Classes Responsibilities & Design Rationale

Zombie attacks

Functionality: Zombie attack using intrinsic weapon (punch/bite)

New classes	Class Responsibility	Design Rationale
ZombieIntrinsicWeapon extends IntrinsicWeapon	Represents the Zombie intrinsic weapon:punch and bite. - Has 2 new **private** attributes: 1. healPoint: the heal point that a Zombie will receive when it launches a successful attack (punch:0, bite:5). 2. missChance: the chance for a Zombie to miss an attack when it uses the zombie intrinsic weapon (punch:50%, bite:70%).	This class is created because the Zombie intrinsic weapon has two more attributes/features than the current implemented intrinsic weapon. It extends the features of a normal intrinsic weapon, thus, it is a subclass of IntrinsicWeapon. Besides, it is not a good design to hardcode and handle the healPoint and missChance values in the ZombieAttackAction class, following the design principles: **Avoid excessive use of literals** and **classes should be responsible for their own properties**.
ZombieAttackAction extends AttackAction	Implements and handles all the attack actions that can be carried out by Zombie. - Override method: 1. execute(Actor actor, GameMap map): handles the different probability for Zombie to miss when it uses different form of attack (punch:50% miss, bite:70% miss, weapon:50% miss) - New **private** method: 1. attackSuccess(Actor actor, Actor target, Weapon weapon, GameMap map): handles the case when the Zombie attacks successfully. Uses source code provided to deduct the	This class is introduced because it is found that the Zombie and Human attack differently and the effects of their attacks are different as well. Following the **Single Responsibility Principle**, the code in AttackAction should be easy to understand and handle only one type of attack. Therefore, since the new functionality has complicated the code, it is better to create new subclasses and let each subclass handle different types of attack action. Following the design principle: **minimize dependencies that cross encapsulation boundaries**, both target and attackSuccess(Actor, Actor, Weapon, GameMap) will be declared as private.

	target's health point, as well as create a HumanCorpse when the target (Human) becomes unconscious	

Modified Classes	Class Responsibility	Design Rationale
Zombie	Represents the Zombie creature in the game. - Has 1 new **private** attribute: 1. intrinsicWeapons: a HashMap that stores verb as key, and an array of integer, i.e. [damage, healPoint, missChance], as the value. - Modified method: 1. getIntrinsicWeapon(): randomly choose an intrinsic weapon from the HashMap, creates the ZombieIntrinsicWeapon object and returns it.	Initially the zombie has only one intrinsic weapon, that is punch, but now it has two: punch and bite. Therefore, we need to store a Collection of intrinsic weapons for each Zombie. The original getIntrinsicWeapon() cannot be used anymore as the Zombie now has more than one intrinsic weapon. In the latest design, each Zombie has 2 intrinsic weapons. However, instead of hardcoding the probability as 50%, our design is to first create an array from the key set of the HashMap intrinsicWeapons and determine the number of intrinsic weapons a Zombie has (n). Then, the method will generate a random number ranging from 0 to n-1 (r). Then, we will use the key value at array[r] to create the ZombieIntrinsicWeapon instance and return it. This implementation gives all the intrinsic weapons an equal chance of being chosen without hardcoding any probability value. This **reduces the implicit dependency on literals** and also provides the system with more flexibility. We can easily increase the intrinsic weapons that a Zombie can have without worrying

		about the probability of choosing each weapon.
AttackAction	Becomes an Abstract class, and has 2 subclasses: ZombieAttackAction & HumanAttackAction - Removes the constructor but retains the declaration of the attribute target. However, the value of target will be assigned in the constructors of HumanAttackAction and ZombieAttackAction. - Retains menuDescription(Actor actor) but execute(Actor actor, GameMap map) is being moved to its subclasses - This class will be useful in the case where we want to refactor code in ZombieAttackAction and HumanAttackAction	It is decided to make AttackAction an abstract class because the attack actions for both Human and Zombie have much difference, hence it is better to create subclasses for them. The AttackAction class is made abstract and not deleted to prevent other classes that rely on it from failing. Since we make use of inheritance, any place that accepts an AttackAction instance will be able to accept a ZombieAttackAction instance or a HumanAttackAciton instance (the **Liskov Substitution Principle**), so this modification will not fail the program.
AttackBehaviour	Represents the attack behaviour of both Zombie and Human. - Modified method: 1. getAction(Actor actor, GameMap map): returns a ZombieAttackAction or a HumanAttackAction by checking the Actor's ZombieCapability	Initially the method returns an AttackAction instance, however, with the new design, AttackAction has been changed to become an abstract class, meaning we cannot instantiate AttackAction anymore. Therefore, an if-else statement will be used to check the team the Actor is at, and decide the type of AttackAction object that will be returned. If it is found that the actor has the capability ZombieCapability.UNDEAD, this means that the actor is a Zombie, and hence a ZombieAttackAction instance will be created and returned. If the actor has the capability ZombieCapability.ALIVE, this means that the actor is a Human, and thus a HumanAttackAction instance will be created and returned.

