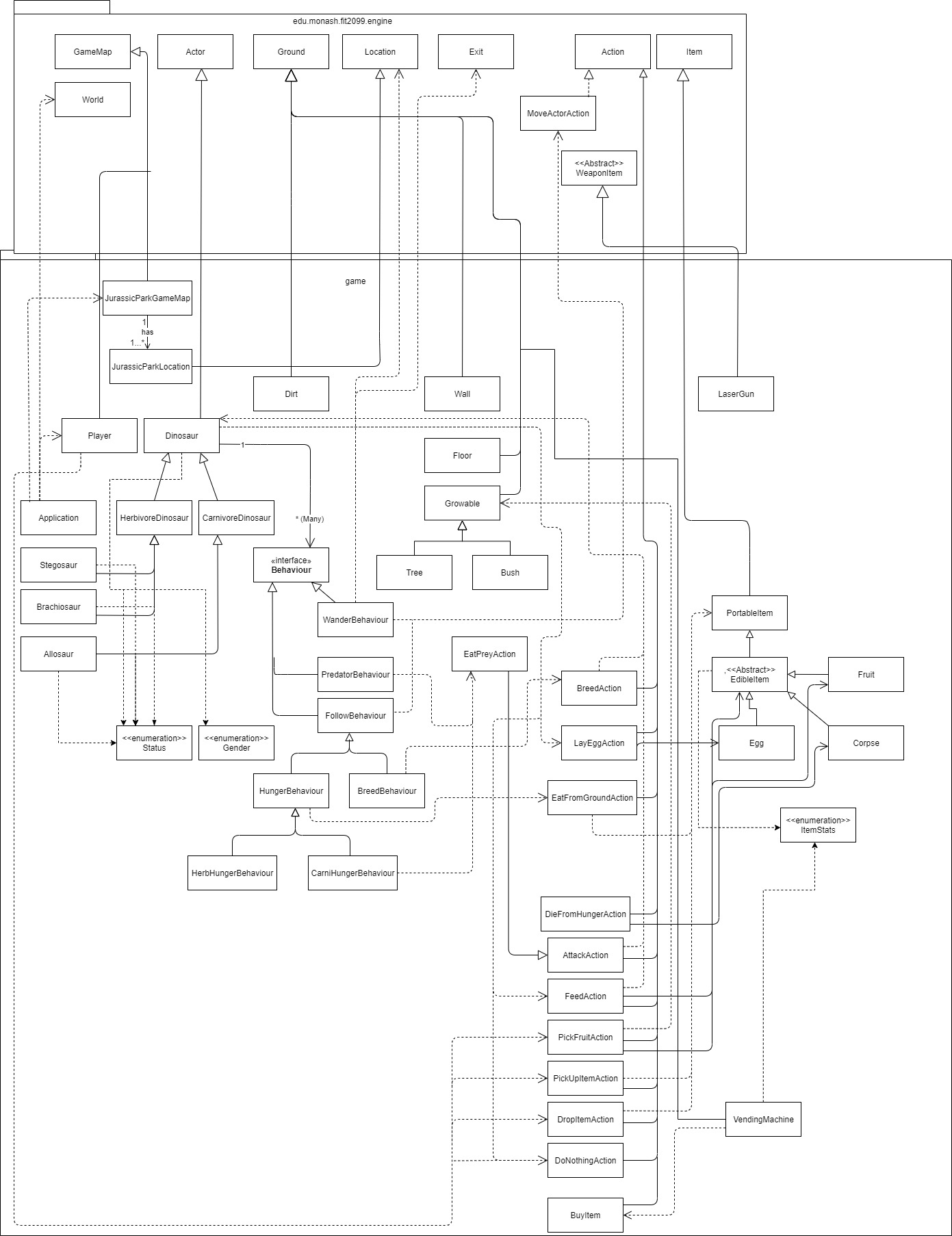
**Design rationale**

**Overall UML Class Diagram**



**Status Enum Class**

This class is created to keep track of the dinosaurs’ statuses, such as pregnant, baby and so on. This is used by the dinosaur class, and its subclasses. Could be included inside the dinosaur class itself, or split into multiple enum classes. This class can also be used to add statuses to grounds, like TALL or SHORT for bushes and trees, reducing dependency on checking class type. This will help to reduce dependency on checking dinosaur class type.

This relates back to the engine using Actor’s capability attribute to store the enum.

**Gender Enum Class**

This enum class is used in determining dinosaurs’ genders. Is a separate enum class from Status as we need to lock what type of enum is it when creating dinosaurs. Using this will be much clearer than using Boolean to indicate female or male, making it easier to understand.

This relates back to the engine using Actor’s capability attribute to store the enum.

