# 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.<br><br>- Has 2 new **private** attributes:<br>1. healPoint: the heal point that a Zombie will receive when it launches a successful attack (punch:0, bite:5).<br><br>2. missChance: the chance for a Zombie to miss an attack when it uses the zombie intrinsic weapon (punch:40% miss, bite:60% miss).<br><br>**UPDATED (for A2):**<br>- Add new methods:<br>1. healPoint(): Getter for the health point that Zombie earns from a successful attack.<br>2. missChance(): Getter for the probability for Zombie to miss an attack using that intrinsic weapon.<br>3. getVerb(): Getter for verb that describes how the Zombie use the intrinsic weapon<br>4. doubleMissChance(): Double the miss chance of the intrinsic weapon. | 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.<br><br>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. | This class is introduced because it is found that the Zombie and Human attack differently and the effects of |

| | | |
|---|---|---|
| | **UPDATED (for A2):**<br>- Add new **protected** methods:<br>1. missAttack(Weapon weapon): This method first determines the weapon the Zombie will use for attack, based on the weapon type, it will determine the chance for Zombie to miss. Then, a random decimal value will be generated and if this value falls into the space where the attack misses, it will return true. Else, the method returns false.<br><br>2. attackSuccess(int damage, Actor actor, Weapon weaponUsed, GameMap map): The Zombie's target, Human, is hurt (health point deducted). If the weapon used is "bite" then the health point of Zombie will be restored by 5 points. | their attacks are different as well.<br><br>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. |

| Modified Classes | Class Responsibility | Design Rationale |
|---|---|---|
| Zombie | Represents the Zombie creature in the game.<br><br>- Has 1 new **private** attribute:<br>1. intrinsicWeapons: a List that stores all the ZombieIntrinsicWeapon instances.<br><br>**UPDATED (for A2):**<br>- Modified constructor:<br>1. Zombie(String name): create and add ZombieIntrinsicWeapon into Zombie's private attribute, intrinsicWeapons.. | 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.<br><br>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%, the implementation used is to randomly choose an index that is within the index range |

| | | |
|---|---|---|
| | - Modified method:<br>1. getIntrinsicWeapon(): randomly chooses an intrinsic weapon from the list intrinsicWeapons and returns the intrinsic weapon. | of the list intrinsicWeapons. Then, the ZombieIntrinsicWeapon that locates at that random index will be returned.<br>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<br><br>- 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.<br><br>- Retains menuDescription(Actor actor) but execute(Actor actor, GameMap map) is being moved to its subclasses<br><br>- This class will be useful in the case where we want to refactor code in ZombieAttackAction and HumanAttackAction<br><br>**UPDATED (for A2):**<br>- Add new **protected** attributes:<br>1. result: Store the String that describes the attack outcome.<br><br>2. rand: a Random object which will be used to | 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.<br>The AttackAction class is made abstract and not deleted to prevent other classes that rely on it from failing.<br><br>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. |

| | | |
|---|---|---|
| | generate random probability.<br><br>- Add new **abstract** methods:<br>1. missAttack(weapon Weapon): determine if the attack at that turn will miss or successful based on a random generated value<br><br>2. attackSuccess(int damage, Actor actor, Weapon weaponUsed, GameMap map): Operations to be carried out when the attack is successful.<br><br>3. killTarget(GameMap map): Create and add corpse to map when the target is killed<br><br>- Override method:<br>1. execute(Actor actor, GameMap map): attacks the target, checks if the target loses consciousness and carries out corresponding course of actions.<br><br>2. menuDescription(Actor actor): returns a string that describes the output of the attack, e.g. “Human attacks Zombie”<br><br>- Add new **default** methods:<br>1. dropCorpseItems(GameMap map): Drops all items in the corpse’s inventory onto the map, at the location of the corpse | |
| AttackBehaviour | Represents the attack behaviour of both Zombie and | **UPDATED (for A2)** |

