**Design Rationale**

For **Zombie attacks**, we created a ZombieAttackBehaviour which inherits AttackBehaviour and BiteAction class to inherit the AttackAction class. Since most of the methods and attributes are the same, we used the principle **DRY** (Don’t Repeat Yourself). Within the BiteAction class, we used named constants to set the probability, damage and health restored of each bite to be the same using the principle of **DRY**. Before beginning each turn a zombie will reuse methods (map.locationOf().getItems()) from Location class to obtain items on the spot it is standing on to pick up weapons.

For **Beating up the Zombies**, we added two attributes, armsNumber and legsNumber to the Zombie class. Using the design principle that **classes should be responsible for their own properties**, it knows its own armsNumber and legsNumber at all times. Since the maximum number of legs or arms is 2 for each zombie, we use assertions to ensure that the value does not exceed 2, following the principle of **FF** (Fail Fast). Since zombies can lose limbs to ANY attack that causes damage, to implement the losing limbs feature, we used the inherited method hurt() from Actor class and override to add the feature using the principle **DRY**. Keeping all the methods and attributes within the Zombie class, we followed the design principle of **grouping elements that must depend on each other together inside an encapsulation boundary** of a class.

For **Crafting Weapons**, we created a crafting class. Crafting is added as a capability of the item zombieArm and zombieLeg using the addCapability() method of the item class. Crafting class has 2 attributes, club and mace which are both WeaponItem objects because the attributes and methods that club and mace are required to use are similar to objects of Weapon item class, using the principle of Don’t Repeat Yourself. The method createClub() takes a zombieArm item as a constructor to return a club while the method createMace() takes a zombieLeg item as a constructor to return a mace.

For **Rising from the Dead**, we add a method toCorpse() which converts the Human to a Corpse object is the human is dead. We check if the human is dead by using the isConscious() that the Human class inherits from the Actor class so that we abide by the **DRY** principle. The method creates a Corpse object at the location using the addItem() method of the Location class.

We created a Corpse class which inherits the Item class as the methods and attributes required by these classes are similar. The Corpse class which has 1 attribute, conversionCounter. We use the Random class to generate an integer between 5 to 10 to set as our conversionCounter. We used Assertions to ensure the conversionCounter is in the specified range, following the **Fail Fast** principle. We use the tick() method of the item class to compare with conversionCounter each turn. When the number of turns that have passed is equal to our conversionCounter(), we use the addActor() method of the Location class to add a zombie at the location.

For **Farmers and Food**, we created three new classes, the Farmer class, the Crop class and the Food class. We made the Farmer class a subclass of the Human class ,and the Crop class and the Food class a subclass of Item class as many of the attributes and methods used for these classes are the same as their parent classes, this is to abide by the **DRY** principle.

The crop class has 1 attribute, ripeCounter. Using the design principle that **classes should be responsible for their own properties**, it knows its own ripeCounter at all times. The isRipe() method is used to determine if the crop is ripe by checking if ripeCounter is equal to 0. We use assertions to ensure that ripeCounter is not less than 0, which abides by the **Fail Fast** principle. Every turn, the ripeCounter reduces by 1 until it reaches 0 by using the tick() method of the item class. The tick() method informs the Crop class of the passage of time.

We used constants to set the probability of the Farmer class to sow a crop on a patch of dirt if next to it. Every turn the sowing() method will reuse the methods (Location.getGround()) of the Location class to check if the location next to the Farmer is a patch of dirt. The fertilize() method uses methods of the Location class and Crop class to check if the Farmer is standing on an unripe crop. If so, the fertilize() method will reduce the ripeCounter of the crop by 10. The harvest() method uses methods of the Location class and Crop class to check if the Farmer is standing on a ripe crop, if so the crop will be harvested into food and reuses the Location class methods(location.setGround()) to turn the location into a patch of dirt. The Food item will be dropped on the ground.

The Food class has 1 attribute, nutrients which is the amount of health points that the player will recover after eating the Food object. We use to constants to set the value of nutrients to abide by **DRY**.