**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

**Limbs [New Class]**

* A type of item that has the properties of a limb.

New Changes and their responsibilities:

1. This class inherits from a class called **PortableItem**.
2. It has a constructor that takes in 2 parameters (**String** name and **char** display ). It uses the superclass constructor to initialize its parameters. A setLimbsCapability() is added into the constructor to add a capability if the display parameter is either “I” or “A”.
3. There is a private method called  setLimbsCapability() that takes in a parameter (**char** display) . Its function is to add Capability depending on the parameter display . If the display is “A”, it will add a capability called LimbsCapability.ARM. If the display is “l”, it will add a capability called LimbsCapability.LEG

Design Choice:

1. When designing the **Limbs** class, it inherits the **PortableItem** class instead of Item class as **Limbs** are portable too in the game. Hence, it follows the DRY( Don’t repeat yourself) principle that prevents repetition of codes.
2. **Limbs** act as a “limb” for the **Zombie** but it is actually an item and it is stored in the inventory.
3. The added capability can then be used to identify whether the item has properties of an arm or a leg.

**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. A setLimbsCapability() is added into the constructor to add a capability if the display parameter is either “I” or “A”.
3. There is a private method called  setLimbsCapability() that takes in a parameter (**char** display) . Its function is to add Capability depending on the parameter display . If the display is “A”, it will add a capability called LimbsCapability.ARM. If the display is “l”, it will add a capability called LimbsCapability.LEG
4. A new **CraftingAction** is added into the allowableActions (attributes in **Item**) so that **SimpleClub** can be crafted by Player.
5. 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 but **Limbs** is an **Item** so it can’t act as an **WeaponItem**. Hence, when a Zombie is about to lose its **Limbs**, **SimpleClub** is instantiated in the **AttackAction** class and drops on the ground while **Limbs** is removed from the inventory.
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.

**AttackAction [Existing class]**

New Changes and their responsibilities:

1. The execute() method is modified to have a chance to “drop limb” when a **Zombie** is being attacked. If the actor is conscious and has a capability of ZombieCapability.UNDEAD , it will have a chance of 25% to drop 1 of its **Limb**. When it hits the chance, the **Limb** from the inventory will be removed and a new **SimpleClub** object will be instantiated and dropped on the ground.

Design Choice:

Technically, the SimpleClub is acting as a ‘limb’ but the inner properties of the limb have been switched from a **Limbs** class to a **SimpleClub** class.

**Zombie [Existing class]**

New Changes and their Responsibilities

1. The private attributes added:
   1. **int** numberOfArms,
   2. **int** numberOfLegs,
   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. checkStatus() 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. The number of items in the inventory that has LimbsCapability.ARM will be assigned to numberOfArms and the number of items in the inventory that has  LimbsCapability.ARM will be assigned to numberOfLegs.  numberOfArms and numberOfLegs shows how many arms and legs the Zombie currently have every turn.
   3. 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%.
4. playTurn() method is modified.
   1. At the beginning of the playTurn() method, checkStatus() method is called.
   2. 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.
   3. When numberOfLegs is 2, the array used to loop through all the elements is behaviours, else when numberOfLegs is 0, behaviourWithoutLegs is used instead.
   4. If the **Action** get by the elements in the selected array is all null, it will return a new DoNothingAction object.

Design Choices / Reasons:

If Zombies have dropped both arms, it will still have pick up behaviour (maybe with its jaw) but the weapon will still drop after picking it up because it is unstable to use a jaw to hold a weapon. The modified playTurn is use to control the the movement of the Zombie. If behaviours is used, the zombie moves normally. If behaviourWithoutLegs is used, zombie can only attack and pick up weapon.

**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

**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 two methods.
3. 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  a class called **SimpleClub** 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 capability of LimbsCapability.ARM (enum) can be crafted into a **ZombieMace** and capability of LimbsCapability.ARM (enum) can be crafted into a **ZombieClub**. If it is successful, it will return a string that indicates the item is crafted.
4. 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 capability of LimbsCapability.LEG.

**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 capability of LimbsCapability.ARM.

Rising from the dead

**Corpse** [New Class]

* An **Item** added to the **GameMap** when an **Actor** (**Human**, **Farmer** and **Zombie**) is killed.

