Design Rationale

Zombie Attacks

Zombies should be able to bite. Give the Zombie a bite attack as well, with a 50% probability of using this instead of their normal attack. The bite attack should have a lower chance of hitting than the punch attack, but do more damage – experiment with combinations of hit probability and damage that make the game fun and challenging. (You can experiment with the bite probability too, if you like.).

A **BiteAction** class that inherits **AttackAction** is created to execute a bite attack by a **Zombie**. It has will create an **InstrinsicWeapon** that will act as its teeth that will deal damage. A **Zombie** will have a 70% chance of missing its bite attack. The decision of using **AttackAction** or **BiteAction** is selected in the **AttackBehaviour.getAction(...)** method and is only available to zombies by checking if **Actor** have a **ZombieCapability.UNDEAD** capability.

Having the bite attack as a separate class allows the system to be easily extendable if more attack types needs to be added, following the Open/Closed Principle. To avoid code duplication, common blocks of code in the **AttackAction.execute(...)** method were refactored into package-private methods. **BiteAction** inherits these methods and uses them in its own execute method which avoids duplicated code and follows the DRY principle.

A successful bite attack restores 5 health points to the Zombie.
The health restore will be done in the BiteAction class after the bite attack has been executed. As the Actor object is already passed as a parameter in execute(...), the health restore can be simply done with actor.heal(5).

If there is a weapon at the Zombie's location when its turn starts, the Zombie should pick it up.
This means that the Zombie will use that weapon instead of its intrinsic punch attack (e.g. it might
"slash" or "hit" depending on the weapon).
A new class called ScanvengeBehaviour which implements Behaviour Is created. It will contain getAction(...) that inspects the items at the actor's location and picks up items that are a weapon. A ScanvengeBehaviour object will be the first element in the behaviours array in the Zombie class so that it will be executed at the start of its turn.

Creating a new class has been decided as the best design as it will group attributes and methods that depends on each other, following the "Group elements that must depend on each other together inside an encapsulation boundary" principle.

Every turn, each Zombie should have a 10% chance of saying "Braaaaains" (or something similarly Zombie-like).
As all AI Actor (including Zombie), iterates through its list of Behaviour objects until it finds an Action, a SpeechBehaviour class that implements Behaviour is created and has a 10% chance of returning a SpeakAction, else it'll return null. The SpeakAction will return a string "Braaaaains" that will be printed.

This was done so a Zombie saying "Braaaaains" will take a turn.

The speak method was not implemented in Zombie as the responsibilities of Zombie class should be kept minimal, ideally one, to follow the SRP.

Beating up the Zombies

Any attack on a Zombie that causes damage has a chance to knock at least one of its limbs off (I suggest 25% but feel free to experiment with the probabilities to make it more fun).

Within the **Zombie** class, the **hurt(...)** method is modified so that there's a 25% chance that least one of its limbs will be knocked off. For each severed limb, a **ZombieArm** or **ZombieLeg** will be created and passed on to a newly created **DropAdjacentItemAction** where it will then be stored in its **actions** attribute. This is so the actions can be executed later, and the **String** returned from the **DropAdjacentItemAction** can be printed with the attack action.

ZombieArm and **ZombieLeg** mentioned above are classes that inherits an abstract class **ZombieLimb** as they both share the same **tick(...)** methods.

The number of limbs knocked off will be determined by **x=rand.nextInt**, if **x > 15** then one limb will be knocked off, else if **5 < x <= 15** it will be two limbs, **1 < x <= 5** for three limbs and **x < 1** for four limbs.

On creation, a Zombie has two arms and two legs. It cannot lose more than these.

There will be two extra attributes in the **ZombieActor** class to keep track on the number of arms and legs the subclasses (eg. **Zombie**) have – **armCount**, **legCount**. These attributes will be set to **2** by default to represent two arms and two legs. There will be a condition that these attributes can only be an integer between **0** and **2** inclusive.

These attributes are stored in the **ZombieActor** class to keep it easily assessible when retrieving them and to ensure validity, following the "Classes should be responsible for their own properties" principle.

If a Zombie loses one arm, its probability of punching (rather than biting) is halved and it has a 50% chance of dropping any weapon it is holding. If it loses both arms, it definitely drops any weapon it was holding.

In the **AttackBehaviour** class, the probability that a normal attack will be executed for an actor with **ZombieCapability.UNDEAD** (i.e. **Zombie**) will be determined by checking their **armCount**. If **armCount** is **2**, then it'll have the normal probability of 50%. If **armCount** is **1**, then the probability will be halved to 25%, else if **armCount** is **0**, then the probability will be 0%. The probability of a bite attack is the remaining percent to make 100%.

