## Zombie Attack

|  |  |  |
| --- | --- | --- |
| **Modified Class** | **Modified parts** | **Design Rationale** |
| **Zombie** | The method getIntrinsicWeapon()   1. returns a ZombieIntrinsicWeapon object instead of IntrinsicWeapon. 2. Using (rand.nextBoolean()), the probability of getting either punch or bite is 50   The method playTurn()   1. Added 10% probability with, (rand.nextDouble()<0.1), it will print out “Braiiinss”. | Random class is used to ensure that the probability is random. |
| **AttackBehaviour** | This class has a method getAction which will return ZombieAttackAction to execute the Action instead of the AttackAction.   1. In the method getAction, added a loop to loop through the list of item at the Zombie’s location so if ever there is an item which is an instanceof Weapon, the actor would pick it up by    1. Creating a pickUpItemAction and then    2. Execute the pickUpItemAction | 1. It returns ZombieAttackAction as this can be implemented on Zombie. There are 2 actors who attack in this game, Player and Zombie. 2. Player can access the action without the behaviour as we are the one controlling it. While Zombie need this behaviour, hence use ZombieAttackAction to implement this behaviour on Zombie. 3. Also it able to add the functionality of a Zombie picking up a weapon when a weapon is at the zombie's location. |

|  |  |  |
| --- | --- | --- |
| **New Added Class** | **Responsibility** | **Design Rationale** |
| **ZombieIntrinsicWeapon** | 1. The class inherits IntrinsicWeapon which gives them the attribute of the parent class. 2. Added **a new attribute**( double probability) so that we can manipulate the hit probability of each ZombieIntrinsicWeapon (punch or bite) . 3. Constructor will consist of the super class’s attribute and the new attribute (probability). 4. A getter for the probability attribute. | Class created specifically for Zombie, to add a new attribute of probability to achieve hit probability, also to distinguish the difference in attack of other characters and Zombie. |
| **ZombieAttackAction** | 1. New action class inherits from AttackAction class. 2. Created a weapon object with (weapon = actor.getWeapon), , Zombie actor without a weapon will acquire a ZombieIntrinsicWeapon object instead. 3. There is few more conditions to check  * Is the weapon instanceof ZombieIntrinsicWeapon? * If it is, then we will check if it is a bite attack as the bite attack has its own low hit probability and higher damage. * The hit probability will be implemented using the unique attribute **probability** of ZombieIntrinsicWeapon Object(rand.nextDouble()> **probability**), then it will return a String indicating Zombie missing the target. * Else, it will be a successful bite attack, actor.heal(5) - which heal 5 hit points to the ZombieActor | (This class is specifically for ZombieAttack so we can expand its functionality further when needed.)   1. Created a new class so that we can make use of the probability attribute we add to ZombieIntrinsicWeapon to have different attack’s hit probability 2. Also, this would be convenient as it helps to make the system extendable like adding the crafting of Zombie weapons to AttackAction instead of ZombieAttackAction as zombies do not have the capability to craft weapons. |

