**Design Rationale**

**Group : Team 2099**

**Members : Siang Jo Yee, Lua Shi Liang**

Zombie attacks

**Zombie [ Existing class ]**

* A type of actor called Zombie.

New Changes and their responsibilities:

1. A new private attribute called PickUpWeaponBehaviour() is added into the existing array of behaviour. Its function is to create a behaviour that allows the **Zombie** to pick up the weapon from its standing location in the map. The order of the array of behaviour is **AttackBehaviour**, **HuntBehaviour**, **PickUpWeaponBehaviour** and **WanderBehaviour**.
2. A double attribute called probability is added into the class to use in the getIntrinsicWeapon() method for comparing the probability.
3. The existing getIntrinsicWeapon() method is modified  and now has 2 possible types of IntrinsicWeapon called bites and punches when the method is called. The chances of getting one of them 50%. Math.random() is used to obtain the random probability between the two.
4. The existing playTurn() method is modified to have a 10% chance of printing the line “ Braaaaains ” every turn. Math.random() is used to obtain the probability of saying the phrase.

Design Choice:

1. The behaviour of a **Zombie** in the game should be wanting to attack and hunt humans first as a particular zombie would do. Then only it should have the instinct to pick up weapons and wander around. In a logical standpoint, they should have picked up a weapon first then only started to attack **Human**. However, **Zombie** has no logical sense .

**PickUpWeaponBehaviour [ New Class ]**

* A type of non-player behaviour that picks up weapons from its standing location.

New Changes and their responsibilities:

1. This class implements an interface called **Behaviour**.
2. It has a method called getAction() that takes in two parameters ( **Actor** actor and **GameMap** map ) and returns an Action. Its function is to pick up a weapon that is on the **Actor**’s location in the map. If there is a weapon in the **Actor**’s location, it will instantiate a PickUpItemAction() and return it.

Design Choice:

1. When designing **PickUpWeaponBehaviour**, it implements Behaviour because behaviour serves as a purpose for a non-player to perform actions and is a guidance for all types of behaviour to use its method which is getAction(). This makes the codes for all types of behaviour class understandable and consistent.
2. When the **Zombie** picks up a weapon, the weapon will be stored in the inventory and it will use the weapon instead of the intrinsic weapon.
3. Since a **Zombie** can only use a weapon at a time, if a zombie picks up multiple weapons, it will only use the first weapon that it picks up. It will use the following weapon in order if it drops the first weapon and so on.

**ZombieAttackAction[ New Class]**

* A type of attack action that only Zombie class can have

New Changes and their responsibilities:

1. This is a class that inherits from a class called **AttackAction**.
2. It has a constructor that takes a parameter (**Actor** actor) and its parameter is initialized by the superclass constructor.
3. It has only one method:
4. The execute() method  that takes 2 parameters (**Actor** actor and **GameMap** map) and returns a String. Its function is to execute the attack action that the Zombies have. This class is similar to the **AttackAction** class but with additional features only available for **Zombie**. **Zombie** will miss more bite attacks than punch attacks. A probability is set by Math.random(). Bites will have a missing rate of 75% and 25% for punch. **Zombie** will also heal 5 points if they successfully landed a bite attack. If **Zombie** is using a weapon, the chances of missing is 50%.

Design Choice:

1. When designing **ZombieAttackAction** class, it inherits from **AttackAction** class as it is a type of attack action. This reduces repetition of codes and hence follows the  DRY( Don’t repeat yourself) principle.
2. I have decided to create a new attack action class only for **Zombie** instead of using **AttackAction** class. The reason for this is for better maintainability and extendable. In the future, we can add more **Zombie** related attack action features in this class.

**AttackBehaviour [ Existing Class ]**

New Changes and their responsibilities:

1. The getAction() method is modified and now has another condition in it. If the actor has a capability of ZombieCapability.UNDEAD (enum), it will return a new **ZombieAttackAction**.

Design Choice:

1. **AttackBehaviour** now has 2 possible types of action, **AttackAction** and **ZombieAttackAction**. **AttackAction** is used by other actors in the game where **ZombieAttackAction** is only used by **Zombie**.

Beating Up the Zombies ( Updated for Assignment 2)

**SimpleClub [New Class]**

* A type of weapon item that is created when a zombie is about to lose its limbs**.**

New Changes and their responsibilities:

1. This class inherits from a class called WeaponItem.
2. It has a constructor that takes a parameter( char display). It uses the superclass constructor to initialize its parameters
3. A new **CraftingAction** is added into the allowableActions (attributes in **Item**) so that **SimpleClub** can be crafted by Player.
4. A capability called  ItemCapability.CRAFTABLE (enum) is added into SimpleClub so that it has a capability of crafting..

Design Choices:

1. When designing the **SimpleClub** class, it inherits the **WeaponItem** class as **SimpleClub** is portable like an **Item** and has damage and verbs like a **Weapon** . Hence, it follows the DRY( Don’t repeat yourself) principle that prevents repetition of codes.
2. According to the game,when a player picks up a dropped limb of a Zombie, it can be wielded as a  **SimpleClub** and it can be used as a weapon **. SimpleClub** is actually the “limb” that is dropped on the ground. Hence, when a Zombie is about to lose its “Limbs”, **SimpleClub** is instantiated in the playturn() method in **Zombie** class and drops on the ground.
3. The added capability can then be used to identify whether the item has properties of an arm or a leg and these properties are important to **CraftingAction.**
4. The capability ItemCapability.CRAFTABLE can also be used by other non-players in the future that wants to craft a weapon.

