Design rationale

Please refer to "Note" to know which part of specification header we are talking about. The design document is split into parts talking about classes instead of the header one by one because it is hard to design one header without the other, whereas having one side of stuff (Dinosaur, then behaviour) is easier to explain our design.

Indexing of where to look at for specification header related document

Refer to this, and the "Note" to know where to look specifically for things related to it.

Dirt, trees and bushes: Section detailing that which consists of JurassicParkLocation, JurassicParkGameMap, Growable, DroppableFruitGrowable, Tree and Bush. For picking fruit, see PickFruitAction in Action section.

Hungry Dinosaur: Dinosaur abstract class, HungerBehaviour and its extended class

Brachiosaur: Dinosaur abstract class, Brachiosaur (Brachiosaur only extends HerbivoreDinosaur and implement abstract method of Dinosaur)

Breeding: Dinosaur abstract class, Behaviour section: BreedBehaviour, Action section: BreedAction

Eco points and purchasing: Item section: Purchasble, and all the item listed that are purchasable (They are all at the last section), VendingMachine (All purchase related, expect for VendingMachine)
For earning ecopoint: See their respective classes that gains us ecopoints:
Tree (producing fruit), PickFruitAction (picking fruit), FeedAction (feeding fruit), Egg (any dinosaur hatcing)

Death: Dinosaur (checking if it's dead from hunger, telling how long a corpse should last), DieFromHungerAction, AttackAction and EatPreyAction (To make it dead and check death from attack), Corpse (The product of being dead: a Corpse, and also corpse removal time)

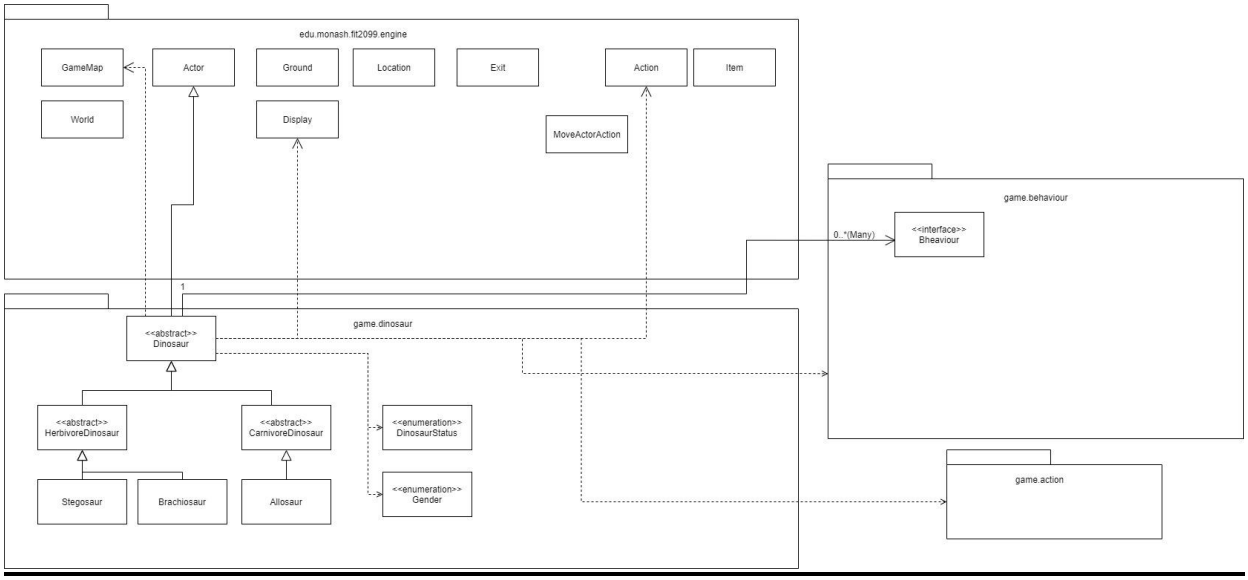
Note: Hungry Dinosaur, Breeding and Death (Check Death) Part 1

Note: Stegosaur , Brachiosaur

Note: Allosaur Part 1

For this first part, we are going to talk about the Dinosaur abstract class and its child classes, Stegosaur, Allosaur and Brachiosaur. We will not be talking about how it is going to find food and breed yet, those are to be covered in the behaviour part. We will however show how the behaviours are going to be used to achieve Hunger and Breeding mechanics.

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.

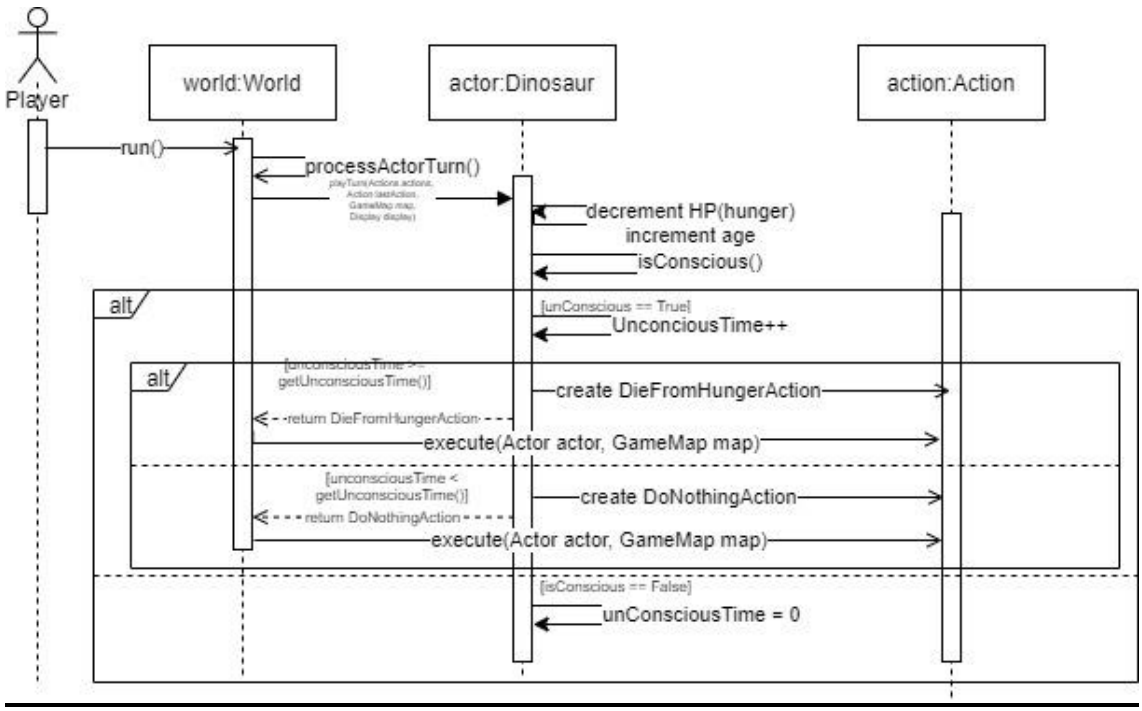
Dinosaur abstract class (extends actor)

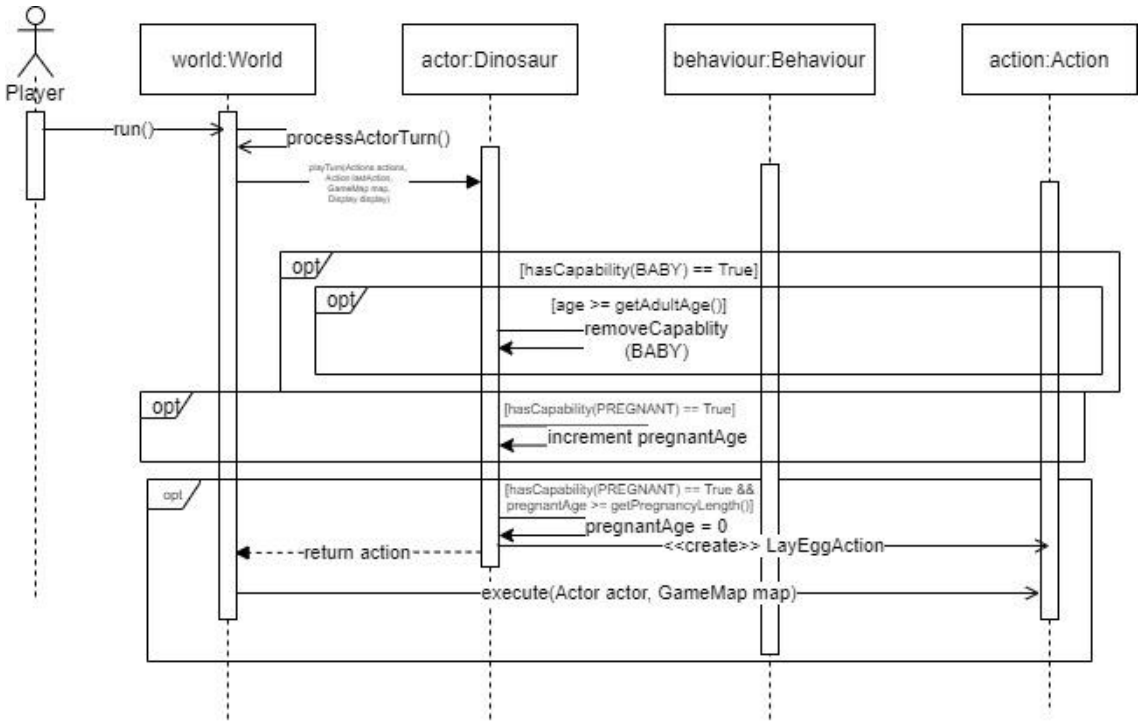
This abstract class is created so we can abstract all the details we need about a dinosaur. Anything from hunger to age, we can simply add it here. Only when we need to have specific dinosaur related stuff, we extend this class to add them. Since this class is still an Actor, we are following Liskov Substitution Principle where when the

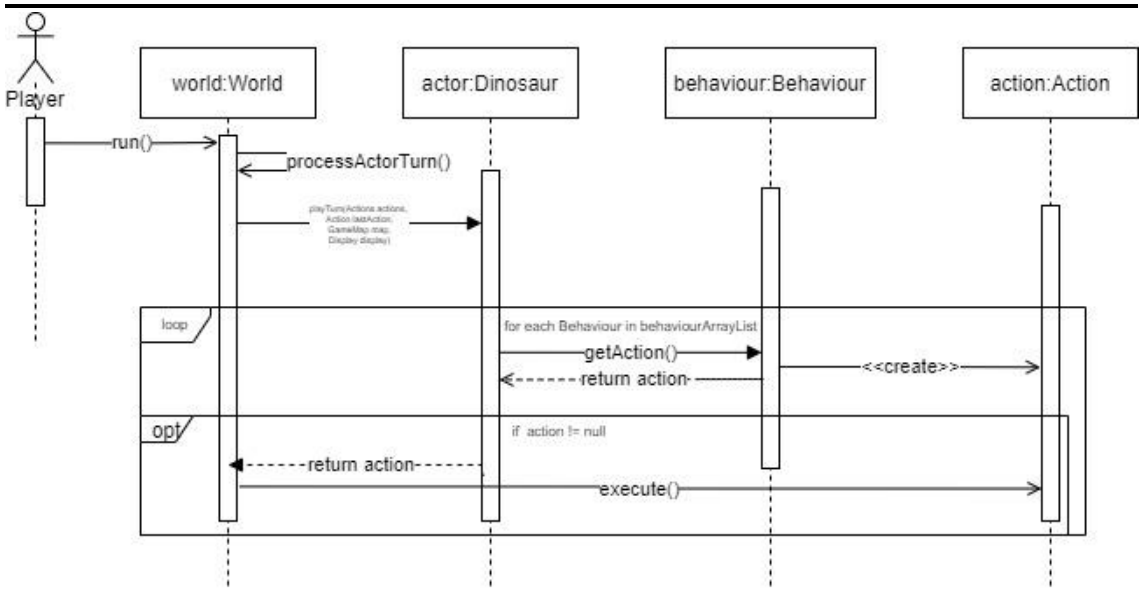
game World code is expecting an Actor, a dinosaur is still an Actor and can perform the same thing as an Actor. We therefore are not increasing the dependencies of the engine code (we can't modify it either way). We are also following the principle of Don't repeat yourself, since if we need to create 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.

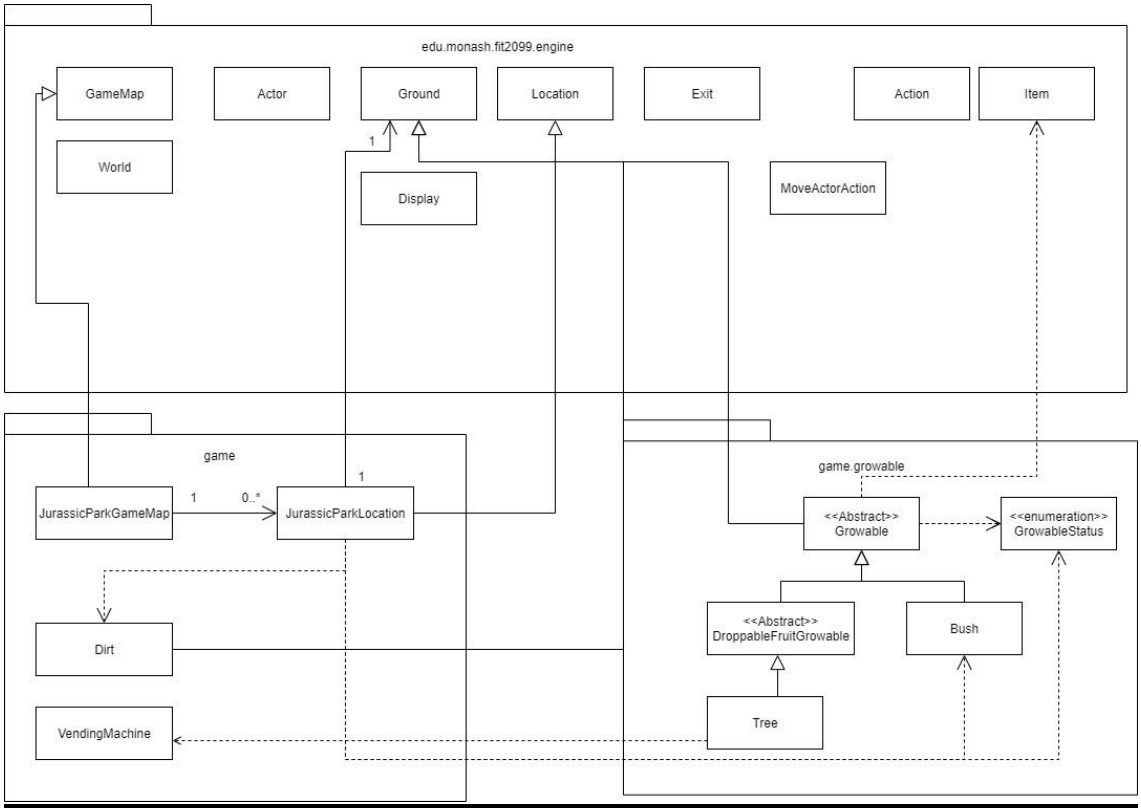
Implementation of these involve extending Herbivore/Carnivore, then implementing methods from Dinosaur (all these getter methods and other stuff), adding extra behaviours as needed (Allosaur's predator behavior of attacking anything nearby), and also adding enumerations that suits the dinosaur.

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

Note: Dirt, Trees and Bushes

For this part, we are going to talk about the implementation of Dirt, Trees and Bushes. The fruits of trees and bushes are going to be implemented as integer. Only once they get picked or dropped it becomes an item. For fruit related document, see further below.

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.

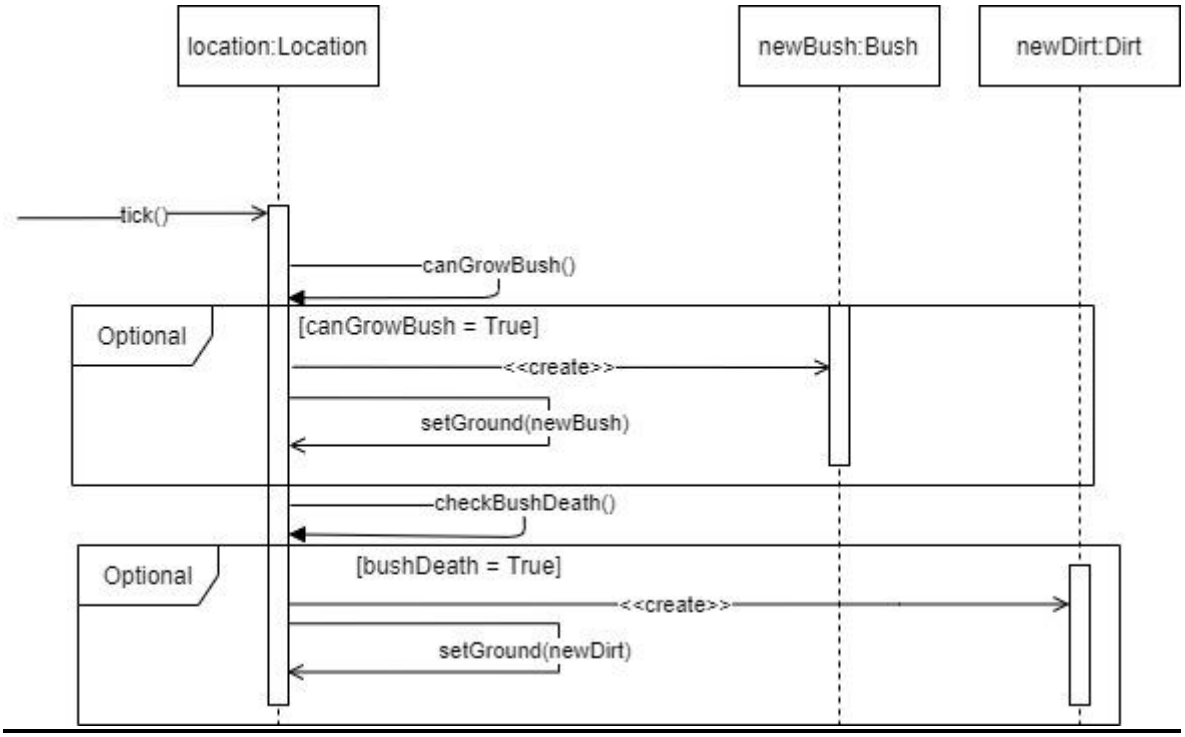
Since JurassicParkGameMap and JurassicParkLocation are extended from their respective parents, this is following Liskov's Substitution Principle, and doesn't increase dependency.