It is assumed that the **Weapon** a **Zombie** is holding is the first **Weapon** in its inventory. If a **Zombie** loses an arm, then a **rand.nextBoolean()** will be checked and if successful then the **Zombie** will drop the first **Weapon** in its inventory. If a **Zombie** loses both arms, then it will definitely drop the first **Weapon** in its inventory.

If it loses one leg, its movement speed is halved – that is, it can only move every second turn,

although it can still perform other actions such as biting and punching (assuming it's still got at least one arm).

This will be implemented in the **Zombie** class. **Zombie** will have a **Boolean** attribute **movedLastTurn** that indicates whether the **Zombie** moved last turn. If **Zombie** only has one leg, then **movedLastTurn** is checked. If the one-legged **Zombie** has moved last turn, then **HuntBehaviour** and **WanderBehaviour** will not be used in **playTurn(...)**.

If it loses both legs, it cannot move at all, although it can still bite and punch.

Like above, however **movedLastTurn** will not be checked. **HuntBehaviour** and **WanderBehaviour** will not be used in the **playTurn(...)** if the Zombie has no legs.

Lost limbs drop to the ground, either at the Zombie's location or at an adjacent location (whichever you feel is more fun and interesting).

The lost limb(s) will be dropped at an adjacent location to give the **Zombie** the option to move and pick up the **Weapon** or to chase/attack a human. If the limb was dropped at the **Zombie's** location, then the **Zombie's** next turn will be determined as the **Zombie** will pick the limb up and end its turn. Limbs dropped on the ground will have an attribute indicating whether it was a leg or an arm that fell off for use in the **ZombieClub** and **ZombieMace** classes later.

Cast-off Zombie limbs can be wielded as simple clubs – you decide on the amount of damage they can do.

An abstract class **ZombieLimb** that inherits **WeaponItem** is created so that subclasses of **ZombieLimb** can be used as a weapon. The two subclasses of **ZombieLimb** are **ZombieArm** and **ZombieLeg.** When a **Zombie** loses a limb, the limb will be created as a **ZombieArm** or **ZombieLeg**. Having the abstract class **ZombieLimb** avoids repeating methods that are common with **ZombieArm** and **ZombieLeg**, such as the tick method which decides the allowable actions of the object. Having two distinct classes to represent types of limbs allows the limb to be crafted into a better weapon more easily as each limb has its own **upgrade()** method. Through this abstraction, **ZombieLimb** can have more subclasses without having to modify the method that crafts weapons from a **ZombieLimb**, following the Open/Closed Principle.

Crafting weapons

If the player is holding a Zombie arm, they can craft it into a Zombie club, which does significantly more damage. If the player is holding a Zombie leg, they can craft it into a Zombie mace, which does even more damage.

A **CraftWeaponAction** class that inherits **Action** is created which accepts an **Item** and executes the **Item.upgrade()** method. The original **Item** is removed from the **Actor** inventory and the new upgraded **Item** is added. When this is done on a **ZombieArm** (which is a **Item** subclass), the **ZombieArm.upgrade()** returns a **ZombieClub** which deals more damage. For a **ZombieLeg**, **ZombieLeg.upgrade()** will return a **ZombieMace**.

This follows the Dependeny Inversion Principle as **ZombieLimb** provides a level of abstraction between **CraftWeaponAction** and **ZombieArm/ZombieLeg**. **CraftWeaponAction** doesn't know the details of **ZombieArm/ZombieLeg** but as they inherit **ZombieLimb** and it has the abstract method **upgrade()**, **CraftWeaponAction** knows all the subclasses of **ZombieLimb** will have that method as well.

Rising from the dead

If you're killed by a Zombie, you become a Zombie yourself. After a Human is killed, and its corpse should rise from the dead as a Zombie 5-10 turns later.

A **HumanCorpse** is created that inherits **PortableItem**. It replaces **Human** when they are killed. **HumanCorpse** has a randomly generated death timer between 5-10 turns. Each turn, the death timer decrements by using the **HumanCorpse.tick(...)** method. Once the death timer is less or equal to 0, a **Zombie** will be spawned at the **HumanCorpse** location or adjacent location if an **Actor** is at the **HumanCorpse** location. If there's no location to spawn, then the **Zombie** will not spawn and will attempt to spawn again next turn. After the **Zombie** is spawned, the **HumanCorpse** will be removed from the map.

This works whether the **HumanCorpse** is on the ground or is being carried by an **Actor**.

Farmers and food

You must create a new kind of Human: Farmers. A Farmer shares the same characteristics and abilities as a Human.

