Design Rationale

Zombie Attacks

Since bite attacks are part of a Zombie's intrinsic attack, we will slightly modify the Zombie's getIntrinsicWeapon() method where it either returns an intrinsic punch attack or an intrinsic bite attack based on a randomiser (e.g. *rand.nextInt(100)* < *biteChance*). As such, the Zombie could either bite or punch.

Since the Player and Zombie share the AttackAction class to attack, it will be unideal to handle the Zombie's properties such as recovering HP from bite attacks and losing limbs within that class. What if a new type of Actor comes in? For example, maybe a lion that bites as well? Lions can definitely bite a Human much more accurately than a Zombie and they definitely do not recover any health from biting. So it is hard to manage if all the implementations are done within AttackAction. As such, 2 methods are introduced from the ActorInterface which are attackMissed() and bonusEffect().

attackMissed() will check if an Actor misses an attack or not. As such, we can change the accuracy of a Zombie's bite attack within the Zombie class as this accuracy difference is unique to Zombies only. Meanwhile, bonusEffect() is a method that allows for additional changes to an Actor's properties. AttackAction will call this method, along with the hurt() method, where in the Zombie class, it will restore the Zombie's health if they bite, or drop their limbs on the map if they are damaged. As such, all the implementation details and knowledge of them are encapsulated within Zombie since such special properties are unique to Zombies alone.

This means that the attacker does not need to know the type of Actor they are hitting or anything about how the Actor should respond when they get hit. The AttackAction class only tells the Zombie to take damage and to trigger their bonus side effects but the Zombie class handles the details. Through this approach, we have applied the "Tell, Don't Ask" principle and encapsulation helps us to minimize a potential amount of dependencies in AttackAction.

In regards to the Zombie saying "Braaaaains", we added a new Behaviour object called SpeakBehaviour into the Zombie's behaviours array. SpeakBehaviour is initialised with 2 arguments which are the sentence the Zombie will speak, and the probability of them saying it, which will be "Braaaaains" and 10 respectively. As a result, it will return a SpeakAction that simply returns the sentence "Braaaaains" 10% of the time.

As for the Zombie being able to pick up weapons, we created another Behaviour class called PickUpWeaponBehaviour and added into their behaviours array. PickUpWeaponBehaviour will only pick up weapons. To do so, we utilise the asWeapon() method given in the Item class. Furthermore, this form of Action is a PickUpItemAction, so we utilise the getPickUpItem() method given in the Item class as well to pick up the weapon. Therefore, we applied DRY in a way that we do not require to design new methods to pick up weapons as they are all provided in the Item class, so we avoided duplicate code functionality.

As for our decision on how the Zombie should behave, it loops through its behaviours array where whichever Behaviour object returns an Action object in order, the Zombie will perform that Action. Therefore, our Zombie's behaviour is prioritised in the following order:

1. Zombie attempts to say "Braaaaains"
2. Zombie attempts to pick up a weapon on the ground
3. Zombie attempts to attack a nearby Human
4. Zombie attempts to run after a nearby Human
5. Zombie wanders around
6. Do nothing if cannot do any of the above actions

Beating Up the Zombies
According to the specifications, the Zombie's limbs can be wielded as simple clubs. In other words, they have to inherit the WeaponItem class. Thus, we have given Zombies another attribute which is a list of Limb objects. A Zombie will have 4 Limb objects, 2 which are their arms, and the other 2 which are their legs. The Limb class inherits a new class called CraftableItem which inherits WeaponItem.

Limb is a subclass of CraftableItem as it is an item that can be crafted into a WeaponItem.The reason why the Zombie's limbs have to be stored in a list as an attribute and not in the Zombie's inventory is because they are weapons. If they are in the Zombie's inventory, the Zombies will be attacking Humans with their limbs instead of using their intrinsic weapon. Another reason is that if the limbs were in the Zombie's inventory and we want them to drop a limb, we have to unnecessarily check through their inventory if the item we are iterating through is a Limb object, which means we are forced to use instanceof or getClass to find their limbs. Therefore, having a separate list to contain the Zombie's limbs allows for better maintainability of our code.