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, DroppableFruitGrowable 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). Implementations in a nutshell just involves abstract getFruitGrowthChance, checkFruitGrowth and overriding tick to use these methods, as well as having a number of fruit integer

DroppableFruitGrowable will handle any Growable that has Fruits that drops from it. This follows Don't repeat yourself principle so that other growable that needs fruit dropping just needs to extend this class. It is still a Ground object so it follows Liskov Substitution Principle. Implementations involves checkFruitDrop and FruitDropChance, overriding tick to use these methods.

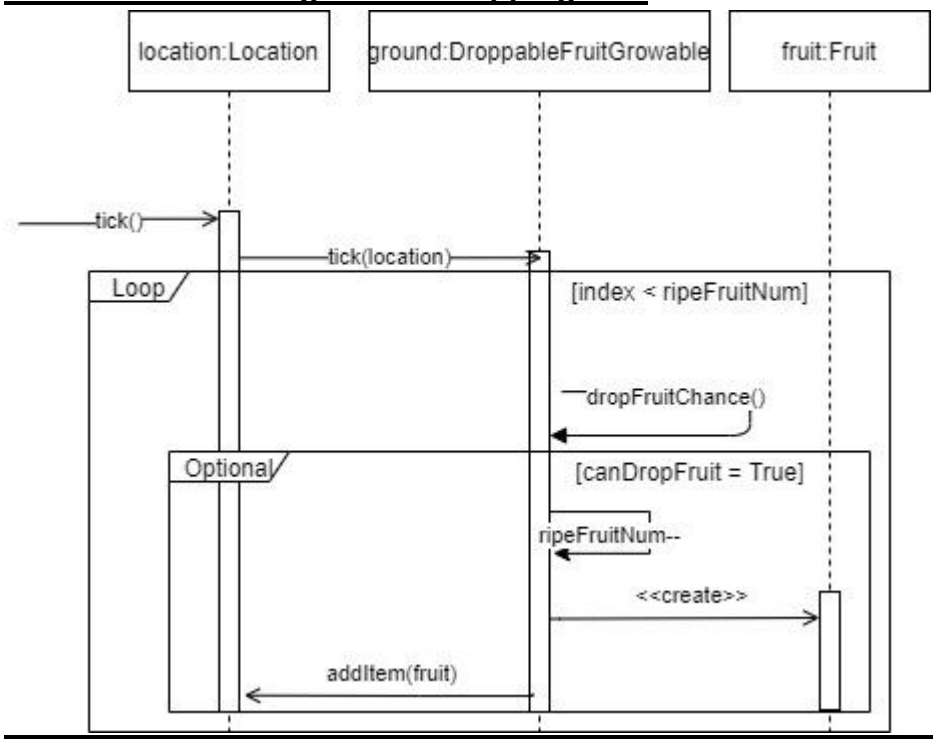
Bush and Tree

To create the bush and tree we need, all we need to do is extend Growable or DroppableFruitGrowable 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 before 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.

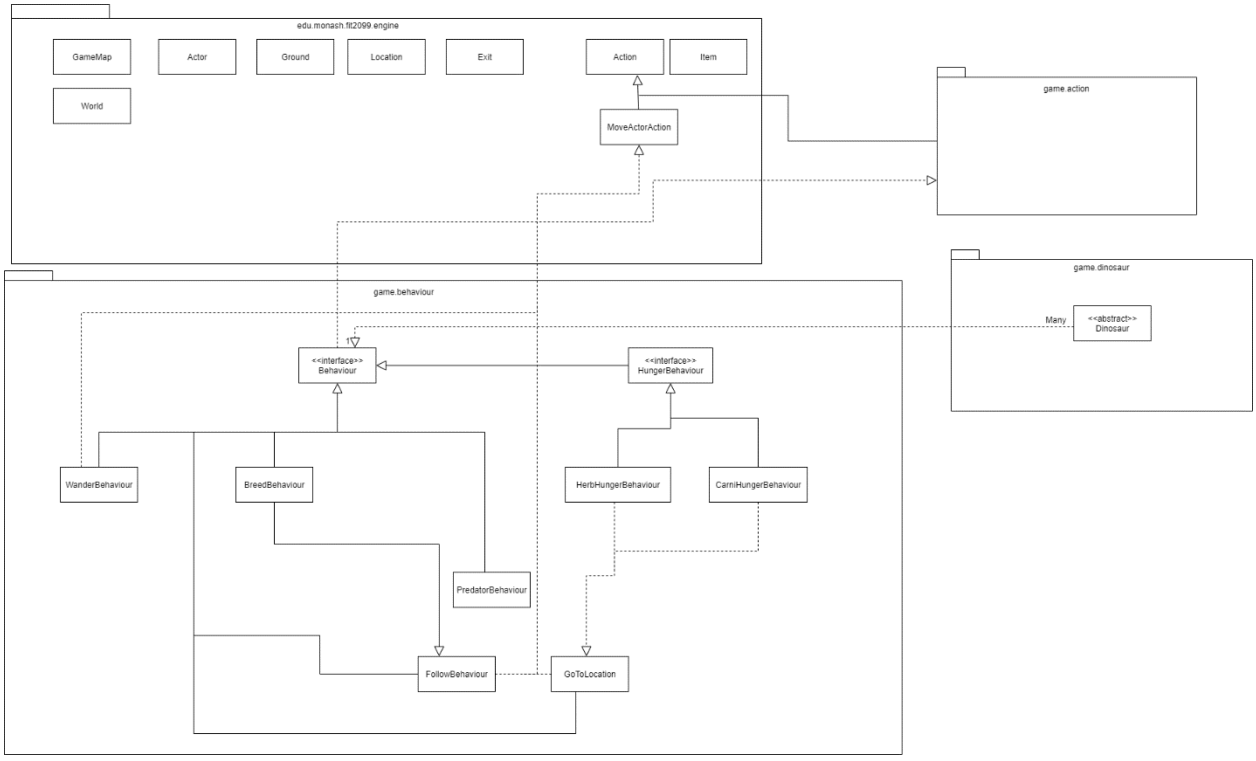
Tree interaction diagram for dropping fruit



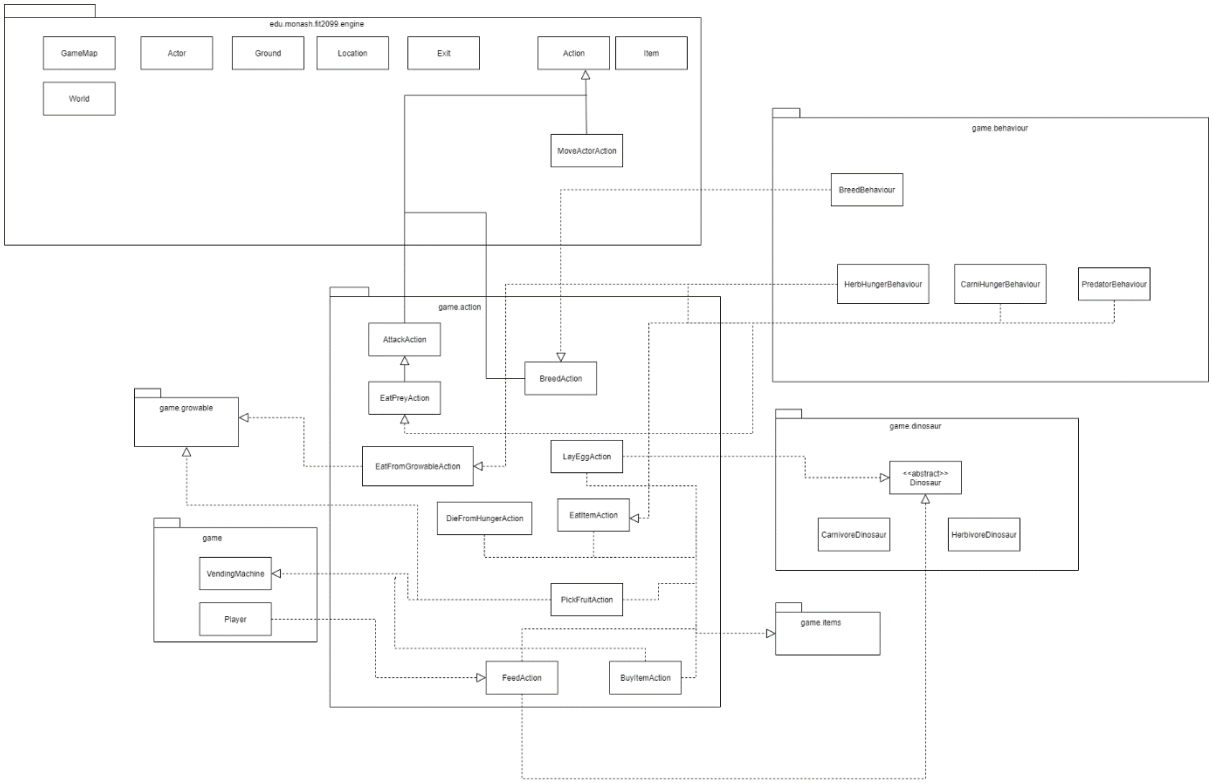
Behaviours and Actions: Hungry dinosaurs and Breeding Behaviour

Note: Hunger and Breeding

Class Diagram of Behaviours



Class Diagram of Actions



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.

All Actions extends the Action class. This means that for all the Actions that we create, we just need to follow the same method name when an Actor calls it.

Following an Actor and Going to a Location

In order to chain movements, we make use of the already built-in FollowBehaviour for an Actor to follow another Actor. However, there are also situations where we want an Actor to go to a specific Location, namely towards a Growable or Item, which do not move. Hence we need another such Behaviour called GoToLocation since even though both chains the Actor's action to move, FollowBehaviour follows a moving target but GoToLocation only brings the Actor towards a certain place, hence if we only use FollowBehaviour for this, we will break the Single Responsibility Principle (SRP)

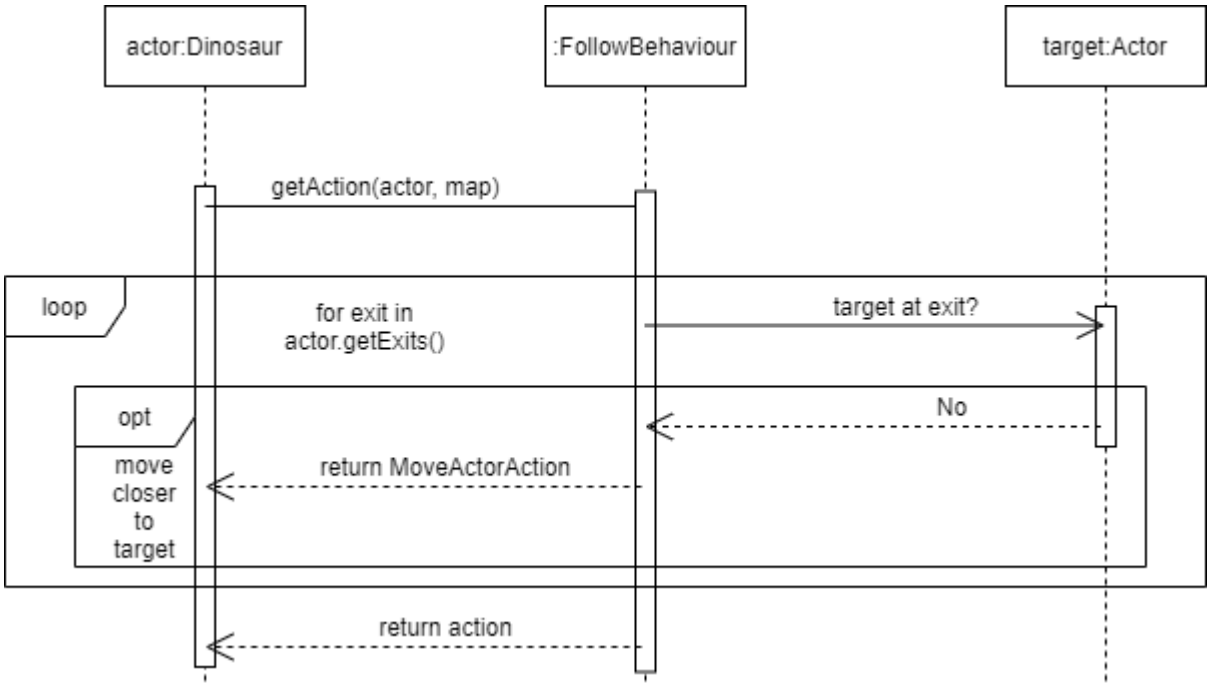
FollowBehaviour implements Behaviour

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.

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 inputted in the constructor when it is adjacent to its target.

Interaction Diagram of FollowBehaviour

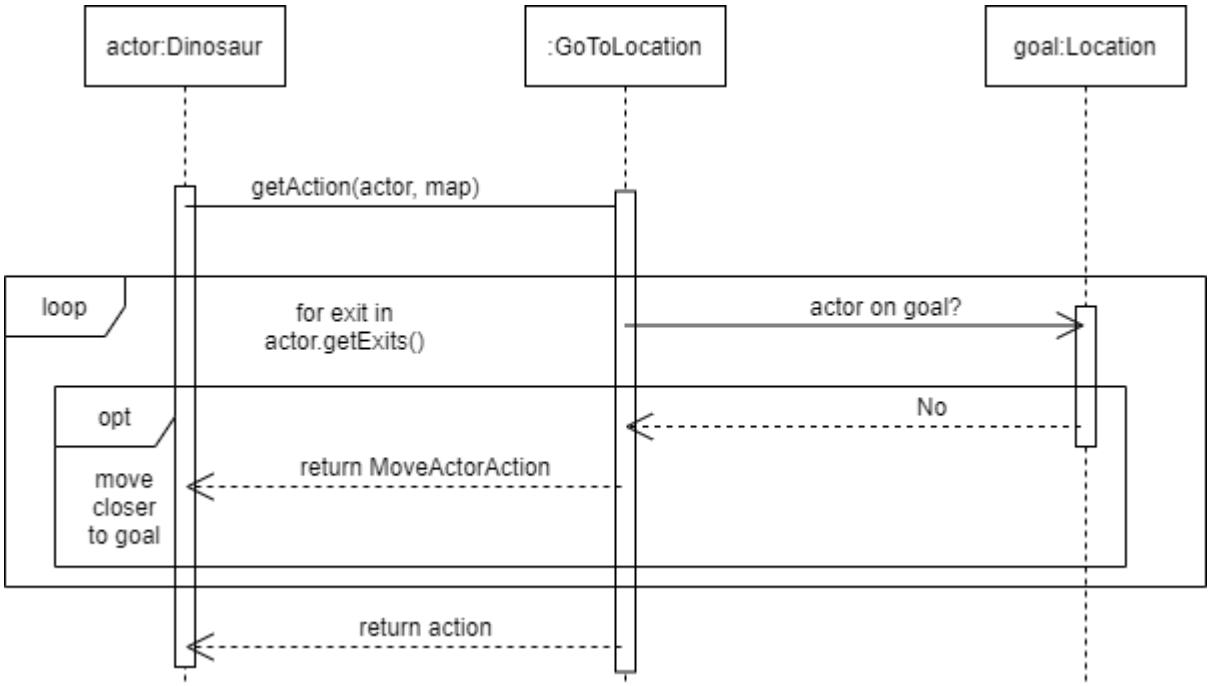


GoToLocation implements Behaviour

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. Hence we will use a difference class called GoToLocation to deal with this so FollowBehaviour won't have too many features to deal with in order to maintain SRP. Similarly, we will keep

overriding getNextAction() of the MoveActorAction until the Dinosaur is on the Location, then return the Action inputted in the constructor.

Interaction Diagram of GoToLocation



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/GoToLocation'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/GoToLocation)

```
if (newDistance < currentDistance) {
    return (new MoveActorAction(destination, exit.getName()){
        @Override
        public Action getNextAction() { return self.getAction(actor, map); }
    });
}
```

Hunger related Behaviours and Actions

When a Dinosaur is hungry, it will start looking for food on the map. Different Dinosaurs have different appetites and different types of food. Hence, we need Behaviours to help differentiate what a Dinosaur can and cannot eat, and then proceed to let them eat something based on different eating Actions.

public interface HungerBehaviour extends Behaviour

After checking if a Dinosaur is hungry, we will need to let it find food if it is indeed hungry. Since there are different types of HungerBehaviour checking for different types of food, it may get complicated if they all had different methods.