**Zombie [Existing class]**

New Changes and their Responsibilities

1. The private attributes added:
   1. **int** numOfArms,
   2. **int** numOfLegs,
   3. **double** probability,
   4. **boolean** isSecondTurn
   5. **Behaviour**[] behaviourWithoutLegs
2. behaviourWithoutLegs is a new array of behaviour that only consist of **AttackBehaviour** and **PickUpWeaponBehaviour**.
3. checkArmStatus() method is added
   1. This method is to check the current number of arms and legs of the **Zombie**, and execute some conditions if the **Zombie** drops its arm.
   2. If the Zombie has only 1 arm, there is a 50% chances of dropping all the weapon from its inventory and the probability will increase since chance of punching will reduce.If the Zombie has no arms , the zombie will drop all the weapons from its inventory and the probability of biting will be 100%.
   3. Zombie will call dropAllWeapon() method from **ZombieActor** to drop weapons.
4. A new method returnAction() is added. It has 2 parameter (**Behaviour[]** behaviourArray**, GameMap** map) . Its function is to return an action depending on the behaviourArray given.
5. The hurt() method from **Actor** class is overridden in **Zombie** class. When the Zombie is hurt, it has a 50% of chances of losing its arms or legs. When it does, **Zombie** will have a capability called LimbsCapability.ARM or LimbsCapability.LEG depending on which limb it dropped. This capability is an indication for playturn() method in **Zombie** class to drop a **SimpleClub.**
6. The playTurn() method is modified.
   1. At the beginning of the playTurn() method, a condtion is added for dropping **SimpleClub**. If it has either LimbsCapability.ARM or LimbsCapability.LEG, a **SimpleClub** will be added to the location of the **Zombie**.
   2. checkArmStatus() method is called after that.
   3. isSecondTurn is used to check whether the **Zombie** with one leg is on its first or second turn. It is a boolean attribute that determines whether the **Zombie** with one leg is on its first or second turn by assigning it to be true and false alternatively and it is used as a condition in the playTurn() method.
      1. When isSecondTurn is false, it is the first turn, **Zombie** with one leg is still able to move, thus behaviours is the array where its elements is looped.
      2. Else when isSecondTurn is true, **Zombie** which is on its second turn will not be able to move, thus, behaviourWithoutLegs is the array where its elements is looped instead.
      3. This fulfils the functionality that **Zombie** with one leg cannot move every second turn.
   4. When numberOfLegs is 2, the array used to loop through all the elements is behaviours, else when numberOfLegs is 0, behaviourWithoutLegs is used instead.
   5. returnAction() method is used in this method for returning an action based on behaviours or behavioursWithoutLegs.
   6. If the **Action** get by the elements in the selected array is all null, it will return a new DoNothingAction object.

Design Choices / Reasons:

The modified playTurn is use to control the the type of behaviour of the Zombie based on its cuurent number of limbs. The design choice is for encapsulating everything that is related to Zombie inside the **Zombie** class.

**ZombieActor [Existing Class]**

New Changes and their Responsibilities

1. A new private method is called dropAllWeapons() is added. The function for this method is to drop all the **WeaponItem** in the inventory of the **Actor.**

Design Choices / Reasons:

Currently, this method is only used by **Zombie**. The reason for adding this method in the **ZombieActor** class rather than the **Zombie** Class is for maintainability and extendability.

Other actors that inherit from **ZombieActor** can use this method in the future. Any new changes on the method is only need to applied one time hence better maintainability.

**ItemCapability [New Enum]**

* A collection of constants which indicate the capability of **Item**

Constant added:

CRAFTABLE - if the item has this capability, it can be crafted

**LimbsCapability[New Enum]**

* A collection of constants which indicate the capability/properties of **Limbs**

Constant added:

ARM- if the item has this capability, it has arm properties

LEG- if the item has this capability, it has leg properties

Crafting Weapons( Updated for Assignment 2)

**CraftingAction[ New class]**

* An action that allows an actor to craft a weapon.

New Changes and their responsibilities:

1. This class inherits from a class called Action.
2. It has a attribute called craftedItem
3. It has two methods.
4. execute() is a method that takes 2 parameters (**Actor** actor and **GameMap** map) and returns a String. Its function is to craft a new weapon out of  craftedItem which is initially a “Limb”. There are 2 possible weapons that this class can craft which are **ZombieClub** and **ZombieMace**. If the actor’s inventory has an item with a capability of ItemCapability.CRAFTABLE (enum), that item can be crafted. An item with the displayChar of ‘l’ can be crafted into a **ZombieMace** and ‘A’ can be crafted into a **ZombieClub**. If it is successful, it will return a string that indicates the item is crafted.
5. menuDescription() is a method that takes a parameter ( Actor actor ) and returns a String. Its function is to return a menu description to show that there’s an item that is craftable in the Actor’s inventory.

Design Choices:

1. When designing **CraftingAction** class, it inherits from **Action** class as it is a type of action. This reduces repetition of codes and hence follows the  DRY( Don’t repeat yourself) principle.
2. The item that is going to be crafted will be removed from inventory and the crafted weapon will be added into the inventory. It’s kind of replacing the item that is going to be crafted with another item, in this case is either **ZombieClub** or **ZombieMace**.

**ZombieMace [ New class ]**

* A type of weapon item that is created from crafting.

New Changes and their responsibilities:

1. This class inherits from a class called **WeaponItem**.
2. It has a constructor that takes no parameter and initializes the attributes of the class with a superclass constructor.

Design Choice:

1. When designing the **ZombieMace** class, it inherits the **WeaponItem** class as **ZombieMace** is portable like an **Item** and has damage and verbs like a **Weapon**. Hence, it follows the DRY( Don’t repeat yourself) principle to prevent repetition of codes.
2. **ZombieMace** will have higher damage than **ZombieClub**.
3. **ZombieMace** will only be created through **CraftingAction** and if the item has a displayChar of ‘l’.

**ZombieClub [ New class ]**

* A type of weapon item that is created from crafting.

New Changes and their responsibilities:

1. This class inherits from a class called **WeaponItem**.
2. It has a constructor that takes no parameter and initializes the attributes of the

class with a superclass constructor.

Design Choice:

1. When designing the **ZombieClub** class, it inherits the **WeaponItem** class as **ZombieClub** is portable like an **Item** and has damage and verbs like a **Weapon** . Hence, it follows the DRY( Don’t repeat yourself) principle that prevents repetition of codes.
2. **ZombieClub** will only be created through **CraftingAction** and if the item has a displayChar of ‘A’.

**Player [Existing Class]**

New Changes and their responsibilities:

1. Playturn() method is modified that a condition is included where if the player has an item that has a capability of ItemsCapability.CRAFTABLE , a new **CraftingAction** is added into the actions.

Rising from the dead

Updated:

* **Corpse** is a **PortableItem**.
* When **Player** picks up the **Corpse**, time freezes and will only continue running if **Player** drops it onto the ground.
* **Corpse** object is created in **ZombieAttackAction** instead of **AttackAction**, thus only created when **Human** and **Farmer** is dead, not including **Zombie**.

**Corpse** [New Class]

* An **PortableItem** added to the **GameMap** when an **Actor** (**Human**, **Farmer**) is killed.
* The corpse of Zombie is not a Corpse object.

New Changes and their Responsibilities:

* This class inherits from a class called **PortableItem**.
* It has a private attribute (**int** turn).
  + turn – initialised to 0 and acts as a counter
* It has a constructor that takes in one parameter (**String** name). It makes use of the superclass constructor to initialise name and displayChar.
* It overrides tick() method to fulfil the required functionality for the **Corpse** to “rise from the dead”.
  + Parameter: **Location** currentLocation - Current location where the corpse lies on
  + This method is executed every turn in the game.
  + turn increments by 1 if **Corpse** is on the ground
  + A random integer is generated between 5 to 10 inclusive.
  + Then, if the turn equals the random integer and there is no **Actor** on the currentLocation, A new **Zombie** object is instantiated.
  + The **Zombie** is added onto currentLocation and the **Corpse** is removed from currentLocation using methods in **Location** called addActor() and removeItem() respectively.
  + A string is printed to notify the **Player** that a dead **Human** becomes a **Zombie**.
  + This will look as though the **Corpse** becomes a **Zombie** 5 to 10 turns later.