Every time a Zombie is attacked, the AttackAction class will call the hurt() and bonusEffect() method on Zombie and any property changes to the Zombie are encapsulated in the Zombie class. The Zombie class will manage dropping the limbs onto the map, as well as the changes to their behaviour after losing those limbs. In their playTurn(), it checks through each Action object that if it is not null, we will also check if the Zombie is allowed to use that Action given they already lost an arm or a leg. These methods that do the checks are all private methods within the Zombie class and are not handled in other classes because these properties are unique to Zombies. We are applying the "Tell, Don't Ask" principle such that the AttackAction class only tells the Zombie class to take damage, but the Zombie class handles whether they have to drop a limb on the map, weaken their movement abilities, and whatnot. We believe this is good encapsulation taking place to help us apply ReD on the AttackAction class in the process as well.

The Zombie will have a *turn* count attribute. *turn* indicates the turn number of the Zombie and it is used to check if they are allowed to move given they lose a leg. If they only have 1 leg and *turn % 2 == 1*, or they have no legs, then they are not allowed to move at all.

The Zombie will also have a *biteChance* attribute. It is defaulted to 50. If the Zombie loses an arm, the chances of them punching is halved. In other words, instead of a 50-50 chance to bite or punch, it is now 75-25. If they have no arms, then it is 100-0. So whenever a Zombie is attacked, the overridden hurt() method in the Zombie class will call a private method called weakenZombie() which increases the Zombie's chances of biting over punching through this formula: *biteChance = Math.min(biteChance + 25, 100)*.

Furthermore, in the Zombie's getIntrinsicWeapon() method, instead of using literals for the randomiser check (e.g. *rand.nextInt(100) < 50*), we can do it as such: *rand.nextInt(100) < biteChance*. This allows us to eliminate indirect dependencies such as literal values that could possibly be changed and so we applied ReD.

Crafting Weapons
As mentioned earlier, the Zombie's limbs are Limb objects, which is a subclass of CraftableItem, which makes them items that can be crafted into weapons. In turn, CraftableItem is a subclass of WeaponItem, so they are already weapons on their own before they are crafted into stronger ones.

CraftableItem is a class that has a WeaponItem attribute which holds the weapon that item can be crafted into. To craft an item, an Actor needs a CraftAction object. We cannot add the CraftAction object into the CraftableItem's allowableActions list because the Player will then be able to craft a weapon before they pick the item up, which violates the specifications. Hence, we have no choice but to modify the Player class' playTurn() such that on every turn, the Player's inventory is iterated through to find if they have a CraftableItem in their inventory. If they do, the CraftAction object is added to their list of actions they can perform on that turn.

Since the Zombie's arms and legs can be crafted into a Zombie club and Zombie mace respectively, we create 2 new WeaponItem subclasses called ZombieClub and ZombieMace, similar to the Plank class.

As for our new CraftAction class, its constructor takes in a CraftableItem object which is the item we want to upgrade. Since a CraftableItem has an endProduct attribute, we use an accessor method to get the endProduct of that item. That way, we can remove the CraftableItem object from the Actor's inventory, and then add the endProduct into the Actor's inventory. As such, the CraftAction class does not need to know which item corresponds to the correct end product to be crafted because it is already provided by CraftableItem. The CraftAction class does not need to depend on all the different WeaponItem subclasses created. It does not need to store a list of WeaponItems to know which weapon can the item be crafted into, nor does it need to return an instance of the corresponding WeaponItem, which will create an unnecessary amount of dependencies as more craftables are added into the game. In a sense, we have applied ReD through this approach.

Rising from the Dead
To implement this feature, we created a new class called HumanCorpse that extends the Item class. We had to modify our existing AttackAction to check if a Human is not conscious or dead, it will create a HumanCorpse which will turn human corpses to Zombies.