## Beating up the Zombie

|  |  |  |
| --- | --- | --- |
| **Modified Class** | **Modified parts** | **Design Rationale** |
| **Zombie** | 1. Create a new method which is able to knock Zombie’s limb off. After knocking the Zombie’s limb, the Zombie’s Limb (newly created ZombieLeg() or ZombieArm() object ) will drop next to the Zombie’s original location. Then, it will check if the Zombie is still alive (checking if it still has at least one limb). If no, set its hitpoints to -1 (dead).  2. Modification of getIntrinsicWeapon to vary the probability of punching.  - before returning the intrinsic weapon (punch or bite), if the Zombie’s choose to punch(50% chance), before returning the “punch” intrinsic weapon, new probability of punch will be equal to (no of arm left / max number of arms a Zombie can have) \* original probability.  3. Modify playTurn  - give Zombie a new attribute call movable (boolean) ,initialise it as true  - when zombie legs is knocked off, if no legs left, set movable to permanently false  - at each playTurn, if ZombieLimb.NoArms == 1, then movable = !movable  - in wanderBehaviour.getAction(), if !actor.movable, then straight return null  4. Override getWeapon method in Actor (parent) to handle dropping of weapon   * If the Zombie has weapon in its inventory, then before returning the weapon back to ZombieAttackAction, we will do the following inside the for loop of the getWeapon method,      * + Set **probability of dropping to weapon** equal to (no of arm left / max number of arm the zombie can have)   + Then, the Zombie now has a **set probability** of returning a null instead of the weapon (dropping the weapon)   + A new method called drop method will be created inside the Zombie Class to drop the item (remove the item from the Zombie inventory) | 1. As limb is an attribute of the Zombie, then the action of knocking off the limb should be in the Zombie Class (class is responsible for their own properties), instead of reducing the Zombie’s Limb in the AttackAction class.  2. By using (no of arm left / max number of arms a Zombie can have), we can reduce the use of numeric constants. If the zombie’s only have one arm left, then probability of punching is halved (½). If two arms still exist, probability is (2/2) = 1. If no arm exists, probability is (0/2) = 0.  3. By adding an extra attribute in the Zombie class, we can keep track of whether Zombie is allowed to move at a particular turn. We only have to manipulate the attribute when the number of legs ==1. If the Zombie has no leg left, the movable attribute will not be changed anymore (as the movable attribute will be set to false when two legs are knocked off). If the Zombie has 2 legs, movable no need to be changed too, as they will be initialised as true.  4. Probability handling is the same as 2). Create a new method to drop the weapon so that the getWeapon method only handle weapon retrieval and not inventory removal. (Each method should only has one purpose). |

|  |  |  |
| --- | --- | --- |
| **Newly Added Class** | **Responsibility** | **Design Rationale** |
| **Limb** | An attribute for Zombie Class.  Attribute:  1. Number of Legs Zombie still has  2. Number of Arms Zombie still has  3. Maximum number of legs (static final)  4. Maximum number of arms (static final)  Methods:  1. Return the number of Arm and the number of Leg the Zombie currently have.  2. Return a boolean stating if the Zombie can still function (meaning still have at least one limb).  3. Knock off the Zombie’s Arm and Leg (either one based on probability). | A new class (instead of two new attributes) are created so that we can grouped related details and functionality into one class -> Class that handles the Zombie’s Limb only.  Static final is used to represent the maximum number of legs and arms a Zombie has so that it cannot be changed (Zombie can only have 2 arms and 2 legs).  Methods are created to manipulate and retrieve the state of the Zombie Limb so that the class which calls it does not have to know the exact implementation. |
| **ZombieWeaponItem**  **(abstract)** | Inherit from the class WeaponItem. It is used to create subclass for WeaponItem that comes from a Zombie (their arm/leg). | It will act as the parent for all Zombie’s related WeaponItem. |
| **ZombieLeg** | A subclass of ZombieWeaponItem. It will be created when the Zombie’s Leg is knocked off and be placed next to the Zombie. It can be picked up as a weapon with damage = 10. | New classes for ZombieLeg and ZombieArm are created so that new features (such as causing continuous damage like poison) can be added to the weapon easily. This makes the system expandable. |
| **ZombieArm** | A subclass of ZombieWeaponItem. It will be created when the Zombie’s Arm is knocked off and be placed next to the Zombie. It can be picked up as a weapon with damage = 10. |

## Crafting Weapon

|  |  |  |
| --- | --- | --- |
| **Modified Class** | **Modified Parts** | **Design Rationale** |
| **Player** | 1. Create a new method called craftZombieLimb. Given a ZombieWeaponItem object (ZombieLeg or ZombieArm), craft a ZombieClub (using Arm) or Zombie Mace (using Leg). It will be called in AttackAction class if the weapon obtained is an instance of ZombieWeaponItem.  a. First check if the ZombieWeaponItem is a ZombieLeg or a ZombieArm.  b. If (ZombieLeg)  - Remove the Leg object from inventory  - Create a new ZombieMace() object  - Add the newly created weapon into inventory  c. If (ZombieArm)  - Remove the Arm object from inventory  - Create a new ZombieClub object  - Add the newly created weapon into inventory  Eventually, the newly crafted weapon will be returned to AttackAction. | The part highlighted in yellow shows that action of removing the Zombie’s Leg/Arm and putting the newly crafted weapon into the inventory are the same. Thus, to avoid repeated code, a new method will be created to handle the removal of the old weapon and the addition of newly crafted weapon into the inventory. |

|  |  |  |
| --- | --- | --- |
| **Newly Added Class** | **Responsibilities** | **Design Rationale** |
| **ZombieClub** | A subclass of ZombieWeaponItem. It is created by the player if the player is holding a ZombieArm object as his primary weapon. It does significantly more damage compared to ZombieArm. | New classes for ZombieClub and ZombieMace are created so that new features (such as causing continuous damage like poison) can be added to the weapon easily. This makes the system expandable. |
| **ZombieMace** | A subclass of ZombieWeaponItem. It is created by the player if the player is holding a ZombieLeg object as his primary weapon. It does significantly more damage compared to ZombieLeg, and even more compared to ZombieClub. |

