**Design Rationale**

For **Zombie attacks**, we created a ZombieAttackBehaviour which inherits AttackBehaviour and ZombieAttackAction 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 ZombieAttackAction 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 an ArrayList containing classes ZombieArm and ZombieLeg that inherit WeaponItem (following **DRY** principle) to the Zombie class. Using the design principle that **classes should be responsible for their own properties**, it holds its own items. ~~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 CraftingAction class which inherits the Action class. Result of crafting items is added as an ArrayList attribute in ZombieArm and ZombieLeg using the design principle that **classes should be reseponsible for their own properties**. The CraftingAction class takes in the parameter as an Item (the item to be crafted). Two new classes were created which inherits WeaponItem which are ZombieClub and ZombieMace using the principle of **DRY**.

For **Rising from the Dead**, we created a ZombieCorpse class which inherits Item class. We further changed the ZombieAttackAction class to be able to knock out humans and create ZombieCorpse instead of normal corpse item. The ZombieAttackAction class uses the addItem() method if the Location Class to add a ZombieCorpse at the location. The ZombieCorpse uses the tick() method from its parent class Item so that we abide by the **DRY** principle.

The ZombieCorpse class 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 a Farmer class which is subclass of the Human class as many of the attributes and methods used by the Farmer class are already used by the Human class. This helps us abide by the **DRY** principle. Since the Farmer has unique behaviours only found in this class, we created a FarmerBehaviour class which implements the Behaviour interface. The FarmerBehaviour class uses the subclasses of the FarmerAction class, which itself is a subclass of the Action class, as its actions. By doing so we have created an encapsulation boundary. This helps us **reduce the dependencies** of the Farmer class as much as possible. This abides by the **principle of grouping elements that depends on each other together in an encapsulation boundary to minimize dependencies that cross the encapsulation boundary.**

The subclasses of the FarmerAction class are the SowingAction class, FertilizeAction class and the HarvestAction class. For the SowingAction class, we use constants to set the probability of the class to sow a patch of dirt, and use constants to set the number of turns it will reduce in a crop for it to ripen in the FertilizeAction class.

We created a Crop class which is a subclass of the Item class. 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.

We created a Food class which is a subclass of the Item class. 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**.