**DESIGN RATIONALE**

ASSIGNMENT 1

Cedric Liang, Nathan Vaughan

**Zombie Attacks**

We inspect the existing code to note that the randomisation for the hit probability is attached to AttackAction.execute. The generation for the hit probability is the same for each weapon, at 50%. We need to handle the possibility for each weapon to have its own hit probability.

Since hit probability is not natively supported by the Weapon class, we can only crudely hardcode it into the AttackAction class. We create new WeaponItem classes ZombieArm, ZombieLeg, ZombieMace, ZombieClub, and new IntrinsicWeapon classes ZombiePunch, HumanPunch and ZombieBite. We then check in the AttackAction class which of these the Weapon is an instance of, retrieving and hardcoding the weapon attack probabilities in there.

We considered creating a subclass of Weapon that would add variables corresponding to the hit probability, then pointing *all* the weapons in the game class to be a subclass of this new subclass of weapon, but this runs into two issues: first, the Actor.getWeapon method only refers to a Weapon and this is in the Engine package, and two, this implementation is ‘cheating’ in a way, and doesn’t seem like good code practice.

If the weapon is of type ZombieBite, we call the Actor.heal method on the attacker to heal the zombie.

We implement some code in the Zombie class before it checks the possible behaviours to check whether it can pick up a weapon. If it can, we simply return the PickUpItemAction in playTurn() before it can check the other behaviours.

We can do the same thing for saying brains, except this should be a simple print calling the println method in the Display class, and it doesn’t consume an action.

**Beating up the Zombies**

We implement new private attributes in the Zombie class that correspond to the number of legs and arms it has.

We implement a new method in ActorInterfaces called takeDamage that handles all actors in the game’s damage taking logic. We make it also have a reference to the location of the actor, by feeding in 'map', then overwrite the takeDamage(int damage, GameMap map) method in zombie to handle gamelogic about the subtraction of limb count and the creating of an object at the location.

The reason for this because the basic Actor.hurt method does not contain a reference to the location of any of the actors, and implementing a method in ActorInterfaces allows for each Actor to handle being damaged with different logic, and reduces bloating of the AttackAction class.

We initialise these attributes in the Zombie class to have two arms and legs.

We can code the impact these private attributes have on the zombie’s attack probabilities by coding the randomisation as being dependent on the zombie’s arm and leg private attributes in the getIntrinsicWeapon override in the Zombie class. We can handle the dropping of the weapons in the takeDamage method by simply calling the dropItemAction method.

Since takeDamage has a reference to the location on the map, we can also create ZombieArm or ZombieLeg as new objects at the location.

World already stores information about what each Actor did in the previous turn, and this can be accessed in the playTurn method in the Zombie class. We can then simply perform a type check on the previous move depending on the number of legs the zombie has (if lastAction instanceof MoveActorAction). This way, we can slow or immobilise the Zombie by altering what behaviours are available to it on a given turn depending on its last action.

**Crafting Weapons**

We need to create a new class called CraftAction. This action takes in an instance of the parent item, the child item, and the Action simply removes the parent Item from the map or the Actor’s inventory and the child item is added to the map or inventory.

To allow the Player to craft certain Items, we need to add CraftAction as an allowableAction to the parent Item, where the argument passed into CraftAction are the parent item itself and a newly created instance of the child Item. As such, the Item associations are stored in each parent item. Each parent knows its child, but the child does not know its parent.

When an Actor stands near a craftable Item, the associated CraftAction will become an option available to the Actor, and the Actor can execute that Action. Since only Players have direct access to Actions (NPCs must have their Actions governed by Behaviours or their own PlayTurn logic), only Players can perform CraftAction through the allowableAction mechanism.

We then create these new weapon classes.

**Rising from the Dead**

It doesn’t make sense to support the possibility to move corpses (almost all industry AAA shooters and RPGs don’t support this capability), so we will create a new class called StationaryItem. We create Corpse as a subclass of StationaryItem. Since we need to differentiate between corpses that were killed by zombies and corpses that were killed by human (to support any future human vs human combat), we create a new class called InfectedCorpse, which is a subclass of Corpse. We give this corpse a private attribute called age, and a private attribute called reanimateWhen. The age starts at 0 and increments one per turn, representing the age. reanimateWhen initialises at a random value between 5 and 10. On the turn when age increments above reanimateWhen, the InfectedCorpse is removed from the map and an instance of Zombie is created at that location.

When a human dies, we perform the check of whether a human or a zombie killed them simply by performing a check in AttackAction of the attacking actor’s undeadness Capability. This is simple to implement because this is already where the code for the *creation* of the corpse item is being handled.

**Farmers and Food**

We create an extension to Human, called Farmer. We add Crop as an extension of Ground.

We create a new set of Action, called SowAction, HarvestAction and FertiliseAction, each of which handle the changing and manipulation of the map around it accordingly, by changing the Ground at that Location. In order to facilitate the behaviour of the Farmer, we create a new class FarmingBehaviour.

FarmingBehaviour will handle the prioritisation of SowAction, HarvestAction and FertiliseAction for the farmer. The farmer will have FarmingBehaviour as a behaviour from which it can choose an action on any given turn. The player will have access to HarvestAction only through the allowableActions mechanism.

We can handle the *permissions for access to these actions* by creating a new Capability class, called CropCapability. This enum will have two modes, *HARVESTABLE and FERTILISABLE.* These correspond to the state a Crop is in and removes the need to create separate classes for ripe crops and unripe crops. A Crop is created with the *FERTILISABLE* capability. We allow a crop to only be fertilised once, and the *FERTILISABLE* capability is removed. When the crop matures, the displayChar is changed and it takes on the *HARVESTABLE* capability.

The presence of the CropCapability associated with Ground will dictate what actions are available to various Actors that can interact with Crop.

We create a new class called Food. This is an extension of PortableItem since food can be carried in the inventory. We create a new Action called EatFoodAction that restores HP to the Actor. We can code this action into a Human’s playTurn method in order to give them the condition to eat food (for example, only a Human that has lost HP can eat food).

In addition to the food mechanism required by the specifications, I added HuntGroundBehaviour and HuntItemBehaviour to facilitate the ability for AI to hunt down more than just Actors. If a Farmer does not perform a farming Action in a certain turn, instead of Wandering, they now move towards the nearest existing Crop within a five square radius so that crops tend to be grown next to existing crops. Humans will now hunt for Food and add it to their inventory if possible, instead of wandering. They will consume this food when they lose HP.