HumanCorpse is a class that has an integer attribute called turns which will keep track of the number of turns after the class has been created. From there, we have a constructor with only a String parameter passed into it to record the name of the Human that died. We have set the display character of the HumanCorpse in the constructor so every time we create a HumanCorpse, we don't have to keep having to input a display character which is a DRY approach.

There will be 2 override methods which are tick(Location) and tick(Location, Actor) to be used to track the number of turns after the HumanCorpse has been created because the tick() methods are called in the GameMap() every turn to simulate time. tick(Location) is used if the HumanCorpse has not been picked up by a player or human and tick(Location,Actor) is used when a player or human has picked up the HumanCorpse item. How it works is it will start with tick(Location) first because a HumanCorpse will start by it being a PortableItem on the ground. Once it has been picked up, then tick(Location, Actor) will be used to track the number of turns and increase the turns attribute. Once turns reaches an integer value of 5 and only 5, it will run a method to create a Zombie at the item location if it's not picked up or adjacently if it's been picked up. The method is called addCorpseZombie.

The method addCorpseZombie will then check if the item has been picked up or not through one of its input boolean parameters. The method has 2 more other parameters, one for the location of the HumanCorpse and another for passing in the new Zombie. If the item has not been picked up, the boolean will be false. Then the idea of adding a Zombie at the location is to add the new Zombie at the location, then we used the given location parameter to utilise the principle of ReD to instead of locating the Item from the GameMap and used removeItem to remove the Item at the location. If the item has been picked up, it will find an adjacent available location to add the new Zombie and remove the Item from the actor’s inventory using the methods from Location to check if an Actor can be placed at the location passed in.

Farmers and Food
For the new kind of Human, we created a new class that inherits Human called Farmer. So this class will inherit the common attributes of the Human class such as hitpoints and capability to utilize DRY. The only thing that varies is the name which is the only parameter of the constructor and we set the display character “F” by default as well just like before to avoid repeating ourselves every time we create a Farmer. The Farmer then will have multiple available behaviours as a list of Behaviours as a class attribute. WanderBehaviour(), SowingBehaviour(), FertilizeBehaviour() and HarvestBehaviour().

The Farmer having the ability to sow a new crop with a probability of 33% will be handled in SowingBehaviour() which extends the Behaviour interface as well as an Action class for the behaviour called SowingAction(). SowingBehaviour() only has 1 attribute which is to obtain a random value to set a probability of sowing a crop. If the Farmer decides to sow a crop, SowingBehaviour() will return a SowingAction() in order for processActorTurn() in the World class in the engine package to execute the code in Action(). Reason for not processing everything into one Behaviour class is that processActorTurn() will call the execute() methods in the returned Action class and also to prevent Farmer to be able to run multiple Behaviours in one turn. But because the logic and checking whether the Farmer is sowing the crop or not will be in the SowingBehavior() class, SowingAction() will have a big reduction of dependencies (ReD) as it will only have to set the ground to a new Crop().

Crop() is a class that extends Ground for the Player and Farmers to interact with. Reason for not extending it to being an Item because Items can be picked up, so if its an Item, before the crop is ripe, a Player or Human, might have just picked it up which is not right. We have set the display character “c” by default because all crops are the same which is utilizing the principle DRY. We then use tick() to track the number of turns after the Crop has been created to add to an integer attribute to simulate the age of the crop towards turning the crop into a ripe crop that can be harvested. We also included a fertilize() method to increase the age if a Farmer were to fertilize the plant to speed up its aging process.

For a Farmer to fertilize a crop, we created 2 new classes as well to handle the logics which are FertilizeBehaviour() and FertilizeAction(). It is quite a simple logic where the FertilizeBehaviour() will check if the ground is a unripe crop or not, if it is, it will call FertilizeAction(), if not then it will return null. To check if the ground has an unripe crop, we only have to get the location of the actor by using the 2 default parameters in getAction() which are Actor *actor* and GameMap *map* to avoid any unnecessary dependencies to obtain an actor’s location such as passing a Location as a parameter into a constructor or find a Location of the actor instead of using the GameMap. This allows the method to be coded in a ReD way. FertilizeAction() will then run the Crop’s fertilize() method to increase the number of turns to simulate fertilization of a crop then return a menuDescription.

