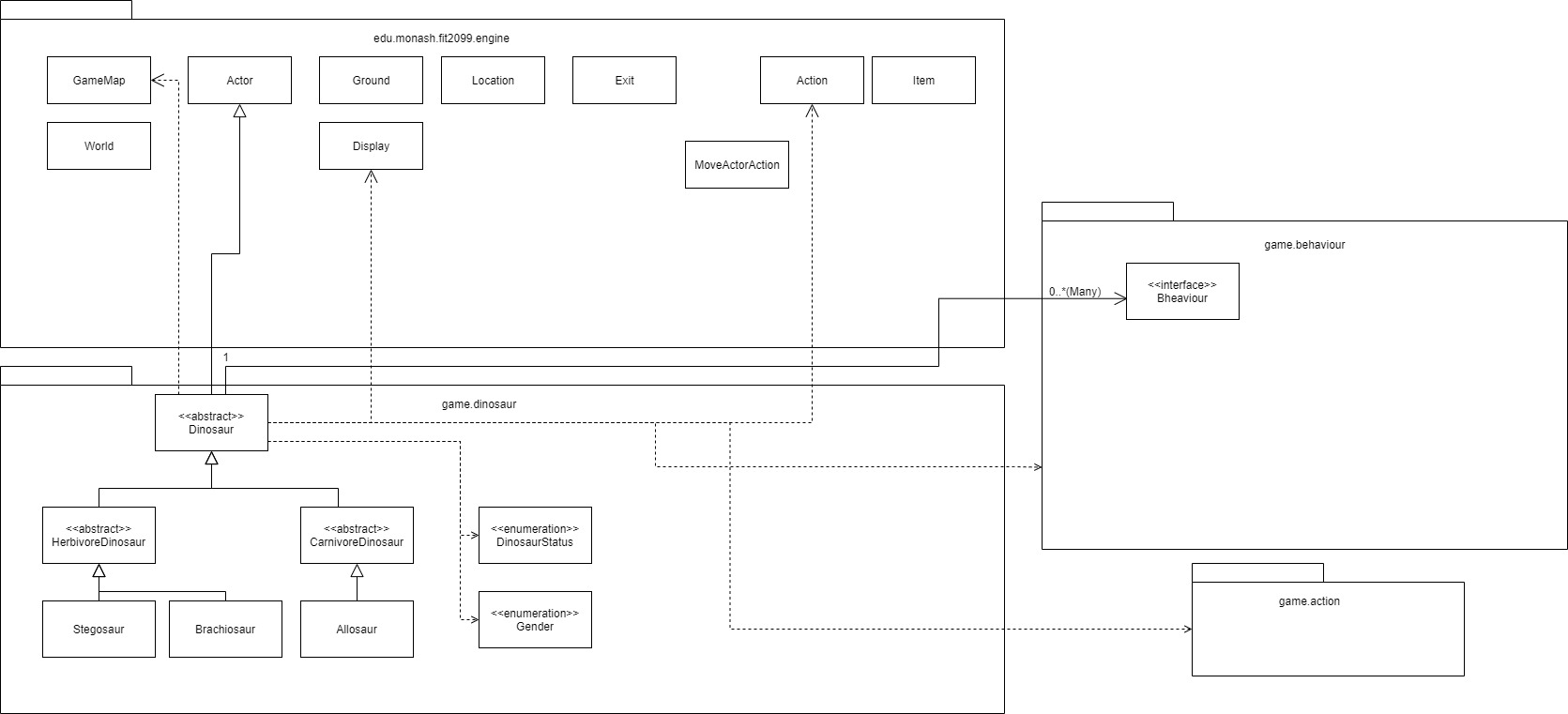
**Design rationale**

**Dinosaur: Stegosaur,Brachiosaur,Allosaur, its hunger and breeding behaviour and other stats (Only the use of it, for actual design of hunger, breeding and other behaviour, look beyond this section)**

**Class Diagram for Dinosaur related classes**

**DinosaurStatus Enum class**

The general idea of having this enumeration class is that we can give dinosaurs their own unique traits. We can then check the traits via the capability interface when needed, for example in actions and behaviors. This eliminates the need to depend on checking if an actor is an instance of a specific dinosaur, thus reducing unwanted dependencies, and instead we depend on the DinosaurStatus enum instead, which is sort of like the dependency inversion principle. However, since all dinosaur status related enums are placed here, there can be a scenario where many class end up depending on this. Another implementation would be not using enum classes and instead checking the class type when deciding what to do, but that would have been disastrous since we have to depend a lot on specific dinosaurs, and refactoring and adding new features would been problematic.