| | | |
|---|---|---|
| | Human.<br><br>- Modified method:<br>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.<br><br>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 the target near the actor has the capability ZombieCapability.UNDEAD, this means that the actor is a Human and target is a Zombie, hence a HumanAttackAction instance will be created and returned.<br><br>If the target near actor has the capability ZombieCapability.ALIVE, this means that the actor is a Zombie and the target is a HUMAN, thus a ZombieAttackAction instance will be created and returned. |
| ZombieActor | Represents the actors in the Zombie world.<br><br>- Modified method:<br>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.<br><br>If the ZombieActor has the capability of ALIVE and otherActor has the capability of UNDEAD, then a ZombieAttackAction instance will be created and returned.<br><br>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.<br><br>- Modified method:<br>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.<br><br>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.<br><br>**UPDATED (for A2)**<br>- Has 1 **private** constant attribute:<br>1. GROWL_CHANCE: a constant that stores the probability of Zombie to growl, that is, 10<br><br>2. rand: A Random object which is used to generate random probability<br><br>- Override method: | **UPDATED (for A2)**<br>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, GROWL_CHANCE is being declared as a constant and used in comparison instead of hardcoding the value 10 to **reduce implicit dependency on literals**.<br><br>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 |

| | | |
|---|---|---|
| | 1. getAction(Actor actor, GameMap map): ensures that the Zombie has a 10% chance of growling at every turn. A random decimal value will be generated from the range 0.0 to 1.0, and this value will be multiplied by 100. If the result is less than GROWL_CHANCE, the Zombie will growl, so a GrowlAction instance will be returned. | attack any Human at that turn. |
| GrowlAction extends Action | Represents the growling action of Zombie.<br><br>- Override methods:<br>1. menuDescription(Actor actor): returns a string that impersonate a zombie growling, e.g. “zombie says Braaains”.<br><br>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 | - Constructor:<br>1. Add GrowlBehaviour into behaviours, behaviours is a Behaviours object that has been initialized in the ZombieActor class | The GrowlBehaviour instance is being added into the behaviours 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 |
|---|---|---|
| **UPDATED (for A2)** | A thin wrapper for java.util.ArrayList<Behaviour> that | All the methods related to handling an ArrayList of |

| | | |
|---|---|---|
| Behaviours implements Iterable<Behaviour> | does not allow nulls to be added.<br><br>- Add 1 **private** attribute:<br>1. behaviours: An ArrayList of Behaviour objects<br><br>- Constructors:<br>1. Behaviours(): Constructs an empty list.<br>2. Behaviours(List<Behaviour> behaviours): Constructs using a list of (non-null) Behaviour.<br><br>- Add **public** methods:<br>1. add(Behaviours behaviours): Appends the contents of another Behaviours list to this one.<br>2. add(List<Behaviour> behaviours): Appends the contents of any List<Behaviour> to this one.<br><br>3. add(Behaviour behaviour): Appends a single Behaviour to this collection, if it is non-null. If it is null, then it is ignored.<br><br>4. add(Behaviour behaviour, int index): Add a new Behaviour to the collection at a specified position if it is not null. If it is null, then it is ignored.<br><br>5. Iterator<Behaviour> iterator(): Returns an Iterator for the underlying collection.<br><br>6. remove(Behaviour behaviour): Remove the first occurrence of a Behaviour from the collection, if it is present. If it is not present, the list is unchanged.<br><br>7. remove(Behaviours behaviours): Remove the | Behaviour objects are encapsulated in this Behaviours class. This provides a simple interface to deal with the ArrayList of Behaviour objects and reduce the number of duplicated code, thus fulfilling the **Don't Repeat Yourself** principle. |

| | | |
|---|---|---|
| | first occurrence of each Behaviour in behaviours from the collection, if it is present. If it is not present, the list is unchanged.<br><br>8. contains(Behaviour behaviour): Check if the same instance class of behaviour exists in behaviours from the collection.<br><br>9. hasBehaviour(Behaviours behaviours): Check if at least one of the behaviour in the behaviours arguments exists in behaviours from the collection. | |

| Modified Classes | Class Responsibility | Design Rationale |
|---|---|---|
| ActorInterface | An interface for Actor<br><br>**UPDATED (for A2)**<br>- Added method:<br>1. takeDamage(int points, Actor otherActor, GameMap map): Do some damage to the actor. | Based on principle to reduce **dependencies as much as possible**, a hurt method is added to ActorInterface to be overridden in Zombie class to implement a more advanced hurt method and avoid downcasting in HumanAttackAction class. |
| ZombieActor | Base class for Actors in the Zombie World<br><br>**UPDATED (for A2)**<br>- Added **protected** attribute:<br>1. behaviours: a Behaviours object which stores a list of Behaviour that can be performed by the actor<br><br>- Added method:<br>1. playTurn(Actions actions, Action lastAction, | Based on the **minimize dependencies that cross encapsulation boundaries principle,** the access modifier of behaviours attribute is protected so that only its subclasses can access and make changes to it.<br><br>Based on the **Don't Repeat Yourself** principle, playTurn method is moved to this class so that its subclasses such as Zombie and Human do not have to implement duplicate code. |