When it comes to harvesting, there will be 2 new classes to compute the logic of harvesting which are HarvestBehaviour() and HarvestAction(). HarvestBehaviour() will first check if an actor is on a crop or not using the @Override getAction method when implementing a behaviour. The idea is to get the location of the actor if it's on the ripen crop, then the actor can harvest it as a Food item in their inventory. If the actor is a Farmer, the Farmer can harvest adjacently if the crop is ripe and drop an item at the location of the harvested crop. HarvestAction() will have the role of setting the Ground back to Dirt using the Dirt class and adding the item at the location or adding the Food item into the player’s inventory. The location is passed in as a constructor parameter when creating a HarvestAction().

There will be a Food class as well that extends Item. The class attributes are just a String to output a message if a player eats a Food item and an integer value to hold a healValue. A constructor will handle the process of naming the food and giving it a heal value for healing hit points once eaten. There will be a getter for the getHealValue(). Picking up the food would involve the usage of PickUpFoodBehaviour() for Humans to pick up Food items on the Ground.

Lastly, there's also EatBehaviour() and EatAction() classes. The EatBehaviour() class has an association with Item as it needs to retrieve a List<Item> from the actor from getAction() to obtain an inventory of items to check if the actor has a Food Item to eat or not. From here, the getAction() will just check if there’s a Food item in the inventory, if there is, then return an EatAction(). In the EatAction() class, it has an association to the Food class which will be initialized when calling a new EatAction() class which will have a Food parameter passed into its constructor. The execute() method will utilize the methods in the Actor class to heal and remove the item from the inventory. Obtaining the healValue will be from the Food attribute that was passed into it so that execute() in EatAction() can avoid any indirect dependencies such as the healValue being changed. Hence we utilized the principle ReD in this class.

Going to Town
The approach to this is similar to how it is implemented in the demo mars package. The new town map is created in the Application class. Two Vehicle objects, which are subclasses of Item, are also added in the Application class where one is placed on the compound map and the other in the town map to allow for to and fro travel. It makes sense to initialize them in the Application class as the GameMaps and Vehicles must be initialized and running when the game begins.

We did consider encapsulating the compound map and the town map as subclasses of GameMap but this brings up a couple of problems:

1. When we want to initialize a Vehicle on the map, it does not know a different map that it can travel to aside from the one it is in. If we try to resolve this by having the compound map class to have the town map as an attribute and vice versa, we create a cyclic dependency!

2. For both subclasses, both of them will have the same set of FancyGroundFactory attributes because both are using the same Ground objects. This results in duplicating our code which goes against the DRY principle.

3. Both subclasses definitely have to initialize Zombies, Humans and Items in their respective maps. As a result, both subclasses will be dependent on the Zombie, Human and Item classes and potentially many other classes in the future as we expand our game. This goes against the ReD principle.

Therefore, we made the decision that we have the maps initialized in the Application class where only the Application class is dependent on the other classes (Zombie, Human, Player, Vehicle, etc.) so that we do not have to repeat initializing the FancyGroundFactory object every time we have a new map as well as avoiding the occurrence of a cyclic dependency. Hence, this approach is more feasible to apply the ReD and DRY principles.

Next is the implementation of Vehicle. Since we have to add a MoveActorAction into the Vehicle object's allowableActions list to allow the Player to travel from one map to another, we can do this where the Vehicle class has an addAction() method so that we can add a MoveActorAction into it by calling that method in Application. We did consider having the Vehicle class to have a MoveActorAction object added into its allowableActions list upon initialization instead of adding it manually in the Application class but this is not appropriate due to the MoveActorAction's constructor.