New Changes and their Responsibilities:

* This class inherits from an abstract class called **Item**.
* It has two private attributes (**int** turn and **Actor** initialActor).
  + turn - acts as a counter
  + initialActor - the dead **Actor** who owns the **Corpse**
* It has a constructor that takes in 2 parameters (**String** name and **Actor** initialActor). It initialises initialActor and also makes use of the superclass constructor to initialise name, displayChar and portable.
* 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 is placed
  + This method is executed every turn in the game.
  + turn increments by 1.
  + If initialActor does not hasCapability(*ZombieCapability.UNDEAD*), in which it is not a **Zombie**, generate a random integer 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 Item takes into account the Do Not Repeat Yourself principle because Corpse also uses the features in Item 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.
* initialActor is 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.

**AttackAction** [Existing Class]

New Changes and their Responsibilities:

* execute() method is modified
  + If an **Actor** is killed after being attacked [existing code], a new **Corpse** object is instantiated [modified] instead of a **PortableItem** [existing code], then added to the **Location** of the dead **Actor** which is removed from the **Location** [existing code].
  + This is because there is a new **Corpse** class created in which it is able to increment the counter every turn in the game to control the turn when it becomes a **Zombie** to fulfil the required functionality.

Design Choices / Reasons:

* **Corpse** is instantiated 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

**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 four private attributes stored in an array of type **Behaviour** - behaviours

* Its elements are new object instantiated from **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 which is the return value of returnEatAction method, then the return value of getAction method called by **HarvestBehaviour**, **FertilizeBehaviour**, **SowBehaviour**, **WanderBehaviour** and lastly return a new **DoNothingAction** object.

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.
* returnEatAction method in Human class is called in the playTurn method instead of instantiating a new **EatBehaviour** object which calls the getAction method to return its **Action.** This is to follow the principle:
  + Do Not Repeat Yourself as to avoid repetition of codes since **Human** and **Farmer** can both eat **Food** when they are damaged [details for method discussed in Human class]
  + Reduce Dependencies as much as possible since it reduces the dependency to EatBehaviour
* 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 implement, it is arranged to be the last before wandering and doing nothing.

**Human** [Existing Class]

New Changes and their Responsibilities:

* returnEatAction method is added
  + Visibility: protected, Parameter: **GameMap** map, Return type: **Action**
  + This method is to return an **Action** to eat if this **Actor** is damaged and when the **Actor** is able to own **EatBehaviour**.
  + It is called in the playTurn method in both **Human** and **Farmer** class, thus also executed at every turn in the game.
  + If hitPoints is smaller than maxHitPoints, and if the **Action** get from new EatBehaviour().getAction(this, map) is not null, the **Action** is returned,
  + Else, return null
    - hitPoints and maxHitPoints are protected attributes in **Actor**
* playTurn method is modified
  + Before returning **Action** get from the getAction method called by the attribute in Human(behaviour- new **WanderBehaviour** object)[existing code], returnEatAction(map) method is called to return its return value if it is not null [modified].

Design Choices / Reasons:

* returnEatAction method is created to avoid duplicated code since this behaviour is shared by both **Human** and its subclass, **Farmer** if they are damaged. This follows the design principle Do Not Repeat Yourself.
* New **EatBehaviour** object is instantiated in returnEatAction method to follow Declare things in the tightest possible scope principle since it is declared as a local variable instead of as an attribute.

**Crop** [New class]

* An item which can be sowed, fertilized when unripe and harvested when it has ripened for food.

New Changes and their Responsibilities:

* This class inherits from an abstract class called **Item**.
* It has two private attributes (**int** age).
  + age - acts as a counter
* It has a constructor that takes no parameter. It makes use of the superclass constructor to initialise name, displayChar and portable. It also calls addCapability() method from **Item** with *ItemCapability.CAN\_BE\_FERTILIZED*  as the argument.
* It has a getter and setter method for the attribute, age.
  + This information is needed in **FertilizeAction** in order to decrease the time left for the **Crop** to be ripened.
* 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.
* In the method, when age reaches 20, the displayChar is reassigned with another **char** to show to the player that the **Crop** has ripen.
* removeCapability method from **Item** is called to remove *ItemCapability.CAN\_BE\_FERTILIZED* from the capability of the **Crop**.
* addCapability method from **Item** is called to add *ItemCapability.CAN\_BE\_HARVESTED* to be the capability of **Crop**.
* A new **HarvestAction** object whose argument is **Crop** itself and location is added in allowableActions, the attribute in **Item**. This is to allow **Crop** to be harvested by the **Player** when the Player is standing on the ripe **Crop**.