ZombieActor	Represents the actors in the Zombie world. - Modified method: 1. getAllowableActions(Actor otherActor, String direction, GameMap map): change the method such that it will create and return ZombieAttackAction or HumanAttackAction instead of AttackAction.	Initially the method returns an AttackAction instance, however, with the new design, AttackAction has been changed to become an abstract class, meaning we cannot instantiate AttackAction anymore. If the ZombieActor has the capability of ALIVE and otherActor has the capability of UNDEAD, then a ZombieAttackAction instance will be created and returned. If the ZombieActor has the capability of UNDEAD and otherActor has the capability of ALIVE, then a HumanAttackAction instance will be created and returned.

Functionality: Zombie picks up weapon

New Class	Class Responsibility	Design Rationale
PickUpItemBehaviour Implements Behaviour	A class which implements Behaviour interface and generates a PickUpItemAction if the current Actor is standing in a location which has a weapon. - Modified method: 1. getAction(Actor actor, GameMap map): returns a PickUpItemAction if the current Actor is standing in a location which has a weapon and null otherwise.	This class is created because it represents a behaviour that is not modelled in the original system. Following the **Single Responsibility Principle**, this PickUpItem behaviour is not merged into other behaviour classes. This implementation is based on the concept of **Polymorphism**, which allows the playTurn() method in Zombie class to loop through the behaviours array and call the getAction() method of each subclass of Behaviour interface in the array to check if the specific action represented by the subclass is possible to be performed by the Zombie in its current turn.

Functionality: Zombie says "Braaaains"

New Classes	Class Responsibility	Design Rationale
GrowlBehaviour implements Behaviour	Represents the Zombie's growling behaviour, where the Zombie will have 10% chance to say "Braaaaaains" at each turn. - Has 1 **private** attribute: 1. growlChance: stores the probability of Zombie to growl, that is, 10 - Override method: 1. getAction(Actor actor, GameMap map): ensures that the Zombie has a 10% chance of growling at every turn. A random integer from range 0 - 100 will be generated, and if the value is less than growlChance, the Zombie will growl, so a GrowlAction instance will be returned.	This class is created because it represents a behaviour that is not modelled in the original system. Following the **Single Responsibility Principle**, this growling behaviour is not merged into other behaviour classes. Besides, growlChance is being used in comparison instead of hardcoding the value 10 to **reduce implicit dependency on literals**. In our game design, a Zombie can only carry out one action at each turn, that is, the Zombie cannot attack and growl at the same time. If it growls, it does not attack any Human at that turn.
GrowlAction extends Action	Represents the growling action of Zombie. - Override methods: 1. menuDescription(Actor actor): returns a string that impersonate a zombie growling, e.g. "zombie says Braaains". 2. execute(Actor actor, GameMap map): calls menuDescription(Actor actor) and returns the string.	This class is created because it represents a Zombie action that is not modelled in the original system. Following the **Single Responsibility Principle**, this action is not merged into other action classes.

Modified Class	Class Responsibility	Design Rationale
Zombie	- Modified **private** attribute:	The GrowlBehaviour instance is being added into the

	1. behaviours: Add GrowlBehaviour into the behaviours array	behaviours array so that it will be looped through when the Zombie's playTurn(Actions actions, Action lastAction, GameMap map, Display display) is being called.

Beating up the Zombies

Functionality: Allow Human to attack Zombie

Modified Classes	Class Responsibility	Design Rationale
Human class	Class representing an ordinary human - Modified default attribute: 1. behaviours: an array of Behaviour objects which store both AttackBehaviour and WanderBehaviour. It is being changed to an ArrayList object and its access modifier is changed to default.	In order to make Human be able to perform multiple actions, it is decided to change Human class attribute behaviour into an array of Behaviour objects. This allows different Behaviour objects to be stored into Human class and allow getAction() method to be called on each Behaviour based on the concept of **Polymorphism**, similar to the implementation in playTurn() for Zombie class. This makes Human class functionality more extensible. The data structure and access control of behaviours are changed so that the Human's subclass, Farmer, can add more Behaviour into the behaviour list easily. Nevertheless, **dependencies that cross encapsulation boundaries are minimized**. Since Farmer and Human are classes in the same package and Farmer needs to modify behaviours, the access modifier of behaviours is changed to the default instead of public.
Human class Zombie class ZombieActor	- Parent class: 1. ZombieActor: an abstract class which acts as the base class for Actors in the Zombie World	Since we change the Human class attribute to an array of Behaviour objects, the playTurn() method in both Human and Zombie class will be exactly the same. Thus, the existing playTurn() method in Human class

	- Subclasses of ZombieActor which are modified: 1. Human: Class representing an ordinary human 2. Zombie: Class representing a zombie - Modification: 1. Remove playTurn() method from Human class 2. Move playTurn() method from Zombie class to ZombieActor class	which only handles one behaviour will be removed. The playTurn() method in Zombie class which handles an array of Behaviours will be moved to ZombieActor Class. This will allow both Human and Zombie which are both subclasses of ZombieActor class to be able to use the same playTurn() method to deal with their own behaviour array. The decision of moving common methods from subclasses into their parent class is based on the **Don't Repeat Yourself principle** to reduce lines of repeated code between the three classes.