**Dinosaur Abstract Class Extends Actor**

The class was created to have a generalisation of what things are to be processed each turn for the dinosaur (playTurn() method), such as hunger(HP) and then printing a message when a dinosaur becomes hungry (Done in checkhunger() ), age, pregnancy, as well as going through the behaviour ArrayList to find an action to do. When extended, the extended class should call its parent’s playTurn() method, reducing repeated code. This class should have various helper methods that will help in playTurn() such as checking how long it was unconscious, to avoid cluterring up playTurn(). Would also have many attributes, and static final attributes that help in playTurn() and other classes, such as age, pregnantAge, unConsciousTime, HUNGRY\_INT, CORPSE\_ROT\_TIME. Will also have an ArrayList of behaviour that is used in playTurn(). The constructor of this class should initialise the age and gender, and add WanderBehaviour, and more can be added to the front of the ArrayList in extended classes, to show more behaviours in a dinosaur. There should be two constructors where one is an one parameter constructor that takes in gender from the status enum class, and will add the gender enum(MALE or FEMALE) into capabilities depending on the gender, and automatically initialises age to the adult age. The other constructor will take in age and gender (or just age and randomise the gender), and initialises the dinosaur normally, with an inputted age, as well as gender.

When extending this class, many of the attributes and methods will be inherited, thus reducing repeated codes. Each dinosaur is still of type Actor, in which the engine will still process, and doesn’t increase dependency.

Any subsequent classes that wish to have its own max HP, starting HP, etc can just simply override the final static attribute, and all other code will still use the same attributes instead of a number that needs to be updated everywhere else.

This relates back to the engine, as Dinosaur is still an Actor, and is accepted by the engine. The engine’s World class will still go through every Actor added into the World, due to polymorphism, even to the children classes of Dinosaur, their playTurn() is still invoked.

**HerbivoreDinosaur Abstract Class Extends Dinosaur**

This class is created so that we can generalise the herbivore food behaviour. This class will extend its parent’s constructors and add the necessary behaviours, BreedBehaviour and HerbHungerBehaviour in the constructor, to give HerbivoreDinosaurs breeding and feeding behaviour which simulates the dinosaur’s breeding and feeding. This class is here just in case we may want to introduce more Herbivore Dinosaurs, and when we do, it will reduce repeated codes.

**CarnivoreDinosaur Abstract Class Extends Dinosaur**

This class is created to generalise Carnivore food behaviour. This class will extend its parent’s constructor and add BreedingBehaviour and CarniHungerBehaviour which simulates the dinosaurs breeding and feeding. This is created in case we may want to introduce more Carnivore Dinosaurs, and when we do, we will reduce repeated codes. Will override getAllowableActions and check if the other actor (Must be a player) has suitable food (Carnivore Kit) to feed to this dinosaur. This is done by looping through the other actor’s inventory.

**Stegosaur Extends HerbivoreDinosaur**

This class is used to represent a Stegosaur. Should have its own starting HP, max HP, adult age that is unique to Brachiosaur. All of the necessary behaviours are already in the parent’s code. The constructor should add a short\_neck enum which will be used in HungryHerbivoreBehaviour. Default constructor should call super and initialise the Stegosaur with its starting HP and age. Will initialize final static attributes inherited from Dinosaur. Will override getAllowableActions and check if the other actor (Must be a player) has suitable food (Herbivore Kit and Fruit) to feed to this dinosaur. This is done by looping through the other actor’s inventory.

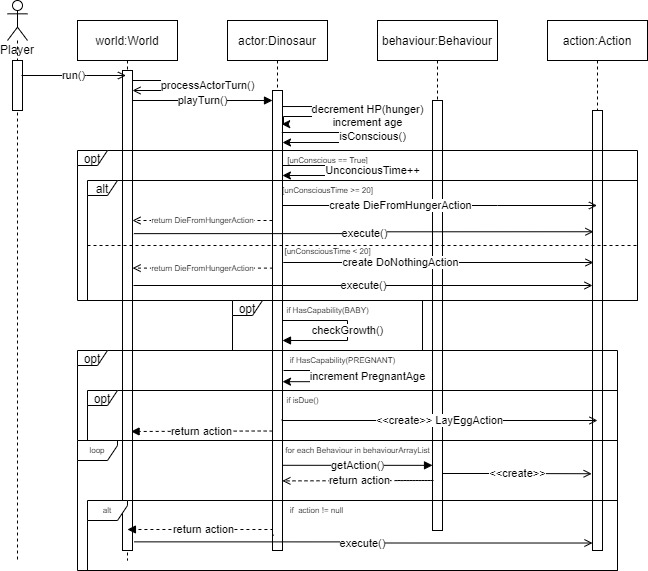
**Brachiosaur Extends HerbivoreDInosaur**

Used to represent a Brachiosaur. Should have its own starting HP, max HP, adult age that is unique to Brachiosaur. All of the necessary behaviours are already in the parent’s code. The constructor should add a long\_neck enum which will be used in HungryHerbivoreBehaviour. Default constructor should call super and initialise the Stegosaur with its starting HP and age. Will initialize final static attributes inherited from Dinosaur.