This interface serves an abstract method for HerbHungerBehaviour and CarniHungerBehaviour called findFood, while extending the other method getAction from Behaviour. This is so that when we ever decide to make the Dinosaur findFood outside of this Behaviour, both HerbHungerBehaviour and CarniHungerBehaviour will share the same name for method findFood while having their own unique purpose. This is made with Open/Closed Principle (OCP) and Interface Segregation Principle (ISP) in mind, since only hunger related Behaviours need the findFood method while other Behaviours don't, and since all Behaviours need the getAction method, we can just extend Behaviour to HungerBehaviour.

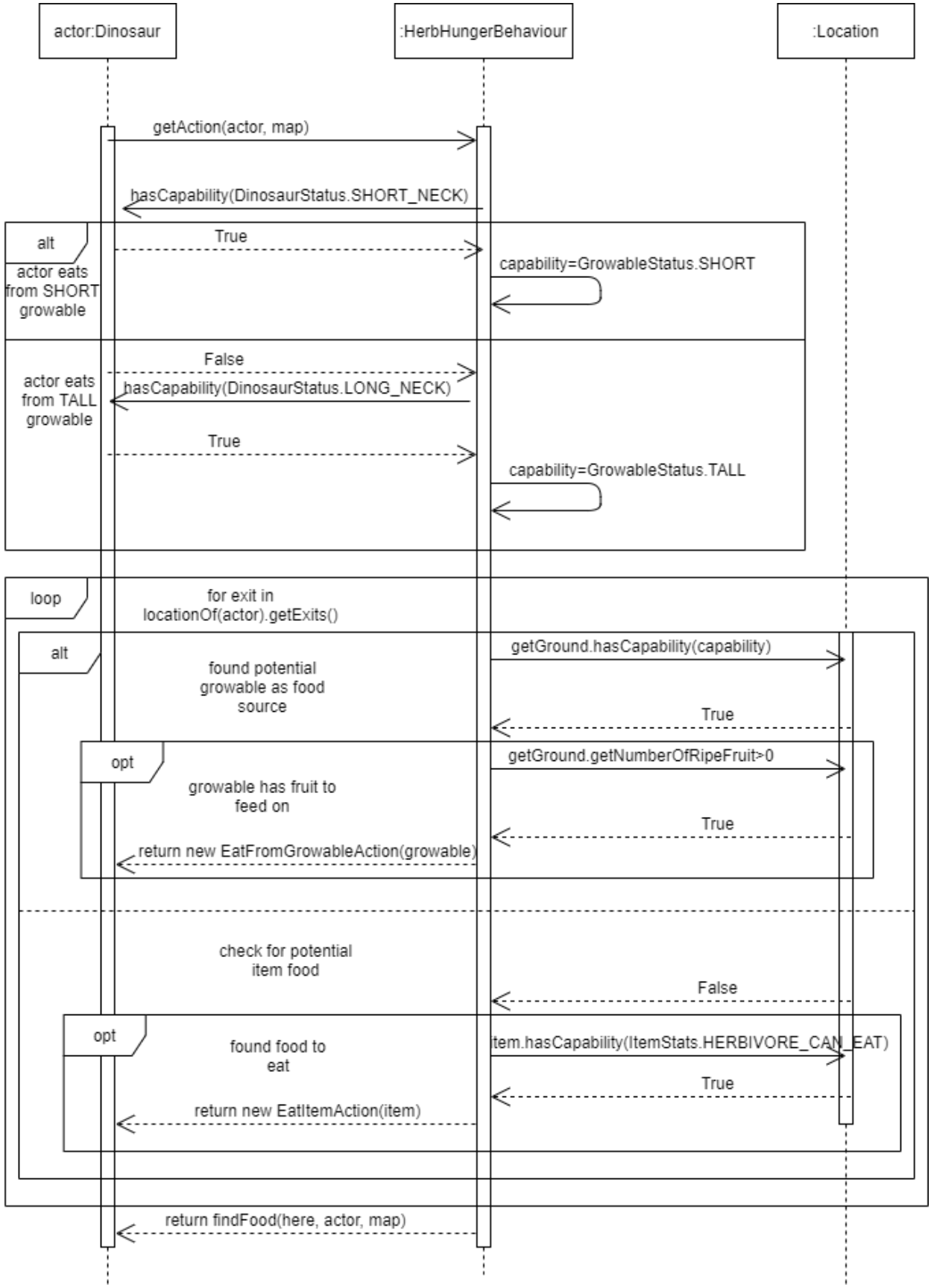
HerbHungerBehaviour implements HungerBehaviour

This Behaviour serves as a condition checker for HerbivoreDinosaurs for every turn whether it wants to eat or not. For every turn if this HerbivoreDinosaur does not already have an objective and wants to eat something, this Behaviour will help it find a food to go to, or eat it immediately if the food is beside the HerbivoreDinosaur. Since Stegosaur and Braichosaur eat different types of food, we should make use the hasCapabilities method to specify which food this Dinosaur is looking for. In HerbHungerBehaviour, it will check whether this HerbivoreDinosaur is a Stegosaur or Braichosaur by looking for their unique enums SHORT_NECK for Stegosaur and LONG_NECK for Braichosaur, and store the type of Growable the Dinosaur feeds from, SHORT for Stegosaur and TALL for Braichosaur. This is to prevent excess usage of the instanceof condition check as much as possible since in the future there may be other Dinosaurs that feed from similar food source as either of the current two HerbivoreDinosaurs, so that the condition checking will not be too convoluted and redundant.

This Behaviour overrides the getAction method to first check if the Dinosaur is hungry, and if so check its surrounding exits if there is a food source which the capability that can be eaten by this Dinosaur, and immediately eat the food if there is one, by using the EatFromGrowableAction if it is a Growable, or EatItemAction if it is an Item. These Actions will be further explained later.

If there are no such food in the Dinosaur's exits it will use the findFood method to look for the nearest food. This is done by overriding findFood to suit the needs of a HerbivoreDinosaur based on its capabilities, when it find the nearest food, the method will create a GoToLocation which that location and growable, and return the getAction of GoToLocation in order to chain movement until the Dinosaur reaches the Growable/Item.

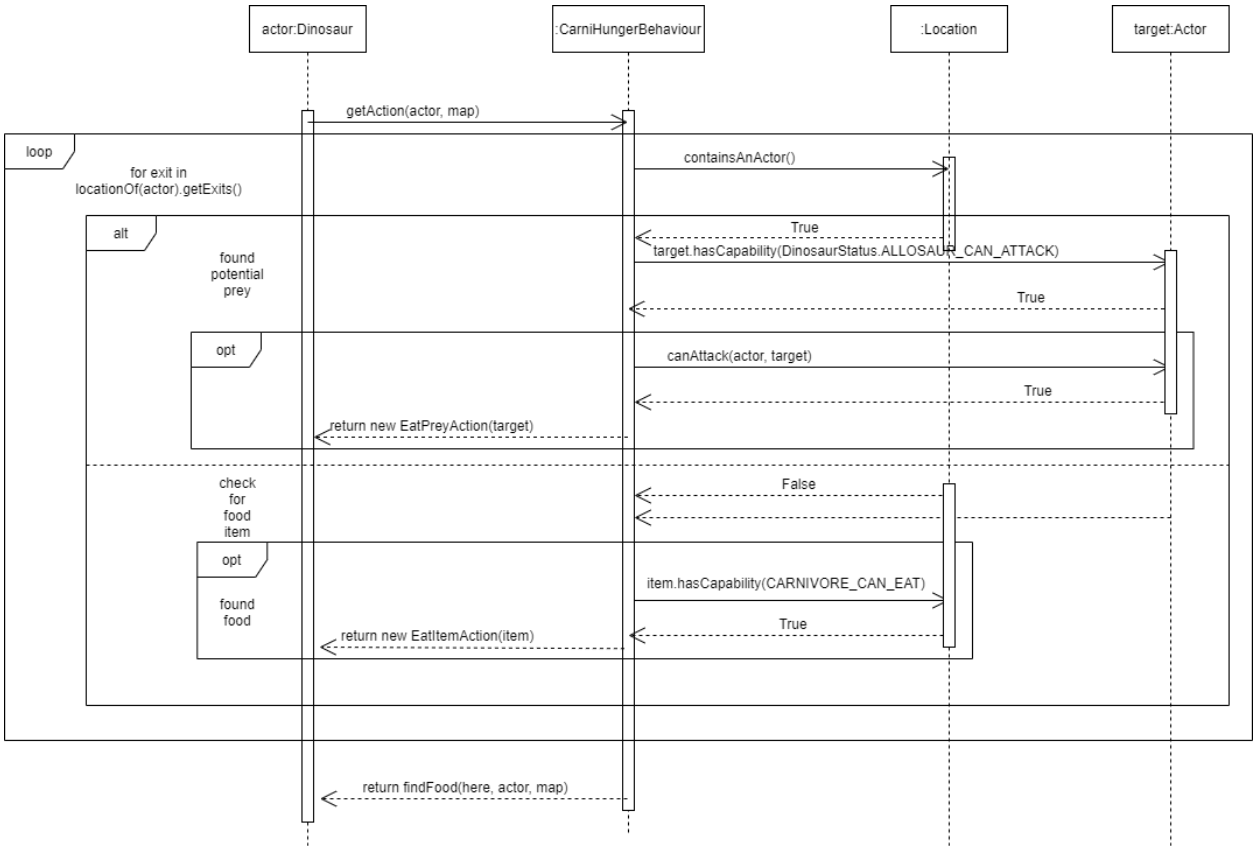
Interaction Diagram of HerbHungerBehaviour


actor:Dinosaur
:HerbHungerBehaviour
:Location
getAction(actor, map)
hasCapability(DinosaurStatus.SHORT_NECK)
alt
actor eats from SHORT growable
True
capability=GrowableStatus.SHORT
actor eats from TALL growable
False
hasCapability(DinosaurStatus.LONG_NECK)
True
capability=GrowableStatus.TALL
loop
for exit in locationOf(actor).getExits()
alt
found potential growable as food source
getGround.hasCapability(capability)
True
opt
growable has fruit to feed on
getGround.getNumberOfRipeFruit>0
True
return new EatFromGrowableAction(growable)
check for potential item food
False
opt
found food to eat
item.hasCapability(ItemStats.HERBIVORE_CAN_EAT)
True
return new EatItemAction(item)
return findFood(here, actor, map)

CarniHungerBehaviour implements HungerBehaviour

This Behaviour serves as a condition check for CarnivoreDinosaurs whether it wants to eat or not. Similar to HerbHungerBehaviour, it will check whether this CarnivoreDinosaur has not current object and is hungry, then find the nearest food possible. However, the food source is different since CarnivoreDinosaurs will attack a prey instead of feed from a Growable. Hence, we override getAction to return the suitable Action, be it EatPreyAction, EatItemAction, or create a FollowBehaviour or GoToLocation and return a MoveActorAction (FollowBehaviour/GoToLocation's getAction method) which achieves polymorphism.

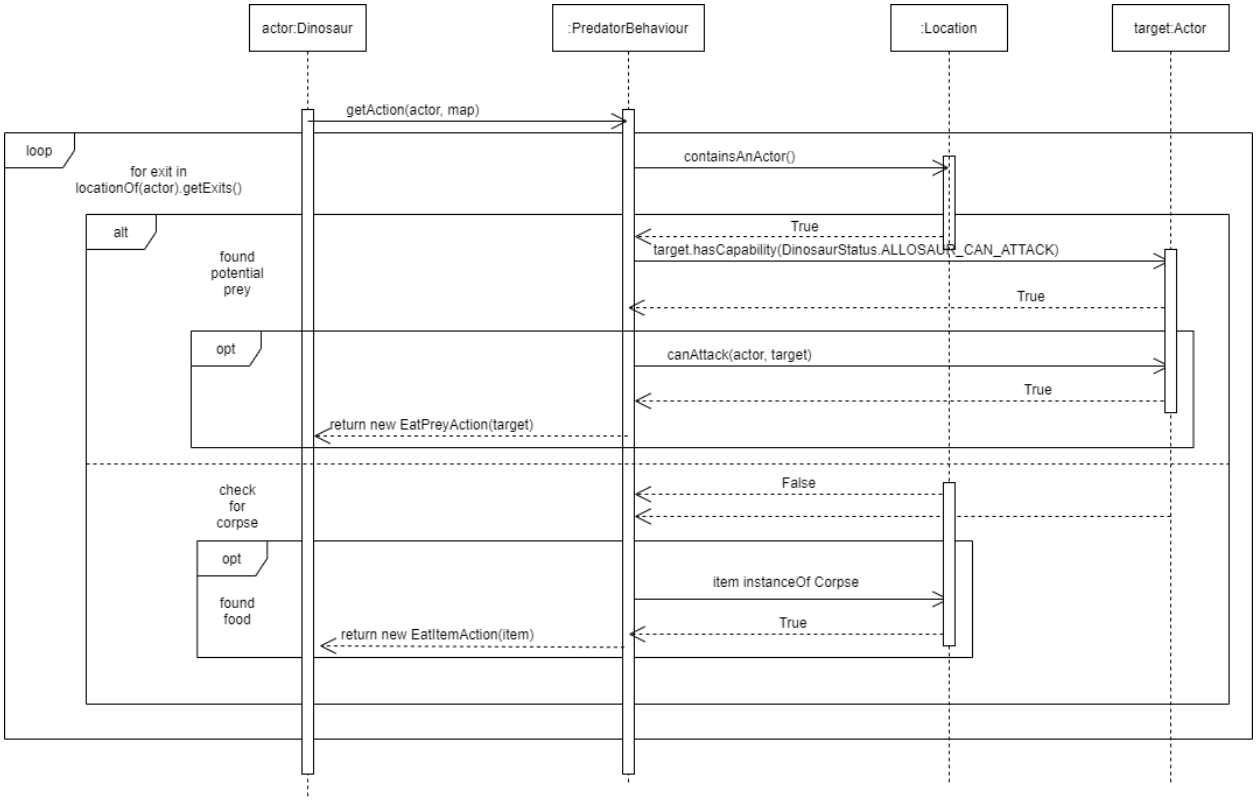
Interaction Diagram of CarniHungerBehaviour



PredatorBehaviour implements Behaviour

Even though this Behaviour also checks for suitable prey and corpse in the CarnivoreDinosaur's surroundings like CarniHungerBehaviour, this Behaviour only checks the exits of the Dinosaur, and doesn't find food if there are not such prey or corpse in its surroundings. Hence, we shouldn't use CarniHungerBehaviour for this so that SRP is maintained. If there is a prey or corpse in one of the exits, this Behaviour will create a new EatPreyAction/EatItemAction for the CarnivoreDinosaur to eat.

Interaction Diagram of PredatorBehaviour



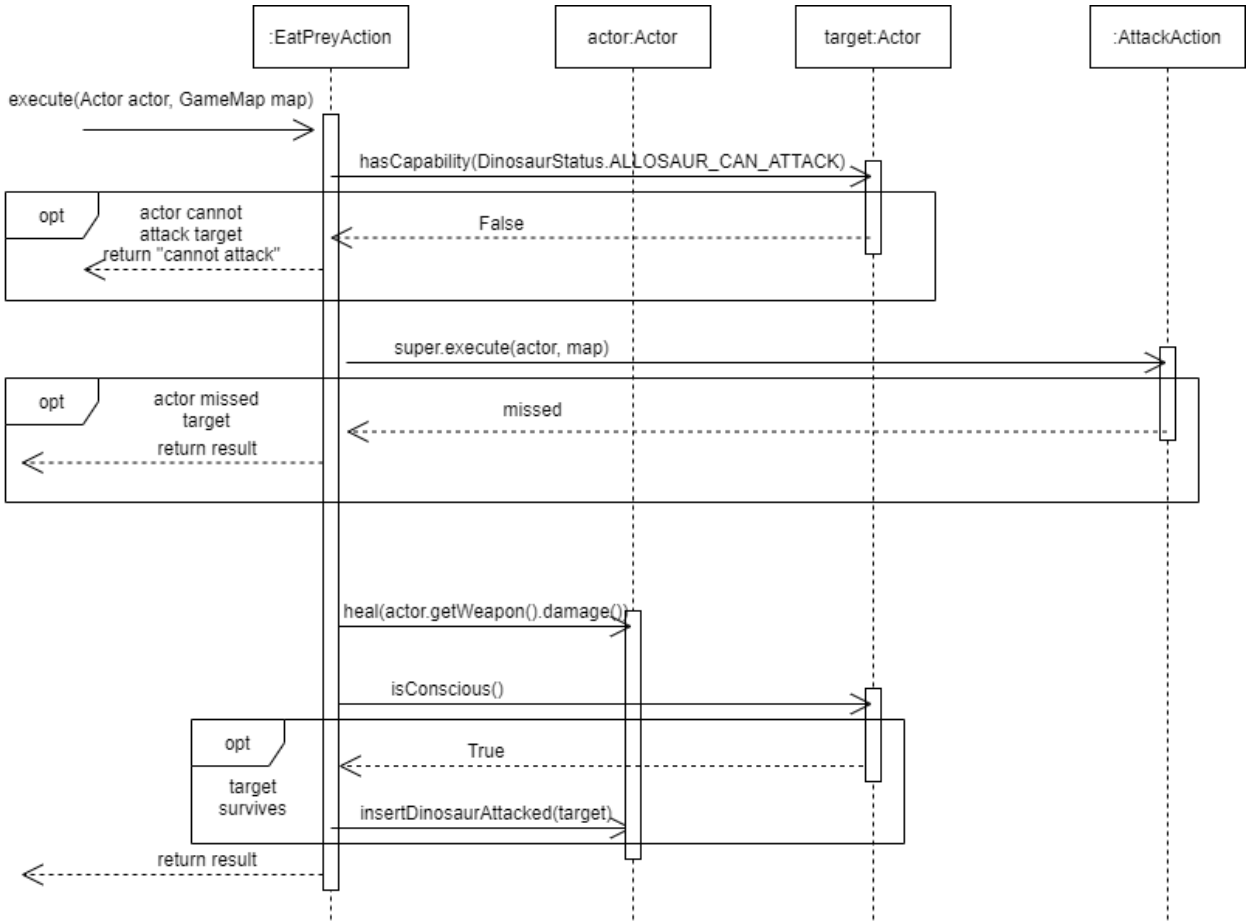
Hunger and Feeding related Actions

When the Dinosaur wants to eat something, it won't just go towards a food and stand there, it will try to eat it. Similarly the Player can also feed a Dinosaur with the appropriate food in hand. These Actions all extend the Action class so that they use the same method name but does different things. For eating related Actions, there are three such Actions: EatPreyAction, EatFromGrowableAction, and EatItemAction. Even though these Actions ultimately have the same objective which is to heal the Dinosaur, their process of healing are all different. In order to avoid breaking SRP, having the Actions divided as such is better. As for FeedAction, all it does is heal the target by the Item's amount regardless of which Actor or Item is it, so just having FeedAction for the Player is fine since we do not want to over-focus of SRP.