A **Farmer** will inherit from the **Human** class. As a **Farmer** will share the same characteristics and abilities of a **Human**, we extend the **Human** class as to adhere to the "Don't Repeat Yourself" principle otherwise we would need to copy over the code from the **Human** class.

When standing next to a patch of dirt, a Farmer has a 33% probability of sowing a crop on it.

A **SowBehaviour** class that implements **Behaviour** is created to be responsible in determining whether **a Farmer** can or will sow. This is done by checking adjacent locations from **Farmer** for a ground with a **GroundCapability.SOWABLE** capability which was only given to **Dirt**. If such ground is found, then a **rand.nextInt(100) < 33** condition is executed and if successful, **SowBehaviour** will return a **SowAction**, with the sow location stored in it. If it is not successful, then it will return **null**. When a **SowAction** is executed, the ground at the stored location is set to a **Crop** ground. (**Crop** is discussed further in the next section)

A **SowBehaviour** will be added to the **Farmer behaviours** attribute so it will be executed if all prior **Behaviour** returns null. This method wasn't implemented in the **Farmer** class as it will add an extra responsibility for it which doesn't follow the SRP.

Left alone, a crop will ripen in 20 turns.

Crop is a class that inherits **Ground** and has an attribute named **ripeTime** which is initialised to 20. It also has a **CropCapability.UNRIPE** capability to indicate it is unripe. Each turn, its **tick(...)** method is called and **ripeTime** decrements each time. Once **ripeTime** is less or equal to 0, its **CropCapability.UNRIPE** capability is removed and a **CropCapability.RIPE** capability is added to indicate it is ripe. It's display character will also be changed so the player can visually identify a ripe crop.

When standing on an unripe crop, a Farmer can fertilize it, decreasing the time left to ripen by 10 turns.

A **FertilizeBehaviour** class that implements **Behaviour** is created to be responsible in determining whether a **Farmer** can fertilize. The **getAction(...)** method is overridden to check if the **Ground**, the **Farmer** is standing on is an unripe crop by checking if it has the **CropCapability.UNRIPE** capability. If it does, then it returns a **FertilizeAction**, else it'll return **null**.

When the **FertilizeAction** is executed, it uses the **Ground.fertilize(...)** method which will reduce the **Crop ripeTime** by 10.

The **FertilizeBehaviour** will be added to the **Farmer behaviours** attribute. The reasoning for this design is the same as **SowBehaviour**.

When standing on or next to a ripe crop, a Farmer (or the player) can harvest it for food. If a Farmer harvests the food, it is dropped to the ground. If the player harvests the food, it is placed in the player's inventory.

A **HarvestBehaviour** that implements **Behaviour** is created to be responsible in checking if there is a ripe crop at or adjacent to the **Actor** location. This is done by checking if the **Ground** has the **CropCapability.RIPE** capability. If it finds a **Ground** with that capability, then it will return a **HarvestAction** with the **Location** of the **Ground** stored in it, else it'll return **null**.

For **Player** to be able to harvest crop, we use the fact that for every turn, each allowable actions of the **Ground** adjacent to the **Player** is retrieved. So, having **Crop.allowableAction(...)** return an **Actions** containing **HarvestAction** if **Crop** has the **CropCapability.RIPE** will allow **Player** to harvest crops.

When the **HarvestAction** is executed, it set the **Ground** to **Dirt** and generate a food **Spinach**. The **Actor** capability is checked, if they have the **ActorCapability.DROPS_HARVEST**, then **Spinach** will be dropped to the ground. If they have the **ActorCapability.POCKETS_HARVEST**, then **Spinach** will be placed in their inventory. **Farmer** will have the **ActorCapability.DROPS_HARVEST** capability and **Player** will have the **ActorCapability.POCKETS_HARVEST** capability.

Again, same as the **SowBehaviour** and **FertilizeBehaviour**, it'll be placed in the **Farmer behaviours** attribute. The reasoning is also the same, if all these behaviours were implemented in **Farmer**, then it will have many responsibilities and the class will be large which will make it difficult to maintain.

Food can be eaten by the player, or by damaged humans, to recover some health points.

A **Food** abstract class that inherits **PortableItem** is created that holds the healing value in its **nutritionVal** attribute and has the **EatCapability.EDIBLE** capability. An **EatBehaviour** class that implements **Behaviour** is created that allows damaged **Human** and its subclasses to eat food to restore health. First, it checks if the **Human** is missing health which is done in the **Human** class and passed on to **Food**. If it is missing health, then the **Human** will check each **Item** in its inventory and look for an **Item** with **EatCapability.EDIBLE**, indicating that that **Item** can be eaten to recover some health points which is done with **EatAction**.