## Rising from the dead

|  |  |  |
| --- | --- | --- |
| Newly Added Class | Responsibilities | Design Rationale |
| **Corpse** | It is a subclass of Item. It is created in ZombieAttackAction(), when the target(Human) that the Zombie attacks is dead. The Corpse Item will be placed in the exact location which the target dies.  Attribute:  1. Turn to revive  2. Name of the target (Human)  Method:  1. tick(Location) - override the parent class  - the method when called in GameMap, will check if the attribute **turn (**to resurrect) is equal to zero. If yes,   1. the current instance of the Corpse object will be remove from the location. 2. A new Zombie object, with the Corpse name will be added to the map and the actor list.   If no, turn to resurrect decreased by 1. | Item is chosen as the parent class of Corpse because the item has attributes and methods that the Corpse class needs.  Attribute would be the name attribute, which can be used to store the name of the dead human. Thus, when the Corpse resurrected, the resurrected Corpse will have the same name as the dead human.  Method would be the tick method. The tick method allows us to track/decrease the number of turn the Corpse would resurrect. The Corpse will then be ticked at each turn.  Inheritance allows fewer repeated code(DRY principle). |

## Farmers and food

|  |  |  |
| --- | --- | --- |
| **New Added Class** | **Responsibility** | **Design Rationale** |
| **Crops** | This class inherits Item class, and make use of the method tick from Item and has the attribute int **age**   1. Tick can be used for the time concept in the map to let the crops ripen. 2. Another method added if fertilise(), this method can be used to increase the age to make it ripen faster. | 1. This class inherits Item because it can make use of the method tick in item. Instead of Ground, item is more versatile and flexible for future extendable functionality as to remove the item at that location 2. Also available for harvesting crops better as if a Player comes close to a crop item, they will have the choice to harvest them or not. This is only possible because in Item class there is an attribute for list of allowableAction and thus if we can add these actions to it. Player is able to harvest crops. |
| **Food** | This class inherits Item and has an attribute int recoverPoints ( to indicate the recovering point of certain food item)   1. It uses the parent class constructor in its Constructor and one attribute of the Item class which is allowableAction to allow the Player to have the option to eat the food shown in the menu. 2. It has another method as a getter for recoverPoints | 1. Food inherits Item as well so that it is able to use the attribute allowableAction. 2. This allows Player to be able to have the option from the menu to eat food from inventory after harvesting it 3. Eating will heal hitPoints hence a method as a getter for recovery points is implemented with the thought that expand the system in the future where different food item might have different recovery points. |
| **Farmer** | This class inherits Human class and thus is a subtype of Human.   1. It has a private attribute of behaviour consist of SowCropBehaviour,FertilizeBehaviour,HarvestingBehaviour 2. The constructor using parent class’s constructor and thus create a Farmer with a different displayChar ‘F’ 3. The playTurn method is overridden. It consist of a loop to loop through the behaviour array and if the action is not null then it will return that action (which it getAction from the behaviours ) 4. If it is null, then the Farmse will wander, this is done by    1. calling super.playTurn() as Farmer has the attribute like Humans as well. | It extends Human because it is a subtype of Human where they would have the similar characteristic but Farmer has a few sets of unique behaviour.   1. It overrides the playTurn and consists of the few behaviours which are able to achieve the functionality it should have.  * Sow crops * Fertilise crops * Harvest crops into food |
| **FarmingBehaviour (abstract )** | It implements the Behaviour interface   1. It has an attribute that is protected which indicates the display character of a nearby ground. 2. It also has a method which can be inherited by all the children class to avoid repeated code of iterating a list of Exits. 3. An abstract getAction method so the children class can use it as well. | This acts as a parent to all farming-related behaviour   * To avoid repeated code hence made it abstract as it reduces dependency. * It consist of method which will be repeatedly used in children class (let them override) * This reduces dependency. |
