestAction`.

## SowBehaviour

Inherits from `Behaviour` and behaviour is only exhibited by Farmer. Must have random number generator for the chances a crop is sown.

### METHODS

1. **@Override**  
**getAction(Actor actor, GameMap map)**  
Checks if the ground is dirt at each exit from the actor. If it is dirt and `rand` is less than 0.33, `SowAction` is created.

## SowAction

Inherits from `Action`, instantiated by `SowBehaviour`.

### ATTRIBUTES

1. **Location cropLocation;**  
The coordinates where the crop is to be sown.

### METHODS

1. **SowAction(Location cropLocation)**  
Initialise this.cropLocation to cropLocation.
2. **@Override**  
**execute(Actor actor, GameMap map)**  
Creates a new Crop object. Set ground at cropLocation as Crop.
3. **@Override**  
**menuDescription(Actor actor)**  
Return a string saying actor sows a crop.

### **FertilizeBehaviour**

Inherits from Behaviour and behaviour is only exhibited by Farmer.

#### METHODS

1. **@Override**  
**getAction(Actor actor, GameMap map)**  
Checks if location where actor is standing on is a Crop. If it is, then FertilizeAction is returned, else, null.

### **FertilizeAction**

Inherits from Action, instantiated by FertilizeBehaviour.

#### ATTRIBUTES

1. **Location cropLocation**  
The location of the crop to be fertilized.

#### METHODS

1. **FertilizeAction(Location location)**  
Initialize cropLocation as location.
2. **@Override**  
**execute(Actor actor, GameMap map)**  
A loop calling cropLocation.tick() 10 times (The crop's time left to ripen is decreased by 10 turns).
3. **@Override**  
**menuDescription(Actor actor)**  
Return a string saying actor fertilizes a crop.

### **HarvestBehaviour**

Inherits from Behaviour and behaviour is only exhibited by Farmer.

#### METHODS

1. **@Override**  
**getAction(Actor actor, GameMap map)**  
Checks if location where actor is standing on and location of exits is a ripe crop. If it is, then HarvestAction is returned, else, null.

### **HarvestAction**

## ATTRIBUTES

### 1. **Location cropLocation**

Location of ripe crop.

## METHODS

### 1. **HarvestAction(Location location)**

Initialise cropLocation with location.

### 2. **@Override**

**execute(Actor actor, GameMap map)**

A Food object is created. If the actor is a Farmer, the Food is dropped at cropLocation. If the actor is Player, the Food is added to Player's inventory. The ground at cropLocation is set to Dirt.

### 3. **@Override**

**menuDescription(Actor actor)**

Return a string saying actor harvests a crop.

## **FindFoodBehaviour**

Inherits from HuntBehaviour (re-use of code follows Do Not Repeat Yourself principle) and behaviour is exhibited by Human.

If Human is at 50% health, then they will 'hunt' for Food type objects.

## **HealAction**

Inherits Action and is called when an item heals an actor.

## ATTRIBUTES

### 1. **Item healingItem**

The item which does the healing.

### 2. **int pointsHealed**

The number of points the actor is going to get healed by.

## METHODS

### 1. **HealAction(Item healingItem, int pointsHealed)**

Initialise healingItem and pointsHealed.

### 2. **@Override**

**execute(Actor actor, GameMap map)**

Calls actor.heal(pointsHealed), then removes healingItem from the actor's inventory.

### 3. **@Override**

**menuDescription(Actor actor)**

Returns a string saying actor healed by pointsHealed.

## **Food**

Inherits from PortableItem and adds HealAction to allowableActions.

## ATTRIBUTES

### 1. **protected int healPoints**

This attribute is protected to allow this class to be extensible in the future (maybe different types of food inherit from this class and have different healPoints)