Design Choices / Reasons:

* The design in which the Corpse inherits from PortableItem takes into account the Do Not Repeat Yourself principle because Corpse also uses the features in PortableItem and inheritance helps to avoid repetition of codes.
* **Zombie** objectis instantiated in the method as a local variable instead of as an attribute to follow Declare things in the tightest possible scope principle to lessen the risk that something can depend on it, thus reduce the risk of possible failure in the future.

**ZombieAttackAction** [Existing Class]

New Changes and their Responsibilities:

* execute() method is modified
  + If an **Actor** is killed after being attacked, a new **Corpse** object is created then added to the **Location** of the dead **Actor** which is removed from the **Location**.
  + T **Corpse** class is created to increment the counter every turn when necessary in the game to control the turn when it becomes a **Zombie** to fulfil the required functionality.

Design Choices / Reasons:

* **Corpse** is created in the method as a local variable instead of as an attribute because Declare things in the tightest possible scope principle is taken into account in this design.

Farmers and food

Updated:

* A boolean isDamage() method is added to **ActorInterface** to check whether an Actor is damaged, and implemented in **ZombieActor**.
* Added **EatBehaviour** to the behaviours array (attribute) for both **Human** and **Farmer**, thus playTurn method in both Farmer and Human is modified.
* **Crop** is changed to be a **Ground** instead of an **Item**, thus related class and method will thus be modified.
* A speedUpAge() method is added to **GroundInterface** to speed up the age of a type of Ground.
* Exception handling is added.

**Farmer** [New class]

* A new kind of **Human** who shares the same characteristics and abilities as a **Human** but also able to sow, fertilize and harvest crops.

New changes and their Responsibilities:

1. This class inherits from a class called **Human**.
2. It has five private attributes stored in an array of type **Behaviour** - behaviours

* Its elements are new object instantiated from **EatBehaviour**, **HarvestBehaviour**, **FertilizeBehaviour**, **SowBehaviour** and **WanderBehaviour** respectively

1. It has a constructor that takes in one parameter (**String** name). It makes use of the superclass constructor to initialise name, displayChar and hitPoints. It also calls addCapability() method from **Actor** with *ZombieCapability.FARM* as the argument.
2. It overrides playTurn() method.
   1. Parameter:

* Actions actions, Action lastAction, **GameMap** map, Display display
  1. This method executes every turn in the game.
  2. it will return the first **Action** which is not null following the sequence from the return value of getAction method called by **EatBehaviour, HarvestBehaviour**, **FertilizeBehaviour**, **SowBehaviour**, **WanderBehaviour**, else a new **DoNothingAction** object is returned.

Design Choices / Reasons:

* The design where **Farmer** inherits from **Human** follows the Do Not Repeat Yourself principle because **Farmer** also uses the features in **Human** and inheritance helps to avoid repetition of codes.
* behaviours are declared as private to follow the Minimize dependencies that cross encapsulation boundaries principle because if we declare it to be more visible, i.e. public initially, it may break other systems if we change a originally more visible attribute to a lower visibility.
* *ZombieCapability* is an Enum and defines a collection of constants to indicate the capability of an **Actor**. **Farmer** is designed to have the capability *ZombieCapability.FARM* to follow the Do Not Repeat Yourself, Reduce dependencies as much as possible and Avoid excessive use of literals principle. It is designed to avoid repetition of code when checking whether an **Actor** is a **Farmer**, and also to avoid using the getDisplayChar method from **Actor** to check whether it is equal to the displayChar of **Farmer**, since if displayChar is changed, we would have to hunt for every place it occurs in the code and change all of them. It would help to avoid having such indirect dependencies in the system.
* An enhanced for loop is used as stated above to avoid repetition of code on checking whether the **Action** returned by the elements in behaviours is null or not and this is also where Do Not Repeat Yourself principle is taken into account in this design.
* In playTurn method, the sequence is arranged like that because logically, Farmer should prioritise eating to recover health points when it is damaged, and then only does its farming action. SowBehaviour is the last farming behaviour because the game map which is full of dirt will always allow Farmer to sow, so to give chances for other farmer behaviour to be implemented, it is arranged to be the last before wandering and doing nothing.

**Human** [Existing Class]

New Changes and their Responsibilities:

* The attribute behaviour is changed to an array named behaviours, which consists of a new **EatBehaviour** object and a new **WanderBehaviour** object.
* playTurn method is modified
  + it will return the first **Action** which is not null following the sequence from the return value of getAction method called by **EatBehaviour** and **WanderBehaviour**, else a new **DoNothingAction** object is returned.

Design Choices / Reasons:

* behaviour is changed to an array named behaviours to avoid duplicated code while checking whether the action get from the behaviour is null or not by using a loop. This follows the Do Not Repeat Yourself principle.

**ActorInterface** [Existing class]

New Changes and their Responsibilities:

* boolean isDamaged() is added

**ZombieActor** [Existing class]

New Changes and their Responsibilities:

* boolean isDamaged() is implemented and defined to check whether an Actor is damaged or not by checking is hitPoints smaller than maxHitPoints

Design Choices / Reasons (**ActorInterface**, **ZombieActor**)：

* This interface method is overridden in **ZombieActor** to avoid duplicated code as its subclass can also check its damage with the help of inheritance. This follows the Do Not Repeat Yourself Principle.

**GroundInterface** [Exisiting class]

* void speedUpAge() method is added.

Design Choices / Reasons:

* This interface method is overridden in the subclasses of Ground.
* For current required functionality, **Crop** implemented this method to speed up the age of an unripe **Crop** when it is fertilized to decrease the time left to ripe.
* For other subclasses, this method can be used for future extensibility.
  + For example, for the living type of ground such as **Tree**, its age can be speed up if fertilized just like **Crop**. For the non-living type of ground such as **Fence**, its age would be defined as it being old and no longer can function well (eg: the fence will be passable as it is broken due to the old age)

**Crop** [New class]

* An type of **Ground** which can be sowed, fertilized when it is unripe and harvested when it is ripe.

