**DESIGN RATIONALE**

**Implementing Crafting weapons**

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 ZombieArs 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.

**Implementing Rising from the dead**

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.

**Implementing Farmers and food**

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.