EatPreyAction extends AttackAction

This Action allows a Dinosaur to attack a target and heal in the process. Instead of extending from Action, we should extend from AttackAction instead. This is because EatPreyAction partly does the same thing as AttackAction, which is to attempt to hurt its target. This is a chance to prevent repetitive code, so we will call its super to attack the target first before healing the attacker. This achieves OCP since we are only adding new features to EatPreyAction instead of writing the same code as AttackAction in the first half of EatPreyAction.

Interaction Diagram of EatPreyAction



EatFromGrowableAction extends Action

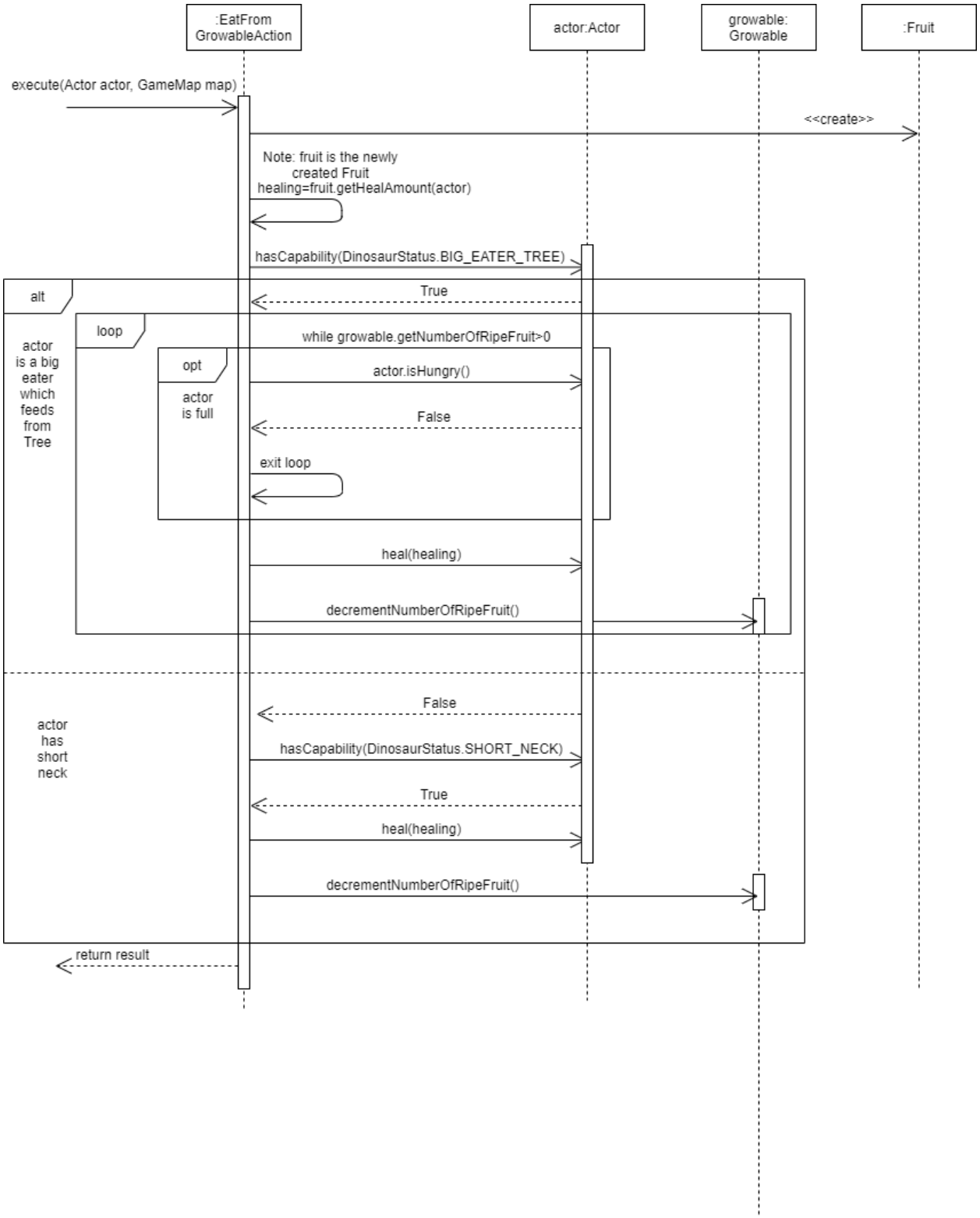
This Action allows a Dinosaur to feed from a Growable and heal accordingly, and then decrement fruit count of that Growable by the number of fruits eaten in one turn. Since all Dinosaurs that uses this Action eats Fruit, we can just heal them based on the healAmount of Fruit. This is so that we don't have to hardcode the healing values inside this Action as the healing amount is already decided based on the Dinosaur's capabilities.

Brachiosaur

Brachiosaur has the ability to eat as many fruits in one turn, with the trade-off that it has bad digestion which heals less, hence we can check if Dinosaur has the BIG_EATER_TREE enum to allow this Dinosaur to eat as many fruits as it wants, then heal it using the Fruits getHealAmount, which will have already knew that this Dinosaur has BAD_DIGESTION, hence healing the correct amount.

Other Dinosaurs will eat only one fruit per turn, so we just have to check whether it has the SHORT_NECK enum since the only other HerbivoreDinosaur right now is the Stegosaur.

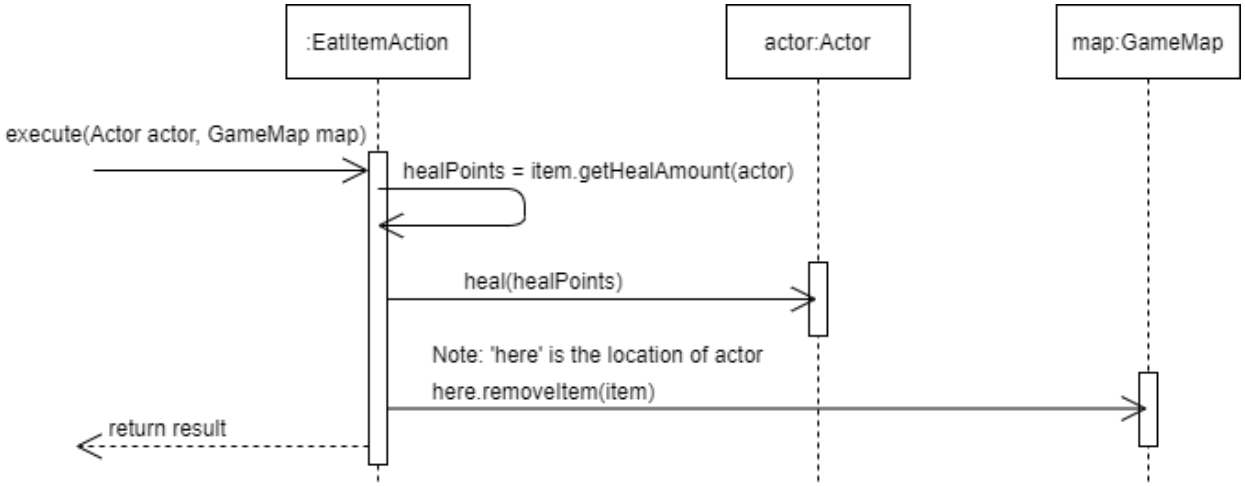
Interaction Diagram of EatFromGrowableAction



EatItemAction extends Action

This Action allows a Dinosaur to eat an Item on the Ground. This Action takes an Item as an input and casts it into an EdibleItem so that it can get the heal amount of this Item. We don't need any condition checking here because the checking part is done in the Dinosaur's respective HungerBehaviour, and all this class does it heal the Dinosaur by the item's healAmount and remove the Item from the map.

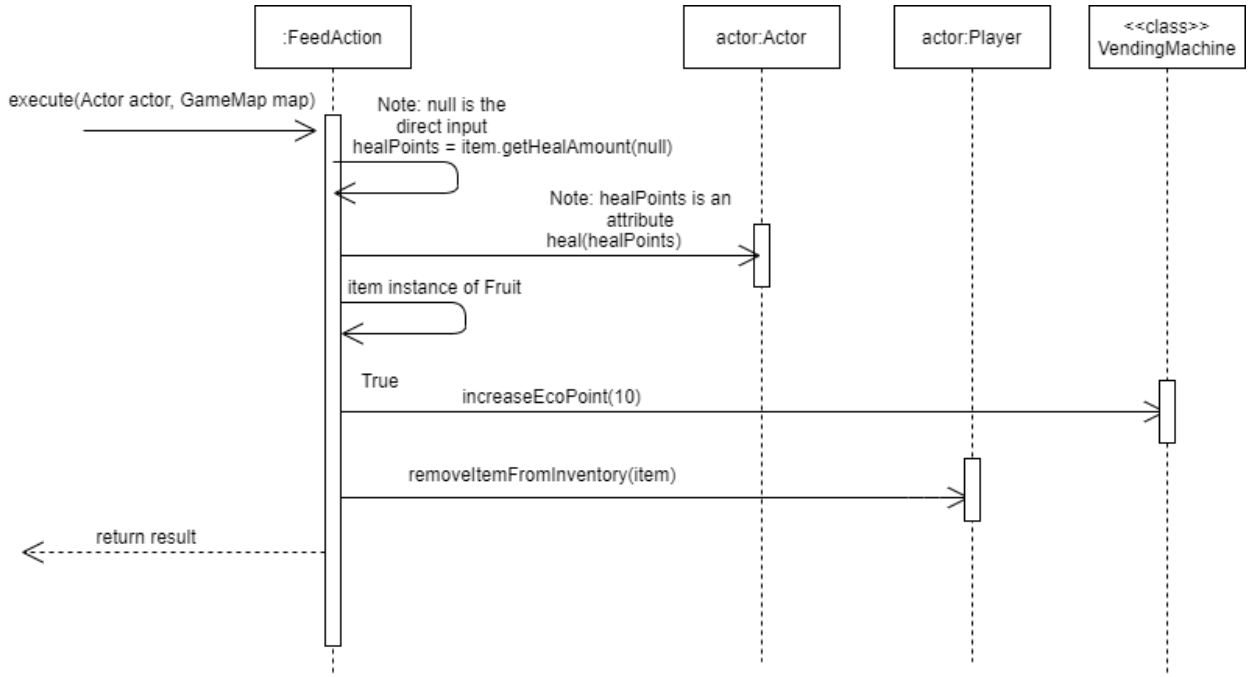
Interaction Diagram of EatItemAction



FeedAction extends Action

This Action is only available to the Player if and only if they are beside a Dinosaur and has an Item that can heal that Dinosaur. Player gets this option from the Dinosaur's getAllowableActions, which they will be able to see it if this Actions menuDescription is shown on the menu. This Action does a similar job as EatItemAction except that it removes the Item from Player's inventory instead of from the map. Hence we have to keep them separate so as to not break SRP. Additionally, if the item Player fed is a Fruit, the Player gains 10 EcoPoints. Since only by feeding Fruit will this happen, we can use instance of since this is unique to this one class.

Interaction Diagram of FeedAction



Breeding

If a Dinosaur is well fed, it may want to breed with another Dinosaur of the same species. We will use the BreedingBehaviour to do the checking and BreedAction to execute the breeding process.

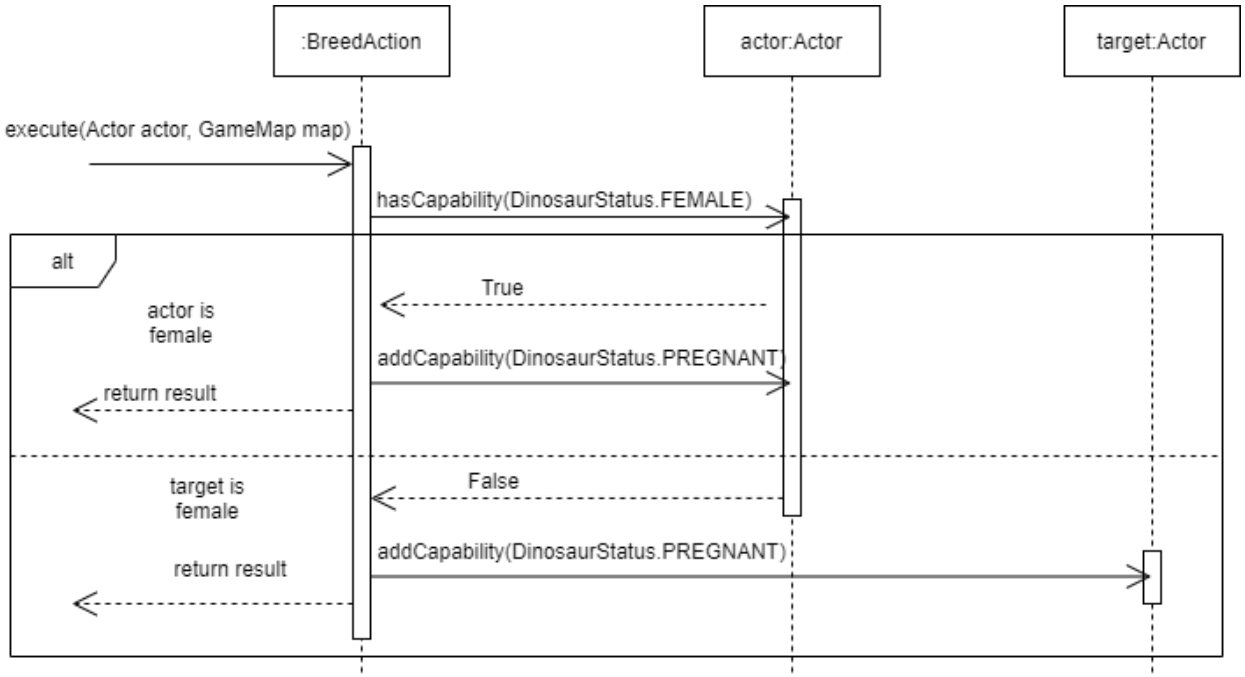
BreedBehaviour implements Behaviour

This Behaviour checks if the Dinosaur is well fed and then finds a potential partner within its 3-tile radius. If it finds a partner, it will go to it. If the partner is beside the Dinosaur, they will immediately mate for that turn. This Behaviour makes use of FollowBehaviour's getAction for the Dinosaur to approach the partner, and BreedAction to breed. As always, we will reuse methods from the Dinosaurs and also using their enums to do most of the checking so as to reduce repeated code. However, the part where we check for different Gender is necessary, so we have no choice but to do it manually.

BreedAction extends Action

This Action allows the Dinosaur to breed with its target and make the Dinosaur or the target pregnant, whoever is the female here. This Action also needs to check who is the female and whether they are already pregnant, which we can readily decide in BreedBehaviour since anything can happen while the two Dinosaurs approach each other. Hence we will have to do the condition checking again to prevent the code from breaking.

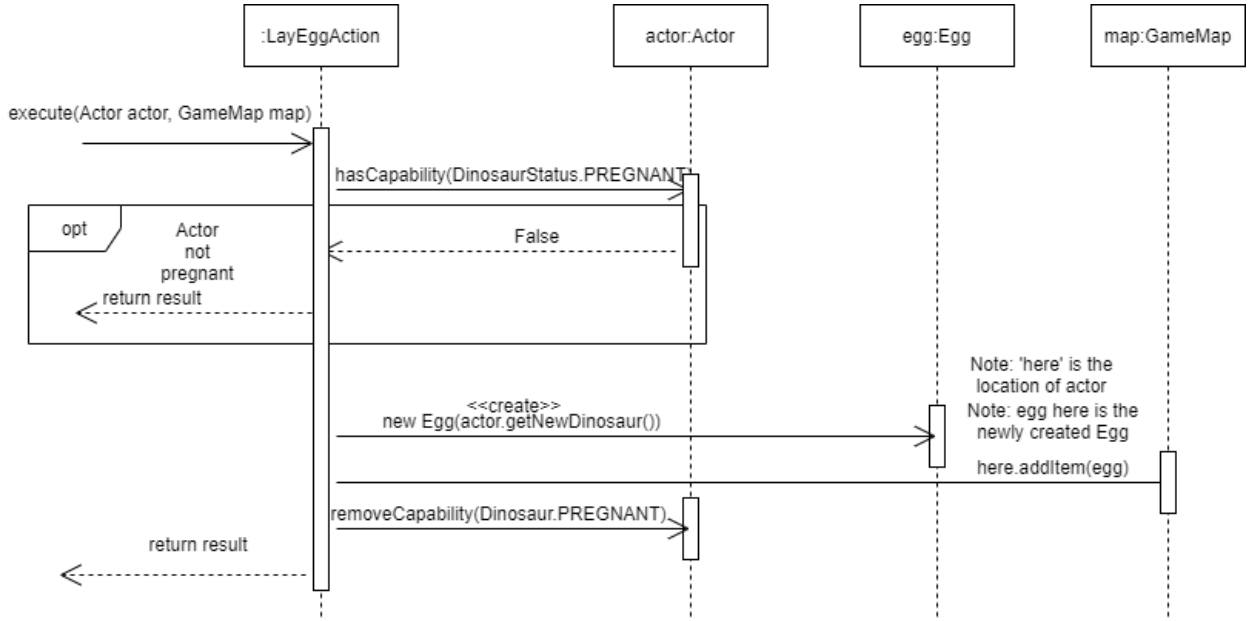
Interaction Diagram of BreedAction



LayEggAction extends Action

This Action allows a pregnant Dinosaur to lay an egg which will hatch a baby dinosaur of this Dinosaur's species. In order to do so, we will cast the input actor in execute to a Dinosaur, use the getNewDinosaur method to create a new Dinosaur of this Dinosaur's species, then create an Egg with this new Dinosaur. This works even we only cast the actor to Dinosaur instead of specific species because of Liskov Substitution Principle (LSP) which since the specific Dinosaur classes should also have the methods in the abstract Dinosaur class thanks to polymorphism. When creating the Egg, we can also make use of this principle by creating PortableItem instead of Egg.