New Changes and their Responsibilities:

* This class inherits from an abstract class called **Ground**.
* It has three private attributes (**int** age, **boolean** isRipe, **final int** RIPE\_AGE).
  + age - acts as a the age of the Crop which increases with the turn in the game and initialised to be 0
  + isRipe – check and determine whether the Crop is ripe and initialised to be false, indicating the Crop is initially unripe.
  + RIPE\_AGE - stores the age when the Crop ripes.
* It has a constructor that takes no parameter. It makes use of the superclass constructor to initialise displayChar. It also calls addCapability() method from **Ground** with *GroundCapability.CAN\_BE\_FERTILIZED*  as the argument.
* It has a setter method for the attribute, age.
  + it throws new Exception if age is negative.
* It overrides tick() method to fulfil the required functionality for the **Crop** to be either ripe or unripe based on the age
  + Parameter: **Location** location - location where the **Crop** is placed
  + This method is executed every turn in the game.
* age increments by 1 each turn.
* In the method, when age reaches RIPE\_AGE, the displayChar is reassigned with another **char** to show to the player that the **Crop** has ripen. removeCapability method from **Ground** is called to remove *GroundCapability.CAN\_BE\_FERTILIZED* from the capability of the **Crop**. addCapability method from **Ground** is called to add *GroundCapability.CAN\_BE\_HARVESTED* to be the capability of **Crop**. isRipe is assigned to be true now that the **Crop** ripes.
* Overrides allowableActions method
  + If is the Crop is ripe, a new HarvestActions is added into the the new collection of Actions, else a new, empty collection of Actions is returned.
* Overrides speedUpAge() method
  + age is added by the increase (for current required functionality, it would be increases by 10 turns) if it is smaller than 10
  + If the current age is already 10 or more, its age would made to be (RIPE\_AGE -1) since in the same turn, the **Action** of the **Actor** will run before the tick method which increment the age, thus the minus 1.

Design Choices / Reasons:

* The design in which the **Crop** inherits from **Ground** takes into account the Do Not Repeat Yourself principle because **Crop** also uses the features in **Ground** and inheritance helps to avoid repetition of codes.
* In the setter for the attribute age, exception is thrown to follow the Fail Fast design principle as the system would fail immediately and visibly when age is set to be negative.
* *GroundCapability* is an Enum and defines a collection of constants to indicate the capability of an **Ground**. Crop is designed to be:
  + Initially, **Crop** is unripe, thus it has the capability *GroundCapability.CAN\_BE\_FERTILIZED*, which explains why this capability is added in the constructor.
  + When the age reaches 20,
    - it means that a **Crop** has ripened and it can no longer be fertilized, thus the capability of *GroundCapability.CAN\_BE\_FERTILIZED* is removed.
    - A ripe **Crop** can be harvested, thus the capability of *GroundCapability.CAN\_BE\_HARVESTED* is added.
  + This is designed to avoid repetition of code and avoid using getDisplayChar method from **Ground** that whether it is equal to the displayChar of ripe or unripe **Crop**, since if displayChar is changed, we would have to hunt for every place it occurs in the code and change all of them. It would help to avoid having such indirect dependencies in the system.
    - It is designed this way as Do Not Repeat Yourself, Reduce dependencies as much as possible and Avoid excessive use of literals principles are taken into consideration for this design.
  + This design can be reused when there are other **Ground** which allows the **Actor** to fertilize and harvest, we can use the same constant in the enum to support the new features, thus improve its extensibility.
* allowableActions method is overridden to allow **Actor** to perform **HarvestAction** if it is at the adjacent location of the **Crop**. This avoid duplicated code (Do Not Repeat Yourself) which needs to get the adjacent locations using getExits method from Location as it is already coded in the processActorTurn method in the **World** class in the engine package.
* RIPE\_AGE is declared as constant as its value is fixed and to follow the Avoid excessive use of literals and Do Not Repeat Yourself principle since its value will be reused repetitively in this class. These principle are applied in speedUpAge() method where the condition does not use too much of literals, but instead, using variables to replace it.

**Food** [New class]

* An **PortableItem** which is edible.

New Changes and their Responsibilities:

* This class inherits from a class called **PortableItem**.
* It has a constructor that takes no parameter. It makes use of the superclass constructor to initialise name and displayChar. It also calls addCapability() method from **Item** with *ItemCapability.EDIBLE* as the argument.
* A new **EatAction** object, whose argument is **Food** itself, is added in allowableActions, which is the attribute in **Item**. This is to allow **Food** to be eaten by the **Player**.

Design Choices / Reasons:

* The design in which the **Food** inherits from **PortableItem** takes into account the Do Not Repeat Yourself principle because **Food** also uses the features in **Item** and is portable. Inheritance helps to avoid repetition of codes.
* *ItemCapability.EDIBLE* is used to avoid repetition of code and avoid using getDisplayChar to check whether an **Item** is a **Food** item in order to execute the action of eating. By doing so, such indirect dependencies will be reduced too. Here, Do Not Repeat Yourself, Reduce dependencies as much as possible and Avoid excessive use of literals principles are taken into account when designing.
  + This design can be reused when there are other **Item** which allows the Actor to eat, we can use the same constant in the enum to support the new features, thus improve its extensibility.

**Dirt** [existing class]

* A type of Ground which allows Farmer who is next to it to sow a crop on it.

New Changes and Responsibilities:

* Its constructor is modified.
  + addCapability() method from **Ground** with *GroundCapability.CAN\_BE\_SOWED\_ON* as the argument is added in the constructor

Design Choices / Reasons:

* **Dirt** is designed to have the capability *GroundCapability. CAN\_BE\_SOWED\_ON* to follow the Do Not Repeat Yourself, Reduce dependencies as much as possible and Avoid excessive use of literals principle.
* It is designed to avoid repetition of code when checking whether the **Ground** allows **Actor** to sow on it, and also to avoid using the getDisplayChar method from **Ground** whether it is equal to the displayChar of **Dirt**, since if displayChar is changed, we would have to hunt for every place it occurs in the code and change all of them. It would help to avoid having such indirect dependencies in the system.
* This design can be reused when there are other **Ground** which allows the Actor to sow on it, we can use the same constant in the enum to support the new features, thus improve its extensibility.

**SowBehaviour** [New class]

- A type of non-player behaviour to sow a crop on a patch of dirt if the actor is next to it. **Farmer** owns this behaviour

New changes and their Responsibilities:

* This class implements an interface called **Behaviour**.
* It overrides getAction() method.
  + Parameter: **Actor** actor, **GameMap** map, Return type: **Action**
  + If it is within the 33% probability, an enhanced for loop is used to get the **Exit** from the location of actor on the map and call getDestination and getGround method to check whether the **Actor** is standing next to a **Dirt**, then only a new **SowAction** object is returned, else return null.