| | | |
|---|---|---|
| | GameMap map, Display display): Return an action which can be performed by the actor in the actor's current turn.<br><br>2. takeDamage(int points, Actor otherActor, GameMap map): Do some damage to the ZombieActor.<br><br>- Override method:<br>1. getAllowableActions(Actor otherActor, String direction, GameMap map): return an Actions object consisting of HumanAttackAction if the current actor is Zombie and the other actor is Human. If the current actor is Human and other actor is Zombie, return an Actions object consisting of  ZombieAttackAction. | |
| Human | Class representing an ordinary human<br><br>**UPDATED (for A2)**<br>- Removed behaviours attribute and playTurn method from Human class<br><br>- Modified constructor:<br>1. Human(String name): Creates a Human instance object using super() constructor and calls addHumanBehaviours().<br><br>2. Human(String name, char displayChar, int hitPoints): This constructor is used to create the subtypes of Human. addHumanBehaviours() is also called. | **UPDATED (for A2)**<br>The behaviours and playTurn method are moved to ZombieActor class and can be inherited in Human class, thus removing the duplicated code and fulfilling the **Don't Repeat Yourself** principle.<br><br>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. |

| | | |
|---|---|---|
| | - Add **private** & **protected** methods:<br>1. addHumanBehaviours(): Add a Behaviours object consist of a list of Behaviour (AttackBehaviour, EatingBehaviour and WanderBehaviour) to protected behaviours attribute in its parent class which is the ZombieActor class.<br><br>2. extendHumanBehaviours(int index, List<Behaviour> newBehaviours): Enables the subclasses of Human that have other behaviours to add more behaviours into the behaviours attribute. | |

**Functionality: Knock zombie's limb off**

| New Classes | Class Responsibility | Design Rationale |
|---|---|---|
| HumanAttackAction extends AttackAction | A class that represents an action for attacking Zombie<br><br>**UPDATED (for A2)**<br><br>- Add new **protected** method:<br>1. missAttack(Weapon weapon): Return Boolean to indicate if the actor misses the target based on 50% chance.<br><br>2. attackSuccess(int damage, Actor actor, Weapon weaponUsed, GameMap map): If the attack is successful, cause damage to the Zombie based on the damage point of the weapon, update the result description 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. |

| | | |
|---|---|---|
| | 3. killTarget(GameMap map): Human successfully kills his target, Zombie. The Zombie becomes a corpse. | |
| Limb extends PortableItem | A new class which represents the arm or leg knocked off from the zombie.<br>- Constructor:<br>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.<br><br>- Has capabilities:<br>1. CRAFTABLE_INTO_CLUB: added for limbs that represent arms<br>2. CRAFTABLE_INTO_MACE: added for limbs that represent legs<br><br>**UPDATED (for A2)**<br>- Override method:<br>1. getAllowableActions(): return an unmodifiable list of Action which consist of CraftWeaponAction(this) | To make Limb able to be picked up, we make Limb to become a subclass of PortableItem which is a subclass 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.<br><br>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.<br>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. |

| | | |
|---|---|---|
| Club extends WeaponItem | A new weapon crafted from a zombie arm.<br><br>- Constructor:<br>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.**<br><br>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 a zombie leg.<br><br>- Constructor:<br>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.**<br><br>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.<br><br>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. |

| New Classes | Class Responsibility | Design Rationale |
|---|---|---|
| ActionInterface<br>**UPDATED (for A2)** | This interface provides the ability to add methods to Action, without modifying code in the engine, or downcasting references in the game.<br><br>- Added **default** methods:<br>1. hasCapability(Enum<?> capability): Check if this Action have the given Capability. By default, return false for Action having the given Capability.<br><br>2. addCapability(Enum<?> capability): Add a Capability to this Action.<br><br>3. removeCapability(Enum<?> capability): Remove a Capability from this Action. | Default methods are added into ActionInterface to allow capabilities to be added into some action classes to be used to identify the action by comparing the constant tag instead of comparing the class type of action. |
| CraftWeaponAction extends Action | A new class which represents an action for crafting zombie arm or leg into club or mace.<br><br>**UPDATED (for A2)**<br><br>- Add new **private** attribute:<br>1. Item: item to be crafted<br>2. Capabilities: capability of the craftWeaponAction<br><br>- Constructor:<br>1. CraftWeaponAction(Item item): Create a CraftWeaponAction with the specified item to be crafted and add | 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.<br><br>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.<br><br>As explained in Limb class, CraftWeaponAction is now made available for the Player. However, it can be |