Interaction Diagram of LayEggAction



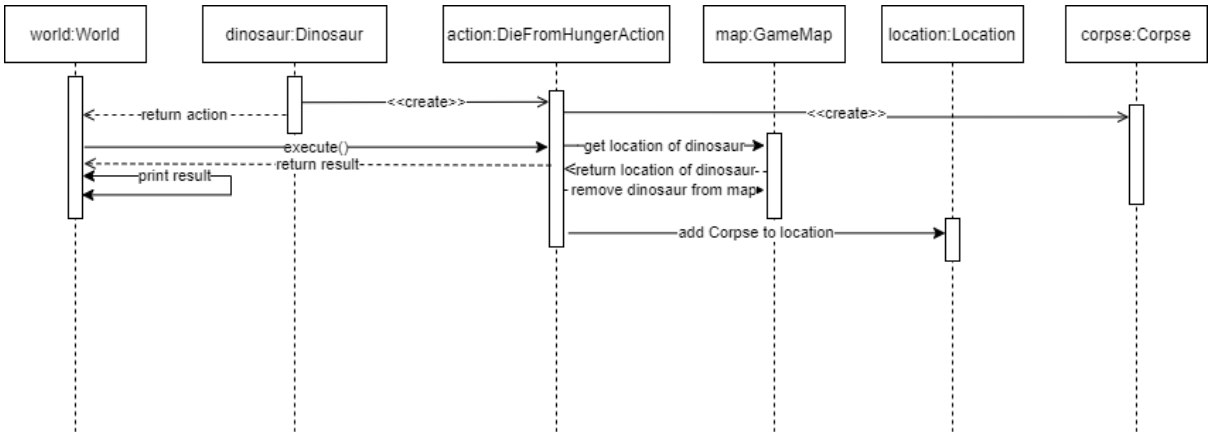
Death part 2 (Executing removal and adding corpse)

When an Actor dies they should be replaced with their corpse on the map. We will do this by implementing a special death Action.

DieFromHungerAction extends Action

This Action will be executed when a Dinosaur stay unconscious for too long and starves to death. Firstly, we should double check that this actor dying is indeed a Dinosaur. Since there are no enums shared among all Dinosaurs, we can use instanceof to check since all we need to know is that this actor is a Dinosaur. We create a PortableItem corpse (LSP) on the map with its rottime and corpsehealamount decided by the type of this Dinosaur. We will do that by using the Dinosaur's getCorpseRotTime and getCorpseHealAmount so as to not repeat ourselves and make this Action usable for all Dinosaurs. Afterwards, the Dinosaur will be removed form the map.

Interaction Diagram of DieFromHungerAction

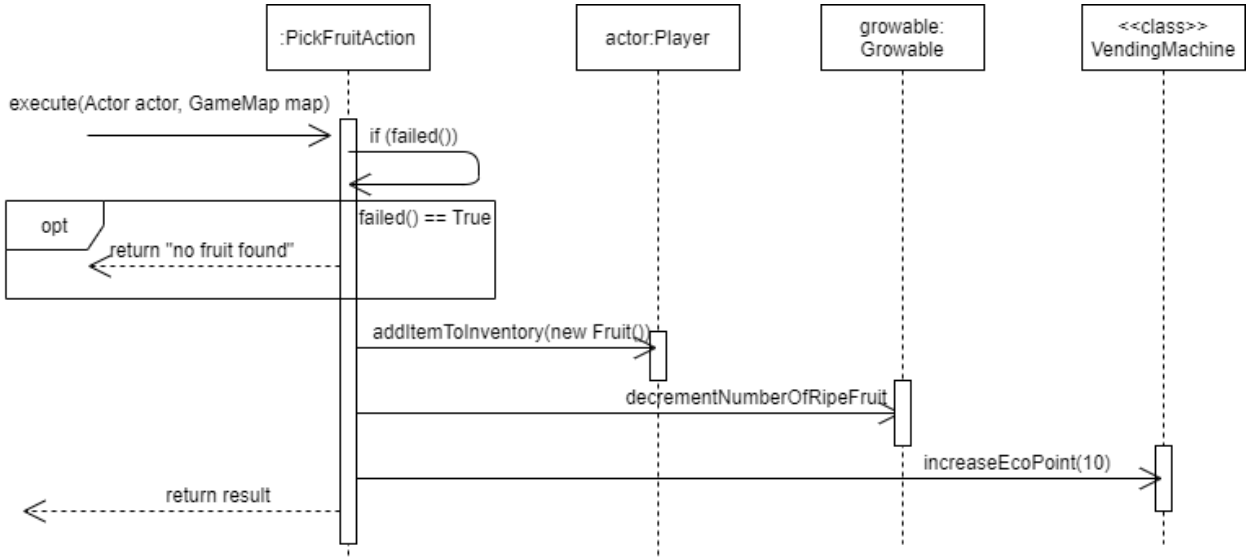


Player interacting with Growables

PickFruitAction extends Action

When a Player is standing on a Growable and it has at least one fruit on it, the Player can attempt to pick it and add to their inventory. However, they have a 60% chance of failing to find a fruit in the Growable.

Interaction Diagram of PickFruitAction



Design Rationale for Items and Vending Machine

Note: EcoPoints and Purchasing, Death Part 3 (Corpse item)

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.

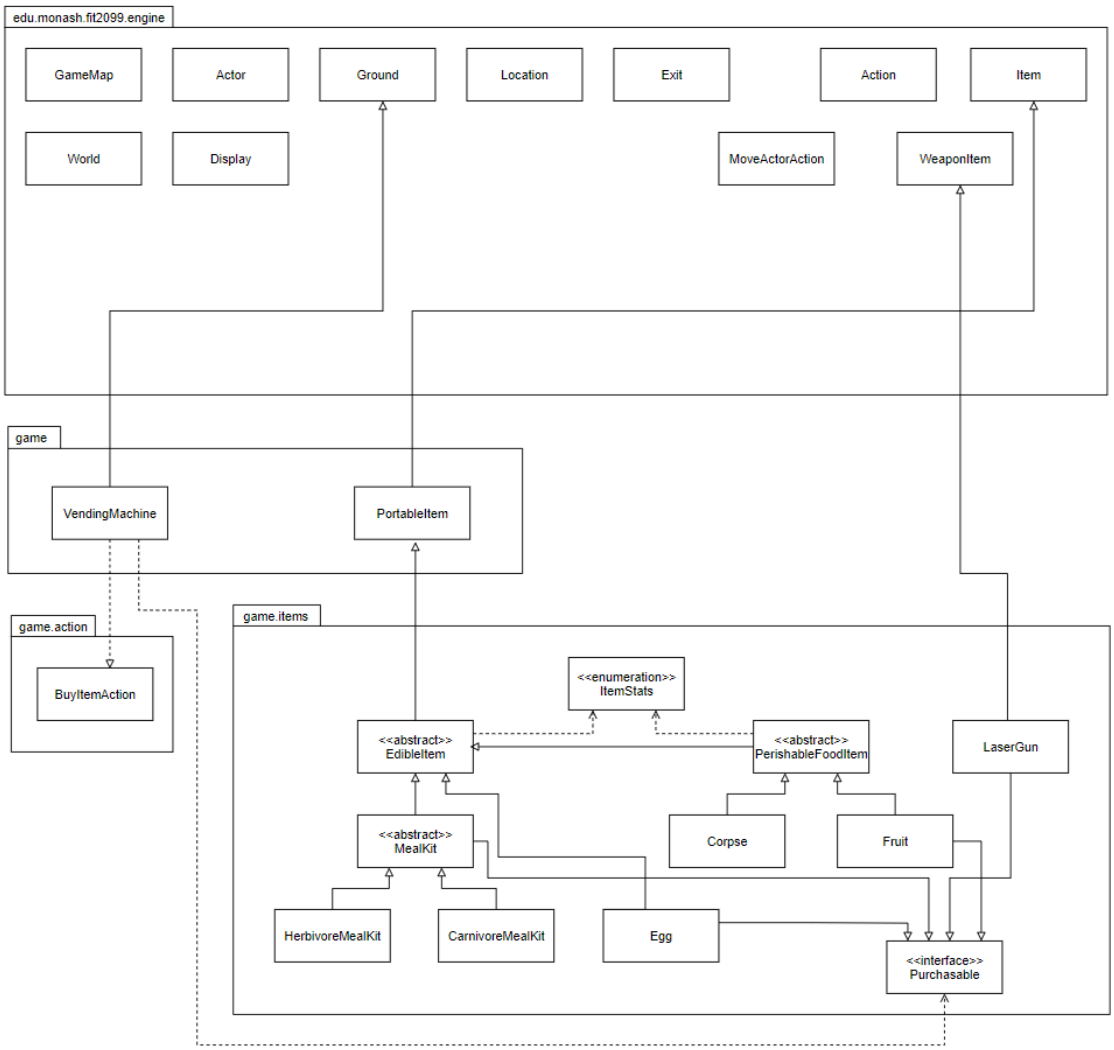
The player, if adjacent to a vending machine, prompts the vending machine to show it’s item menu, and buy items from it as long as they have enough points to do so.

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.

Design Rationale for Items and Vending Machine

Class diagram of items and VendingMachine



ItemStats enum class

With an enumeration class, we can give all items their own unique attributes and check the attributes by calling the capability interface as needed. This essentially eliminates the need for multiple dependencies for checking if an item is edible, or if the item is a carnivore only food and so on. With this in mind, unwanted dependencies are reduced and the items will depend on the enum class instead, which is the dependency inversion principle. There is an issue though as many classes other than the items will end up using this enum class. Checking class type might be implemented but it'll be a pain to refactor and add new features. Therefore the enumeration class as of now is best.

Actions for items

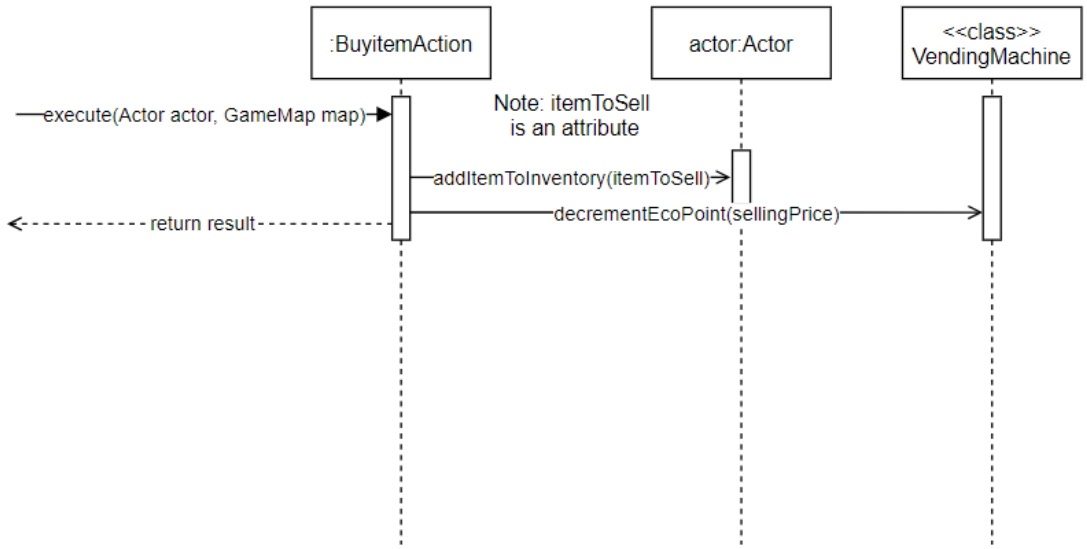
BuyItem extends Action
This class is called when a player wants to purchase items at the vending machine.

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.

The process starts by using an item menu for the player to select when adjacent to it. Each time the player wants to buy something, BuyItemAction 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.

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

Interaction Diagram of BuyItem



Items misc

EdibleItem abstract class extends PortableItem
This abstract class represents all edible items. This class is created so all details on an item can be abstracted here in one place. The principle of Don't repeat yourself is followed here as if a new item that is edible is need in the event of refactoring or addition of more items, we can just extend this class over to the new class and simply add any unique attributes that new edible item might have. A bad solution would be to create new edible items and repeat the same block of code in all the new items. This class also gets the heal amount of each edible item.

PerishableFoodItem extends EdibleItem

Similar to the EdibleItem class, PerishableFoodItem extends from it to achieved even more abstraction. This class denotes all edible food items that can rot, keeps track of and removes any rotted item after a set number of turns that the item specifies.
We can place all this information inside of EdibleItem class but this violates the Single Responsibility principle which states that every class should only have one responsibility. We are not increasing dependency as well as PerishableFoodItem is still an Item, thus Liskov's Substitution principle holds.

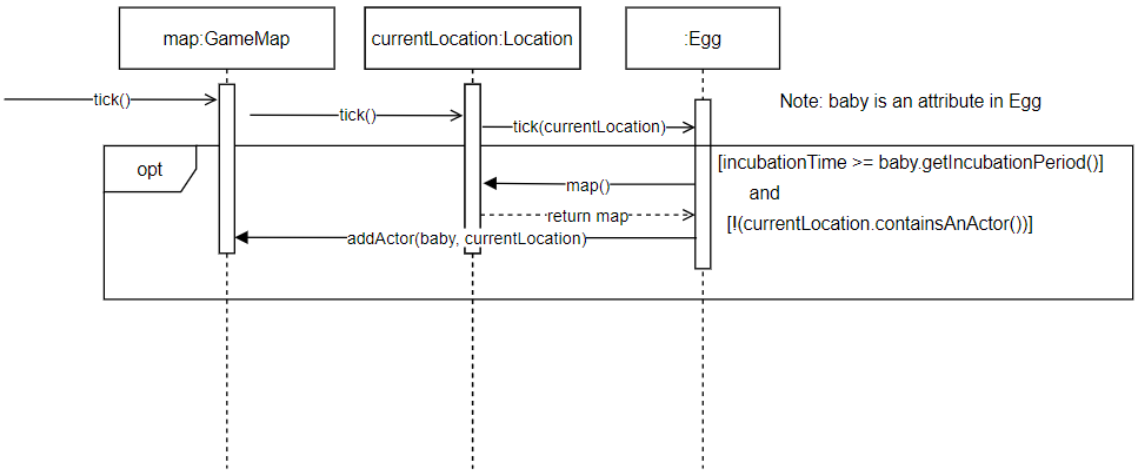
Corpse extends PerishableFoodItem

This class represents a dinosaur corpse, created when a player or an Allosaur kills a Stegosaur, or when a dinosaur is unconscious and not fed for a certain number of rounds. Food source for only the carnivore classes which is the Allosaur as of now.
Overrides super's tick() method to initiate the set rotting timeframe (e.g. 30 turns) if it is placed on the map.

Egg extends EdibleItem implements Purchasable

This class contains all current and future dinosaur eggs and their characteristics. This allows for future modifications to all eggs and lessens unnecessary code.
Fetches attributes of the eggs such as hatch time needed via the getIncubationPeriod method and type of dinosaur hatched from the Dinosaur classes. Has a heal amount of 10 points for carnivorous dinosaurs.
If any eggs are picked up and carried in a player's inventory, override super's tick() method is overridden to stop incubation or hatching.
If an egg is placed on the ground or laid by a dinosaur, override super's tick() method to start the countdown until the egg is hatched.
EcoPoints are incremented when an egg hatches. The eggs of each dinosaur are purchasable, as this class implements the Purchasable Interface.

Interaction Diagram of Egg



Fruit extends PerishableFoodItem implements Purchasable

This class represents a piece of fruit, which is produced by either a tree or bush. This is a food source of all herbivorous dinosaurs. This class also hold the heal amount for each herbivore dinosaur and the vending machine purchase price.
The fruit has a rot time for 15 turns and is a portable item. This class is a purchasable item, as Fruit implements the Purchasable Interface.

Purchasable Interface class

This class is used to represent an item that is purchasable. Any item classes that implements this will make said item purchasable. The purpose of this interface is to make it easy for any future items to be made available for purchase at the vending machine. Since the class reduces dependencies and makes use of only one function, the Single Responsibility principle holds.