Design Choices / Reasons:

* It implements **Behaviour** because **Behaviour** is the objective an **Actor** has to help decide which **Action** is to be performed next by overriding the getAction method in **Behaviour**.For the **Actor** (**Farmer** in this case) to sow, it will return a new **SowAction** object in the method.
* It follows the Declare things in the tightest possible scope principle since **Exit** is declared in the enhanced for loop instead of at the beginning of the method.
* It uses the *GroundCapability* enum to check whether a **Ground** is capable of being sowed on. [Details of design reasons discussed in **Dirt**]

**SowAction** [New class]

* An action which allows the actor (Farmer) to sow a crop on a specific location.

New changes and their Responsibilities:

* This class inherits from an abstract class called **Action**.
* It has one private attribute (**Location** sowLocation)
  + sowLocation - the **Location** where an **Actor** sows a **Crop** on it
* It has a constructor that takes in one parameter (**Location** sowLocation) and initialises the attribute sowLocation.
* It overrides:
  + execute() method,
    - Parameter: **Actor** actor, **GameMap** map, Return type: **String**
    - A new **Crop** object is set as the **Ground** at sowLocation by using setGround method from **Location.**
    - A string describing the actor successfully sows crop is returned.
  + menuDesciption() method.
    - Parameter: **Actor** actor, Return type: **String**
    - A string describing the actor sows a crop is returned.
    - It should be displayed in the menu for the player to choose as a next action. However, since the **Player** does not sow, thus this method is
    - not used for any current required functionality.

Design Choices / Reasons:

* The design where **SowAction** inherits from **Action** follows the Do Not Repeat Yourself principle because **SowAction** also uses the features in **Action** and inheritance helps to avoid repetition of codes.
* A new **Crop** object is instantiated in the method as a local variable rather than being an attribute follows the Declare things in the tightest possible scope principle to lessen the risk of something depending on it which will thus reduce the risk of possible future failure.
* sowLocation is declared as private to follow the Minimize dependencies that cross encapsulation boundaries principle because it may break other systems if we change a originally more visible attribute to be less visible.

**FertilizeBehaviour** [New class]

* A type of non-player behaviour to fertilize an unripe crop if the actor is standing on it. **Farmer** owns this behaviour.

New Changes and Responsibilities:

* This class implements an interface called **Behaviour**.
* It overrides getAction() method.
  + Parameter: **Actor** actor, **GameMap** map, Return type: **Action**
  + Get the ground from the location of actor on the map and if it is has the capability *GroundCapability.CAN\_BE\_FERTILIZED*, it will return a new **FertilizeAction** object, else return null.

Design Choices / Reasons:

* It implements **Behaviour** because **Behaviour** is the objective an **Actor** owns to help decide which **Action** is to be performed next by overriding the getAction method in **Behaviour**.
* For the **Actor** (**Farmer** in this case) to fertilize, it will return a new **FertilizeAction** object in the method.
* It uses the *GroundCapability* enum to check whether a **Ground** is capable of being fertilized. [Details of design reasons discussed in **Crop**]

**FertilizeAction** [New class]

* An action which allows the actor (Farmer in this case) to fertilize an unripe crop.

New Changes and their Responsibilities:

* This class inherits from an abstract class called **Action**.
* It has one private attribute (**Ground** unripeCrop)
  + unripeCrop – the unripe **Crop** to be fertilized
* It has a constructor that takes in one parameter (**Ground** unripeCrop) and initialises the attribute unripeCrop.
* It overrides:
  + execute() method,
    - Parameter: **Actor** actor, **GameMap** map, Return type: **String**
    - speedUpAge() method is called to decrease the time left for the unripeCrop to ripe.
    - A string describing the actor fertilizes a crop is returned.
  + menuDesciption() method.
    - Parameter: **Actor** actor, Return type: **String**
    - A string describing the actor fertilizes a crop is returned.
    - It should be displayed in the menu for the player to choose as a next action. However, since the **Player** does not fertilize, thus this method is not used for any current required functionality.

Design Choices / Reasons:

* The design where **FertilizeAction** inherits from **Action** follows the Do Not Repeat Yourself principle because **FertilizeAction** also uses the features in **Action** and inheritance helps to avoid repetition of codes.
* unripeCrop is declared as private to follow the Minimize dependencies that cross encapsulation boundaries principle because it may break other systems if we change a originally more visible attribute to a lower visibility in the future.
* make use of speedUpAge() method to avoid duplication of codes (Do Not Repeat Yourself).

**HarvestBehaviour** [New class]

* A type of non-player behaviour to harvest a ripe crop if the actor is standing on it or next to it. **Farmer** owns this behaviour. (Player does not have behaviour)

New Changes and Responsibilities:

* This class implements an interface called **Behaviour**.
* It overrides getAction() method.
  + Parameter: **Actor** actor, **GameMap** map, Return type: **Action**
  + An ArrayList of type **Location,** locationList is created to store the location and all the adjacent locations of the actor on the map.
  + A loop is used to get the **Ground** from the elements in the locationList by calling getGround method, and if it has the capability *GroundCapability.CAN\_BE\_HARVESTED*, it will return a new **HarvestAction** object, else return null.

Design Choices / Reasons:

* It implements **Behaviour** because **Behaviour** is the objective an **Actor** owns to help decide which **Action** is to be performed next by overriding the getAction method in **Behaviour**.
* For the **Actor** (**Farmer** in this case) to harvest, it will return a new **HarvestAction** object in the method.
* It follows the Declare things in the tightest possible scope principle since **Exit** is declared in the enhanced for loop instead of at the beginning of the method when getting the adjacent location of the **Actor.**
* It uses the *GroundCapability* enum to check whether a **Ground** is capable of being harvested. [Details of design reasons discussed in **Crop**]

**HarvestAction** [New class]

* An action which allows the actor (Farmer and Player) to harvest a ripe crop at a specific location.

New changes and their Responsibilities:

* This class inherits from an abstract class called **Action**.
* It has one private attribute (**Location** harvestLocation):
  + harvestLocation - the **Location** of the ripe **Crop** to be harvested by the **Actor**
* It has a constructor that takes in one parameter (**Location** harvestLocation) and initialises the attribute harvestLocation.
* It overrides:
  + execute() method,
    - Parameter: **Actor** actor, **GameMap** map, Return type: **String**
    - A new **Food** object is instantiated and stored in a variable called food.
    - The ground of harvestLocation is reset to be **Dirt**. If the actor is capable of farming (i.e **Farmer**), food is added onto the harvestLocation. Else (i.e. **Player**), food is added into the **Player**’s inventory.
      * This will look as though after harvesting a ripe crop, the farmer drops the food on the ground, while player place the food into the inventory.
    - A string describing the actor harvests a crop is returned.
  + menuDesciption() method.
    - Parameter: **Actor** actor, Return type: **String**
    - A string describing the actor harvests a crop is returned.
    - It will be displayed in the menu for the **Player** to choose as the next action in the turn when this action is allowed.