To initialize a MoveActorAction object, its constructor takes in 2 parameters: a Location on the map with its x and y coordinates and a String which is the description where the Actor will go to. This makes it difficult to have the MoveActorAction object initialized within the Vehicle's constructor because we have to hardcode the coordinates as well as the description message in. As such, it is not possible to have the MoveActorAction object initialized within the Vehicle class because this incurs indirect dependencies which should be avoided as much as possible.

Another problem is when some other Actor (e.g. a Zombie or Human) is standing on the Vehicle in the town and the Player wants to travel to the town from the compound map. Since only 1 Actor can be on a Ground at any given time, the moment when the Player tries to move to the town, the game will crash. We deduced a few methods to solve this:

1. Instead of adding the MoveActorAction in the Application class, add it in the Player class' playTurn() method only when there are no Actors standing on the Vehicle in the other map.
    - The Player will then have to know the map where the Vehicle will take them to. As such, the Player can only know this if the Vehicle has a GameMap attribute to access from.
    - However, there is no reason why a Vehicle should have a GameMap attribute if it is not going to be used in Vehicle.
    - The Player class will also then have a dependency on Vehicle. How would this scale as we add more features that are only interactable with the Player? The Player will then have more dependencies as time goes on.
    - The Application class is already dependent on Vehicle and Player, and through this approach, Player will also be dependent on Vehicle. Therefore, as more Player intractable items are initialized in Application, the more dependencies the Player class will have. This is not good.

2. Creating a new subclass of Ground called PlayerGround to be placed at the same location of the Vehicle. PlayerGround is a Ground that can only be entered by the Player based on their ZombieCapability.
    - No additional dependencies added to the Player class.
    - As new Items or special Grounds that are only interactable by the Player are added, PlayerGround can be reused.
    - Trade-off: PlayerGround location must be the same as Vehicle to work, so both of them must have the same coordinates hardcoded. However, it is negligible since this is only done in the Application class where hardcoding coordinates to add Actors and such onto maps is inevitable. So dependencies are minimized to only be within the Application class.

As a result, we decided to go for the 2nd approach to handle this problem. We believe it will potentially reduce dependencies on other classes such as the Player class (ReD) as well as the PlayerGround class can be useful for future Player-only interactables.

New Weapons : Shotgun and Sniper Rifle

Both weapons will extend WeaponItem which have their own damage values that can be picked up by the Player. The way of how these two new weapons will inflict damage to the Actors is that the shotgun will deal damage within an area of effect in the shape of a 90° cone and for Sniper it will be target based.

Ammo: Firstly, both of these weapons being ranged weapons that use ammunition, we've implemented a new class that extends Item for each type of ammunition (shotgun and sniper). The new classes are ShotgunAmmunition and SniperAmmunition. The way how ammunition is consumed is by using a method in each ammunition class called shotOnce(). By using this method, ammunition is consumed in a way like consuming a bullet from an ammo box. So by calling this method, it will decrease the number of ammo in the ammunition class. We will then track if the number of ammo in the ammunition class is 0 by overriding the tick() method. Each stack of ammunition is displayed with the displayChar of 'o' for Shotgun ammunition and 'O' for Sniper ammunition.

Shotgun: To implement the new weapon Shotgun, the way how the player could use the weapon is to first pick up a Shotgun which will be a WeaponItem with the displayChar 's'. The Shotgun will have no ammunition inside at all, so the player will have to first get ammunition. We then will check if the Player's current weapon is a Shotgun or not, if it is, we will then check the Player's inventory for ShotgunAmmunition to be able to select an option in the Menu to fire the weapon. From there onwards, if both conditions of having the shotgun and ShotgunAmmunition, we will display a submenu of choices for the player to fire the shotgun in the available directions based on the player's location.