Functionality: Knock zombie's limb off

New Classes	Class Responsibility	Design Rationale
HumanAttackAction extends AttackAction	A class that represents an action for attacking Zombie - Add new **private** attribute: 1. int DROP_LIMB_CHANCE: a constant integer which represents the probability of the attack knocking zombie's limb off, which is 25 - Override method: 1. execute(Actor actor, GameMap map): attack the target zombie and each success attack has a 25% chance to knock one of the zombie limbs off by calling dropLimb() method in Zombie class. Return string which describes the result of the attack.	Based on the **Single Responsibility Principle**, we created a new class HumanAttackAction to separate the attack action for human or player and attack action for zombie, which have quite different implementations. The constant DROP_LIMB_CHANCE is being used in comparison instead of hardcoding the value 25 to **reduce implicit dependency on literals** and **avoid excessive use of literals.** In order to **minimize dependencies that cross encapsulation boundaries**, we declare the DROP_LIMB_CHANCE to be private and constant to HumanAttackAction class.
Limb extends PortableItem	A new class which represents the arm or leg knocked off from the zombie.	To make Limb able to be picked up, we make Limb to become a subclass of PortableItem which is a subclass

	- Constructor: 1. Limb(String name, Char displayChar, Enum<?> capability): super() is used to create a PortableItem which represents the Limb and addCapability is called inside the constructor to add the capability to the Limb. - Has capabilities: 1. CRAFTABLE_INTO_CLUB: added for limbs that represent arms 2. CRAFTABLE_INTO_MACE: added for limbs that represent legs - Override method: 1. getAllowableActions(): return CraftWeaponAction(this)	of Item, the existing getPickUpAction() in Item class will be able to return a PickUpItemAction for the Limb. Therefore, we do not have to add additional code to associate a PickUpItemAction for Limb and thereby fulfilling the **Don't Repeat Yourself principle** and **reduce dependencies as much as possible** as no new association is created between PickUpItemAction and Limb. In order to **reduce implicit dependency on literals** and **avoid excessive use of literals,** instead of hardcoding the name of the Limb,"arm" and "leg", the two constant ZombieCapability, CRAFTABLE_INTO_CLUB and CRAFTABLE_INTO_MACE are added into limb to be used to compare and decide if an item can be crafted into weaponItem. In our game design, the Zombie is able to pick up their own limbs after dropping them but they will not be able to craft the limb into weapons. So far, in our gameplay design, only Player can craft Zombie limbs into weapons. This is because only Player makes use of the Actions object in playTurn() to decide on the action being carried out at a certain turn. Furthermore, we did not implement a CraftWeaponBehaviour for Human or Zombie, hence they will not be able to craft the limbs into weapons.
CraftWeaponAction extends Action	A new class which represents an action for crafting zombie arm or leg into club or mace. - Add new private attribute: 1. Item: item to be crafted	This class is created because it represents an action that is not modelled in the original system. Following the **Single Responsibility Principle**, this crafting weapon action is not merged into other action classes.

	- Override method: 1. execute(Actor actor, GameMap map): The method will first check the item's capability. If the item has the capability of ZombieCapability.CRAFTABLE_INTO_CLUB, remove the item from the actor inventory and store a Club weaponItem into the actor inventory. If the item's capability is ZombieCapability.CRAFTABLE_INTO_MACE, remove the item from the actor inventory and store a Mace weaponItem into the actor inventory. 2. menuDescription(Actor actor): Describe the craft weapon action in a format suitable for displaying in the menu.	Based on the principle of **Don't Repeat Yourself**, we make the CraftWeaponAction to inherit from the Action class as both have similar methods that can be reused or overridden and by doing so, we can reduce the number of duplicated code. As explained in Limb class, CraftWeaponAction is now made available for the Player. However, it can be extended to be used by Human, by creating a new CraftWeaponBehaviour class to be added into Human behaviour list attribute, which will return a CraftWeaponAction to the Human object if there is an item which can be crafted into a weapon in the Human inventory.
Club extends WeaponItem	A new weapon crafted from zombie arm. - Constructor: 1. Club(): call super("club", "~", 30, "use club to hit") to create a club WeaponItem	By making Club a subclass of WeaponItem which is a subclass of Item, we can use all the methods existing in the parent classes to implement all the functionalities of club, such as having attributes damage and verb and method damage(), getDropAction(), etc. This reduces lines of repeated code, thus fulfilling the **Don't Repeat Yourself principle.** Based on the **Single Responsibility Principle**, Club and Mace are created separately instead of merging into one class to make each class responsible for the data that defines itself such as the damage value and the verb used for attacking using this weapon.