Design Choices / Reasons:

* The design where **FertilizeAction** inherits from **Action** follows the Do Not Repeat Yourself principle because **FertilizeAction** also uses the features in **Action** and inheritance helps to avoid repetition of codes.
* harvestLocation is declared as private to follow the Minimize dependencies that cross encapsulation boundaries principle because it may break other systems if we change a originally more visible attribute to a lower visibility in the future.
* A new **Food** object is instantiated in the method as a local variable rather than being an attribute follows the Declare things in the tightest possible scope principle to lessen the risk of something depending on it which will thus reduce the risk of possible future failure.

**EatBehaviour** [New class]

* A type of non-player behaviour to eat food if the actor is damaged. **Human** and **Farmer** own this behaviour. (Player does not have behaviour)

New Changes and Responsibilities:

* This class implements an interface called **Behaviour**.
* It overrides getAction() method.
  + Parameter: **Actor** actor, **GameMap** map, Return type: **Action**
  + An enhanced for loop is used to get the **Item** from the location of the actor on the map by calling getItems method. If the actor is damaged by calling the boolean isDamaged method and the item has the capability *ItemCapability.EDIBLE*, a new **EatAction** object will be returned, else return null.

Design Choices / Reasons:

* It implements **Behaviour** because **Behaviour** is the objective an **Actor** owns to help decide which **Action** is to be performed next by overriding the getAction method in **Behaviour**.
* For the **Actor** (**Farmer** and **Human** in this case) to eat, it will return a new **EatAction** object in the method.
* It follows the Declare things in the tightest possible scope principle since **Item** is declared in the enhanced for loop instead of at the beginning of the method.
* It uses the *ItemCapability* enum to check whether an **Item** is edible. [Details of design reasons discussed in **Food**]
* Make use of isDamaged method to check is an actor damage to avoid duplicated code (Do Not Repeat Yourself).

**EatAction** [New class]

* An action which allows the actor (Human, Farmer, Player) to eat food which can recover some health points.

New changes and their Responsibilities:

* This class inherits from an abstract class called **Action**.
* It has two private attributes (**Item** food, final **int** HEAL\_POINTS)
  + food - the **Item** to be eaten and heal the **Actor**
  + HEAL\_POINTS - the points a **Food** can provide to heal the **Actor**
* It has a constructor that takes in one parameter (**Item** food) and initialises the attribute food.
* It overrides:
  + execute() method,
    - Parameter: **Actor** actor, **GameMap** map, Return type: **String**
    - The heal method is called by the actor, with HEAL\_POINTS as the argument to recover the actor’s health points
    - For **Player**,foodwill be removed from the inventory or its location depending on when **Player** performs this action as **Player** can choose to eat or pick up when standing on a food. As for **Human**, food will be removed from the actor’s location on the map.
      * This will look as though the actor has eaten the food and recovers some health points.
    - A string describing the actor eats food and the amount of HEAL\_POINTS recovered is returned.
  + menuDesciption() method.
    - Parameter: **Actor** actor, Return type: **String**
    - A string describing the actor eats food and the amount of HEAL\_POINTS is returned.
    - It will be displayed in the menu for the **Player** to choose as the next action.

Design Choices / Reasons:

* The design where **EatAction** inherits from **Action** follows the Do Not Repeat Yourself principle because **EatAction** also uses the features in **Action** and inheritance helps to avoid repetition of codes.
* HEAL\_POINTS is declared as final, thus it is a constant where it can only be declared once and never change. This follows the Do Not Repeat Yourself and Avoid excessive use of literals principles where I can reuse this final field instead of coding the amount of health points recovered repetitively.
* food is declared as private to follow the Minimize dependencies that cross encapsulation boundaries principle because it may break other systems if we change an originally more visible attribute to be less visible in the future.

***ZombieCapability*** [Existing enum]

* A collection of constants which indicate the capability of **Actor**

New Changes and Responsibilities:

* Added constant: *FARM* [discussed in **Farmer**]

***GroundCapability*** [New enum]

* A collection of constants which is indicate the capability of **Ground**

New Changes and Responsibilities:

* + Constants :
    - *CAN\_BE\_SOWED\_ON* [discussed in **Dirt**]
    - *CAN\_BE\_FERTILIZED* [discussed in **Crop**]
    - *CAN\_BE\_HARVESTED* [discussed in **Crop**]

***ItemCapability*** [New enum, also used in Beating Up the Zombies]

New Changes and Responsibilities:

* + Constants added: *EDIBLE* [discussed in **Food**]

**Application** [Driver class]

New Changes and Responsibilities:

* Add new **Actor** (**Farmer**) onto the **GameMap**
  + randomly place 2 farmers within the fence area after ensuring the location has no other actor.
  + Place another 2 farmers outside of the fence area (one left side and the other right side of the map)

**Assignment 3**

**Going to town**

**Vehicle** [New Class]

* An **Item** which represents a vehicle that moves **Actor** to the other map.

New Changes and their Responsibilities:

* This class inherits from a class called **Item** and it is not portable.
* It has a constructor that takes in one parameter (**String** name). It makes use of the superclass constructor to initialise name and displayChar and a boolean variable portable as false.
* An addAction method is added
  + Parameter: **Action** action – the action to be added into allowableActions

Design Choices / Reasons:

* The design in which the **Vehicle** inherits from Item takes into account the Do Not Repeat Yourself principle because **Vehicle** also uses the features in **Item** and inheritance helps to avoid repetition of codes.
* The addAction method is added in this class so that we can add MoveActorAction that move the actor to the other map into allowableActions in Application (driver class), as actions can only be added to allowableActions in Item class or its subclass.

**Application** [Driver class]

New Changes and Responsibilities:

* A new town map is added in the game world
* Two **Vehicle** objects are created
  + Added in compound map and move actor to town at a fixed location
  + Added in town map and move actor to compound at a fixed location
* addAction method is called by the **Vehicle** objects to add the newly instantiated **MoveActorAction** at a fixed location in respective map

**New Weapons: Shotgun and SniperRifle**

**RangedWeapon** [New Class]

* A ranged weapon which is a type weapon item which uses ammunition.

New Changes and Responsibilities:

* This class inherits from a class called **WeaponItem**.
* It has one private attribute:
  + **Item** ammunition – an item which represents the ammunition to be loaded into the **RangedWeapon**
* It has a constructor that takes in four parameters (**String** name, **char** displayChar, **int** damage, **Item** ammunition). It makes use of the superclass constructor to initialise name, displayChar, damage and verb, and initialise the attribute ammunition.
* loadAmmunition method is added to let the **RangedWeapon** loaded with its respective ammunition if the actor’s inventory contains the ammunition.

