**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 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. Zombies also check if they are standing on a WeaponItem and picks it up, instead of having another independent Action class, following **reduce dependencies** principle. 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. We used a private method to prevent repeated code in tick() when a ZombieCorpse is on the ground or being carried to follow **DRY**. 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.

**Going to Town**

We created a new GameMap with its own FancyGroundFactory with new terrain classes which **inherits** an **abstract class** SolidTerrain within the Application class to follow the principle of **of grouping elements that depends on each other together in an encapsulation boundary to minimize dependencies that cross the encapsulation boundary.**

To allow Players to travel between GameMaps, we created a Vehicle class that **inherits** Item class to follow **DRY**. The Vehicle is placed on both GameMaps and uses a MoveActorAction to transport players between GameMaps. Since we did not create a new Action to move the Player, we followed the principle of **DRY**.

**Shotgun and Sniper Rifle**

Firstly, we created **abstract** classes GunItem and Ammo which **inherits** WeaponItem and PortableItem respectively. This is to **prevent repeated code** when calling methods in the different GunItems and Ammo Items following the principle of **DRY**. In the Player class, we looped over the Player’s inventory every turn to check for GunItem instances and if they were loaded. If their ammo is in the Player’s inventory, their respective are Actions are added. Each GunItem **knows its own Action** to be added and **its** **own ammo type**, following the principle that **classes should be responsible for their own properties.** Since each GunItem knows its own ammo and Actions, we **reduced dependencies** because Player class is not dependent on them.

**Shotgun**

We created Items for shotgun: Shotgun and ShotgunAmmo classes which **inherit** GunItem and Ammo respectively. We created a ShotgunAction class which **inherits** Action that provides a submenu for Players that choose to use the Shotgun. This follows the **DRY** principle as we did not create a new class to show the submenu. We further created two new Action classes, ShotgunShootAction and ShotgunAttackAction which **inherits** Action and AttackAction respectively.

ShotgunAction **reuses methods** from the engine to obtain the Exit directions the Player can shoot in and creates a ShotgunShootAction for each direction.

In ShotgunShootAction, it scans the area blasted in the direction chosen by **reusing methods** in the Exit class. Since it mostly depends on the Exit class to obtain the area affected, we **reduced dependencies** to other classes. ShotgunAttackAction is created for every Actor affected in the area. It also reduces the ammo by calling the reduceAmmo() method in Shotgun, which is **inherited** from GunItem. Since the Shotgun is responsible for handling its own ammo, it follows the principle that **classes should be responsible for their own properties.**

In ShotgunAttackAction, it has **private static final constants** representing the damage and the miss chance of the Shotgun. It **reuses protected** **methods** from its parent class, AttackAction, to decrease the health of the Actors affected, following **DRY**.

**Sniper Rifle**

We created Items for sniper rifle: SniperRifle and SniperAmmo classes which **inherit** GunItem and Ammo respectively. We created a SniperAction class which **inherits** Action that provides a submenu for Players that choose to use the SniperRifle. We also created SniperAimAction and SniperShootAction for aiming and shooting Actions which **inherit** Action and AttackAction respectively. This follows the **DRY** principle. In the Player class, we added a **private** **aims** **attribute** which tracks the number of aims the Player has on a target. We **added getter and setter methods** to add and clear the aims to reduce privacy leaks. Since using the SniperRifle is only an Action the Player can do, we followed the principle that **classes should be responsible for their own properties** by allowing the Player class to know its own aim count at all times.

In SniperAction, it checks the number of aims the Player has before proceeding. If the Player has not aimed before this, they are given a new set of SniperAimAction and SniperShootAction for Zombie Actors on the map in a submenu. We **reused methods** from the engine to scan the whole map for valid targets. Each SniperAimAction is **responsible for its own SniperShootAction**, and can be accessed by the getNextAction() method. This follows the principle that **classes should be responsible for their own properties.** If the Player has aimed before this, it clears the number of aims and returns the same previous Actions available.

In SniperAimAction, it adds to the Player’s aim count.

In SniperShootAction, it has **private static final constants** representing the damage and the miss chance of the SniperRifle. It **reuses protected** **methods** from its parent class, AttackAction, to decrease the health or kill the Actors affected, following **DRY**. It also reduces the ammo by calling the reduceAmmo() method in the SniperRifle, which is **inherited** from GunItem. Since the SniperRifle is responsible for its own ammo type, it follows the principle that **classes should be responsible for their own properties**

**Mambo Marie**

We created a MamboMarie class which **inherits** the ZombieActor class as it is a new type of actor. We created counters to check if the number of turns for Mambo Marie to disappear and for her to chant have been met. We created **getter methods to access the counters to prevent any privacy leaks**.

We created a CompoundMap class which **inherits** the GameMap class. The CompoundMap class **reuses** methods from the GameMap class to keep track of the number of turns that Mambo Marie has been spawned on the map. This abides by the **DRY** principle. As Mambo Marie only spawns in the compound, CompoundMap is **responsible** for the spawning and removal of Mambo Marie as **classes should be responsible for their own properties**. CompoundMap uses Boolean checks to check if Mambo Marie has already spawned and checks whether Mambo Marie has been killed.

We created a ChantAction class that **inherits** the Action class. The ChantAction summons Zombies at random locations of the map. The ChantAction class **reuses the methods** in the GameMap class to get the range of x values and range of y values of the game map to generate a random coordinate to spawn a Zombie. This abides by the **DRY** principle.

**Ending the Game**

We created a NewWorld class that **inherits** the World class as many of the methods used by NewWorld are already implemented in World. This abides by the **DRY** principle as NewWorld **reuses** methods instead of creating new redundant ones. NewWorld uses methods to keep track of the number of humans and zombies in the compound every turn. NewWorld terminates the loop that runs the game when the conditions to win or lose have been met. NewWorld checks at the start of every turn if the Player has chosen to quit the game, if they have NewWorld will end the game immediately.

We changed the type of game map used by the compound to an instance of CompoundMap. CompoundMap keeps track of all the zombies and humans in the map by using counters. The counters are updated every time an actor is added to or removed from the map. CompoundMap also checks if Mambo Marie is still alive every turn. This allows CompoundMap **to be responsible for its own actors at all times**. The Player wins when all the zombies in the compound and Mambo Marie have been killed. The Player loses if all the other humans in the compound have been killed or the player has been killed. We created **getter methods to access these attributes to prevent any privacy leaks**.

We created a QuitGameAction which **inherits** the Action class. The Player is given the option to quit the game every turn. If selected, the game ends immediately.

**Bonus Features**

**Bracelet of Kyo**

An item that **inherits** PortableItem. When picked by the Player, it increases the Player’s health by using **mutator methods** present in the Player class and decreases the health of the Player as if they haven’t worn it when dropped. The use of **mutator methods prevents privacy leaks from occurring**. We added **private static final constants** to BraceletOfKyo class as the health recovered due to the bracelet is constant. We used methods in the Player class to increase and decrease the health of the Player proportionally using percentage when the bracelet is worn and removed. This prevents any exploitation of the feature.