**Design Rationale**

**Zombie attacks**

Firstly, we created new classes **ZombieAttackBehaviour** which **inherits** AttackBehaviour and **ZombieAttackAction** which **inherits** AttackAction. We changed Zombie class to use ZombieAttackBehaviour instead of AttackBehaviour. We used inheritance to follow the principle **don’t repeat yourself (DRY).** Since most of the methods and attributes are the same, we **reused methods** and **added protected methods** to AttackBehaviour and AttackAction to prevent repetitive code when overriding methods.

In ZombieAttackBehaviour, Zombies check if they are standing on a WeaponItem and picks it up, instead of having another independent class, following **reduce dependencies** principle. In ZombieAttackAction, we used **private static final constants** to set the miss chances, bite damage, health restored of each bite to be the same using the principle of **DRY** and to **prevent privacy leaks**. This also follows the principle of **classes should be responsible for their own properties** because ZombieAttackAction holds its own properties. We **created private methods** to prevent repetition of code for a normal or bite attack, following **DRY** principle.

In the Zombie class, we added **private static constants** to set the string of zombie dialogue and the chance for the dialogue to be printed. In the playTurn method, we allowed the dialogue to be printed based on the constant chance set and to return DoNothingAction for the turn instead of creating and using another Action subclass. This is keeping with the principle of **reducing dependencies** as using another Action subclass will add a dependency. We also followed the principle of **grouping elements that must depend on each other together inside an encapsulation boundary** of a class and **classes should be responsible for their own properties** since the Zombie class holds its own dialogue and chance.

**Beating up the Zombies**

Firstly, we created new classes **ZombieArm** and **ZombieLeg** which **inherit** WeaponItem (following **DRY** principle). We added four attributes in the Zombie class, armsNumber, legsNumber, LIMB\_LOSS\_CHANCE (**static final**) and isSecondTurn since these values were going to be used multiple time, so we followed the principle of **DRY**. Using the design principle that **classes should be responsible for their own properties**, Zombie instances knows its own number of limbs 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 lose limbs to an attack that causes damage, to implement the losing limbs feature, we used AttackAction class to call a method in the target to reduce their limbs and drop the limbs on the ground. To **avoid downcasting**, we added a loseLimb method in ActorInterface. We used random doubles between 0 and 1 to choose which limb to be knocked off and to follow the **DRY** principle we **created private methods** to be used within this method to prevent code from being repeated. Keeping all the methods and attributes within the Zombie class, we followed the design principle of **reducing dependencies** by **grouping elements that must depend on each other together inside an encapsulation boundary** of a class.

To implement changes to zombie attacks depending on number of arms, we allowed ZombieAttackBehaviour and ZombieAttackAction to accept armsNumber and refactored the getAction and execute methods to add these features. We created **private methods** to be used within these methods to prevent code from being repeated, using the **DRY** principle.

To implement changes to zombie movement we separated attack behaviour and move behaviours in to different attributes and used a the Boolean attribute, isSecondTurn to determine the current turn according to the principle that **classes should be responsible for their own properties** (whether it is this Zombie’s second turn or not) and **reducing dependencies** by **grouping elements that must depend on each other together inside an encapsulation boundary** (isSecondTurn and legsNumber) of a class. The Boolean will check if it is a second turn and respond accordingly. We used the design principle of **avoid variables with hidden meanings** by clearly defining the attribute name.

**Crafting Weapons**

We created a CraftingAction class which **inherits** the Action class. Result of crafting items is added as an attribute in ZombieArm and ZombieLeg using the design principle that **classes should be reseponsible for their own properties**. CraftingAction takes in the item to be crafted into and removes the item from player’s inventory and adds crafted item. We added a new method in the Item Interface which allows CraftingAction to access the result of crafting an Item. By using the interface, we have prepared all our Item subclasses for expansion if we need to expand CraftingAction to these classes. This follows the principle of **DRY** so that we do not need to add a new item every time we expand CraftingAction to an Item subclass. Two new classes were created which **inherits** WeaponItem which are ZombieClub and ZombieMace using the principle of **DRY**.

**Rising from the Dead**

Firstly, we created a ZombieCorpse class which **inherits** PortableItem class (following **DRY**). ZombieCorpse has a conversion counter attribute, a zombie attribute and two **final static** attributes for minimum and maximum turns it takes for a corpse to convert to a Zombie. These attributes follow the **DRY** principle. Since a ZombieCorpse knows its own conversion counter at all times, it follows the principle that **classes should be reseponsible for their own properties.**

We further **refactored methods** in the ZombieAttackAction class to be able drop a ZombieCorpse instead of normal corpse Item after killing a target following the **DRY** principle. This also **reduces redundancy** of creating another action class to create a ZombieCorpse instance. ZombieCorpse uses the tick() methods from its parent class Item so that we abide by the **DRY** principle.

We used 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 decrease conversionCounter every turn and check if it is 0. If conversionCounter is 0, it removes the corpse from the ground and adds a Zombie actor in its place. Since everything is within the ZombieCorpse class, we followed the principle **of grouping elements that depends on each other together in an encapsulation boundary to minimize dependencies that cross the encapsulation boundary.**

**Farmers and Food**

We created a Farmer class which **inherits** 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 **subclasses** of the Action class are the SowAction class, FertilizeAction class and the HarvestAction class. For the SowAction class, we used random doubles between 0 to 1 in FarmingBehaviour to get the probability of the class to sow a patch of dirt, and used **constants** to set the number of turns it will reduce in a crop for it to ripen in the FertilizeAction class. Usage of constants follows the **DRY** principle.

We created a Crop class which is a **subclass** of the Ground class. The crop class has 1 attribute, ripeAge. Using the design principle that **classes should be responsible for their own properties**, it knows its own ripeAge at all times. The ripeAge attribute is a constant to abide by the **DRY** principle. We used **Assertions** to ensure that ripeAge never decrements below 0. This follows the **Fail Fast** principle.

We created a Food class which is a **subclass** of the Item class. The Food class has 1 **constant** attribute, NUTRIENTS(static final) which is the amount of health points that the player will recover after eating the Food object. The use of constants to set the value of NUTRIENTS abides by **DRY**.

We created an EatAction class which **inherits** Action. This follows the principle **DRY**. This action allows Players and Humans to be able to restore their health points by eating food.

To allow Human actors to eat Food when damaged, we changed the playTurn() **inherited** method to check for Food on the ground if the Human is damaged and consume it. This follows **DRY** as we **reuse methods** from Actor.