| **SowCropBehaviour** | It inherits FarmingBehaviour class   1. Constructor uses the super constructor to indicate the displayChar of the nearbyGround should be a dirt to sow crops on it. 2. It has a few methods that are unique to its own class.  * suitableChar * To identify this the nearbyExit (parent class method) has a ground that has the displayChar which equals the attribute of nearbyGround which was set in the constructor. if yes, it returns that exit, else null. * correctGround * To check whether it is suitableChar!= null hence return true * getAction * If the correctGround returns true:   It will have 33% probability of returning a SowCropAction. | It extends FarmingBehaviour as it is a part of Farmer’s Behaviour and it has a common code where they need to iterate through list of Exits to check which exit has required condition (displayChar is dirt)   1. As for Sow Crop, we are dealing with nearbyGround’s displayChar so it would be for that specific location’s ground displayChar instead of item hence these methods can only be in SowCrop and not other behaviour like Fertilize or Harvesting.   Used a random’s class, random.nextDouble() <0.33 to achieve the sowing probability. |
| **GrowingBehaviour (abstract )** | This class inherits FarmingBehaviour,   1. Constructor has a parameter so that the children can input a wanted displayChar which satisfies the requirement. 2. It also contains 3 methods that will be inherited by its children class.  * suitableItem * To identify this the nearbyExit (parent class method) has a ground that has the **item**.getdisplayChar which equals the attribute of nearbyCharacter which was set in the constructor. if yes, it returns that exit, else null. * correctItem * To check whether it is suitableItem!= null hence return true * getAction is abstract for the children to inherit it. | This class is needed as it acts as a sub- parent to the other 2 behaviour, fertilizeBehaviour and HarvestingBehaviour as these 2 behaviour both check an **Item**’s displayChar instead of a Ground type object (only involves SowCropBehaviour)   1. This class is abstract as it acts as a parent to other classes as well, thus reducing dependency while reducing repeated code as well as similar code will be implemented in methods in this class and used by the children class. This class focus more on checking item.displayChar than Ground Types like the SowCropBehaviour hence there must be a difference to make it simple which one of the design principle is to keep things simple 2. It inherits the Farming Behaviour because the children class have also some similarity that can override its method to avoid repeated code. |
| **FertilizeBehaviour** | It inherits GrowingBehaviour   1. Constructor uses the super constructor to indicate the displayChar of the nearbyItem should be an unripe crop to fertilise it. 2. getAction method will indicate if the method correctItem (from parent class) is true, it will return a FertilizeAction, else null | This class involves in checking the item.displayChar→ which are crops objects hence it extends GrowingBehaviour as it will inherit these methods which are involved in checking the displayChar of an item.   1. After all the checking it returns an Action   **One Behaviour dedicated to one Action fulfils the design principle of keeping it simple. Also the single responsibility principle, this has impacted the behaviour classes to mainly have one responsibility, this would help in reducing or identifying bugs and also increase its capability to extend its functionality further.** |
| **HarvestingBehaviour** | It inherits GrowingBehaviour   1. Constructor uses the super constructor to indicate the displayChar of the nearbyItem should be a ripe crop to Harvest it. 2. getAction method will indicate if the method correctItem (from parent class) is true, it will return a HarvestingAction, else null | This class also involves checking the item.displayChar which is a ripened crop, and it will then return an Action if the conditions are met. Hence inherit GrowingBehaviour to avoid repeated code. |
| **SowCropAction** | This class inherits Action   1. It has 2 overridden method:  * execute   - this will identify whether the exit location will have no item, if there isn't, then it will add a Crops object then return the menuDescription.   * menuDescription   To show the statement on the menu (a toString method) | This class is an Action class where it will execute an Action for the Actor.   1. This will check if the exit that was previously checked in SowCropBehaviour has no other item at that location then it will add a Crop object. This will eventually == sowing a crop. |
| **FertilizeAction** | This class inherits Action   1. It has 2 overridden method:  * execute   - this will identify whether the exit location has a Crops item, if there is, then it will run the fertilise method of Crops to increase age so that it ripens quicker then return the menuDescription.   * menuDescription   To show the statement on the menu (a toString method) | This class is an Action class where it will execute an Action for the Actor.   1. This will check if the exit that was previously checked in FertilizeBehaviour there is a unripe Crops object at that location then it will run the method in Crops which is fertilise(). This is so that we can extend the attribute age in Crops and increase it so the Crops have fewer turns to ripe now. 2. Eventually, this will lead to ‘fertilising the crop’. |
