## \*PLEASE Take note that Humans can only eat if they are not of full health, please refer to the Farmers and Food section for more info.

## 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”. 2. 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 | Random class is used to ensure that the probability is random.   1. Also it able to add the functionality of a Zombie picking up a weapon when a weapon is at the zombie's location.   (changes from assignment 1 as we realise that it is inappropriate to let the functionality of zombie to pick up weaponitem to be at AttackBehaviour, since this is a feature for Zombie only, it is more appropriate to add it in Zombie class) |
| **AttackBehaviour** | This class has a method getAction which will return ZombieAttackAction to execute the Action instead of the AttackAction. | 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. |

|  |  |  |
| --- | --- | --- |
| **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. Override the hurt method. Now, there is a 25% of Zombie dropping its limb after it has been hurt/  2. Create a new private method called knockOffLimb which is able to knock Zombie’s limb off (by calling the KnockLimbOff method in ZombieLimb class). 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).  Then, it will also check if the Zombie is movable (at least one leg). If not, set movable to be permanently false(Note: only when the Zombie has exactly one leg that the movable attribute is able to be changed).  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  - at each playTurn, if ZombieLimb.NoArms == 1, then movable = !movable  - Note that the original behaviour attribute is splitted into behaviours and moveBehaviour. This separation is due to the fact that moveBehaviour will not run (the loop) if movable is false.  4. Create an overloading method of getWeapon to take one more parameter(map), which is required in order for us to know where to place the dropped 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 **probability** of returning a null instead of the weapon (dropping the weapon)   + A DropItemAction object will be created if the Zombie drops the weapon. | 1. Overiding the hurt method to include knockingOffLimb allows abstraction. Thus, the function which hurts the Zombie will not have to handle knocking off limb. The program is designed in such a way so that **each class is only responsible for its properties.** (knocking off limb is a property of the Zombie).  2. The action of knocking off the limb should be in the ZombieLimb Class (**class is responsible for their own properties**), instead of reducing the Zombie’s Limb in the Zombie class, again abstraction.  3. 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 will not need to be changed, as they will be initialised as true (the Zombie can move).  The reason of splitting the bahaviour attribute is to **avoid connascence of execution.** Originally, if we were to prohibit the Zombie to move after realising that Zombie cannot attack, we can simply put an if statement to check if the Zombie can move before continuing the loop (Hunt and Wander Behaviour). However, this approach would fail if a new behaviour is added (we will stop after the first iteration if the Zombie cannot move). This makes the program hard to be expand. Therefore, behaviours is spliited into normal behaviour and moving behaviours.  4. Probability handling is the same as 2). A DropItemAction object is created to drop the weapon in order to reuse the code (reduce repeated code). |

|  |  |  |
| --- | --- | --- |
| **Newly Added Class** | **Responsibility** | **Design Rationale** |
| **ZombieLimb** | An attribute for Zombie Class.  Attribute:  1.HashMap containing part(string) as key, and number of parts left(int) as value.  2. Static final maxLimb  Methods:  1. Return the proportion of Arm and the proportion 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. SRP principle.  A hash map is used so that the class is more extendable and less repeated code.  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). |
| **CraftableWeaponItem**  **(abstract)** | Inherit from the class WeaponItem. It is used to create subclass for WeaponItem that can be crafted (such as Zombie Leg and Arm) | It will act as the parent class for all craftable WeaponItem. |
| **ZombieLeg** | A subclass of CraftableWeaponItem. 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 CraftableWeaponItem. 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. |

A few downcasting is used.

a) Casting target (in AttackAction class ) to type Zombie in attack action

- casting is required because when the Zombie is hurt, there is a chance that its limb will be knocked off. To handle on where to place the dropped limb in the Zombie class, we need to know the map (type GameMap) where the Zombie is standing. Thus, the hurt parameter now needs to take on an extra parameter called map. This means that we need to have overloading method where the hurt method now takes two parameters. Since target is of actor class, it would not know if Zombie has this overloading method. Hence, casting is required.

b) Casting actor (in ZombieAttackAction) to type Zombie to getWeapon.