Steps on how we will display the choices of directions is to first check for possible exits based on the Player's location. We did this in order for managing edge detection to make sure that if the Player is on the edge of the map, implementing the area of damage won't throw an error that the index is out of range. A new menu will be displayed with all the possible locations to shoot the shotgun instead of using the existing one to avoid needless dependencies on the first menu of action to meet the principle of ReD. The player will choose which direction and a ShootingShotgunAction() will be executed. The way we implemented our range and area of damage of the shotgun is to implement the area of damage in terms of Pairs of (x,y). Then to implement the damage in the execute method, we will just extract all Pairs of x and y values to implement the damage if there's an actor at the location and not out of bounds. Reason behind implementing the coordinates to implement damage in Pairs is to avoid repeating the execute code to check which direction it is, so instead of creating multiple execute codes for each direction, we hardcoded the 90° cone area of damage of where the damage will be based on the player's location. This utilizes the principle of DRY. The probability of 75% hitting the actors within the 90° cone range of 3 is implemented by the attackMissed() method that was predefined in the ActorInterface.

To consume shotgun ammo after implementing damage to the area, I've created a private method shotAmmo() in the action class to get the ammunition stack in the actor's inventory and implement the method in the ammunition stack shotOnce() to decrease the number of ammunition left.

Sniper: To implement this new weapon, the way how it will work is that the player will be given an option to fire the sniper in the main menu if there's an ammo stack in the player's inventory and the player's current weapon is the Sniper. From there another sub-menu will appear to choose which target to fire the sniper at. The sub-menu will consist of all the zombies and mambo marie on the map for the player to choose which target the player wants to choose. After choosing, another sub-menu will appear with 2 options to either shoot straight away or aim at the target once. Here's a breakdown of the classes:

1. ChooseTargetMenu - To display the possible targets on the map for the player to snipe.
2. ChoosingTargetAction - To display the 2 options to either spend a round aiming or shoot straight away.
3. ShootingSniperAction - To shoot the target using AttackAction and consume ammo from the SniperAmmunition item in the player's inventory

After checking that the current weapon of the player is a sniper, the player must have a SniperAmmunition item in its inventory to only be able to choose a target to snipe. An action class ChooseTargetMenu() will be executed to get the menu of targets the player can select after choosing to snipe a target. After a player has selected, we will produce another sub-menu to give the player the option to fire the straight away or spend the turn to aim at the target. This menu is produced by ChoosingTargetAction() that will separate the option into the sub-menu to avoid cluttering the main menu to meet the principle of ReD from the main menu.

With a changing accuracy depending on the number of times a player aims, a tracker is implemented in the player's class to keep track of the number of times the player has aimed at the target. From there, we will use AttackAction and obtain the aimCount value to increase the damage accordingly that will run the attackMissed() method with its respective probability to have many times the player has aimed at the target. We reused AttackAction to avoid reproducing another method to attack a target to meet the principle of DRY such as not needing to repeat the method to check if the actor is not conscious anymore to place a corpse on the map.

To detect if the player is done another move besides aiming or shooting the target after selecting a target to snipe, we implemented another counter in the player class to track the aimCount increment. How it works is that trackAimCount will be 1 digit behind aimCount when initialized. The trackAimCount will always increase the moment aimCount is not 0 meaning the player has aimed at a target previously so trackAimCount should always be one digit behind aimCount. If the player decides to do anything beside aiming or shooting, aimCount wont increase but trackAimCount will still increase. Hence, if trackAimCount and aimCount are the same value, means the player has lost concentration of the target the player was aiming at. To check if the player is attacked when still aiming at a target, we set the current hit points of the player to an attribute in player, then when a player chooses to snipe, we will add the current hit points of the player into another attribute called hitpointCheck. So later on we will check every turn after the player has chosen to aim at a target, if the currentHitPoints is less than the hitpoint value, it notifies that the player has been attacked. Hence, the player will lose concentration of the target.