| **HarvestingAction** | This class inherits Action   1. It has 2 overridden method:  * execute   - this will identify whether the exit location will have a Crops item, if there is, then there is a condition:  - is an actor instanceof Player?  - if yes then it will add food object to the inventory  -else if actor instanceof Farmer  - it will add a food object at that location  Then after these are complete, it will remove the Crop object at that location  then return the menuDescription.   * menuDescription   To show the statement on the menu (a toString method) | This class is an Action class where it will execute an Action for the Actor.   1. This will check if the exit that was previously checked in HarvestingBehaviour there is a ripe Crops object at that location then it will add Food object to the Player inventory, else it would add Food Item at that location if a Farmer harvest it . 2. Then remove the Crops Object at that location. 3. From here we can see that Crops and Food extends Item will very easily manipulate the object at that location or not while giving the Player that ability to choose this action as it is added in the allowableAction in Item. 4. Player has a playTurn method which will involve an Action object. Hence other than trying to uphold the design principles of single responsibility and keep it simple, we need an action for the player to have the option in the menu to run this functionality. |
| **EatingHealAction** | This class inherits Action   * It has an int attribute howMuchToRecover * Its constructor sets the howMuchToRecover attribute.  1. It has 2 overridden method:  * execute   - this will heal the actor based on the attribute of howmuchToRecover  Then it return the menuDescription   * menuDescription   To show the statement on the menu (a toString method) | This class is an Action class where it will execute an Action for the Actor   1. Eating food will heal the actor (damaged human or Player) 2. In this class it will heal the actor dependent on the recovery points a food has 3. It is a class which act as a parent to PlayerEatingAction as both Human and Player will heal after eating but Human pick up the Food on the Ground and eat it while Player have to iterate through inventory to eat Food 4. Doing this will keep it simple and definitely able to extend the system further as now changes to Player eating will not affect the Human 5. Also since Farmer extends Human, this means Farmer will also be able to have these characteristics. |
| **EatingBehaviour** | It implements Behaviour interface   1. getAction method  * Will iterate through the item list at that location, if item is instanceof Food, then check if Human is full health? * If not then: * Item will be removed from that location and if the actor (Human or Farmer ) is not of Full Health, then return EatingHealAction | This class implements Behaviour   1. Instead of just removing the Food object and heal the actor immediately, it return EatingHealAction 2. This decision is based on the single responsibility principle and keep it simple because at this point Behaviour classes that are newly added are all to check whether conditions are met for an Action to be return or not |
| **PlayerEatingAction** | It extends EatingHealAction   1. it will iterate the item in Player’s inventory and the item which is instanceof Food will be remove from inventory and Player will heal with returning execute of its parent class(EatingHealAction) | This class is specifically for Players   1. As mentioned, Player would have a playTurn method which will receive an Action and display the menuDescription in the menu then it's up to us to pick one of these options available. So based on this, we need an Action class which can specify for the Player as the condition for Player to eat and for Normal Humans or Farmer is different. 2. With the design principles of keeping it simple and single responsibility and don't repeat yourself also along with other considerations,  * this class will then inherit EatingHealAction, to not repeat codes of healing the actor. * Then it is kept with single responsibility is to check the Players inventory to have any Food, if yes then it is removed and then return the super.execute() * This would be more efficient as now we can do adjustment for Players without affecting the Human’s EatingBehaviour * We actually thought of using if trap with the conditions of (actor instanceof Player or Farmer) then proceed from there, however it shows it is a bad design habit hence we implemented this new class. |

|  |  |  |
| --- | --- | --- |
| **Modified Class** | **Modification** | **Design Rationale** |
| **Human** | Added a method called isFullHealth()  Which returns a boolean on (hitPoints == maxHitPoints)  If hp is not max, then it will return false. | So that the health status of the actor can be known without changing the code in Actor class and also without changing the value of that attribute. |