| | | |
|---|---|---|
| | ZombieCapability.CRAFT_WEAPON_ACTION to the capabilities of the action.<br><br>- Override method:<br>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.<br><br>2. menuDescription(Actor actor): Describe the craft weapon action in a format suitable for displaying in the menu.<br><br>3. hasCapability(Enum<?> capability): Check if this Action has the given Capability.<br><br>4. addCapability(Enum<?> capability): Add a Capability to this Action.<br><br>5. removeCapability(Enum<?> capability): Remove a Capability from this Action. | 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.<br><br>**UPDATED (for A2)**<br>CRAFT_WEAPON_ACTION is added so that we can determine if the action is CraftWeaponAction using the constant instead of checking the class type of the action by using instanceof. |

| Modified Class | Class Responsibility | Design Rationale |
|---|---|---|

| | | |
|---|---|---|
| ZombieCapability | Stores all the enumeration values that are used in the Zombie world.<br><br>**UPDATED (for A2)**<br>- Add enum:<br>1. CRAFTABLE_INTO_CLUB: indicates that the object can be crafted into club (Zombie limb: arm)<br>2. CRAFTABLE_INTO_MACE: indicates that the object can be crafted into mace (Zombie limb: leg)<br>3. CRAFT_WEAPON_ACTION: indicates that the action is a CraftWeaponAction | These two capabilities are added so that we can determine if the Zombie limb picked up can be crafted into club or mace.<br><br>**UPDATED (for A2)**<br>CRAFT_WEAPON_ACTION is added so that we can determine if the action is CraftWeaponAction using the constant instead of checking the class type of the action by using instanceof. |
| Zombie | Class which represents the zombie.<br>**UPDATED (for A2)**<br>- Removed behaviours attribute<br>- Has new **private** attributes:<br>1. int arm: The number of arms of the zombie, which is 2, as specified in the specification rule<br>2. int leg: The number of leg of the zombie, which is 2, as specified in the specification rule<br>3. boolean slowMotion: Indicates if the Zombie is in slow motion (only have one leg). It is initially set as false.<br>4. Behaviours MOVING_BEHAVIOUR: a constant Behaviours object which consists of a list of Behaviour objects which will change location of zombie | To **reduce implicit dependency on literals** and **avoid excessive use of literals,** the constant Behaviours object 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.<br><br>In order to **minimize dependencies that cross encapsulation boundaries**, we declare all of the variables and some methods to be private to the class.<br><br>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. |

| | | |
|---|---|---|
| | 5. DROP_LIMB_CHANCE: a constant integer which represents the probability of zombie dropping one of its limbs from an attack, by default is 25<br><br>6. DROP_WEAPON_CHANCE: a constant integer which represents the probability of zombie dropping one of its weapons when dropping one of its arms, by default is 50<br><br>- Constructor:<br>1. Zombie(String name): Creates a Zombie object using super() constructor and add a Behaviours object consist of a list of Behaviour (AttackBehaviour, PickUpItemBehaviour, HuntBehaviour, GrowlBehaviour and WanderBehaviour)  to protected behaviours attribute in its parent class which is the ZombieActor class.<br><br>- Add **public** method:<br>1. takeDamage(int points, Actor otherActor, GameMap map): Do some damage to the zombie and have a fixed 25% chance  to knock at least one of the zombie's limbs off<br><br>- Add **private** methods:<br>1. dropLimb(Actor actor, GameMap map): Randomly choose to drop one of the zombie's limbs if there is still any, else return without doing anything. Each limb has an equal probability of getting knocked off.<br><br>2. dropArm(Actor actor, GameMap map): Perform | |

| | | |
|---|---|---|
| | operations when one Zombie's arm is knocked off. This method decrements the value of arms attribute by one and creates a limb object which represents the arm at the attacking actor location. If the zombie only has one arm left, its probability of punching is halved and there is a 50% chance of the zombie dropping the weapon it is holding. If the zombie does not have any arm left, drop all of its weapons and it cannot punch anymore.<br><br>3. dropWeapon(Actor actor, GameMap map): Return an Actions consisting of all the dropping actions of weapons in the actor's inventory<br><br>4. dropFirstWeapon(Actor actor, GameMap map): Drop the first weapon in the zombie's inventory to the location of the attacking actor, if there is any.<br><br>5. dropAllWeapon(Actor actor, GameMap map): Drop all of the weapons in the zombie's inventory to the location of the attacking actor, if there is any.<br>6. dropLeg(): Perform operations when one Zombie's leg is knocked off. This method decrements the value of legs attribute by one, create a limb object which represents leg at the attacking actor location. If the zombie only has one leg left, set its slowMotion attribute to true to indicate that zombie can only move every second turn after this.If the zombie does not have any leg left, remove all moving behaviours from zombie behaviours attribute so that zombie | |