**Allosaur Extends CarnivoreDinosaur**

Used to represent an Allosaur. Constructor will add PredatorBehaviour to the front of the behaviour ArrayList to simulate the Allosaur attacking adjacent Stegosaurs. Will have a hashmap to keep track of the Stegosaur it has attacked. playTurn() is overridden, and we will loop through the hashmap and increment the turns elapsed since Stegosaur attacked in it. If it’s more than 20, we will remove it, then we call super’s playTurn(). Will initialize final static attributes inherited from Dinosaur.

**General Interaction Diagram of Dinosaurs**



**JurassicParkLocation Extends Location**

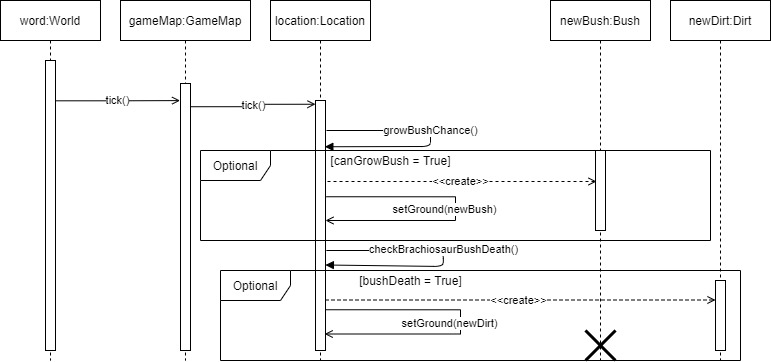
Used to represent a location in our Jurassic Park game. This is used so we can check whether a bush can grow in the tick() method, as well as implementing the chance a bush dies if there’s a brachiosaur on it. This is the same as how the Conway demo implemented it. Due to polymorphism, other methods that expects Location will still work since JurassicParkLocation is a location, thus not increasing dependency.

The initial idea was to have Bush Growth checked in dirt, but it was better to have it in Location to centralise everything. If we needed to add more things related to location every turn we could do it here. This will not increase dependency since World still uses Location

**JurassicParkGameMap Extends GameMap**

Used to represent a GameMap in our Jurassic Park game. This is used so we create a JurassicParkLocation instead of a standard location. This is the same as how the Conway demo implemented it. The game engine will still expect a normal GameMap but due to polymorphism this is still accepted and will work, thus not increasing dependency. This will not increase dependency since World still uses GameMap

**Interaction Diagram of Location**



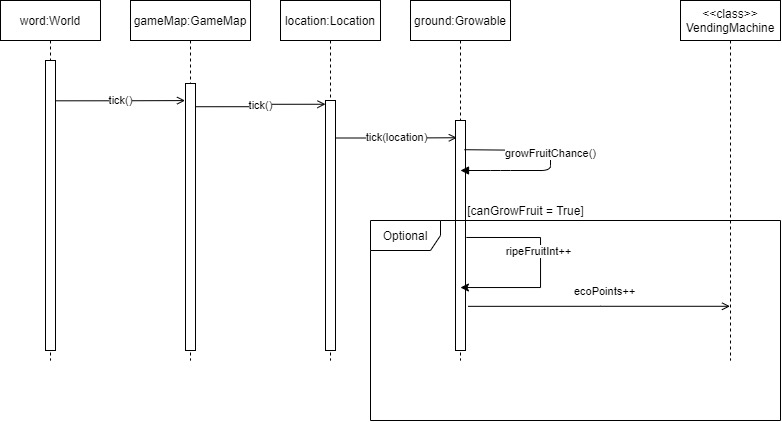
**Growable Abstract Class extends Ground**

Used to represent anything that can grow fruit. We will override tick() so we can add a check to see if we can grow fruit. This class will have a final static variable to store the chance of growing the fruit and the check will use that variable. By doing this, we can easily extend the class to add things like bushes and trees that grow fruits, where they can override the growth chance, and the chance is still passed over to the parent Growable class when using super’s tick() to check whether a fruit will grow, reducing repeated code. The class should also keep track of how many ripe fruits are there (integer). Should have helper methods for the number of ripe fruits.

Relates back to the engine since anything of Ground type is still ticked through if it’s assigned to a Location. Location will tick() the ground it has, and these methods described above will run, as well as its children’s tick(), provided they called they called the super’s tick then run their own.

Tree and Bush will not override AllowableAction() since we need to be **on the exact location of the bush** to pick the fruit. The action will be added in Player’s playTurn().