Implementation is straightforward as putting the enumerations in.

**Gender Enum Class**

I chose to separate this from the DinosaurStatus enum class because there is a need to tightly separate what type of enum to take in for Dinosaur’s constructor. If it allows any enum to be selected, if another user comes along and use the Dinosaur code, they might put some other enum that may break the code. The downside to this is it may add extra dependency since it’s another class Dinosaurs and potentially behaviours and actions need to depend on, however the upside is that we are strict in mentioning what type of enum we need. This could be included into DinosaurStatus instead, but as I have mentioned, there is a need to lock the type just in case.

Implementation is straightforward as putting the enumerations in.

**Dinosaur abstract class (extends actor)**

This abstract class is created so we can abstract all the details we need about a dinosaur. Anything from hunger to age, we can simply add it here. Only when we need to have specific dinosaur related stuff, we extend this class to add them. Since this class is still an Actor, we are following Liskov Substitution Principle where when the game World code is expecting an Actor, a dinosaur is still an Actor and can perform the same thing as an Actor. We therefore are not increasing the dependencies of the engine code (we can’t modify it either way). We are also following the principle of Don’t repeat yourself, since if we need to create a new type of dinosaur, we can simply extend this class, and just give it its unique attribute and that’s it. Any other mechanics such as hunger, pregnancy and AI behavior is still ran from this parent class, the extended class only adds to it, which also technically follows Open Closed Principle.

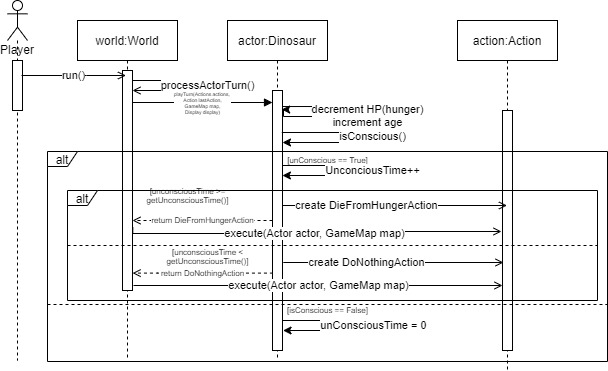
The downside here is that a lot of general dinosaur related code is all centralized here, so a lot of things are going to be sensible to changes in this class.

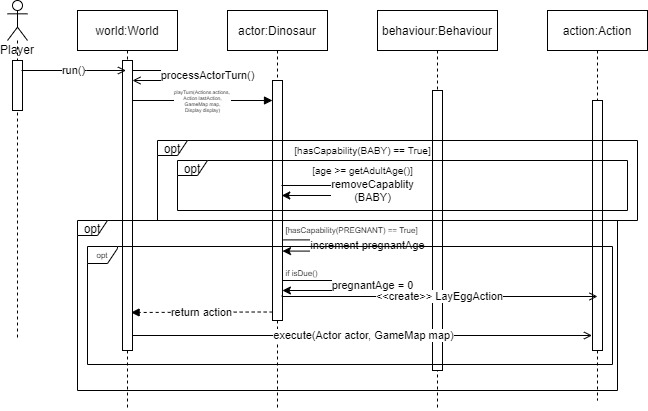
The worst alternative to this would have been creating each dinosaur class (Stegosaur, Allosaur and Brachiosaur) and have repeating hunger codes and everything, so that’s the reason why I chose to have this class.

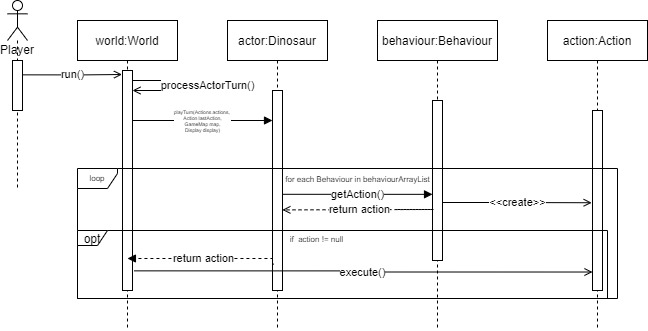
This class will have methods and attributes that all Dinosaurs should have, for example methods to retrieve a dinosaur’s pregnancy length, how much its egg cost, etc. They should also have an age attribute and timer attribute to count how long it was unconscious for. It should also have an arraylist of behavior for the playturn to use.