Mambo Marie
The engine package has a World class where it checks its actorLocations attribute to see if MamboMarie is found on the compound map or the town map. If she is not found on either one of them, the game treats that she is dead. This is a problem because MamboMarie must always exist when the game starts till the Player actually kills her. So our method of handling this is to initialize a new simple map (i.e. only has one space ".") and add it to our World object in Application. This simple map is to allow MamboMarie to hide in and is inaccessible by the Player. As such, MamboMarie can constantly exist throughout the game and no changes are needed to the game's state to keep track of her existence. The downside is that we must add this simple map to the World object in Application but this decision is deliberate to avoid the following alternative approach:

If we were to create a subclass of World that has some boolean attribute or so to keep track of whether MamboMarie is alive or not, that subclass must know the current condition of MamboMarie throughout the game's lifetime. But why must a World subclass keep track of an Actor's state? It is not helpful when it comes to reusability and it should be sufficient that the default World class only checks if the Actor is on the map or not to indicate that it is alive or dead. What if in the future, we decided to add multiple MamboMaries? We then have to refactor the entire subclass that we made because of its dependency on the state of a single MamboMarie. It will not be good at all if we were to use this approach whenever we want to add new bosses in the future as well because all of this goes against the ReD principle.

Therefore, Mambo Marie will have 2 attributes that helps her to exist in the game as well as allowing her to appear and disappear in the game:

- An ArrayList of GameMaps which are the maps that she can appear in (i.e. the compound map and the town map)
- A GameMap which is the map she will hide in and continue to exist in the game (i.e. the simple map that we previously mentioned)

We do this because this allows us to implement MamboMarie's appearing and disappearing character within MamboMarie's class. So when we initialize MamboMarie, we have to put in the GameMaps where she can appear in the game and the GameMap that she can hide in. So, whatever features such as having her appear in the compound map or town map as well as vanishing after 30 turns, we will handle this in the MamboMarie class itself. We do not have to create a World subclass to handle and manage all of this since these features are unique to MamboMarie herself. Therefore, the knowledge of appearing in maps and the implementation details of it are encapsulated within MamboMarie since these special properties of appearing and disappearing are unique to MamboMarie alone. We believe this is good encapsulation taking place and it definitely reduces the need of creating other World or GameMap subclasses to constantly check MamboMarie's state. As such, we have applied ReD.

As for summoning Zombies, this is also handled within MamboMarie since it is a unique feature to herself. A SummonZombieAction object is returned whenever she attempts to chant and summon Zombies on the field. To make the SummonZombieAction class modular, we design it such that we have to pass in the number of Zombies to spawn, their names, and the verb the Actor used to summon the Zombies (for MamboMarie, it is “chant”). Hence, if in future expansions, there are other enemies that can summon Zombies similar to MamboMarie, these enemies can reuse the following SummonZombieAction class to summon their own numbers of Zombies. Therefore, we applied DRY through this approach such that the feature of summoning Zombies is made reusable.

Ending the Game
Currently, the World class in the engine package only runs the game as long as the Player is alive. Since we have additional win/lose conditions such as the Player loses when all Humans are dead and the Player wins when all Zombies including MamboMarie are dead, we create a new subclass of World named ZombieWorld. Since ZombieWorld inherits World, it can access its parent's attributes and methods where their access modifiers are set to protected. Not only that, we can override some of the methods such as stillRunning() and endGameMessage() to suit our game's win/lose conditions.

The ZombieWorld class can then have its own private methods such as playerAlive(), humansAlive() and zombiesAlive() to check each kind of Actor respectively based on their ZombieCapability. Then we can override the stillRunning() method to include these private methods to modify our game's running state. endGameMessage() can be modified to return a different description based on the result of the game such as a statement telling the Player loses not because they died but because all of the Humans are dead, or a statement telling the Player they won because all of the Zombies including MamboMarie are dead. As such, there is no need to unnecessarily override the World's run() method just to change the while loop condition but then copy back the rest of the code. Instead, we can simply override the stillRunning() method and the run() method remains unchanged. Therefore, we have applied the DRY principle where there is no need to have repeated code implemented into our ZombieWorld class to add new win/lose conditions for our game.