Mace extends WeaponItem	A new weapon created from zombie leg. - Constructor: 1. Mace(): call super(“mace”, “$”, 40, “use mace to hit”) to create a mace WeaponItem	The design rationale for making Mace class is exactly the same as Club class. By making Mace a subclass of WeaponItem which is a subclass of Item, we can use all the methods existing in the parent classes to implement all the functionalities of mace, such as having attributes damage and verb and method damage(), getDropAction(), etc. This reduces lines of repeated code, thus fulfilling the **Don’t Repeat Yourself principle.** Based on the **Single Responsibility Principle**, Club and Mace are created separately instead of merging into one class to make each class responsible for the data that defines itself such as the damage value and the verb used for attacking using this weapon. Besides, our game design has followed the specification rule, in which the weapons crafted from Zombie limbs cause more damage than all currently available weapons and Mace object has higher damage than Club object.

Modified Class	Class Responsibility	Design Rationale
ZombieCapability	Stores all the enumeration values that are used in the Zombie world. - Add enum: 1. CRAFTABLE_INTO_CLUB: indicates that the object can be crafted into club (Zombie limb: arm) 2. CRAFTABLE_INTO_MACE: indicates that the	These two capabilities are added so that we can determine if the Zombie limb picked up can be crafted into club or mace.

	object can be crafted into mace (Zombie limb: leg)	
Zombie	Class which represents the zombie. - Has new **private** attributes: 1. int arm: The number of arms of the zombie, which is 2, as specified in the specification rule 2. int leg: The number of leg of the zombie, which is 2, as specified in the specification rule 3. boolean slowMotion: Indicates if the Zombie is in slow motion (only have one leg). It is initially set as false. 4. Behaviour[] MOVING_BEHAVIOUR: a constant array consist of Behaviour objects which will change location of zombie - Add new methods: 1. dropLimb(): randomly choose to drop one of the zombie limbs if there is still any and change the properties of the Zombie such as behaviour list or punch miss chance according to the remaining number of arms and legs of the zombie by calling dropArm() or dropLeg() method in the Zombie class. Otherwise, return without doing anything. 2. dropArm(): decrement arm attribute by 1 and create and drop an arm which is of Limb type onto the ground at the position of the actor who	To **reduce implicit dependency on literals** and **avoid excessive use of literals,** the constant array MOVING_BEHAVIOUR is used to determine the fixed set of Behaviour to be added and removed from the zombie alternating turn after it loses one leg. In order to **minimize dependencies that cross encapsulation boundaries**, we declare all of the variables to be private to the class. In our game design, if the Zombie limb is knocked off, then we will randomly knock off any of the remaining limbs. The limb will be dropped at the location of the Zombie.

	is attacking the zombie (adjacent to zombie's location). Then check if arm>0, double the miss chance of punch stored in the intrinsicWeapon attribute of Zombie and use the chance of 50% to drop the first weapon in Zombie inventory if there is any. Else, drop all the weapons in the zombie inventory, set the miss chance of punch stored in the intrinsicWeapon attribute of Zombie to 100 and remove PickUpItemBehaviour from the behaviours array. 3. dropLeg(): decrement leg attribute by 1 and create and drop a leg which is of Limb type onto the ground at the position of the actor who is attacking the zombie (adjacent to zombie's location). Then check if leg>0, set the slowMotion flag to True. Else, remove all behaviours which involve moving location (WanderBehaviour and HuntBehaviour) from the behaviours array. - Override method: 1. playTurn(Actions actions, Action lastAction, GameMap map, Display display): this method will first check if the slowMotion attribute is true. If yes, check if Behaviour objects in MOVING_BEHAVIOUR exist in the behaviours array, remove them from the behaviours array. Otherwise if they do not exist, add them into the behaviours array. This method will then operate in the exact same way as original code, which is to loop through the behaviours array and return an action at the end.	

Rising from the dead

Functionality: Human dies and revives as Zombie after 5-10 turns

New Classes	Class Responsibility	Design Rationale
HumanCorpse extends Ground	Represents the human corpse. Each human corpse is represented by the character 'C' on the map. - Constructor parameter: 1. deadHumanName: the name of the dead Human, to be used when the corpse revives as Zombie - Add new **private** attributes: 1. revive_turn: stores the number of turns it takes for a HumanCorpse to revive, the number is a random number generated at the range of 5 to 10. 2. turn: keeps track of the number of turns that the HumanCorpse has been created 3. name: the deadHumanName being passed in the constructor - Override methods: 1. tick(): Increments turn every time it is being called. If turn value has reached revive_turn, it	In our game design, we treat human corpses as an obstacle on the ground instead of an item. This decision is made due to several reasons: 1. The engine code was implemented such that each Location can have many items but can have only one Actor at it. If a HumanCorpse revives when another Actor is standing on the same location, the game will crash. Thus, we decide to make HumanCorpse a Ground object that does not allow any Actor to enter. 2. In our design, HumanCorpse is not portable and will not interact with other Actors in the game. Therefore, it does not need most of the methods in Item class, e.g. getAllowableActions(), PickUpItemAction() and DropItemAction(). Hence, Ground is a more suitable parent class for HumanCorpse. 3. Another reason we did not make HumanCorpse a PortableItem is to prevent other actors from picking the corpse up, which might cause the corpse to revive and turn into Zombie in an actor's inventory.

