**Design Rationale**

**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 **reused methods** and added a performAttack method to ZombieAttackAction to prevent duplicate code for normal and bite attacks. ~~to AttackBehaviour and AttackAction~~ to prevent repetitive code to follow 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**. We changed Zombie class attributes to use ZombieAttackBehaviour instead of AttackBehaviour to **reduce dependencies** (ReD). Within the Zombie class, we added a private String **constant** zombieDialogue and a **constant** chance for the dialogue to be printed. This is keeping with the principle of **ReD and** **grouping elements that must depend on each other together inside an encapsulation boundary** of a class because we chose not to create a new Behaviour class.

**Beating up the Zombies**

We created classes ZombieArm and ZombieLeg that **inherit** WeaponItem (following **DRY** principle). 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 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 can lose limbs to ANY attack that causes damage, to implement the losing limbs feature, **we reused methods** the from Actor class (hurt()) 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.

To implement changes to zombie attacks depending on number of arms, we allowed ZombieAttackBehaviour to accept armsNumber and refactored its methods to add these features. We did not create a new AttackBehaviour class to accommodate this feature to follow the design principle of **reduce redundancy**.

To implement changes to zombie movement we separated attack behaviour and move behaviours in to different attributes and added a new Boolean attribute, isSecondTurn according to the principle that **classes should be responsible for their own properties** and **grouping elements that must depend on each other together inside an encapsulation boundary** 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. Two new classes were created which **inherits** WeaponItem which are ZombieClub and ZombieMace using the principle of **DRY**.

**Rising from the Dead**

We created a ZombieCorpse class which **inherits** Item class. We further **added** to the ZombieAttackAction class to be able to knock out humans and create ZombieCorpse instead of normal corpse item since ZombieCorpse can only be created by Zombies knocking out Humans, abiding by the **DRY** principle. This also **reduces redundancy** of creating another action class to create ZombieCorpse. The ZombieCorpse uses the tick() method from its parent class Item so that we abide by the **DRY** principle.

The ZombieCorpse class has 2 attributes, conversionTurn and conversionCounter. We use the Random class to generate an integer between 5 to 10 to set as our conversionTurn. We **used Assertions** to ensure the conversionTurn is in the specified range, following the **Fail Fast** principle. We use the tick() method of the item class to increment conversionCounter every turn and compare with conversionTurn. If conversionCounter equal to conversionTurn, 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 use **constants** in FarmingBehaviour 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. Usage of constants follows the **DRY** principle.

We created a Crop class which is a **subclass** of the Ground class. The crop class has 2 attributes, ripeCounter and ripeAge. Using the design principle that **classes should be responsible for their own properties**, it knows its own ripeCounter and ripeAge at all times. The ripeAge attribute is a constant to abide by the **DRY** principle. We use assertions to ensure that ripeCounter is not less than 0, which abides by the **Fail Fast** principle. To allow different outcomes when harvested by farmers and by the player, we created PlayerHarvestAction class which **inherits** HarvestAction class which follows principle of **DRY**.

We created a Food class which is a **subclass** of the Item class. The Food class has 1 **constant** attribute, nutrients 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 to 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.