| | | |
|---|---|---|
| | cannot move anymore.<br><br>7. checkSlowMotion(): If the Zombie is in slow motion (lost one leg), make sure the zombie can only move every second turn.If slowMotion is true, check if the zombie behaviours attribute has any behaviours which involve moving its location. If yes, means in previous turn, zombie might probably moved, thus removing all moving behaviours from the behaviour attributes for current zombie turn. If no, means the zombie did not move in the previous turn, thus adding moving behaviour into the behaviour attributes for current zombie turn. By doing so, we can make sure the zombie can only move in every second turn.<br><br>- Override method:<br>1. playTurn(Actions actions, Action lastAction, GameMap map, Display display): Check if zombie is in slow motion (left one leg) and return an action which can be performed by the zombie in its current turn. | |

**Rising from the dead**

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

| New Classes | Class Responsibility | Design Rationale |
|---|---|---|
| **UPDATED (for A2)**<br>HumanCorpse extends PortableItem | Represents the human corpse. Each human corpse is represented by the character ‘C’ on the map.<br><br>- Add new **private** attributes:<br>1. revive_turn: stores the number of turns it takes | **UPDATED (for A2)**<br>In our game design, the human corpse is being treated as a portable item, which means, it can either stay on the ground, or be picked up by other actors who have the pick up ability/behaviour. Due to this, there are a |

| | | |
|---|---|---|
| | for a HumanCorpse to revive, the number is a random number generated at the range of 5 to 10.<br><br>2. turn: keeps track of the number of turns that the HumanCorpse has been created. It is initialized as 0.<br><br>3. MIN_TURN: a constant that stores the minimum number of turns needed for a human corpse to revive, which is 5.<br><br>4. MAX_TURN: a constant that stores the maximum number of turns needed for a human corpse to revive, which is 10.<br><br>- Override methods:<br>1. tick(Location location): This method ensures that HumanCorpse experiences the passage of time when it is on ground. The method increments turn every time it is being called. If the turn value has reached revive_turn, it will call the revive() method to create a new Zombie instance, then it will remove HumanCorpse from the map.<br><br>2. tick(Location location, Actor actor): This method ensures that HumanCorpse experiences the passage of time when it is in another actor's inventory. If turn value has reached revive_turn, it will call the revive() method to create a new Zombie instance, then it will remove HumanCorpse from the actor's inventory. | few issues that can arise at the turn where the human corpse revives, as illustrated below:<br><br>At the turn where human corpse revives,<br>1. There is an actor standing on the location where the human corpse lies. Since each location can only have one actor at a time, the revived corpse actor (Zombie) cannot be added to its current location.<br><br>2. The human corpse revives as Zombie when it is in another actor's inventory. Since by default, the actor can only drop items at the location he is standing on, if the actor chooses to drop the revived human corpse, there will be two actors at the same location at one time, and this will crash the game.<br><br>These issues are resolved in the revive() method. If there is an actor standing at the location where the revived human corpse (Zombie) will be added, then the method will find the nearest empty adjacent location and add the new Zombie instance at that location.<br><br>Furthermore, revive_turn, turn, name and revive() are declared as private, following the design principle: **minimize dependencies that cross encapsulation boundaries.**<br><br>Additionally, MIN_TURN and MAX_TURN are introduced to **reduce implicit dependency on literals** when a random number is generated between the range of MIN_TURN and MAX_TURN. |

| | | |
|---|---|---|
| | - Add new **private** method:<br>1. revive(): Impersonate the revival of a human corpse. It creates a new Zombie instance and places it at the corpse's location. If there is another actor standing on the corpse's location, then it will find the nearest adjacent position that does not have any actor to place the new Zombie instance. | |