Design Choices / Reasons:

* The design in which the **Crop** inherits from **Item** takes into account the Do Not Repeat Yourself principle because **Crop** also uses the features in **Item** and inheritance helps to avoid repetition of codes.
* *ItemCapability* is an Enum and defines a collection of constants to indicate the capability of an **Item**. Crop is designed to be:
  + Initially, **Crop** is unripe, thus it has the capability *ItemCapability.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 *ItemCapability.CAN\_BE\_FERTILIZED* is removed.
    - A ripe **Crop** can be harvested, thus the capability of *ItemCapability.CAN\_BE\_HARVESTED* is added.
  + This is designed to avoid repetition of code and avoid using getDisplayChar method from **Item** 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.

**Food** [New class]

* An item 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.

**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.SOW* as the argument is added in the constructor

Design Choices / Reasons:

* **Dirt** is designed to have the capability *GroundCapability.SOW* 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 maintability and 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**
  + Use an enhanced for loop 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**, if so it will have 33% probability that 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. [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 instantiated and added onto sowLocation.
    - 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**
  + Use an enhanced for loop to get the **Item** from the location of the actor on the map by calling getItems method and check whether the item is capable of being fertilized , if so 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 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 capable of being fertilized. [Details of design reasons discussed in **Crop**]

**FertilizeAction** [New class]

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

New Changes and their Responsibilities:

* This class inherits from an abstract class called **Action**.
* It has one private attribute (**Crop** unripeCrop)
  + unripeCrop - the **Crop** to be fertilized
* It has a constructor that takes in one parameter (**Crop** unripeCrop) and initialises the attribute unripeCrop.
* It overrides:
  + execute() method,
    - Parameter: **Actor** actor, **GameMap** map, Return type: **String**
    - Get the age by using the getAge method called by unripeCrop. Then, add the age by 10 to look as though the time left to ripen is decreased by 10 turns because it is fertilized to fulfil the required functionality.
    - If the current age is already 10 or more, its age is made to be 19 manually, since a **Crop** will ripen in 20 turns and the tick method which increases the age by 1 in **Crop** class will only run after this **Action** is called.
    - 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.

**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 all the location and adjacent locations of the actor. locationList stores the location of the actor on the map.
  + A loop is used to get the **Exit** from the location of the actor on the map and call getDestination to get the adjacent locations and store them in the locationList.
  + Use another loop to get the **Item** from the element in the locationList by calling getItems method, if it is capable of being 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** and **Item** are 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 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 crop at a specific location.

New changes and their Responsibilities:

* This class inherits from an abstract class called **Action**.
* It has two private attributes (**Item** item, **Location** location)
  + item - the **Item** to be harvested
  + location - the **Location** of the **Crop** to be harvested by the **Actor**
* It has a constructor that takes in two parameters (**Item** item, **Location** location) and initialises the attributes item and location.
* 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.
    - If the actor is capable of farming (i.e **Farmer**), food is added onto the location and the item is removed from location.
    - Else (i.e. **Player**), food is added into the **Player**’s inventory and the item is removed from location.
      * Both conditions above are to fulfil the functionality required when they harvest. It will look as though they are harvesting for food.
    - 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.
* item and location are 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 and check whether the item is edible, if so it will return a new **EatAction** 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** 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**]

**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
    - Then, for **Player**,foodwill be removed from the inventory. 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 recoverd 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.
* item and location are 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 : *SOW* [discussed in **Dirt**]

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

New Changes and Responsibilities:

* + Constants added:
    - *CAN\_BE\_FERTILIZED* [discussed in **Crop**]
    - *CAN\_BE\_HARVESTED* [discussed in **Crop**]
    - *EDIBLE* [discussed in **Food**]