- casting is required as there is a chance that when getting a weapon, Zombie will drop the weapon if some of its arms have been knocked off. Thus, to drop the weapon, once again we need the map attribute. However, the getWeapon method in the Actor class does not take in the map parameter. Hence, casting is used. (Here we can pretty sure that the actor will be of Zombie type, as this AttackAction is exclusively for Zombie).

**An alternative** that I have think of before casting is to place a GameMap attribute inside the Zombie class. This would prevent casting as the Zombie contains an attribute of GameMap which tells the map it is currently on. Problem with this approach is that it will add an extra dependency between Zombie and GameMap.

**Final Decision:** After much consideration, I decide to use casting instead of adding unnecessary dependency to the code.

## Crafting Weapon

|  |  |  |
| --- | --- | --- |
| **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. |
| **CraftWeaponAction** | Action created for weaponItem that is craftable. It can be used by the two weapons (Zombie Leg and Zombie Arm), which can be crafted. | It will be added to every craftable WeaponItem’s AllowableAction. Thus, it reduces repeated code needed.  By creating a CraftWeaponAction class instead of adding method into Player Class will allow Player to make a choice on each turn on whether or not he wants to concede a turn to craft a new weapon (instead of it happens automatically depending on condition). This would make the game more fun.  Also, if there is another weapon that happens able to be crafted, the weapon class can apply the same logic that ZombieArm and ZombieLeg use (craftWeapon method). This allows polymorphism. |

## 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.When the Corpse is picked up, the number of turn to resurrect will not decrease until the Player drop the Corpse item (freeze).  Attribute:  1. Turn to revive  2. Name of the target (Human)  3. Freeze - boolean stating if the turn to revive is freeze  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.  2. Setter for freeze attribute   * Used by PickFreezeItemAction and   DropDefreezeItemAction | PortableItem class is chosen as the parent class of Corpse because it 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. **Notes that there is a dependency between Coprse and Map here.** It is unavoidable as new location (which requires a map parameter) has to be created if the existing location contains an actor already by the time the Corpse needs to resurrect as a zombie.  Inheritance allows fewer repeated code(DRY principle).  Furthermore, adding the freeze attribute will mean adding an advantage to the Player. After a human is dead, it will revive after a certain number of turns. However, Player can choose to pick it up to stop the Corpse from resurrecting(set freeze to true) so that he does not have to face the resurrected Zombie in the future. (A way to prevent a lot of enemies). |
| **PickFreezeItemAction** | Inherit from PickItemAction Class. Essentially, it does all the things that PickItemAction can do, and additionally, it will set the item (of Corpse type)’s freeze attribute to true (now the Corpse cannot resurrect). | Reduce repeated code. |
| **DropDefreezeItemAction** | Inherit from DropItemAction Class. Essentially, it does all the things that DropItemAction can do, and additionally, it will set the item (of Corpse type)’s freeze attribute to false (now the Corpse can resurrect). | Reduce repeated code. |