Its playturn method will go through all turn-related stats mechanics (Hunger decrease, age increase), unconsciousness, checking pregnancy, baby growth, and finally finding something to do

**Dinosaur class interaction diagrams**

****

****

****

**HerbivoreDinosaur and CarnivoreDinosaur abstract classes (Extends Dinosaur)**

Similar to Dinosaur class, we extend from Dinosaur into two types of Dinosaur for more abstraction, Herbivores and Carnivores. Both of these would have their own things these dinosaurs can do, for example when the player are near them, they can feed them according to what type of dinosaur they are. We have these two classes so that when we want a herbivore dinosaur, we can just extend HerbivoreDinosaur and create a new dinosaur from that, for example Stegosaur extends HerbivoreDinosaur, and it will have all Herbivore related actions and stuff, plus the things we inherited from the Dinosaur class. This allows us to reduce repeated codes, achieving don’t repeat yourself. We are also not increasing dependency, as this is still an Actor, following Liskov Substitution Principle.

The alternative is to have all Herbivore and Carnivore related stuff inside Dinosaur itself. However, this probably would violate the Single Responsibility principle, since it’s all put inside Dinosaur. Carnivore and Herbivore dinosaurs should have their own code instead. Thus, it might be better to split it into two sub abstract classes instead. Plus, if extra things are needed for all herbivore or carnivore dinosaurs, it can be easily added in here.

Implementation here involves checking player’s item and see if they can feed it, and adding behaviours and enumerations befitting a herbivore/carinivore dinosaur.

CarnivoreDinosaur has a special attribute of attackedDinosaur hashmap to store all the dinosaurs it has attacked (To apply attack restrctions), and has methods associated with operations on the hashmap. playTurn is overridden to update the contents in it every turn.

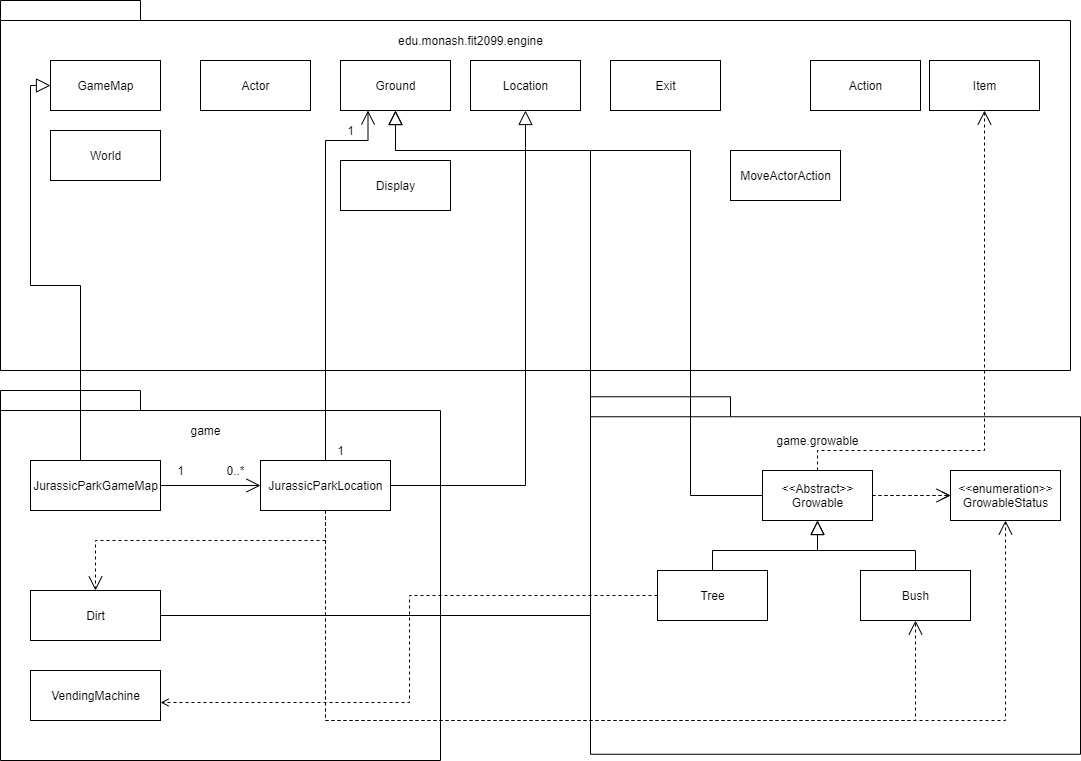
**Stegosaur, Allosaur and Brachiosaur (extends their respective Herbivore/Carnivore)**