**Interaction diagram of Growable**



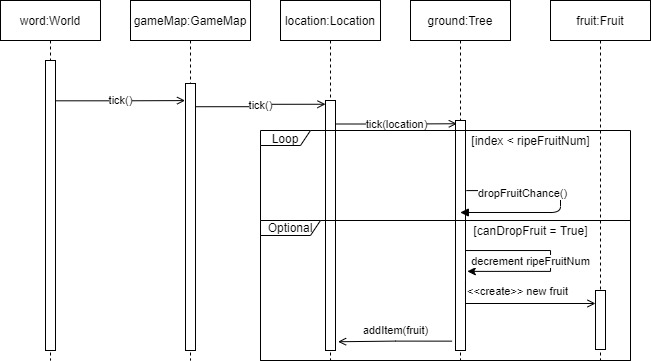
**Bush extends Growable**

Represents a bush. Would just call super’s constructor to indicate its character on the map. Everything here should be done in the parent’s code already. The grow chance should be set to something like 0.1, which when tick() is called, will be used in chance calculation. Should add an enum SHORT in capability which can be used in behaviour to check whether a dinosaur can eat from this.

**Tree extends Growable**

Represents a tree. Calls super constructor to indicate its character. This class will add a final static dropFruitChance. The tick() method is overridden and we will call super’s tick() to run the fruit growth chance, and run something like a dropFruitCheck(), which takes dropFruitChance and sees if a fruit will drop, by looping through numberOfRipeFruit. Also will have age and will increment age and change the character on the map. The allowableAction is also overridden and adds pickFruit (If is player). The grow chance would also be set here, like 0.5, and will be used in super’s tick(). Should add an enum TALL in capability which can be used in behaviour to check whether a dinosaur can eat from this.

**Interaction Diagram for Tree**



**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**

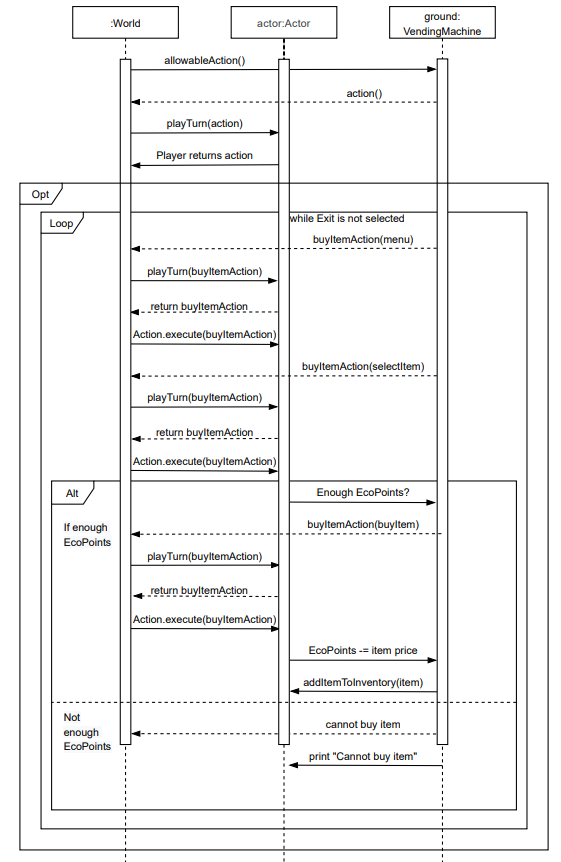
This class is called when a player wants to purchase items at the vending machine.

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.

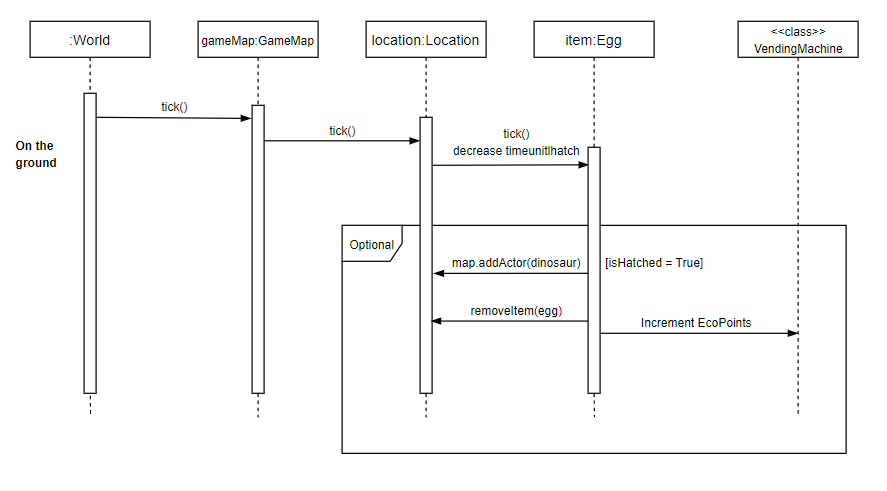
**Fruit extends EdibleItem**

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.