	will call the revive() method 2. canActorEnter(): Override the same method in Ground class to return false (prevent other Actors from standing on the same location as the corpse). - Add new **private** method: 1. revive(): Impersonate the revival of a human corpse. It creates a new Zombie instance and places it at the corpse's location. Then, it removes the human corpse by setting the Ground type of that location to DIrt.	Furthermore, revive_turn, turn, name and revive() are declared as private, following the design principle: **minimize dependencies that cross encapsulation boundaries.**

Modified Class	Class Responsibility	Design Rationale
ZombieAttackAction	- Modified method: 1. execute(Actor, GameMap): Once the target(Human) is found unconcious, it will remove the Human and create a new HumanCorpse object, and place the corpse into the game map.	In our game design, if a Human is dead at a certain turn, then the human corpse will immediately be created and added to the map at the same turn.

Farmers and Food

Functionality: Farmer sow and fertilise crops

New Classes	Class Responsibility	Design Rationale
Plant extends Ground	An abstract class that represents all types of plants in the Zombie world. - Has 2 **private** attributes: 1. matureTime: an ArrayList that stores the number of turns left until the plant enters the next growth phase. The length of the list depends on the number of growth phases the plant has. 2. matureChar: an ArrayList that stores the characters used to represent the plant at different growth phases. Again, the length of the list depends on the number of growth phases the plant has. - Override method: 1. tick(Location location): Decrease all the values in matureTime by 1, indicating that there's now one less turn until the plant grows/ripes. Once the value in matureTime reaches 0, it means that the plant will grow/ripe at that turn, and hence the displayChar of the plant will be changed to the corresponding character stored in the matureChar.	This class is created because it is found that Crop and Tree are very similar, both of them are plants and have growth phases (Tree grows taller while Crop ripes). Therefore, we decide to create an abstract parent class for both Crop and Tree so that we only need to implement the common method once, with respect to the **Don't Repeat Yourself** design principle. Besides, all the attributes and methods are declared such that they follow the design principle: **minimize dependencies that cross encapsulation boundaries**. For example, both the new attributes and changeGrowthPhase() will only be used in the Plant class, hence they are declared as private. addGrowthPhase(int, char) has a default access modifier because it is a method that will be used by Tree and Crop to change the value of matureTime and matureChar. Since both Tree and Crop are in the same package as Plant, the method is not declared as public.

	- Add new methods with **default** and **private** access modifier: 1. addGrowthPhase(int mTime, char mChar): Takes in the mature time of a plant and the character that will be used to display the grown plant. The mature time will be added to matureTime while the character will be added to matureChar 2. changeGrowthStatus(): change the status of crop from UNRIPE to RIPE	
Crop extends Plant	Represents the crops in the game world. Crop can only be sowed/planted by Farmer, on Dirt. Crop will grow over time, and it grows faster when it is being fertilised.Once a Crop is ripe, it can be harvested by either Farmer or Player to become Food. - Constructor: 1. Use parent class's constructor 2. The initial displayChar is 'x' 3. Add one growth phases: 20, 'X' - Has capabilities: 1. UNRIPE: indicates that the Crop object is still unripe 2. RIPE: indicates that the Crop object is ripe 3. FERTILISE: indicates that Crop object can be	In the new design, Tree and Crop share common methods, therefore following the **Don't Repeat Yourself** principle, an abstract class Plant is created and both Tree and Crop become the subclass of Plant. With this design, we only need to implement the growing method once in the Plant class. Besides, the method fertilise() will be called in the class FertiliseAction. Since the two classes are in the same package, the method fertilise() is declared using the default access modifier instead of public. This follows the design principle: **minimize dependencies that cross encapsulation boundaries.**