These dinosaur classes will extend either HerbivoreDinosaur or CarnivoreDinosaur accordingly. Since we have already implemented in the parent classes before, all we need to do is give these classes their own values for the attributes and override the method implemented in Dinosaur to return class variables needed. This achieves our Don’t Repeat Yourself principle since all the codes are now in the classes before, and we have not increased dependencies. Allosaur needs its own behaviors since the attack restriction seems unique to it, we will have add its needed behaviours and attributes here. We are following the Open close principle here from the classes before since we added extra features to the classes before instead of fully changing its purpose.

Implementation of these involve extending Herbivore/Carnivore, then implementing methods from Dinosaur (all these getter methods and other stuff), adding extra behaviours as needed (Allosaur’s predator behavior of attacking anything nearby), and also adding enumerations that suits the dinosaur.

**Dirt, trees and bushes: Bush Growth, Fruit growth and dropping, and Bush Death**

**Class Diagram for Growable related, JurassicParkGameMap and JurassicParkLocation**

****

**GrowableStatus Enum Class**

This is used to give bushes and trees its unique stats, like tall, short, obstructs bush growth and so on. This is done to reduce dependencies, sort of like dependency inversion principle. Implementation is just creating the class and adding enumerations in it.

**JurassicParkGameMap and JurassicParkLocation**

In order to check for Bush Growth and Death, it would have made more sense to centralize it in the Location code, but you couldn’t modify the engine code. However, we could instead expand on the code and add our features, which is why these classes were created. Similar to the Conway’s demo, we extend both Location and GameMap. The Gamemap will override a method to create new JurassicParkLocation instead and JurassicParkLocation will have its tick() overridden to introduce the bush growth and death mechanics.

The alternative to this was checking it in Bush and Dirt’s tick method, and it would still work since we have access to the location from the tick’s input. The pros of having it in location means you get to directly work with the location instead and have the codes centralized. However it can be argued that these mechanics should be inside Bush or Dirt instead since it’s related to them. But anyway both methods are more or less the same.

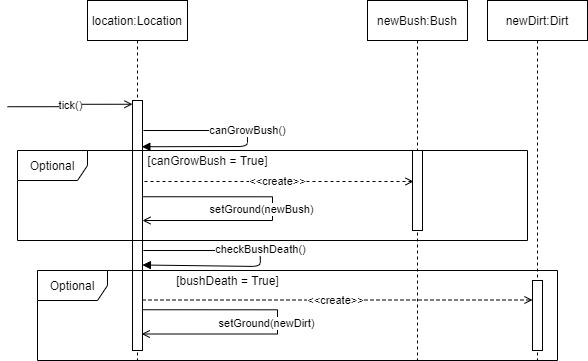
Since JurassicParkGameMap and JurassicParkLocation are extended from their respective parents, this is following Liskov’s Substitution Principle, and doesn’t increase dependency.

I can’t say this violates Single Responsibility Principle, since it is handling whatever is happening to the ground in the Location. I would say having the Location handle the lifecycle of its ground is better, so I chose this method.

Implementation of JurassicParkGameMap just involves extending the GameMap and overriding the create new location method to create a JurassicParkLocation instead. It also will involve the first-time initialization for the map by running bush growth check before the game starts via JurassicParkLocation.

JurassicParkLocation implementation involves extending Location, then overriding tick method to update the ground accordingly on bush death and bush growth. Has private methods to help do so.

**JurassicParkLocation interaction diagram**

****

**Growable extends Ground, and Bush and Tree extends Growable**

This will be the base abstract class for any plants that bear fruit. Bear in mind this can be still used for plant, but the chance of growing fruit need to be overridden to 0. This class was created so we could have centralized code for ground types that can grow fruit, as stated above. This allows us to reduce repeated codes (aka having to repeat fruit growing codes in every plants that grow fruit), which in fact follows the Don’t repeat yourself principle. We are also not increasing dependency here since Growable is of type ground and thus the game engine accepts it (Liskov Substitution Principle).

To create the bush and tree we need, all we need to do is extend Growable and add its stats accordingly. Since Tree has an extra thing (eco point and fruit dropping), we would need to override some methods.

The idea of the abstract class is to follow the Open-close principle which we are allowing it for more features but yet not modifying it, which I am following here, we add to the class but we do not do major modifications that change what the class do originally.