| Modified Class | Class Responsibility | Design Rationale |
|---|---|---|
| ZombieAttackAction | **UPDATED (for A2):**<br>- Add new **protected** method:<br>1. killTarget(GameMap map): The target of Zombie is Human. Hence, this method will create a new HumanCorpse instance and add it onto the 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 |
|---|---|---|
| **UPDATED (for A2)**<br>Crop extends Ground | 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.<br><br>- Add new **private** attributes:<br>1. age: keeps track of the number of turn since | **UPDATED (for A2):**<br>The method ripe() will only be called in Crop. Therefore, the method ripe() is declared using the private access modifier.This follows the design principle: **minimize dependencies that cross encapsulation boundaries.**<br><br>In the game design, any new Crop will be unripe and can only be fertilised and not harvested. This explains |

| | | |
|---|---|---|
| | the Crop object has been created. Age is initialised as 0.<br><br>2. RIPE_TURN: a constant that stores the number of turns needed for crop to ripe, which is 20<br><br>**UPDATED (for A2):**<br>- Has capabilities:<br>1. UNRIPE: indicates that the Crop object is still unripe<br><br>2. FERTILISE: indicates that Crop object can be fertilised<br><br>- Add new method with the **private** access modifier:<br>1. ripe(): Changes the displayChar of Crop from ‘x’ to ‘X’. Changes the capability of Crop from UNRIPE to RIPE and from FERTILISE to HARVEST.<br>**UPDATED (for A2)**:<br>- Override method:<br>1. tick(Location location): Increments the age by 1 each time this method is called. When the age of Crop reaches RIPE_TURN, the corp ripes.<br><br>2. fertilise(): Increases the age of Crop by 10. If the new age is greater than or equal to RIPE_TURN, the crop ripes. | why the Crop only has the capability of UNRIPE and FERTILISE when it is first initialized. Once the crop becomes ripe, it can be harvested but cannot be fertilised anymore. |
| Farmer extends Human | Represents the Farmer in the gameworld. A farmer can sow, fertilise and harvest crops. | 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 |

| | | |
|---|---|---|
| | - Capabilities:<br>1. SOW: indicates that Farmer can sow crops<br>2. FERTILISE: indicates that Farmer can fertilise crops<br><br>- Add new **private** method:<br>1. addFarmerBehaviours(): extend the human behaviours with more behaviours that are only associated to Farmer, e.g. SowingBehaviour, FertiliseBehaviour. | Class.<br><br>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.<br><br>Besides, the method addFarmerBehaviours() is declared as private because it is only used in the Farmer class. This follows the principle: **minimize dependencies that cross encapsulation boundaries**. |
| SowingBehaviour implements Behaviour | Represents the farmer's sowing behaviour.<br>- Add new private attributes:<br>1. SOW_PROBABILITY: a constant that stores the probability for Farmer to sow crop at a turn, which is 33%<br><br>2. rand: a Random object that is used to generate a random probability<br><br>- Override method:<br>1. getAction(Actor actor, GameMap map): This method first 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 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 | 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.<br><br>The constant SOW_PROBABILITY is introduced to **reduce implicit dependency on literals**.<br><br>In our game design, a farmer can only sow a crop at one turn. Using ZombieCapability.SOW, we also limit 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). |

| | | |
|---|---|---|
| | constructor. Else, it will return null. | |
| SowingAction extends Action | Represents the farmer’s sowing action.<br>- Add new **private** attributes:<br>1. int x: x coordinate of the ground to be sowed<br>2. int y: y coordinate of the ground to be sowed<br><br>- Constructor:<br>1. SowingAction(int x, int y, GameMap map): take in the x and y coordinates of the ground to be sowed. Take in map to check if the x and y coordinate values are in range.<br><br>**UPDATED (for A2)**<br>- Override method:<br>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 call menuDescription(Actor actor) and returns the String.<br><br>2. menuDescription(Actor actor): This method returns 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.**<br><br>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.<br><br>Besides, x and y are declared as private, following the design principle: **minimize dependencies that cross encapsulation boundaries.** |
| FertiliseBehaviour implements Behaviour | Represents the farmer’s fertilising behaviour.<br><br>- Override method:<br>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 | 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 |

| | | |
|---|---|---|
| | 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 returns null. | only need to modify this class.<br><br>With our implementation for sowing, there will only be one crop at one location at a time. Therefore, at one location, there will only be at most one Crop for a Farmer to fertilise. |
| FertiliseAction extends Action | Represents the farmer's fertilising action.<br><br>**UPDATED (for A2)**<br>- Add new **private** attributes:<br>1. x: the x-coordinate of the location of unripe crop<br>2. y: the y-coordinate of the location of unripe crop<br><br>- Override method:<br>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. It will also call menuDescription(Actor actor) and return the String.<br><br>2. menuDescription(Actor actor): This method returns 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.<br><br>Since we ensure that each location has only one Crop, we can identify the Crop to be fertilised without much difficulties.<br><br>Besides, x and y are declared as private, following the design principle: **minimize dependencies that cross encapsulation boundaries.** |