## Farmers and food

|  |  |  |
| --- | --- | --- |
| **New Added Class** | **Responsibility** | **Design Rationale** |
| **Crops** | This class inherits Ground class, and make use of the method tick from Ground 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. 3. Added an override method allowableActions so that Ripened Crops are able to be harvest surrounding the farmer or player as well. | 1. This class inherits Item because it can make use of the method tick in Ground. (updated at Assignment 2) as instead of Item, Ground can avoid 2 crops to be sow at the same time as we realise our code has this bug, so we changed it to Ground which can also help in reducing dependency for other class as no need to loop through a list of item at that specific location but easily get Ground’s displayChar as a crop. |
| **RipenCrops** | (Updated)  New class added during Assignment2 :  The class is created to represent ripened crops. As ripen crops can be harvested hence it needs to extend Item as Item has an attribute “allowableAction” which can allow the Player to be able to harvest it. | 1. Also available for harvesting ripen crops better as if a Player comes close to a RipenCrops 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 HarvestingAction to it. Players are able to harvest ripen crops. 2. This also can help in Crops class as if a Crops is ripen, they will add a RipenCrops object (for the ability of harvest) and also change their displayChar to ‘C’ (this avoids from planting more than one crop at a location. |
| **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 and add PlayerEatingAction 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. (Future feature going to be added ) |
| **Farmer** | This class inherits Human class and thus is a subtype of Human.   1. It has a private attribute of behaviour consist of HarvestingBehaviour,FertilizeBehaviour,SowCropBehaviour 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.  * Harvest crops into food * Fertilise crops * Sow crops   It goes in this order so that whenever a Farmer sees a ripen crop it would be a priority to harvest it first, then if it is a crop, fertilise it. Lastly if there is only dirt all around it will then sow crop (33%probability like it stated in the assignment specs) this would prevent it from always sowing crops. |
| **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.   **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.** |
| **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 private method  * correctGround * 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. * 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 there is no similarity with other farming- related behaviour hence a private method   Used a random’s class, random.nextDouble() <0.33 to achieve the sowing probability. |
| **FertilizeBehaviour** | It inherits FarmingBehaviour   1. Constructor uses the super constructor to indicate the displayChar of the nearbyGround should be an unripe crop to fertilise it. 2. getAction method will indicate if the location of the actor has a unripe crop, it will return a FertilizeAction, else null | It extends FarmingBehaviour as it is a part of Farmer’s Behaviour and it needs to set the requirement of the ground needs to be a Crops object hence the need to use FarmingBehaviour’s constructor (displayChar is unripe crop ‘c’)  (why doesn't it need to loop through Exit because in the Assignment specs it states “ when a Farmer stands on a crop” hence all i need is the actor’s location to check the Ground type) |
| **HarvestingBehaviour** | It inherits FarmingBehaviour   1. Constructor uses the super constructor to indicate the displayChar of the nearbyGround should be a ripen crop to harvest it. 2. It has a private method:  * correctItem   → loop through exits with inherited method nearbyExits to check whether the list of item at that exit has displayChar same as nearbyGround, if yes then return that exit.   1. getAction method will indicate if the method correctItem is true or the ground’s displayChar at the location of actor is equal to nearbyGround, 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 FarmingBehaviour to avoid repeated code of looping exits (nearbyExits method)  (2 conditions so that Player or Farmer can harvest crops at adjacent exits and at actors location) |
| **SowCropAction** | This class inherits Action  Private attribute location  = to assigned to a valid location as target location to sow   1. Constructor have Location as parameter to get the valid target location from the behaviour and set it as the target location to sow. 2. It has 2 overridden method:  * execute   - add a Crops object at target 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. Sow a crop at target location where SowCropBehaviour has check a valid location for it. |
| **FertilizeAction** | This class inherits Action   1. It has 2 overridden method:  * execute   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. 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   - iterate list of items at that location and if it is a RipenCrops object 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 RipenCrops object at that location, set the Ground to Dirt  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. Iterate list at target location which was checked by the Behaviour there is a RipenCrops 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 RipenCrops Object at that location. 3. From here we can see that RipenCrops 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) 2. menuDescription  * Return super class’s menuDescription | 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. |

**TAKE NOTE:**

**Humans can only eat food if they are not of full health : using the method isFullHealth() from Human class.**

2 downcasting was used :

**EatingBehaviour→ Human**

Casting to use a method in Human isFullHealth(), to determine whether or not the Humans should eat. If they are of full health they shouldn't be eating as this may cause Farmer (a subtype of Human ) to keep eating after harvesting. This only applies to Human and Farmers as Players do not iterate Behaviours hence the method was written in Human class which its characteristics can then be inherited by Farmer as well.

so we attempt to not cast it in humans but this will also then create dependency between classes and also a need to create new classes as we cannot modify the Actor class (in the engine package) and hence downcasting is needed as this reduces much dependency between classes.

**FertilizeAction → Crops**

FertilizeAction only will involve Crops objects. Casting to use a method in Crops. fertilize() this method is written in crops so that we can add on to the age (to hasten the growth of crops), we figure out some ways of using getter and setter but this will also increase dependency to the Crops class which is not efficient as the dependency only exist for this.

With much thought, we decided to reduce dependency instead.