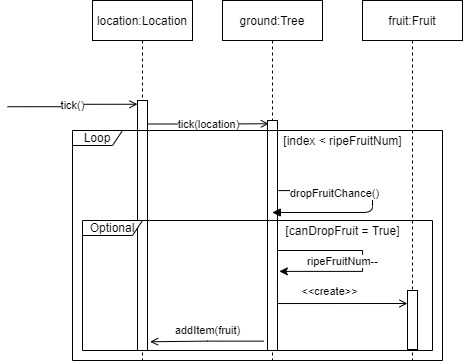
Tree and Bush is then extended from Growable and just needs to override chances. Tree also just needs to override grow fruit method to include ecopoint increments, which adheres to Open Close Principle.

For now I didn’t add another abstract class to represent growable that has fruit dropping, since it seems unique to tree for now. However, it is very easy to refactor it since all I need is to create a DroppableFruitGrowable class and move the fruit dropping code to that class and extend Tree from it, which itself will be a technical debt that has to be repaid in the future for a quick implementation right now.

Implementation of Growable involves extending Ground and overriding tick() to check fruit growth. It has integer attribute to count how much fruit is on it (creating an arraylist of fruit will waste memory, integer is better, we create fruit only when we need it). Also has other private method to help in tick. Also has methods that returns the growth chance of the fruit, increment and decrement, as well as getter for the number of fruit.

Bush just needs to extend from Growable and override the fruit growth chance whereas Tree extends Growable and adds its fruit dropping methods and fruit growth chance. Both adds appropriate enum in constructor.

**Tree interaction diagram for dropping fruit**

****

**Dinosaur Behaviours**

Each turn, every Dinosaur will behave in a certain manner. If it was already behaving a certain way in the previous turn and has not reached its goal, it will continue the same behaviour until the action is done.

All the behaviours below implements the Behaviour interface. That means we can have as much Behaviour as we want, and Dinosaur will still support it. This is due to polymorphism, since Dinosaur will have an ArrayList of type Behaviour, and all of these behaviours come from Behaviour interface, thus not increasing dependency.

**FollowBehaviour implements Behaviour (modified FollowBehaviour)**

**Following an Actor**

If the Dinosaur has found either a breeding partner or prey on the previous turn, it will continue walking towards its target by using FollowBehaviour. When a Dinosaur starts a FollowBehaviour, it will lock its target and keep moving towards it every turn until the Dinosaur reaches its target. Meanwhile, it will ignore other actions and only focus on its target, unless the target is no longer available or useful for the Dinosaur, then it will have a new Behaviour next turn.

In order to do so, we will make the constructor of FollowBehaviour to take in an Actor and an Action, where the Actor will be the target and the Action will be the action to be done when reaching the target. Each turn we will override getNextAction() to be the same as the previous action (MoveActorAction), that brings the Dinosaur closer to its target for each turn, and stop overriding and return the Action from the constructor when it is adjacent to its target.

**Going to a Location**

If the Dinosaur’s target is on the Ground (eg. Item) it will need a different type of FollowBehaviour as it is not following an Actor in this situation. In this case we will make another constructor that takes in a Location and an Action, where the location will be the goal and the Action will be the action to be done when reaching the location. Similarly, we will keep overriding getNextAction() of the MoveActorAction until the Dinosaur is on the Location, then return the Action.

When we create a MoveActorAction and we need to execute a multi-turn action, then we will override its getNextAction to keep returning the FollowBehaviour’s getAction().

Below is an example of overriding getNextAction so that the Dinosaur will continue moving towards the Actor target. (self here refers to the FollowBehaviour)



**Interaction diagram of FollowBehaviour**

**Diagram

Description automatically generated**

If the Dinosaur does not have a goal nor action to be made, it will go through a list of Behaviours to enable a new action.

**The Hierarchy for Behaviours is as shown:**

**PredatorBehaviour > BreedBehaviour > HungerBehaviour > WanderBehaviour**

**The diagram for how the Behaviour is checked through is above, in Dinosaur’s playTurn().**

**PredatorBehaviour implements Behaviour**

This Behaviour caters for predatory Dinosaurs which will attack other Dinosaurs if they are in range. So far it is only used by the Allosaur class since it is the only carnivore and the only one that can attack others. This Behaviour will be first priority among all other Behaviours for Allosaur, and does not need FollowBehaviour as it would be next to its target already, instead it will create EatPreyAction(Dinosaur, self) right away.