Design Choices / Reasons:

* The design in which the **RangedWeapon** inherits from **WeaponItem** takes into account the Do Not Repeat Yourself principle because **RangedWeapon** also uses the features in **WeaponItem** and inheritance helps to avoid repetition of codes.
* loadAmmunition method is to be inherited by its subclass, **Shotgun** and **Sniple Rifle** by following the Do Not Repeat Yourself principle to reduce repetitive codes. This method checks whether the inventory has the **Item** ammunition, and add the *ItemCapability* given in the parameter according to the subclass that is executing this method. The *ItemCapability* to be added is the capability of ranged weapon after it is loaded with its respective ammunition.

**Shotgun** [New Class]

* An shotgun which is a type of ranged weapon that has a short range

New Changes and Responsibilities:

* This class inherits from a class called **RangedWeapon**
* It has a constructor that takes in one parameters (**Item** ammunition). It makes use of the superclass constructor to initialise name, displayChar, damage, verb and ammunition
* It overrides tick method to call loadAmmunition method

Design Choices / Reasons:

* The design in which the **Shotgun** inherits from **RangedWeapon** takes into account the Do Not Repeat Yourself principle because **Shotgun** also uses the features in **RangedWeapon** and inheritance helps to avoid repetition of codes.
* It calls loadAmmunition method from its parent class, and here shows the our design to reduce repeated codes as both sniple rifle and shotgun use ammunition to be fired.

**FireRangedWeaponAction**[New Class]

* An action that allows Actor to fire ranged weapon.

New Changes and Responsibilities:

* This class inherits from a class called **Action**
* It has three private attributes:
  + **Item** rangedWeapon
    - An item which represents the ranged weapon
  + **Item** ammo
    - An item which represents the ammunition
  + **Menu** menu
    - A submenu created if this action is executed
* In the execute method, submenu will be displayed for the chosen ranged weapon if it is loaded with ammunition and fulfil the criteria to be fired if there is any
* For shotgun, it could be fired towards eight directions, which are the cardinal and ordinal directions from the location of the actor.
* For shotgun, the Item ammo is removed if shotgun is chose to be fired to indicate that it has used up the ammunition.
* For sniper rifle, it could fire within a range of 10 spaces horizontally from actor and 6 spaces vertically from actor.

Design Choices / Reasons:

* The design in which the **FireRangedWeaponAction** inherits from **Action** takes into account the Do Not Repeat Yourself principle because **FireRangedWeaponAction** also uses the features in **Action** and inheritance helps to avoid repetition of codes.

**FireShotgunAction**[New Class]

* An action that allows Actor to fire shotgun towards the direction chosen.

New Changes and Responsibilities:

* This class inherits from a class called **Action**
* It has five private attributes:
  + **String** direction
    - A String that specifies the direction where the shotgun is fired towards
  + **int** damagePoints
    - An integer that specifies the damage points of the target
  + **String** hotKey
    - A String that specifies the key used in the menu
  + **boolean** isxPoint1
    - A boolean variable to check if x is point 1 to ensure the coordinate is arranged to be (x,y) and not (y,x).
  + **Actor** target
    - An actor which is the target of the fired shotgun
* It has a constructor that takes in three parameters (**String** direction, String hotkey, int damagePoints to initialise direction, hotkey and damagePoints)
* In the execute method, after checking which direction it is, it would call fireTowardsCardinalDirection or fireTowardsOrdinalDirection method to determine the range of the area of effect respectively.
* fireTowardsCardinalDirection method is added
  + This method is to loop through all the x-coordinates and y-coordinates within the range of area of effect, expect for player itself, to get the actor in the area to be the target of the fired shotgun. **ShootAction** is instantiated to shoot the target.
  + This method is called if the shotgun is fired towards one of the cardinal directions, which is North, South, East or West.
* fireTowardsOrdinalDirection method is added
  + It has the same functionality as fireTowardsCardinalDirection but it is called when the shotgun is fired towards one of the ordinal directions, which is North-West, South-West, North-East or South-East.

Design Choices / Reasons:

* The design in which the **FireShotgunAction** inherits from **Action** takes into account the Do Not Repeat Yourself principle because **FireShotgunAction** also uses the features in **Action** and inheritance helps to avoid repetition of codes.
* fireTowardsCardinalDirection and fireTowardsOrdinalDirection methods are created as different methods with the same objective which is to shoot the actor within the range of the area of effect.
  + This is because the area of effect when shooting towards the cardinal direction is different from the ordinal direction.
  + These two methods are created to reduce repetitive codes to follow the Do Not Repeat Yourself principle.
  + In these methods, ShootAction is instantiated and executed instead of coding the process of hurting the Actor in this class because both shotgun and sniple rifle shoot and hurt actor/s. Here, we also take the Do Not Repeat Yourself principle into consideration while designing the code.
  + **NumberRange** is used to loop through the specific coordinates of the map to reduce code redundancy. NumberRange allows deepcopy of the list and this helps us to avoid privacy leaks. Privacy leaks is important to be prevented as it will lose the benefits of encapsulation and have the risk to increase dependency. Here, Reduce dependencies as much as possible principle Minimize dependencies that cross encapsulation boundaries principle is followed.
  + ArrayIndexOutOfBoundsException is handled in both methods as the exception will be thrown if we try to fire shotgun towards the location which is out of the range of the game map.

**ItemCapability**[Existing enum]

New Changes and Responsibilities:

* LOADED\_WITH\_SHOTGUN\_AMMO
  + item (shotgun) is loaded with shotgun ammo
* LOADED\_WITH\_RIFLE \_AMMO
  + item (sniper rilfe) is loaded with sniple rifle ammo
* LOADED\_IN\_SHOTGUN
  + ammo which is loaded in shotgun
* LOADED\_IN\_RIFLE
  + ammo which is loaded in sniple rifle

**FireSniperRifleAction [New Class]**

* An action that allow actor to fire sniper rifle within a range

New Changes and their responsibilities:

* 1. It has four private attributes which are target, damage,ammo and menu.

where target is the actor that is fired, damage is the damage of the weapon, ammo is the weapon’s ammo and menu is to show option for two possible actions which are shoot and aim action.