MealKit abstract class extends EdibleItem implements Purchasable

This abstract class represents all meal kits, of which are edible. This class is created so all details on a meal kit can be abstracted here in one place such as the heal amount. The principle of Don't repeat yourself is followed here when a new meal kit is added, we can just extend this class over to the new class and only add in any unique attributes that new meal kit might have. A bad solution would be to create new meal kits and repeat the same block of code in all the new meal kits.
This class also gets the heal amount of each meal kit. The meal kits are purchasable items, as it implements the Purchasable Interface.

HerbivoreMealKit extends MealKit

This class represents a meal kit only for the carnivorous dinosaurs Also contains the vending machine purchase price for this item. Heals a herbivorous dinosaur to full health upon feeding from a Player.
The meal kit has no rot attribute and is portable and purchasable.

CarnivoreMealKit extends MealKit

This class represents a meal kit only for the carnivorous dinosaurs.
Also contains the vending machine purchase price for this item. Heals a carnivorous dinosaur to full health upon feeding from a Player.
The meal kit has no rot attribute and is portable and purchasable.

LaserGun extends WeaponItem implements Purchasable

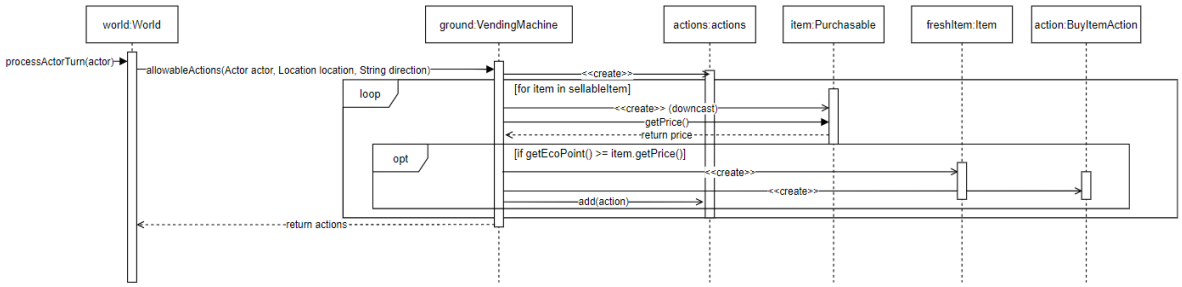
This class represents a laser weapon used to kill dinosaurs for population control and to provide easy food for the Allosaurs.
This class uses a zero parameter constructor. Also contains the price of the Laser Gun. The Laser Gun is a purchasable item, as it implements the Purchasable Interface.

Vending Machine extends Ground

This class represents the vending machine and has a stored amount of EcoPoints as a private static variable. All actions that result in EcoPoint rewards will accumulate in the private static variable. Also contains all purchasable items and fetches their prices. Players are not able to enter (step on) the vending machine and can only use the vending machine while adjacent to it.

The machine gives some options to the player to make a purchase and will continue to do so until the player walks away. If the player has enough EcoPoints, VendingMachine will create a new BuyItemAction to purchase and add the item into the player's inventory.

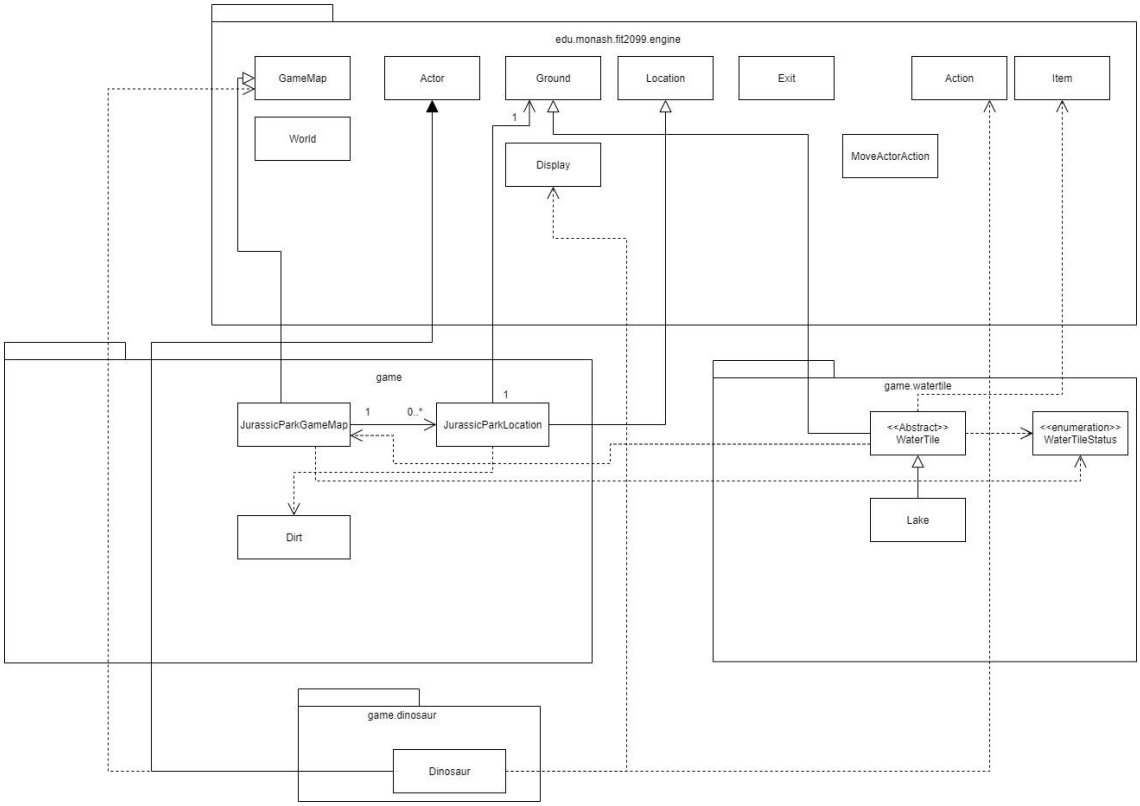
Interaction Diagram of VendingMachine



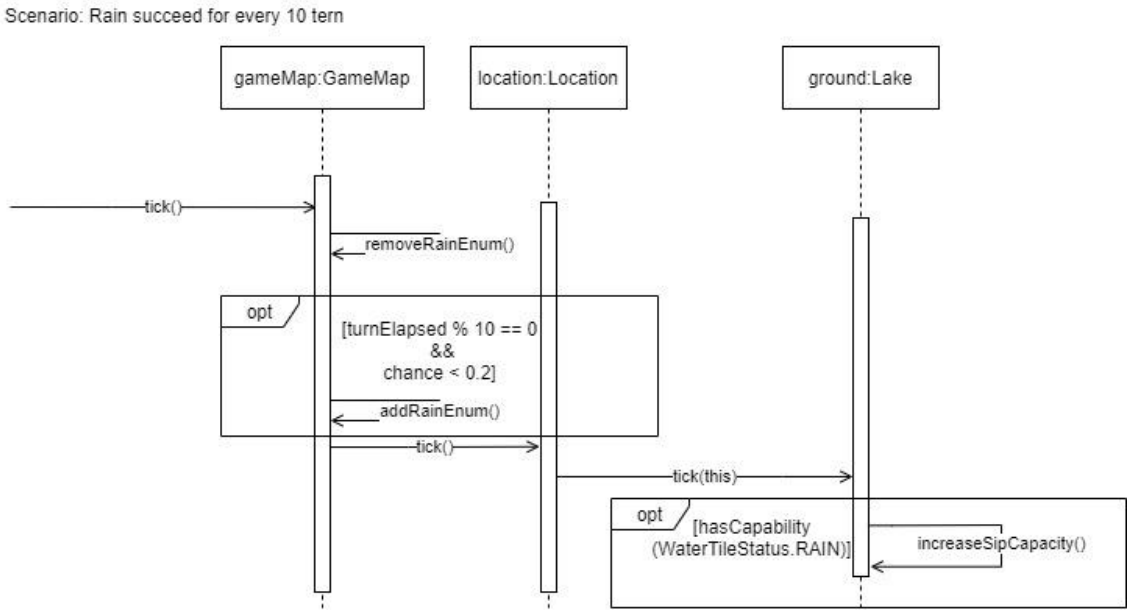
Assignment 3 Design Rationale

Lakes, water and rain

Class Diagram



Sequence diagram



This section was firstly implemented by having a **WaterTile** abstract class. This is because we might expect other tiles that may function the same (Maybe sea, river, etc). By doing this, we can potentially reduce the amount of code and thus achieve Don't Repeat Yourself.

JurassicParkGameMap extends GameMap

This is class we have added more. I have add more to the class by overriding tick() to check for rain, and to pass down the rain, I've used enums, by looping through all the location and

getting its ground to pass down the enum. This helps not increase dependency, keeps with Open/Closed Principle, since we add more instead of heavily modifying, and to check for rain we only need to check for enum, instead of doing something else like downcasting GameMap to JurassicParkGameMap. We follow the pre-existing Dependency Inversion Principle via the use to Capability ENUM for the rain.

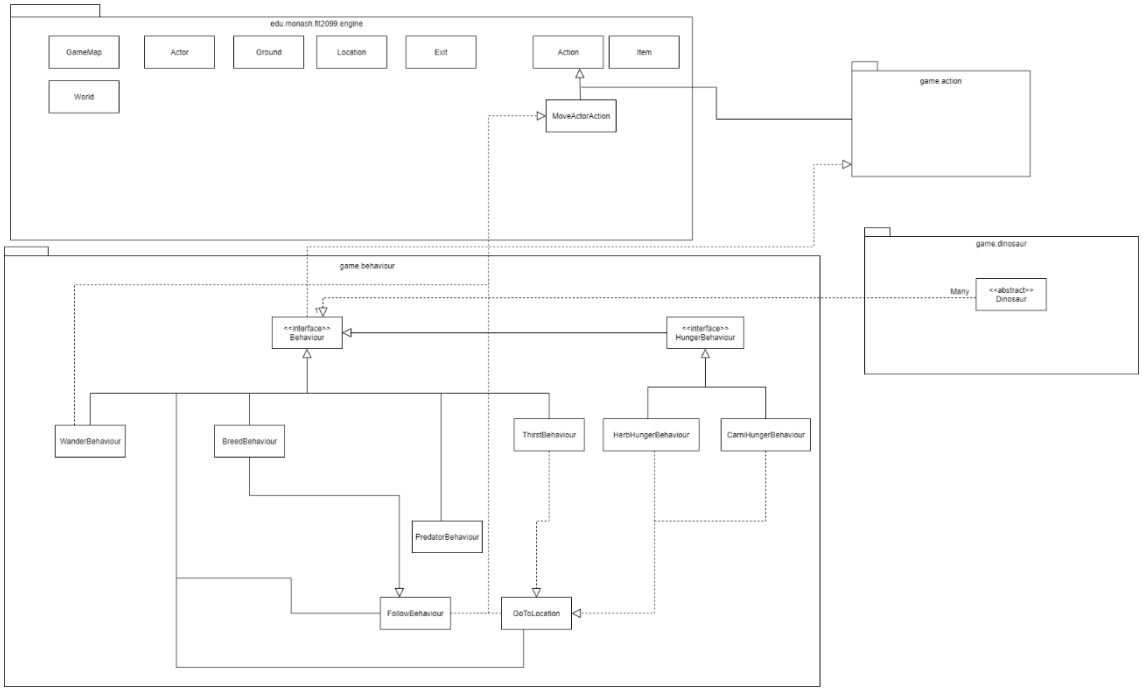
WaterTile abstract class extends Ground

As mentioned before, this class is used so we can potentially reuse it in the future, helping us achieve Don't Repeat Yourself. This also acts as a ground, so other classes can still use it, which achieves Liskov Substitution Principle. We follow the pre-existing Dependency Inversion Principle via the use to Capability ENUM for the rain.

Checks if it has the RAIN enum to check rain and add sips accordingly, and has other methods to manipulate and get sips, as well as fish.

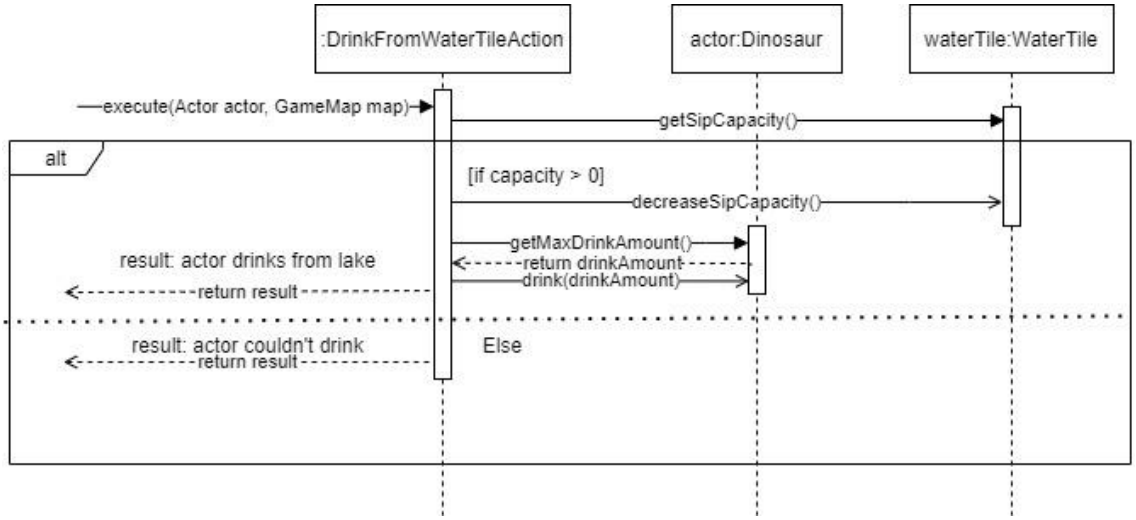
Thirsty Dinosaur

Class Diagram

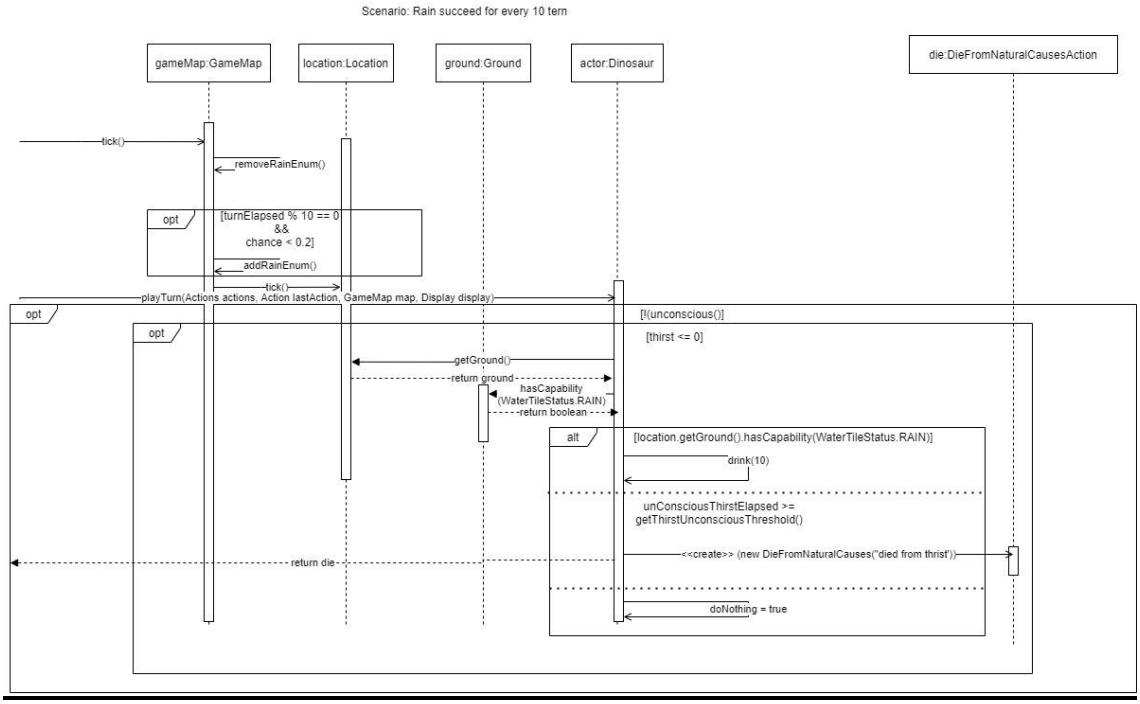


Note: Dinosaur also depends on WaterTileStatus for the ENUM, which is added to class diagram before this.