If the Dinosaur could not find any prey around it, it will go over the other Behaviours to see what it can do.

**BreedBehaviour implements Behaviour**

This behaviour checks if this Dinosaur is able to breed. If the Dinosaur is above a level of hunger, it is eligible for breeding.

After checking the conditions, the Dinosaur will scan through the map to see if it finds a potential partner to breed with. If the Dinosaur does find one, it will create a new BreedAction(this, target), lock the partner as its target and start following it, by creating a new FollowBehaviour(target, BreedAction).

If there are no partners, the Dinosaur will check the next possible Behaviour.

**abstract HungerBehaviour implements Behaviour**

This is an abstract class that contains some common features between two child classes: HerbHungerBehaviour and CarniHungerBehaviour. In this class, the Dinosaur will check if it is hungry. If it is, then we can continue on to the child class that caters to this Dinosaur. This class has an abstract method findFood which will be used by the two child classes.

If the Dinosaur is not hungry, it will go on to the last Behaviour, WanderBehaviour.

**HerbHungerBehaviour extends HungerBehaviour**

This Behaviour is used by Dinosaurs of the HerbivoreDinosaur group, which the herbivores will use to find the closest fruit either from a Bush, Tree or lying on the ground when they are hungry. This is done by overriding findFood from HungerBehaviour.

Firstly, the Dinosaur scans through the map to check for its desired source of food. It will then pick the closest food, create a new EatFromGroundAction(Item/Growable) and start moving towards it by creating a new FollowBehaviour(Location, EatFromGroundAction).

If the HerbivoreDinosaur cannot find any food in the entire map, it will go to the next possible Behaviour.

CarniHungerBehaviour extends HungerBehaviour

This Behaviour is used by Dinosaurs of the CarnivoreDinosaur group, which the carnivores will use to find the closest prey, corpse, or egg when they are hungry. This is done by overrding findFood from HungerBehaviour

Firstly, the Dinosaur scans through the map to check for its desired source of food. Then, it will pick the closest food, create the EatPreyAction/EatItemFromGroundAction and start moving towards it by creating a new FollowBehaviour(Actor/Location, EatPreyAction/EatFromGroundAction).

If the CarnivoreDinosaur cannot find any food in the entire map, it will check the next Behaviour.

WanderBehaviour implements Behaviour

This Behaviour is the final possible Behaviour, and should only be accessed if and only if all other Behaviours are not accessible for the Dinosaur. This simply lets the Dinosaur wander aimlessly with no goal in mind.

**Interaction diagram of Dinosaur Behaviour**

Text

Description automatically generated

**Dinosaur Actions**

Each turn, every Dinosaur will do something based on its current Behaviour. Below are Actions that a Dinosaur will do based on its Behaviour.

**Dinosaur**

These Actions are common actions shared among all Dinosaurs and will be under the Dinosaur class.

**BreedAction extends Action**

This Action allows a female Dinosaur to breed and start carrying an egg.

BreedAction uses a constructor that takes a Dinosaur as input. It will then call Dinosaur.Addcapability(PREGNANT) if the Dinosaur is female to add the PREGNANT enum to the Dinosaur, and do nothing if it is male.

**LayEggAction extends Action**

This Action allows a female Dinosaur to lay an egg when the baby is due.

LayEggAction uses a constructor that takes a Dinosaur as input. It will then remove the PREGNANT enum from the Dinosaur and reset its pregnantAge to 0. Additionally, an Egg of this Dinosaur’s species will be created on the ground.

**DieFromHungerAction extends Action**

This Action is created when a Dinosaur starves to death.

DieFromHungerAction uses a constructor that takes Dinosaur as input. It then creates a Corpse of that Dinosaur. After which it will remove the Dinosaur Actor and replace it with the Dinosaur Corpse on the same Location.

**Interaction Diagram of DieFromHungerAction**

****

**EatFromGroundAction extends Action**

This Action allows a Dinosaur to eat an Item from the Ground.

EatFromGroundAction is going to have these two constructors:

EatFromGroundAction(Actor eater, Ground targetGround)

EatFromGroundAction(Actor eater, Item targetItem)

Both will change the value of their respective attributes from Null to the target (example this.targetGround = targetGround). Then, in the execute() method, we will check which one’s null. We will execute the non-null one accordingly, so if our target’s a bush, we will heal accordingly (Maybe a method inside the Bush, or we do it here), then remove item or decrement accordingly.