| Modified Classes | Class Responsibility | Design Rationale |
|---|---|---|
| ZombieCapability | Stores all the enumeration values that are used in the Zombie world. | 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 |

| | | |
|---|---|---|
| | - Add new enum:<br>1. UNRIPE: Object with this capability is unripe(Crop)<br>2. RIPE: Object with this capability is ripe (Crop)<br>3. SOW: Object with this capability is either sowable (Dirt) or carries out the sowing action (Farmer)<br>4. FERTILISE: Object with this capability can either be fertilised (Crop) or carries out the fertilise action (Farmer) | or fertilised and whether the actor has the ability to carry out certain actions, i.e. sow and fertilise. |
| **UPDATED (for A2)**<br>GroundInterface | An interface for all Ground.<br><br>- Add default method:<br>1. fertilise(): Fertilise the ground. By default, this method does not have any implementation. | This method is added to remove downcasting from Ground to Crop in FertiliseAction. |
| Dirt extends Ground | Represents bare dirt in the game. Crop can be planted on Dirt.<br><br>- Has Capability:<br>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. |

**Functionality: Farmer and Player harvest crop**

| New Classes | Class Responsibility | Design Rationale |
|---|---|---|
| HarvestBehaviour implements Behaviour | Represents the farmer’s harvest behaviour.<br><br>- Override method:<br>1. getAction(Actor actor, GameMap map): This | 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 |

| | | |
|---|---|---|
| | 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. 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 the x and y int value into the class. Else, it will return null. | Behaviour class. With this design, if we need to modify the harvest behaviour in the future, we will only need to modify this class.<br><br>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. |
| HarvestAction extends Action | Represents the farmer's harvest action.<br><br>**-** Add new **private** attributes:<br>1. int x: x coordinate of the ground which has ripe crop to be harvested<br>2. int y: y coordinate of the ground which has ripe crop to be harvested<br><br>- Constructor:<br>1. HarvestAction(int x, int y, GameMap map): assign the x and y coordinates of the ground which has ripe crop  to be harvested. Map is taken in as a parameter to check whether the x and y coordinates are in range.<br><br>- Override method:<br>1. execute(Actor actor, GameMap map): This method will first create a new Food instance. | 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.**<br>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.<br><br>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. |

| | | |
|---|---|---|
| | 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. It also sets the ground of the harvested location to Dirt again as a means to remove Crop.<br><br>**UPDATED (for A2)**<br>2. menuDescription(Actor actor): This method returns a String that states that the Farmer/Player has harvested the crop at location (x, y). | |
| **UPDATED (for A2)**<br>Crop extends Ground | Represents crops in the game.<br><br>- Override method:<br>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. | Needs to override allowableActions() so that the HarvestAction will appear in the list of possible actions the Player can carry out when there is a ripe crop at the Player's surrounding. |
| Farmer extends Human | Represents farmers in the game.<br><br>- Add capability:<br>1. HARVEST: indicates that Farmer can harvest crops | 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. |

| | | |
|---|---|---|
| | - Add new **private** method:<br>1. addFarmerBehaviours(): extend the human behaviours with more behaviours that are only associated to Farmer, e.g. HarvestBehaviour | Besides, the method addFarmerBehaviours() is declared as private because it is only used in the Farmer class. This follows the principle: **minimize dependencies that cross encapsulation boundaries**. |
| Food extends PortableItem | Represents food that contains nutrients in the game.<br><br>**UPDATED (for A2):**<br>- Has a **private** attribute:<br>1. 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. Currently set as 10.<br><br>- Constructor:<br>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.<br><br>- Add new enum:<br>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. |
| **UPDATED (for A2)**<br>Player | Represents the player of the game (the user's avatar).<br><br>- Add capability:<br>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. |

| | | |
|---|---|---|
| | - Modified method:<br>1. filterActions(Actions actions, GameMap map): Add HarvestAction into the Player's Actions list if there is a ripe crop at the Player's location. | |