For **Player** to be able to eat **Food**, **Food.getAllowableActions()** method will return an **Actions** containing **EatAction** if it's in the **Player** inventory. Since **Player** retrieves allowable actions for each **Item** in its inventory per turn, this is possible.

When the **EatAction** is executed, the healing value is retrieved with the **Item.getNutrition()** and is passed to **Actor.heal(...)** which will the **Actor**. The **Food** will then be removed from their inventory.

So far, **Spinach** is the only subclass of **Food** but more types of food can be added easily, hence it follows the Open/Closed Principle.

Mambo Marie

Mambo Marie is a Voodoo priestess and the source of the local zombie epidemic. If she is not currently on the map, she has a 5% chance per turn of appearing. She starts at the edge of the map and wanders randomly.

A **VoodooPriestess** class that inherits **ZombieActor** to act as Mambo Marie. At the start of the game, the **VoodooPriestess** is not on the compound map (i.e. the main map), instead it is in its own map – **VoodooHome** which inherits **GameMap**. **VoodooHome** is just a 1x1 map, capable of holding one **Actor** – the **VoodooPriestess**. It's a turn-less map, meaning the actor on the map skips its turn. This is so the **VoodooPriestess** does not summon Zombies when it's not on the compound map.

The **VoodooPriestess** enters the compound map with a probability of 5% every turn which is determined in the **tick()** method in **VoodooHome**. **VoodooHome** is associated with the compound map so the **VoodooPriestess** can be moved to the compound within the **tick()** method. The **VoodooPriestess** will enter the compound map at a random boundary location.

Every 10 turns, she will stop and spend a turn chanting. This will cause five new zombies to appear in random locations on the map.

A **SummonBehaviour** class that implements **Behaviour** is created to keep tracking of the number of turns for each summon. The **VoodooPriestess** is associated with **SummonBehaviour**, along with **WanderBehaviour** and for each turn of the **VoodooPriestess** a counter increment by 1. When the counter is greater or equal to 10, then the counter will reset to 0 and the **VoodooPriestess** will begin chanting for a turn. This is done with a **ChantAction** class that inherits **Action**. Once the chanting is done, 5 **Zombie** will spawn at random locations on the map the **VoodooPriestess** is on (which should be the compound map). The zombie spawn is done with a **SummonZombieAction** class which inherits **Action**.

The **SummonBehaviour** was created instead of being in the **VoodooPriestess** class to minimise the responsibilities **VoodooPriestess** have. The **VoodooPriestess** class shouldn't be responsible of the process of summoning zombies and by separating the process into smaller processes and creating them as classes – **ChantAction** and **SummonZombieAction**, then **VoodooPriestess** isn't responsible for it and each class has one responsibility (SRP).

If she is not killed, she will vanish after 30 turns. Mambo Marie will keep coming back until she is killed.

The **VoodooPriestess** holds a counter that increments every turn. Once it's greater or equal to 30, then the counter resets and the **VoodooPriestess** returns to its home – **VoodooHome**, by creating and executing a **VoodooLeaveAction** This is possible as **VoodooPriestess** is associated with **VoodooHome** and can pass the **VoodooHome** to **VoodooLeaveAction** which will handle the **VoodooPriestess** leaving the compound map**. VoodooHome** then handles when the **VoodooPriestess** will come back as described before.

VoodooLeaveAction is created so **VoodooPriestess** don't need to be responsible for how it leaves the map (SRP).

Ending The Game

At the moment, there is no way to end the game. You must add the following endings:
• A "quit game" option in the menu
• A "player loses" ending for when the player is killed, or all the other humans in the compound are killed
• A "player wins" ending for when the zombies and Mambo Marie have been wiped out and the compound is safe

A **Quit** class that inherits **Action** is created and is added to the actions **Player** can choose from in the **playTurn(...)**. A new world class called **MultiMapWorld** inherits the **World** class and overrides the **stillRunning()** method. In this method, the last action of **Player** is checked and if it's a **Quit** action, then the game will end.

Also, within **stillRunning()**, the world is checked if it doesn't contain **Player** (i.e. **Player** is dead) or if doesn't contain any **Actor** (excluding Player) with **ZombieCapability.ALIVE** (i.e. All other humans are dead). It this check returns true, then the player loses and the game ends. The check of humans in the map is done by iterating through **actorLocations** which contains all the **Actor** in the world. It will return true if none of the **Actor** have a **ZombieCapability.ALIVE** capability and the game will end, and the player loses

Similar to how humans are checked, zombies and Mambo Marie are checked by searching for an **Actor** that have a **ZombieCapability.UNDEAD** capability. If an **Actor** with that capability is not in the world, then the game will end, and the player will win.