**Interaction Diagram of EatFromGroundAction**

Diagram

Description automatically generated

**CarnivoreDinosaur**

These Actions are common actions shared among the Carnivores and will be under the CarnivoreDinosaur class.

**EatPreyAction extends AttackAction**

This Action allows a CarnivoreDinosaur to attack its desired prey and feed from them.

EatPreyAction uses a constructor that takes a Dinosaur (prey) and CarnivoreDinosaur (predator) as input. The prey’s hunger is decreased by a certain amount, while increasing the predator CarnivoreDinosaur’s hunger by the same amount. The prey, if it survives the attack, is then stored into the CarnivoreDinosaur’s hashmap with a countdown of how many turns left before it can be attacked again.

**Interaction Diagram of EatPreyAction**

**Diagram

Description automatically generated**

**Player Actions**

Every turn, the Player can choose to do something given their allowed Actions. Below are Actions the Player can do in certain situations.

**Dinosaur**

These Actions are allowed for the Player if they are beside a Dinosaur.

**FeedAction extends Action**

The Player is able to feed any Dinosaur as long as they have the suitable food to feed, increasing their hunger.

FeedAction uses a constructor that takes a Dinosaur and Item as input. It then increases the Dinosaur’s hunger based on the food given by the Player, while removing the Item from the Player’s inventory. If the Item is a Fruit, increase EcoPoint by 10 points.

**Interaction Diagram of FeedAction**

**Diagram

Description automatically generated**

**Growable**

These Actions are allowed for the Player if they are beside or on a Growable object.

PickFruitAction extends Action

The Player is able to pick fruits from the Growable object given that there actually are fruits on it. Player can only interact with Growable if they are standing on the same tile of that Growable.

PickFruitAction uses a constructor that takes a Growable as input. The Player has a set chance of failing to pick a fruit. If the Player managed to pick a fruit, decrement the fruit count on the Growable object by 1, increment the fruit count in the Player inventory by 1, and increase EcoPoints by 10 points.

**Interaction Diagram of PickFruitAction**

**A picture containing diagram

Description automatically generated**

**VendingMachine**

These Actions are allowed of the Player if they are beside the vending machine.

BuyItem extends Action

The player prompt the vending machine to show its menu, and buy items from it as long as they have enough points to do so.

BuyItem uses a non-parameter constructor. It first prints out a menu of Items for the Player to choose. Each time the Player buys something, the Item will be added into the Player’s inventory and the Player’s eco points will be deducted. This Action will keep prompting the Player for what Item to buy until the Player chooses to exit the Vending Machine.

**Design Rationale for Items and Vending Machine**

**ItemStats enum class**

This class is created to handle constants such as the prices of items to be sold in the Vending Machine, the amount of HP each edible item gives to a dinosaur, and the amount of damage each weapon gives. Used by PortableItem and all of its subclasses.

***Actions for items***

**BuyItem extends Action**

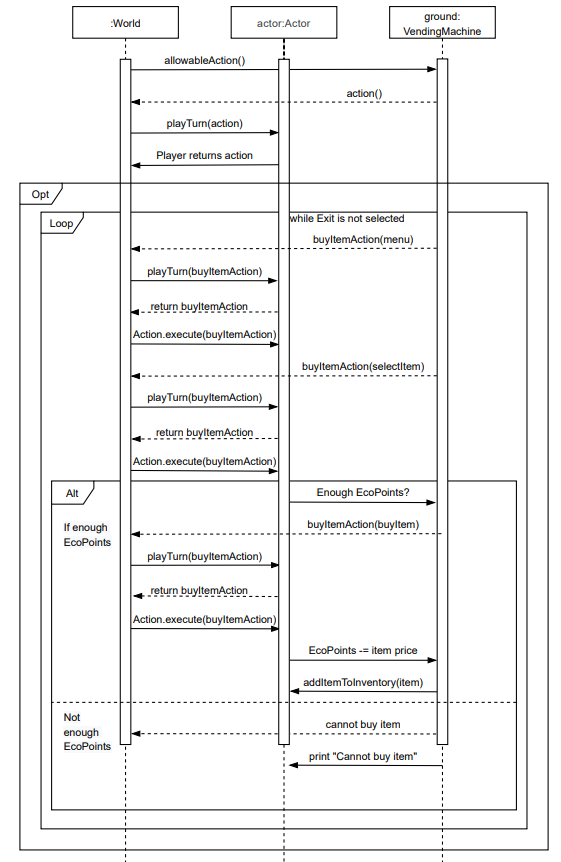
This class is called when a player wants to purchase items at the vending machine.

