**DESIGN RATIONALE: IMMANUEL CHRISTABEL & ABHIRAM AROOP**

**Zombie attacks:**

* For the Zombie bite, I decided to have a random number generator in the getIntrinsicWeapon() method (in the Actor class). A number between 0 and 1 is generated. If the number is 1, return a new intrinsic weapon called “bite” that does 11 damage. Else, we return a new intrinsic weapon called “punch” that deals 10 damage.

I did it this way because bite is a type of intrinsic weapon that has a 50% chance of being used but also dealing more damage than punch.

* For the method executed in the AttackAction class, I check what weapon the Zombie has. If the Zombie has the intrinsic weapon “bite”, then I generate a number between 1 and 3. If the number is 1, I deal the bite damage and call the heal method on the Zombie to restore 5 health. If the number is not 1, the Zombie misses its bite. I did it this way because the execute method in AttackAction has a 50% success rate. I modified this to alter the success rate when the Zombie uses bite, has 1 arm or no arms.
* For the playTurn method in the Zombie class, I check the game map to see if the Zombie is standing on any weapons. If it is, then I use the getPickUpAction() on the Weapon to add it to the Zombie’s inventory. I did it this way because it checks the starting spot of the Zombie, which guarantees it to pick up the weapon if there is one.
* For the playTurn method in the Zombie class, before it returns all the allowable actions for each behaviour, I generate a number between 1 and 10. If the number is 1, we call the speakAction class and have it return “Braaaaains” on the screen. I did it this way because speakAction can also be extended to have humans say something as well as opposed to just the Zombies saying “Braaaaains”.

**Beating up Zombies:**

* I have expanded on the Zombie class to have an availableLimbs array list. This array list holds 4 weaponItems: 2 ZombieArm and 2 ZombieLeg. In addition to this, I overrode the Zombie hurt method so that on top of it decreasing its health, it generates a number between 1 and 4. If the number is 1, we call a new method from the Zombie class called loseLimb which checks the length of the availableLimbs list. If it’s greater than 0, generates a number between 0 and the length of the array list. We then remove the string from the array list and based on what we removed, we use the new class setItemAction which is the same as dropItemAction except you do not need to have the item in your inventory to drop it. The execute method in this class drops the limb at that index to the next available spot adjacent to the Zombie’s position. If there are 2 arms in the availableLimbs and we drop an arm, generate a number between 0 and 1. If it’s a 1, we iterate through the Zombie’s inventory and call dropItemAction on the first weapon item. If there’s only one arm and we drop the arm, we call dropItemAction on the first weapon item and place it where it's standing. I implemented it this way because the Zombie limbs are only ever used as weapons when it drops, so I decided to just have the Zombie hold an array of weapon items which can be thought of as its available limbs. I decided to make a setItemAction because the Zombie’s starting limbs cannot be stored in its inventory since when AttackAction is used, it may use a limb it still currently has as a weapon and Zombie’s can only use limbs once it has been dropped. I also made the Zombies drop their limbs in adjacent spots rather than its current spot because I reserved that spot specifically for when it drops a weapon item.
* As stated above, the zombie will have an availableLimbs array list to indicate what and how many limbs it has left.
* Expanding on the getIntrinsicWeapon method, I have made it so that it checks how many limbs the Zombie has before it returns the intrinsic weapon it uses. I have also edited the execute method in the AttackAction class so that it checks what weapon the Zombie is using. If it’s using bite, it has a 1 in 3 chance of hitting. Else, it has a 50% chance of hitting with any other weapon (weapon item or intrinsic weapon). The last thing I changed was the playTurn method in Zombie. I altered this to constrict what the Zombies can and can’t do based on their available limbs.
* in order to check what limbs the Zombie has in order to alter what it can and can’t do:
  + No limbs lost: if getIntrinsicWeapon finds that the Zombie has lost no limbs, the probability of getting “bite” instead of “punch” still stands at 50%.
  + 1 arm: if getIntrinsicWeapon finds that the Zombie only has 1 arm. It generates a number between 1 and 3. If the outcome is not 1, it returns “bite” instead of “punch”. 0 arms: if getIntrinsicWeapon finds that the Zombie has no arms, the Zombie is guaranteed to bite (i.e. getIntrinsicWeapon will only return bite).
  + 1 leg: Check how many legs the Zombie has in the playTurn method. If the Zombie has 1 leg, use the new variable in the Zombie class called moved which is a boolean that indicates if the Zombie moved in the previous turn. Edit the playTurn method in the Zombie class to only return a MoveActorAction if the moved state was false. Change the move state back to True.
  + 0 Legs: Check how many legs the Zombie has in the playTurn method. If it has no legs, restrict playTurn from being able to return a MoveActorAction.