* 1. If target has no any of the AimCapability

Design Choice:

**SniperRifle [ New Class]**

New Changes and their responsibilities:

* This class inherits from a class called **RangedWeapon**
* It has a constructor that takes in one parameters (**Item** ammunition). It makes use of the superclass constructor to initialise name, displayChar, damage, verb and ammunition
* It overrides tick method to call loadAmmunition method

Design Choice:

* The design in which the **SniperRifle** inherits from **RangedWeapon** takes into account the Do Not Repeat Yourself principle because **SniperRifle** also uses the features in **RangedWeapon** and inheritance helps to avoid repetition of codes.
* It calls loadAmmunition method from its parent class, and here shows the our design to reduce repeated codes as both sniple rifle and shotgun use ammunition to be fired.

**AimAction [ New Class]**

New Changes and their responsibilities:

Design Choice:

**ShootAction[ New Class]**

New Changes and their responsibilities:

Design Choice:

**AimCapability [ New Enum]**

* A collection of constants which is indicate the capability of aiming

New enum added:

ROUND1

ROUND2

CONCENTRATION

**Mambo Marie**

**MamboMarie[ New Class]**

* A type of actor called Zombie.

New Changes and their responsibilities:

* 1. It has two private attributes which are **int** turn and array of behaviours that consist of 2 **behaviour** that are ChantingBehaviour and Wanderbehaviour.
  2. ZombieCapability.Mambo is added into the class.
  3. Every time when playTurn is called the turn will have an increment to indicate that she has been in the game for a turn. Every 10th turn Mambo Marie will have ChantingBehaviour by adding a capability called ZombieCapability.CHANTand when turn reaches the 30th turn, Mambo Marie will have VanishAction. Other turns she will just have WanderBehaviour.

Design Choice:

Mambo Marie is neither a human nor a zombie, she is a unique actor which will summon zombies and vanish. She has 5% chances of reappearing and this cycle will not stop until she is killed. Zombies will not attack Mambo Marie.

**NewWorld [New Class]**

* A type of World called NewWorld.

New Changes and their responsibilities:

1. This class inherits from a class called **World**.
2. It has two private attributes which are **Actor** marie and **Location** mamboLocation.

marie is the **MamboMarie** and mamboLocation is the location of the **MamboMarie** before she vanished.

1. It has a private method called getMambo. Its function is if Mambo Marie has ZombieCapability.VANISH which means she has vanished, it will added Mambo Marie back to the map by using mamboLocation. If she is dead, she will not be added into the game
2. The public method run is overridden to add getMambo() into the method.

Design Choice:

When designing NewWorld, it inherits World to prevent repetition of codes which follows the DRY principle. It runs pretty much the same as its parents class with the addition of adding mambo marie into the game.

**ZombieCapability [Existing class]**

New enum is added:

CHANT

MAMBO

VANISH

**SummonZombieAction [ New Class]**

* A type of action that summons Zombie

New Changes and their responsibilities:

1. It has a private attribute of numOfZombies which is the number of zombie to be summoned.
2. In the execute method, the names of **Zombie** to be summoned will be randomly generated. Bases on numOfZombies, Zombie objects are initialized and added all over the map in random loactions.

Design Choice:

1. When designing **SummonAttackAction** class, it inherits from **Action** class. This reduces repetition of codes and hence follows the  DRY( Don’t repeat yourself) principle.
2. Names of Zombies are randomly generated instead of using a set of names. The reason for this is to benefit user so that user will not be confused when playing the game as using a set of names will cause **Zombie** to have the same names.

**VanishAction [ New Class]**

* A type of action that makes an actor disappear.

New Changes and their responsibilities:

1. The execute method will add a capability called ZombieCapability.VANISH to indicate that the actor has vanished. Then, the actor will be remove form the map.

Design Choice:

1. The ZombieCapability.VANISH enables NewWorld to know that marie has really vanish not dead.
2. When designing **VanishAction** class, it inherits from **Action** class. This reduces repetition of codes and hence follows the  DRY( Don’t repeat yourself) principle.

**ChantingBehaviour [New Class]**

* A type of behaviour that allows the actor to summon zombies

New Changes and their responsibilities:

1. if the actor has ZombieCapability.CHANT , the method getAction will remove ZombieCapability.CHANT and summon 5 zombies. The reason to remove the capability is so that it will not summon zombies in every turn.

Design Choice:

1. I have decided to create **ChantingBehaviour** to called **SummonZombieAction**. This is because it is easier extend and maintain if there are additional features to be added in **ChantingBehaviour**, not just summoning zombies.

**Ending the Game**

**NewWorld [ New Class]**

* A type of World called NewWorld.

New Changes and their responsibilities:

1. A new private attribute is added into the class called GameMap compundMap.

This is the map of the Compound.

1. Three private methods are added into the class which are win, lose and getAllActors

getallActors is a method that returns an ArrayList of actors on the compound that are conscious in the game. win is a Boolean method that returns true if all **Zombie** and **MamboMarie** are dead where as lose is a Boolean method that return true if **Humans** and **Player** are dead.

1. A new public method called setCompoundMap is added. Its function is to set the compoundMap attribute to be the compound map.
2. 2 other methods from the parents class are also overridden that are stillRunning and endGameMessage. stillRunning method now return false of one of the three types of actors are dead which are **Human, Zombie&MamboMarie and Player**. This method makes use of win and lose to determine whether all zombies or human are dead, and uses its parent’s stillRunning method to determine Player’s dead. endGameMessage method now returns three possible types of String message as player is not the only factor that will end the game.

Design Choice:

When designing **NewWorld** class, the idea of encapsulation is used as some methods need to be private as it will only be called in the class.

**QuitGameAction[ New Class]**

* A type of action that allows an actor to quit game

New Changes and their responsibilities:

1. If it is called, the actor will be removed from the map.

Design Choice:

1. Once the player is removed from the map, the game will stop running.

**Player [ Existing class]**

New Changes and their responsibilities:

1. Inside the execute method, a new QuitGameAction is added into the action.

Design Choice:

This is to give the player to have an option to quit game in the menu.

**Bonus Feature**

**Bomb**

****

* A bomb which is a type of Item and hurt Actors when it explodes.

New Changes and Responsibilities:

* It has a constructor that takes no parameters and makes use of the superclass constructor to initialise **String** name, **char** displayChar and int portable
* It overrides the tick() method.
  + This method is called in every turn.
  + The bomb has a 20% probability to explode if an Actor steps on it.
  + In this method, we can see that once the bomb explodes, any type of items and grounds will vanish and turn to **Dirt** respectively. These dirts lose the capability to allow crops to be sowed on it anymore. The bomb wil also hurt all the actors located on the location of the bomb itself and the adjacent locations.

Design Choices / Reasons:

* The design in which the **Bomb** inherits from **Item** takes into account the Do Not Repeat Yourself principle because **Bomb** also uses the features in **Item** and inheritance helps to avoid repetition of codes.