The player, if adjacent to a vending machine, prompts the vending machine to show it’s item menu, and buy items from it as long as they have enough points to do so.

BuyItem uses a non-parameter constructor. The process starts by printing out an item menu for the player to select. Each time the player wants to buy something, BuyItem will check if there are sufficient EcoPoints in class VendingMachine. If there is enough, the selected item will be added into the player’s inventory and the EcoPoints will be deducted, else a message (e.g Not enough EcoPoints!) will be returned and the player will again be prompted with the item menu.

This action will continue until the player quits the item menu.

**Interaction Diagram of BuyItem**

****

***Items* *misc***

**PortableItem extends Item**

This is the base class for any and all items that can be picked up and dropped.

If the item can be picked up, PortableItem will call super’s constructor and will have a boolean attribute type called portable that returns true, else false if the item is not portable. This class also deals with removing items from a player’s inventory when it is consumed.

**Corpse extends EdibleItem**

This class represents a dinosaur corpse, created when a player or an allosaur kills a stegosaur, or when all three dinosaurs are unconscious and not fed for a certain number of rounds. Food source for only the allosaur as of now.

Overrides super’s tick() method to initiate the set rotting timeframe (e.g 30 seconds) if it is placed on the map.

**EdibleItem abstract class extends PortableItem**

This abstract class represents all edible items. Manages the corpses of dinosaurs, vegetarian and carnivore meal kits, fruit and all dinosaur eggs. Allows for further implementation of other edible items .

**Egg extends EdibleItem**

This class contains all current and future dinosaur eggs and their characteristics. This allows for future modifications to all eggs and lessens unnecessary code.

Holds attributes of the eggs such as hatch time needed and type of dinosaur hatched.

If Eggs are picked up and carried in a player’s inventory, override super’s tick() method is overridden to ensure that they won’t incubate or hatch.

If an egg is placed on the ground (getInventory.remove()) or laid by a dinosaur (layEggAction), override super’s tick() method to start the countdown until the egg is hatched. EcoPoints are incremented when an egg hatches.

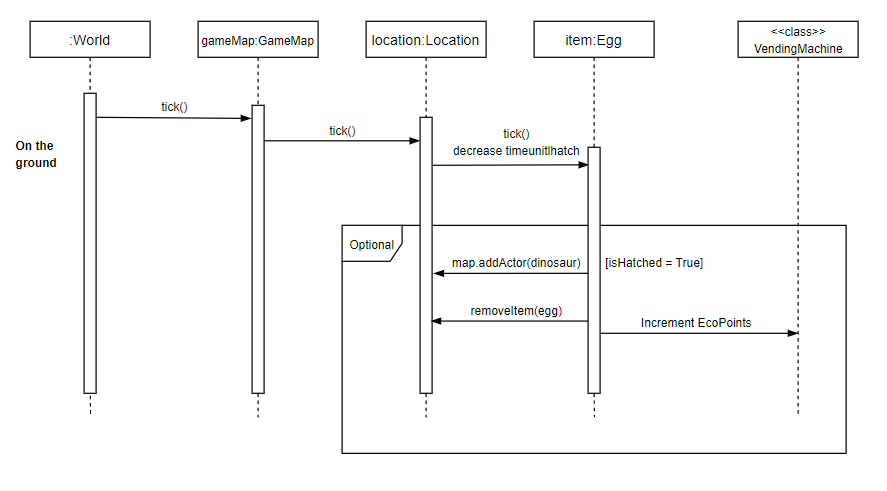
**Fruit extends EdibleItem**

This class represents a piece of fruit, which is produced by a tree or bush.

This is a food source of all herbivorous dinosaurs.

Overrides super’s tick() method to start the rotting process (15 turns) as soon as it is dropped.

**Interaction Diagram of Egg**



**LaserGun extends abstract class WeaponItem**

This class represents a laser weapon used to kill dinosaurs for population control or to provide food for the Allosaurs.

Will use a constructor with an actor (e.g Stegosaur) as input and reduce the actors HP by a set amount depending on the weapon used by overriding damage() and using verb() to return an appropriate message.

**Vending Machine extends Ground**

This class represents the vending machine and has a stored amount of EcoPoints as a static variable. All actions that result in EcoPoint rewards will accumulate in the static variable. Also contains all purchasable items with their prices.

Players are not able to enter (step on) the vending machine and can only use the vending machine while adjacent to it.

Displays an item menu for the player to make a purchase.