**Functionality: Human and Player eat food**

| New Classes | Class Responsibility | Design Rationale |
|---|---|---|
| **UPDATED (for A2)**<br>ItemInterface | An interface for all items<br><br>- Added **default** method:<br>1. getHealPoint(): Getter for heal point of the item. By default, it returns 0. | getHealPoint() is added to remove downcasting from Item to Food. |
| Food extends PortableItem | Represents food that has nutrients in the game.<br><br>**UPDATED (for A2)**<br>- Override methods:<br>1. getAllowableActions(): return an unmodifiable list of Action consisting of EatingAction<br><br>2. getHealPoint(): Getter for the heal point of the food. | 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.<br><br>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.<br><br>- Modified method: | 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. |

| | | |
|---|---|---|
| | 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 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.<br><br>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.<br><br>**UPDATED (for A2)**<br><br>- Add new **private** attribute:<br>1. Item food: an item which represents the food<br>2. Capabilities: capability of the EatingAction<br>- Constructor:<br>1. EatingAction(Item food):Create an EatingAction instance with the food arguments as Item attribute and add ZombieCapability.EATING_ACTION to the | 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 not merged into other action classes.<br><br>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.<br><br>EATING_ACTION is used to identify the EatingAction and avoid the use of instanceof to compare the class |

| | | |
|---|---|---|
| | capabilities of the action.<br><br>- Override method:<br>1. execute(Actor actor, GameMap map): The method will first check whether the Actor is a Player or a Human using the capability PLAYER. 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.<br><br>2. menuDescription(Actor actor): A string describing the eating action suitable for displaying in the UI menu.<br><br>3. hasCapability(Enum<?> capability): Check if this Action has the given Capability.<br><br>4. addCapability(Enum<?> capability): Add a Capability to this Action.<br><br>5. removeCapability(Enum<?> capability): Remove a Capability from this Action. | type of action. |
| ZombieCapability<br>**UPDATED (for A2)** | Stores all the enumeration values that are used in the Zombie world.<br><br>- Add enum:<br>1. EDIBLE: Indicate that the food  can be eaten by Player or Human with hit points less than half of its maximum hit points. | EDIBLE is added to act as a constant tag to  identify the food  throughout the game system. Thus, this **reduces the implicit dependency on literals.**<br><br>EATING_ACTION is added so that we can determine if the action is EatingAction using the constant tag instead of checking the class type of the action by using |

| | | |
|---|---|---|
| | 2. EATING_ACTION; Indicate that the action is an EatingAction | instanceof. |

| Modified Classes | Class Responsibility | Design Rationale |
|---|---|---|
| Player | A class which extends Human and represents the Player<br><br>**UPDATED (for A2)**<br>- Added Capability:<br>1. PLAYER: indicate that the actor is the player<br><br>- Added **private** method:<br>1. filterActions(Actions actions, GameMap map): Remove CraftWeaponItem and EatingAction for items on the ground by checking if the action has the ZombieCapability of CRAFT_WEAPON_ACTION or EATING_ACTION so that Player has to pick up the item from the ground before performing any of the two actions. Return a collection of possible Actions for the Player.<br><br>- Modified method:<br>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 | 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. |

| | | |
|---|---|---|
| | 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.<br><br>**UPDATED (for A2)**<br>- Add new method declarations:<br>1. maxHitPoints()<br>2. 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<br><br>- Override method:<br>1. maxHitPoints(): return the maxHitPoints of the actor<br>2. hitPoints(): return the current hitPoints of the actor | Based on the **Don't Repeat Yourself** principle, we override the maxHitPoints() and hitPoints() methods in 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.<br><br>- Add new enum:<br>1. CRAFTABLE_INTO_CLUB: Object with this capability can be crafted into club (Limb: arm)<br>2. CRAFTABLE_INTO_MACE: Object with this | 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.** |

| | | |
|---|---|---|
| | capability can be crafted into mace (Limb: leg)<br>3. UNRIPE: Object with this capability is unripe (Crop)<br>4. RIPE: Object with this capability is ripe (Crop)<br>5. SOW: Object with this capability is either sowable (Dirt) or carries out the sowing action (Farmer)<br>6. FERTILISE: Object with this capability can either be fertilised (Crop) or carries out the fertilise action (Farmer)<br>7. HARVEST: Object with this capability can be harvested (Crop) or can carry out the harvest action (Farmer, Player).<br><br>**UPDATED (for A2):**<br>8. EDIBLE: Object with this capability can be eaten by Player or Human with hit points less than half of its maximum hit points.<br>9. PLAYER: Object with this capability is the Player.<br>10. CRAFT_WEAPON_ACTION: Object with this capability is an instance of the CraftWeaponAction.<br>11. EATING_ACTION: Object with this capability is an instance of the EatingAction. | |