I implemented it like this because the success rate of the Zombie’s attack is dependent on the execute method of AttackAction. By modifying this class, I’m able to set new probabilities for different events (i.e. the Zombie using “bite”). Furthermore, I edited the getIntrinsicWeapon because it helps return the appropriate intrinsic weapon based on what available limbs the Zombie has. Lastly, I edited the playTurn since it’s executed every turn for the Zombie. This helps ensure that if the Zombie has 1 or 0 leg(s), it will only ever move every other turn, or not move at all.

**Crafting weapons:**

* If the player picks up a Zombie arm, they can craft it into a Zombie club via the craftAction method. The craftAction method holds a hashmap that allows the player to exchange their current weapon for a better one if it exists in the hashmap. In this case, we return a Zombie club which deals 14 damage, has a display character “K”, and the verb “whack”. The Hashmap is hardcoded with the key weaponItems and value weaponItems as it is a simple and effective way to keep track of all the possible craftable weapons.
* If the player picks up a Zombie leg, we have a similar scenario as the one above, however, when the craftAction is called, we instead return a Zombie mace which deals 17 damage and has a display character “M” and the verb “thump”.

**Rising from the dead:**

* newZombie Class extends the Location Class and is used to convert a Dead Human Actor into a Zombie Actor, given that the Human has been dead for 5 turns.
* It has a method called checkConvertionStatus(Actor), that checks whether the Status of the Actor is Dead for 5 turns, given that the Actor is a Human. The return value is a Boolean, if the Human has been DEAD for 5 turns, then return True, else False. This method simplifies the process of the conversion method as it makes testing and understanding the code much simpler.
* newZombie also has another method called convert(Actor), given checkConvertionStatus(Actor) is True, it will remove the current Human Actor at the Location and add a new Zombie Actor instead.
* Doing it this way makes it so that we do not need to find the Location of the Human Actor, since newZombie is just an extension of Location.

**Farmers and food:**

* The Farmer Class extends the Human Class, in order to keep the same traits as the Human. It has a ‘F’ symbol on the map, in order to help the player to tell the difference between a normal Human and a Farmer.
* A random number generator is used to generate a random number between 1-3 and if the number is 1 and the Ground at the Location of the Farmer is a Dirt, then the Dirt is removed from the Location and replaced by a Crop. The random number generator is the best method in order to randomise the chance of sowing a Crop.
* A Crop Class extends the Ground Class, it also has an age variable (integer). The age of the crop increases for every tick() (turns of player). It also has an enum classification of UNRIPE or RIPE. UNRIPE means the crop is in the process of growing (unripe), and SAME means that the crop has ripen. If the age reaches 20, then the Status of the Crop changes from UNRIPE to RIPE. This method is similar to that of the way in which a Tree grows overtime, hence it will be much simpler to implement than looking at the actual age of the crop.
* A Farmer or a Player will receive the option to conduct the harvestAction on the Crop if its Status is RIPE. The harvestAction extends Action, once the harvestAction is called, it will replace the Crop at the Location and replace it with a Dirt object. If a Farmer calls harvestAction, a Food object is dropped in the Location using the setItem Class (mentioned above). If a Player calls harvestAction, a Food object (extends portableItem) is added into the Player’s inventory. This is done using PickUpItemAction and setItemAction, where the item first goes into the Actor’s inventory and is dropped if the Actor is a Farmer. This way, we don’t have to have completely different conditions and operations for the Farmer and Player (avoids repeated code).
* The Food Class has a eat method that takes a Actor parameter and adds 10 hitpoints to the Actor, using the eatAction Class. The heal operation is already implemented, hence does not need to be added.
* The Player has a method called eat, that allows a player to heal from a Food Object from the actor’s inventory( using removeItemFromInventory() method in Actor Class). This method will call the eatAction Class. This makes the eatAction Class have fewer conditions and can be used by Player as well as Human.
* The Human also calls the eatAction class if there is a Food Item at the current Location of the Human. The eatAction Class extends Action and contains a eat method that takes an Actor and heals them for 10 health (cannot exceed max hitpoints). This method makes use of the heal method in the Actor class, instead of implementing a new one.
* If a Farmer’s Location is on a Crop and the Status of the Crop is UNRIPE, the addAge method will be called to increase the age of the crop by 10 turns (using ticks()). Keeping track of the ‘age’ of and using the similar concept of heal(not exceed max hitpoints) to not exceed the age of 20 will ensure that this method will only increment the age to a maximum of 20.