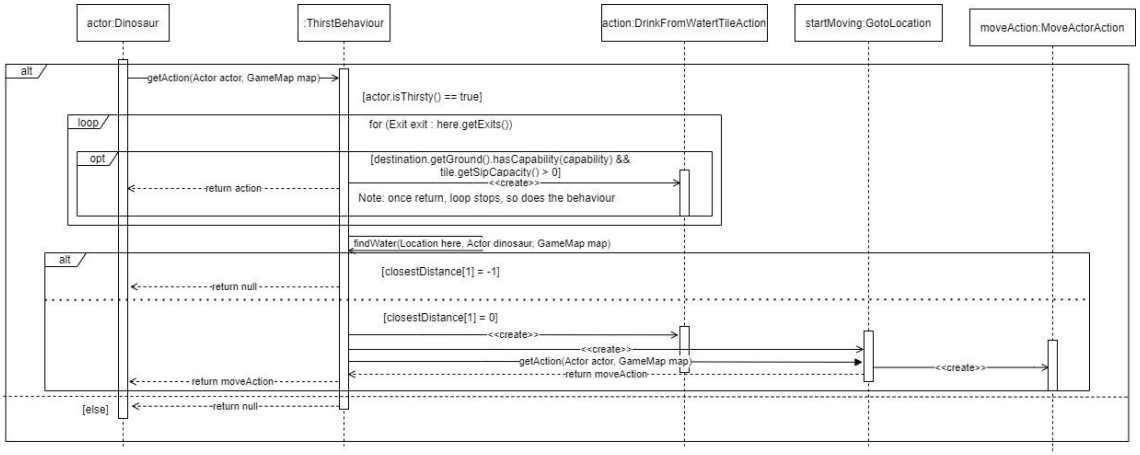
Sequence diagram: Drink Action



Sequence diagram: Dinosaur unconscious drink from rain

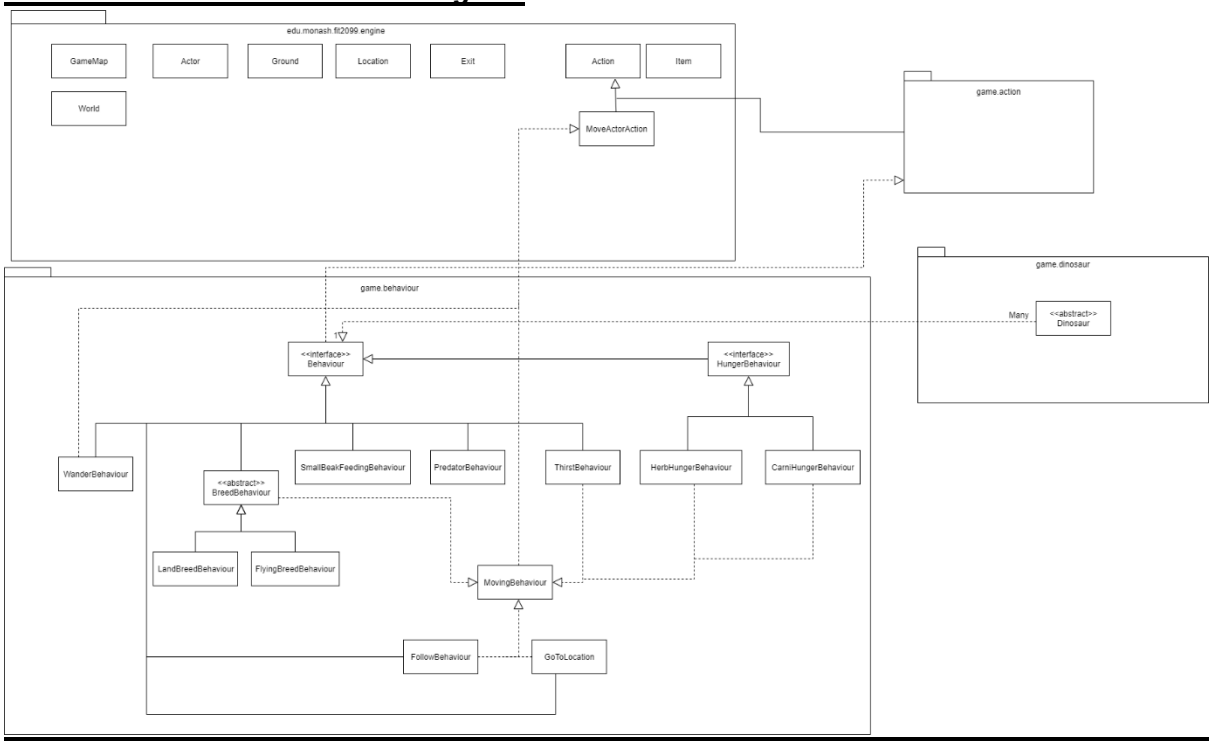


Sequence diagram: ThirstBehaviour getAction()

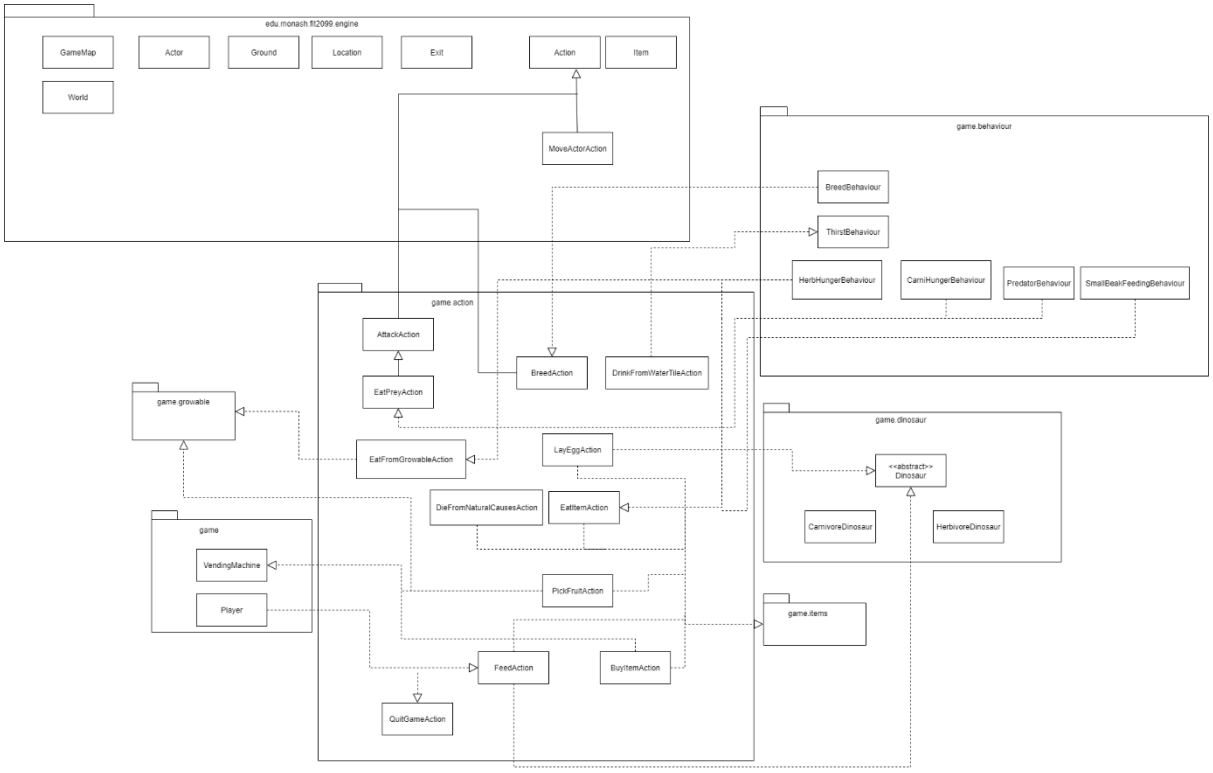


This section is easily addable since we already have a base Dinosaur abstract class, in which we can just easily add to it, so we can follow Don't repeat yourself in subsequent individual dinosaurs.
Dinosaurs will have new methods to get their MaximumDrinkAmount, ThirstyThreshold, and we have overridden isConscious to include thirst too. The playturn method just needs to add extra check for thirst for unconsciousness and kill it if past limit and that's it. Also has method to drink and decrease water level.
Dinosaurs that can't enter water tiles are done in the Lake (WaterTile) class itself, where we check if the dinosaur has an enum status to even enter it in the first place (FLYING). Since we are using the enum to do that, we have reduced dependency on other classes and instead just depend on the enum class, so we have reduced dependency (Depedency Inversion via Capability)
Thirst behaviour was created for dinosaur to use to find water when they are thirsty. It is implemented by implementing Behaviour. The code inside to find water is very similar to how the code to find food is. However, since there are no centralised way currently to find 'something', I had to copy and paste the code from the hunger behaviour to implement the thirst behaviour instead, and it is a technical debt I am willing to pay. We are following Liskov Substitution Principle here since Dinosaur requires a behaviour and ThirstBehaviour is still a behaviour. However, we did not fulfil Don't repeat yourself since I had to copy the code from HungerBehaviour to do this.

Overall new Behaviour class diagram



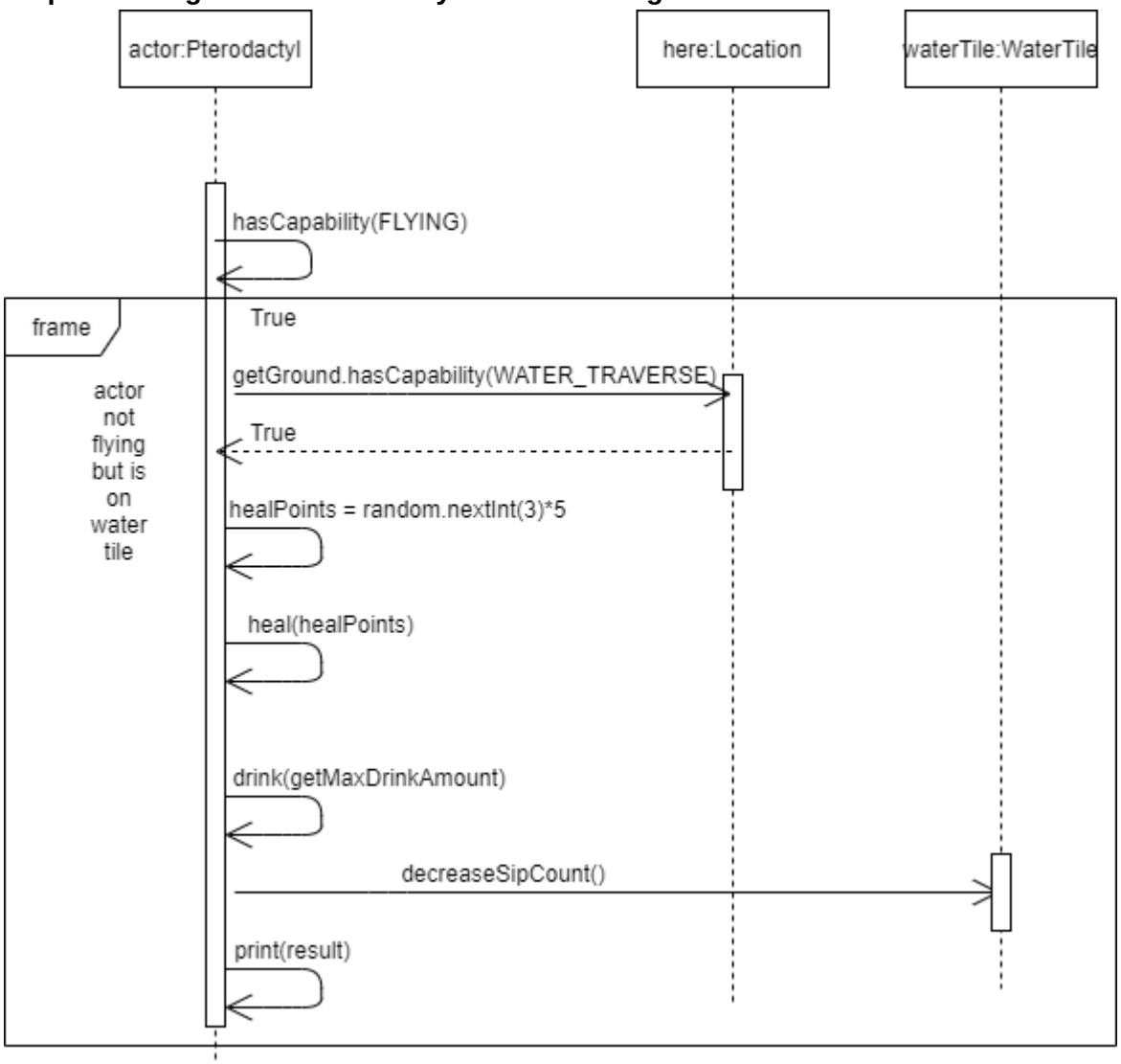
Overall new Action class diagram



Pterodactyls

With a new addition to the Dinosaurs with very different features compared to the rest, we had to modify and add several classes to make things more dynamic. The Pterodactyl class isn't much different, since it just holds the stats and adds more capabilities that other Dinosaurs should not have. However, we decided to add a special method in Pterodactyl that calculates how much a Pterodactyl eats and drinks when crossing a lake. Since Dinosaur class should not have this method due to SRP, and Pterodactyl is the only flying dinosaur in the game, this method should be fine to stay just in Pterodactyl.

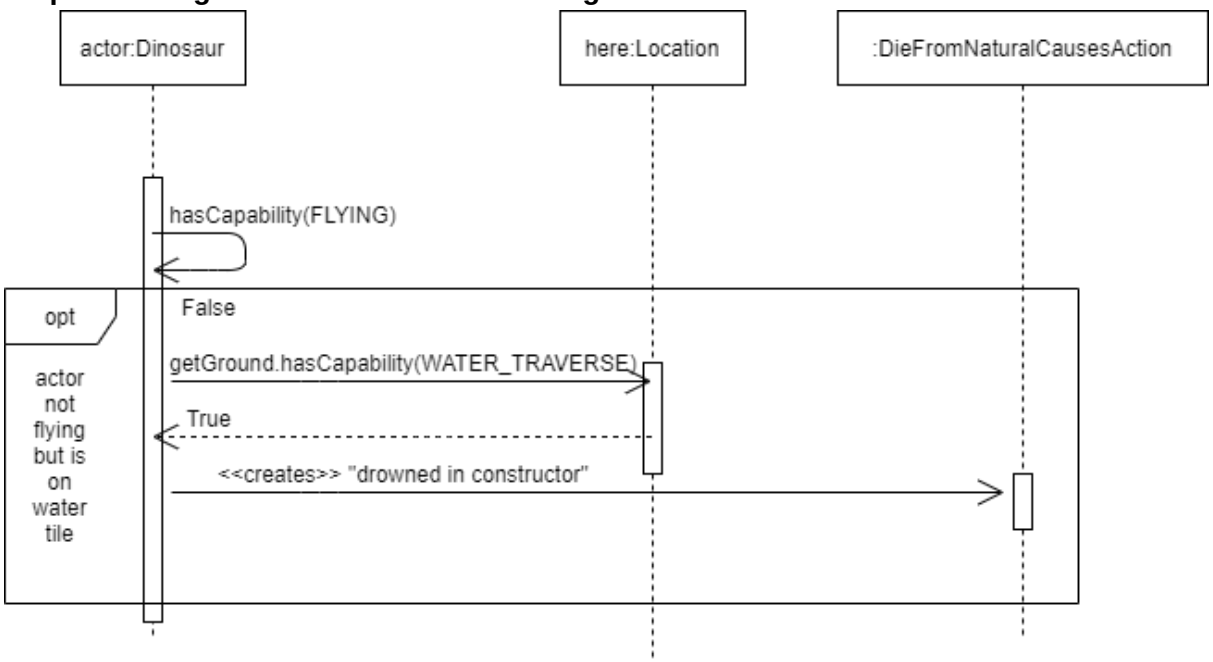
Sequence diagram for Pterodactyl when crossing Lake



Additions to Dinosaur class

With the introduction of flying dinosaurs, we need check new conditions and add new capabilities to these dinosaurs. Firstly we will need CAN_FLY and FLYING enums for flying dinosaurs. Then we need to check if the dinosaur is flying, and if so, increase the flyCounter and kill the dinosaur if flyCounter >= maxFlyCounter and if on a WaterTile.

Sequence diagram for dinosaurs drowning in a WaterTile



DieFromNaturalCausesAction

Previously named DieFromHungerAction, we decide to change this to be more general since there are now more ways to die due to the environment (hunger, thirst and now drowning). Hence, we will rename this to fit the causes more appropriately, and the constructor now takes in a String which will describe the cause of death. This allows us to prevent repeated code and also will not break SRP since the causes are predetermined and does not change the way dying works.

Abstract MovingBehaviour

We decided to add a parent class MovingBehaviour for GoToLocation and FollowBehaviour since both calculate distance the same way. This is to reduce repeated code.

Slight additions to GoToLocation

We will also add a new constructor to this class which will accept an action only to allow an actor to go to a location without any restrictions, so we do not have to do checking for Growable or Item. Action is defaulted to null in case actor ever wants to just go to a Location for reason.

BreedingBehaviour revamped

We assume that flying dinosaurs and land dinosaurs will have different behaviours when it comes to breeding. This calls for some reworking for BreedingBehaviour, which previously only caters to land dinosaurs. Thus, we decide to make BreedingBehaviour an abstract class that contains all common methods for LandBreedBehaviour (previously just BreedBehaviour),

and new child class FlyingBreedBehaviour. By doing so, we will be able to reduce several repeated code and also meet SRP and LSP.

Abstract Class BreedBehaviour

As said above, abstract class BreedBehaviour will have the common methods, namely findPartnerInRadius, breedChance and wantsToBreed, since all breeding behaviours will make use to these methods. Additionally, we will want to make all condition checking in this class too since all such behaviours will check on the same thing in order to meet SRP.