	fertilised - Add new method with the **default** access modifier: 1. fertilise(): Reduce the matureTime of the Crop object by 10. If a negative value is obtained, the Crop matures immediately.	
Farmer extends Human	Represents the Farmer in the gameworld. A farmer can sow, fertilise and harvest crops. - Capabilities: 1. SOW: indicates that Farmer can sow crops 2. FERTILISE: indicates that Farmer can fertilise crops - Modifies parent class's attribute: 1. behaviours: add SowingBehaviour, FertiliseBehaviour into the behaviour list	It is specified very clearly that Farmer shares the same characteristics and abilities as a Human, and can do other things that Human cannot do, i.e. sow, fertilise. Hence, we decide to make Farmer inherits the Human Class. In our game design, the Farmer cannot multitask, that is, a farmer can either sow, fertilise, harvest, attack or wander at one turn, but cannot do any two of these actions simultaneously at the same turn.
SowingBehaviour implements Behaviour	Represents the farmer's sowing behaviour. - Has a constant: 1. SOW_PROBABILITY = 33 - Override method: 1. getAction(Actor actor, GameMap map): Determines whether the Farmer will sow at that turn using SOW_PROBABILITY. If the Farmer will sow, then the method will check if there is any Dirt at the surrounding of the Farmer by checking if the ground at a location has ZombieCapability.SOW. If there is one, it will	This class is created because it represents a behaviour that is not modelled in the original system. Besides, following the **Single Responsibility Principle**, all farmer behaviours are not merged into one single Behaviour class. With this design, if we need to modify the farmer's sowing behaviour in the future, we will only need to modify this class. The constant SOW_PROBABILITY is introduced to **reduce implicit dependency on literals**. In our game design, a farmer can only sow a crop at one turn. Using ZombieCapability.SOW, we also limit

	get the x and y coordinates of the location using x() and y() method of Location class. Then, it will create and return a SowingAction by passing the x and y values into the constructor. Else, it will return null.	that each Dirt location can have at most one crop only (because Crop does not have the capability of SOW, and hence in the next turn, the location that has a Crop planted will not be a possible location for Farmer to plant another crop).
SowingAction extends Action	Represents the farmer's sowing action. - Add new **private** attributes: 1. int x: x coordinate of the ground to be sowed 2. int y: y coordinate of the ground to be sowed - Constructor: 1. SowingAction(int x, int y): take in the x and y coordinates of the ground to be sowed - Override method: 1. execute(Actor actor, GameMap map):This method will create a new Crop object and add it to the location at x and y coordinates. It will then return a String that says the Farmer has sowed a Crop at location x,y.	Instead of storing the Location object to be sowed, we store the x and y coordinates of the location only. The Location object can later be retrieved using the at(int x, int y) method in GameMap. This avoids the dependency between Location and SowingAction class, thus fulfilling the principle of **reducing dependencies as much as possible.** This class is created because it represents an action that is not modelled in the original system. Following the **Single Responsibility Principle**, this sowing action is not merged into other action classes. Besides, sowingLocation will be declared as private, following the design principle: **minimize dependencies that cross encapsulation boundaries.**
FertiliseBehaviour implements Behaviour	Represents the farmer's fertilising behaviour. - Override method: 1. getAction(Actor actor, GameMap map): This method will check if the Farmer is standing on a location that has an unripe crop (it checks whether the ground has the capability of ZombieCapability.FERTILISE and ZombieCapability.UNRIPE). If there is one, it will create and return a FertiliseAction. Else, it	This class is created because it represents a behaviour that is not modelled in the original system. Besides, following the **Single Responsibility Principle**, all farmer behaviours are not merged into one single Behaviour class. With this design, if we need to modify the farmer's fertilising behaviour in the future, we will only need to modify this class. With our implementation for sowing, there will only be one crop at one location at a time. Therefore, at one

	returns null.	location, there will only be at most one Crop for a Farmer to fertilise.
FertiliseAction extends Action	Represents the farmer's fertilising action. - Override method: 1. execute(Actor actor, GameMap map): This method will first identify the Crop object at the Farmer's location. Then, it will call fertilise() of the Crop object and return a String stating that the Farmer has fertilised the Crop at location x,y.	This class is created because it represents an action that is not modelled in the original system. Following the **Single Responsibility Principle**, this fertilising action is not merged into other action classes. Since we ensure that each location has only one Crop, we can identify the Crop to be fertilised without much difficulties.

Modified Classes	Class Responsibility	Design Rationale
ZombieCapability	Stores all the enumeration values that are used in the Zombie world. - Add new enum: 1. UNRIPE: Object with this capability is unripe(Crop) 2. RIPE: Object with this capability is ripe (Crop) 3. SOW: Object with this capability is either sowable (Dirt) or carries out the sowing action (Farmer) 4. FERTILISE: Object with this capability can either be fertilised (Crop) or carries out the fertilise action (Farmer)	These new enumeration values are added so that we can use them to check whether a Crop is ripe or unripe, whether the Ground at a certain Location can be sowed or fertilised and whether the actor has the ability to carry out certain actions, i.e. sow and fertilise.
Dirt extends Ground	Represents bare dirt in the game. Crop can be planted on Dirt. - Has Capability: 1. SOW: indicates that Dirt is sowable	Among all the Ground, only Dirt has a capability of ZombieCapability.SOW. The capability is added so that we can determine if a certain location is Dirt and hence decides whether the Farmer can sow on the location or not.

Tree extends Plant	Implements a tree in the game world. The tree will grow as time passes. - Constructor: 1. Use parent class's constructor 2. The initial displayChar is '+' 3. Add two growth phases: 10, 't' and 20, 'T' - Removed private attribute: 1. age - Removed method: 1. tick(Location location)	Age and tick() are removed because both Crop and Tree have similar method/behaviour (they grow). Hence, we decide to implement the growing method in the Plant class and make Tree inherits Plant, following the **Don't Repeat Yourself** principle.

Functionality: Farmer and Player harvest crop

New Classes	Class Responsibility	Design Rationale
HarvestBehaviour implements Behaviour	Represents the farmer's harvest behaviour. - Override method: 1. getAction(Actor actor, GameMap map): This method has a local ArrayList object named possibleLocations, which contains the actor's current location as well as all the surrounding locations. These are the locations where ripe crop might be located. Then, the locations in the list will be looped through and check if it contains a ripe crop, by checking whether the ground of the location has the capability of HARVEST and RIPE. If there is one, it will get the x and y coordinates of the location using x() and y() method of Location class. Then, it will create and return a HarvestAction by passing	This class is created because it represents a behaviour that is not modelled in the original system. Besides, following the **Single Responsibility Principle**, all farmer behaviours are not merged into one single Behaviour class. With this design, if we need to modify the harvest behaviour in the future, we will only need to modify this class. In our game design, each location will only have at most one Crop. Therefore, at one turn, at one location, Farmer or Player can only harvest a Crop.

	the x and y int value into the class. Else, it will return null.	
HarvestAction extends Action	Represents the farmer's harvest action. **-** Add new **private** attributes: 1. int x: x coordinate of the ground which has ripe crop to be harvested 2. int y: y coordinate of the ground which has ripe crop to be harvested - Constructor: 1. HarvestAction(int x, int y): assign the x and y coordinates of the ground which has ripe crop to be harvested - Override method: 1. execute(Actor actor, GameMap map): This method will first create a new Food instance. Then, it will determine whether the Actor is a Farmer or the Player by using the ZombieCapability.SOW (only Farmer has this capability). If it is Farmer, then the Food object will be added to the location at the x and y coordinates. If it is a Player, then the Food object will be added into the Player's inventory.	Instead of storing the Location object which has ripe crop to be harvested, we store the x and y coordinates of the location only. The Location object can later be retrieved using the at(int x, int y) method in GameMap. This avoids the dependency between Location and HarvestAction class, thus fulfilling the principle of **reducing dependencies as much as possible.** This class is created because it represents an action that is not modelled in the original system. Following the **Single Responsibility Principle**, this harvesting action is not merged into other action classes. In our game design, HarvestAction will only be created for either Farmer or Player. This is because only Farmer has the HarvestBehaviour and only Player will make use of allowable actions to decide the action being carried out at a certain turn.
Crop extends Plant	Represents crops in the game. - Add capability: 1. HARVEST: indicates that Crop object can be harvested	Needs to override allowableActions() so that the action harvest crop will be within the list of possible actions that Player can carry out.

	- Override method: 1. allowableActions(Actor actor, Location location, String direction): This method will first check if the actor has the capability to harvest a crop by looking at ZombieCapability.HARVEST.. Then, it checks if the Crop object is ripe or unripe by checking its ZombieCapability. If the crop is ripe, it will create an Actions() instance with HarvestAction in it. Else, it creates and returns an Actions() instance without any action in it.	
Farmer extends Human	Represents farmers in the game. - Add capability: 1. HARVEST: indicates that Farmer can harvest crops - Modifies parent's class attribute: 1. behaviours: Add HarvestBehaviour into the behaviours list as well	HarvestBehaviour is added into the Farmer's behaviour list so that this behaviour will be looped through when the Farmer's playTurn() is called, and hence the Farmer will be able to carry out the harvest action.
Food extends PortableItem	Represents food that contains nutrients in the game. - Has a private attribute: 1. Int HEAL_POINT: a constant integer which represents the heal point of the food which will be added to Human's or Player's hitPoints when they eat the Food - Constructor: 1. Food(): call super("Food", "^") to create a PortableItem which represent the food	As Food has similar methods with PortableItem, we make Food a subclass of PortableItem, to reduce the number of repeated code, thus fulfilling the principle **Don't repeat yourself.**

Modified Classes	Class Responsibility	Design Rationale
ZombieCapability	Stores all the enumeration values that are used in the Zombie world. - Add new enum: 1. HARVEST: Object with this capability can be harvested (Crop) or can carry out the harvest action (Farmer, Player).	This new enumeration value is added so that we can use them to check whether a ground can be harvested and whether an actor can carry out the harvest action.
Player	Represents the player of the game (the user's avatar). - Add capability: 1. HARVEST: indicates that Player can harvest crops too	HARVEST is added as a capability of Player so that the HarvestAction will be appearing within the list of possible actions, and hence allowing the Player to choose to harvest for food.

Functionality: Human and Player eat food

New Classes	Class Responsibility	Design Rationale
Food extends PortableItem	Represents food that has nutrients in the game. - Override methods: 1. getAllowableActions(): return a list which consists of EatingAction	The getAllowableActions() method is overridden so that if the Food appears in the Player inventory, it will return an EatingAction which will then be added into a list of possible actions that the Player can take and hence allows the Player to choose an action to eat the food.

		In order to **minimize dependencies that cross encapsulation boundaries**, we declare the healPoint to be private to HumanAttackAction class.
EatingBehaviour implements Behaviour	A new class which implements Behaviour interface and generates an EatingAction if the current Actor is standing in a location which has a food. - Modified method: 1. getAction(Actor, GameMap): returns an EatingAction if the current Actor has hit point of value less than half of his maximum hit points and is standing in a location which has a food, else return null.	This class is created because it represents a behaviour that is not modelled in the original system. Following the **Single Responsibility Principle**, this eating behaviour is not merged into other behaviour classes. This implementation is based on the concept of **Polymorphism**, which allows the playTurn() method in Zombie class to loop through the behaviours array and call the getAction() method of each subclass of Behaviour interface in the array to check if the specific action represented by the subclass is possible to be performed by the Zombie in its current turn. In our game design, we have made it such that the Human will only eat Food if and only if their health points drop by half and they encounter a Food. Since their health is in a quite critical condition, we allow Human to directly eat the Food without picking up. However, in the case of the Player, which is controlled by the user, the Player can decide the time he eats a Food (even at the time when the health point is not critical). Therefore, we have made it such that a Player can only eat a Food that is in his inventory. In other words, the Player must pick up Food before eating.
EatingAction extends Action	A new class which represents an action to eat the food.	This class is created because it represents an action that is not modelled in the original system. Following the **Single Responsibility Principle**, this eating action is

	- Add new **private** attribute: 1. Item food: an item which represents the food - Override method: 1. execute(Actor actor, GameMap map): if the actor is a Player, remove the item food from the Player inventory and call the heal method in the Player. Otherwise, the actor is a Human, remove food from the location of the map and call the heal method in the Human.	not merged into other action classes. Based on the principle of **Don't Repeat Yourself**, we make the EatingAction to inherit from the Action class as both have similar methods that can be reused or overridden and by doing so, we can reduce the number of duplicated code.

Modified Classes	Class Responsibility	Design Rationale
Player	A class which extends Human and represents the Player - Modified method: 1. playTurn(Actions actions, Action lastAction, GameMap map): filter the actions list to remove the EatingAction for food items on ground. Then, if there is multi-turn action found, return the action, else return a menu to be shown to the player.	In order to only allow the Player to have the option to only eat food that appears in his inventory, If there is any Food object at the Player's location, he has to pick the Food up first and only in the next round he can choose to eat the Food or not. Thus, the EatingAction returned by the Food on the ground has to be removed from the actions list before passing the option menu to the player.
ActorInterface	An interface which provides the ability to add methods to Actor, without modifying code in the engine, or downcasting references in the game. - Add new method declarations: 1. maxHitPoints() 1. hitPoints()	Based on the **Single Responsibility Principle**, we added new methods maxHitPoints() and hitPoints() in ActorInterface to make Actor to be able to return its own maximum hit point and current hit point without changing code in other classes in the engine.
ZombieActor	A base class for Actors in the Zombie World	Based on the **Don't Repeat Yourself** principle, we override the maxHitPoints() and hitPoints() methods in

	- Override method: 1. maxHitPoints(): return the maxHitPoints of the actor 1. hitPoints(): return the current hitPoints of the actor	ZombieActor abstract class so that this method can be used by all the subclasses of ZombieActor and thus reduce the number of duplicated code.

A Summary for ZombieCapability

Modified Classes	Class Responsibility	Design Rationale
ZombieCapability	Stores all the enumeration values that are used in the Zombie world. - Add new enum: 1. CRAFTABLE_INTO_CLUB: Object with this capability can be crafted into club (Limb: arm) 2. CRAFTABLE_INTO_MACE: Object with this capability can be crafted into mace (Limb: leg) 3. UNRIPE: Object with this capability is unripe (Crop) 4. RIPE: Object with this capability is ripe (Crop) 5. SOW: Object with this capability is either sowable (Dirt) or carries out the sowing action (Farmer) 6. FERTILISE: Object with this capability can either be fertilised (Crop) or carries out the fertilise action (Farmer) 7. HARVEST: Object with this capability can be harvested (Crop) or can carry out the harvest action (Farmer, Player).	All the constants stored in ZombieCapability act as tags which are used to identify specific Objects throughout the game system. Thus, can be used to compare and determine actions to be performed onto the specific objects and **reduces the implicit dependency on literals.**