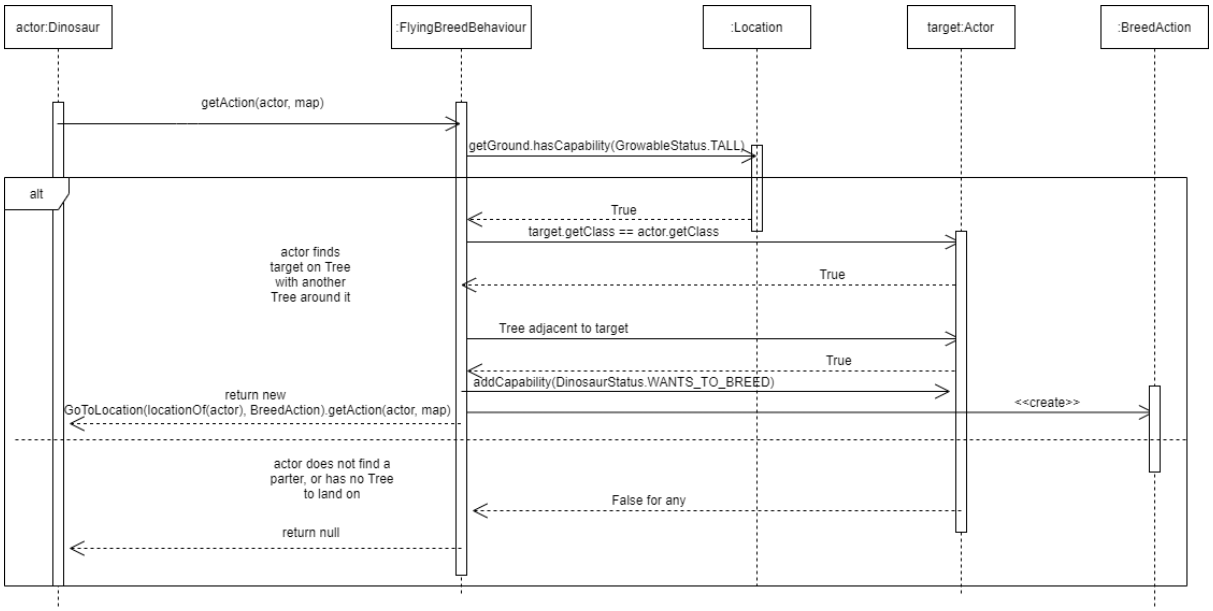
LandBreedBehaviour

Not much to say about this since it is just the old BreedBehaviour without the common methods, only changes we will make is to call the new checking conditions method from parent class and also override hasPartner but the code stays the same.

FlyingBreedBehaviour

Also similar to LandBreedBehaviour except for the type of partner to find and how to meet them. The way we want this breed behaviour to work is to first scan the area within a set radius, then find an opposite gender who is currently on top of a Tree, which the target will check if there is another Tree adjacent to it, and if so, add the capability WANTS_TO_BREED and stay there for a period of time for the actor to arrive. Flying dinosaurs will have a bigger radius to find a partner, since the conditions are tighter for them due to partner having to be already at a Tree.

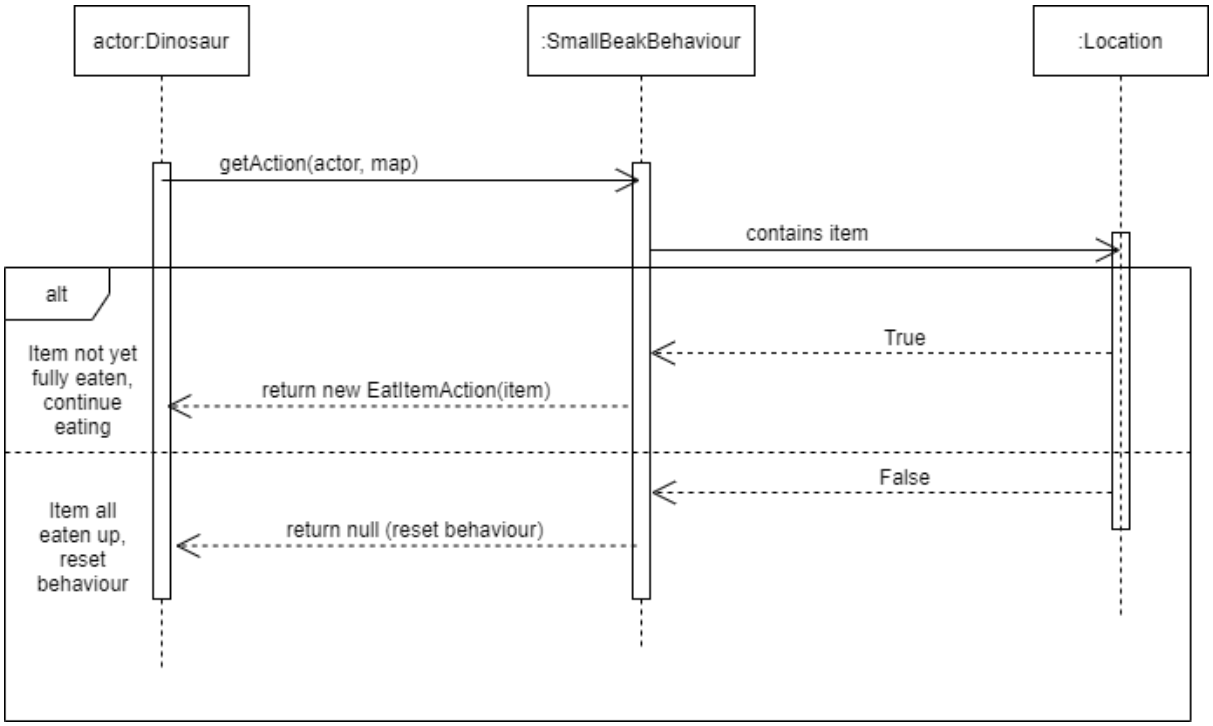
Sequence diagram for FlyingBreedBehaviour



SmallBeakFeedingBehaviour

This behaviour caters to dinosaurs that require multiple turns to eat an Item. If the dinosaur is flying, they have to land to eat the Item. We decided for this behaviour to cater for all dinosaur types with small beaks so that land dinosaurs can also make use of this class (in case there are small beak land dinosaurs). This behaviour locks a dinosaur in “eating mode” where the getAction will keep overriding itself with EatItemAction until the item is gone or the dinosaur is full.

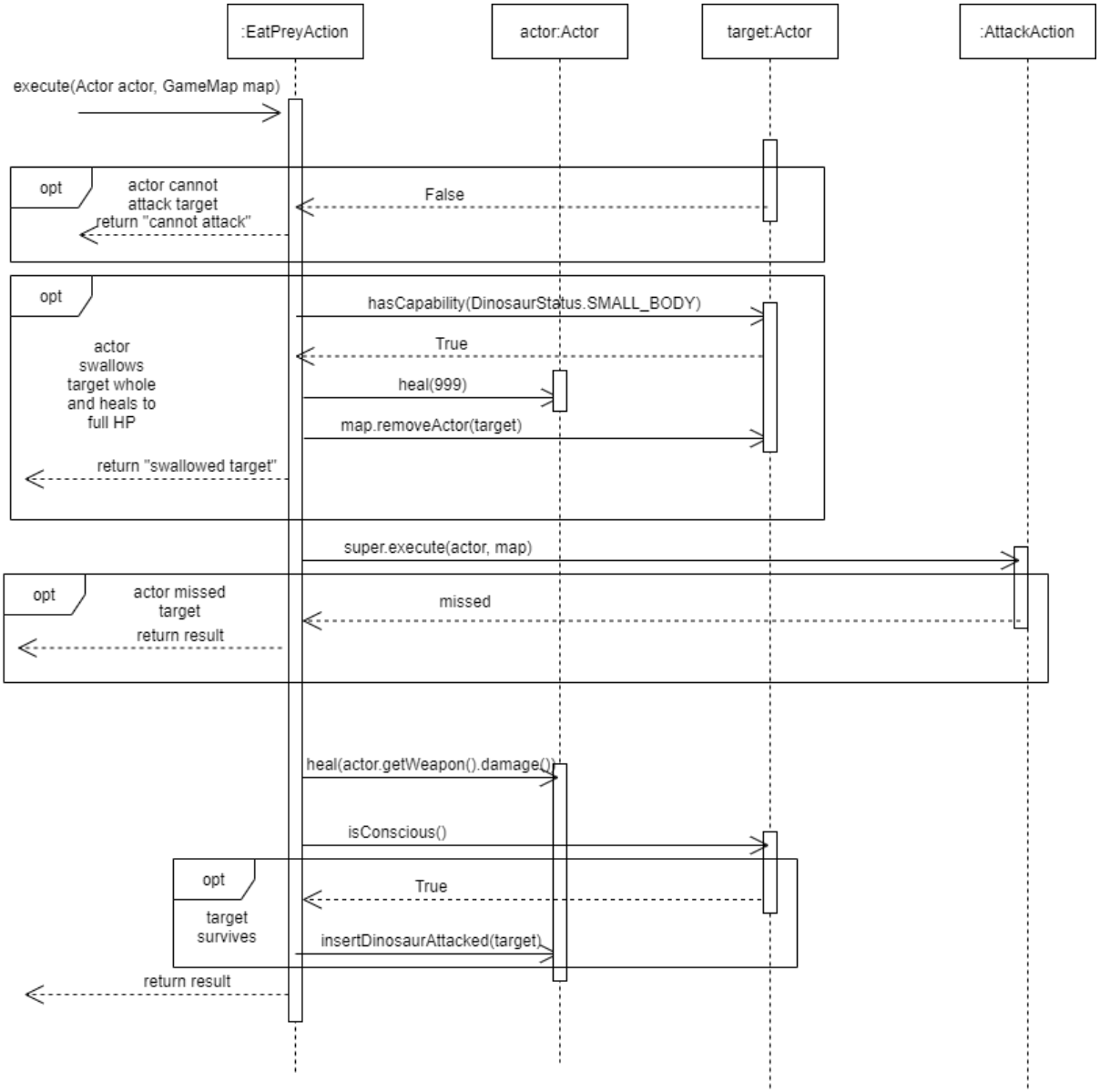
Sequence diagram for eating for multiple turns



Modifications to CarniHungerBehaviour

With the addition to Pterodactyls which have small bodies, predators can swallow them whole so there will not be Pterodactyl Corpse if that happens. Hence, we have to make a new check if the target has the SMALL_BODY enum which will then heal the predator to max health and remove the target without adding any Corpse.

Sequence diagram for predator eating target with small body



Changes to EdibleItem

Since we now need to keep track of the remaining heal points for items due to dinosaurs eating for multiple, we decided to add an “HP” to items that decreases by the amount eaten by a dinosaur. All items will have a default HP of 1 and will be removed from the map if HP < 0. With this, we will also need a way to “hurt” the item which essentially decreases the HP by a certain amount.

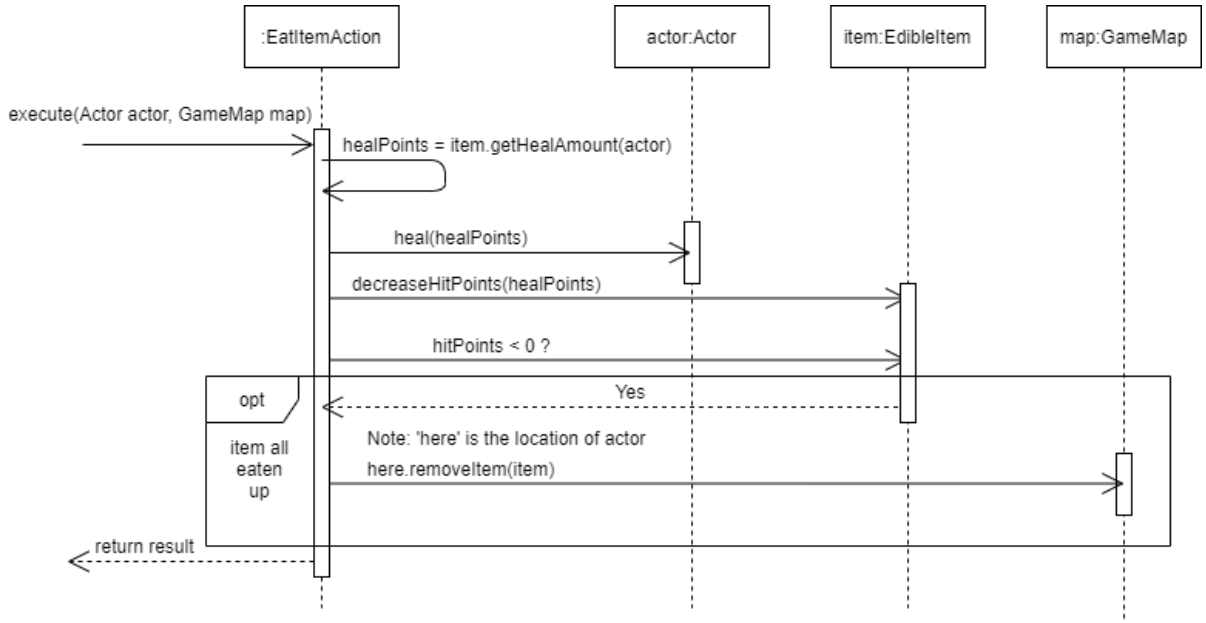
Corpse HP

Corpse is the only one with multi-turn eating capability, but we will make it general in EdibleItem. With this we will also need to return appropriate healing values based on which dinosaur is eating the corpse, since only dinosaurs with small beaks eat the Corpse for multiple turns. We will check if the dinosaur has a small body in getHealAmount.

Changes to EatItemAction

When eating and item, we now decrease the HP of the item by the amount healed to the actor, and then only remove it from the map if they have less than 1 HP.

Sequence diagram for dinosaur eating an Item



Second Map

In order to connect maps with each other, we will add a method that does this in the JurassicParkGameMap. Additionally, we will design the second map and directly add it in Application to initialize both maps and allow the Player to traverse between both maps using the newly added method in JurassicGameMap.

QuitGameAction extends Action

This Action allows the player to quit the game when the option is used. When selected a descriptive message is returned telling the player that they have quit the game. A menu prompting the player to play another game will be shown.

Changes to Application class

A while loop has been implemented to print out a selection menu for the player to choose the modes they wish to play, as long as the player hasn't selected the quit option. The player selects a mode by inputting the number that corresponds with the choice they selected. If the sandbox option is chosen, modeBoolean will be set to False and the game will run as normal like in assignment 2. If the challenge mode is selected, modeBoolean is set to True and the player will have to input the EcoPoints goal that they think they can achieve in a also set number of turns in order to win the game. The player is able to quit the game directly from here, thus the game can be ended without terminating the process.

Changes to Player class

New private attributes were added, int targetTurn, int targetPoint and boolean challengeMode.A challenge mode check is added into the method playTurn to check if the challenge set by the player has been achieved. This method checks if the timer is up before the player's target turn is up, if so a message is printed to notify the player that time is up. This method also checks if the player EcoPoints have exceeded the set target EcoPoint goal. If it has been surpassed, a message is returned stating that the target has been achieved. The player wins the game at this point. If the goal has not been surpassed, the player loses the

game. QuitGameAction is called regardless of the outcome to end the current game session and the player is then prompted to play another game.

Recommendations for extensions to the game engine

To be honest, I'll argue that the game engine is sufficiently well designed and is pretty good, albeit having some places that leaves more to be desired. I will be explaining how the engine achieved this with the SOLID and some other principles to justify why I/we have a good opinion of the game engine.

Single Responsibility Principle

The engine is designed with Single Responsibility Principle in mind, so that anyone that wants to use the engine can follow this principle easier. It also helps us achieve many other principles like reduce dependencies and don't repeat yourself. To further elaborate this, the engine has separated responsibilities, like the World class which will generally process every actor's turn, and then ticks the GameMaps. The GameMaps are responsible for holding all the Locations in a map. It will then tick all the Locaitons it has within the game, after possibly doing some map related mechanics. Then each location will do their job, telling the items at the location to tick as well as the Ground. The point here is that each thing here is separated into its own class. The location and the ground tiles are its own class, the map are its own class and it makes the code neater and easier to maintain. Due to this, the game engine is easy to use, so this is one of the point that makes the game engine good.

Open/Closed Principle

This is very self-explanatory. A game engine's purpose is to provide a base for others to use in creating a game, so it must be easily extended. The engine has done just that, allowing the engine to be extended but closed for modifications (modifications might be needed there and there though, will explain in a bit). To further explain this, the engine provides base classes of Actor, Item and Ground. For us to make our own game, we can just easily extend Actor to add a player, extend Actor to add custom NPC into the game, for instance Dinosaur in the assignment's Jurassic Park game. Extending Ground allows us to create our own tree, bushes, etc. Extending item allows us to have custom items that are usable, and can be picked up and dropped, and will still work with the existing engine classes (Will be further explored in Liskov Substitution Principle). The engine made it so much easier to just add a new Item and so on, and when I need to add a new function, I can easily do so, without modifying the engine. This easily helps us reduce repeated codes and achieve Don't Repeat Yourself.

However there is some modifications that were needed in Assignment 3, to implement the rain. Since we needed a way to make it have a chance to rain every 10 turns, we had to modify the run() loop, which meant we had to extend and undo whatever was done in the super class, and copy over the code for modifications. This is the part it needs some improvement but we will not be going over it. However, there are other ways to implement this so this is not really a con of the engine.

In conclusion for this, since it follows this principle, it allows us to easily use the engine to create our game by just extending engine classes, giving us a good opinion on the engine.

Liskov Substitution Principle

Due to the way the engine is designed, it is very easy to achieve Liskov Substitution Principle. The engine's World class for instance, processes all the Actor in the game, and will be expecting an Actor class to process. Since it follows Open/Closed Principle, if we needed to make a custom Actor, all we need to do is extend the Actor class and add our own function into it, and it will still be an Actor that is accepted by the World class. Same goes for Item, Ground, GameMap and Location. In the Jurassic Park game, in order to implement the Dinosaurs and the items, all I need is to extend the classes and it will still work, since they are still of the expected classes. This helps reduce dependencies since the World only depends on the overall Actor, Item, Ground, etc's implementation.

Since the engine follows this principle, it allows us to just use the previous principles to easily add new classes to our game (new Actors, Items) without touching the engine at all, saving a lot of trouble, which makes life easier and thus why the engine is good.

Interface Segregation Principle

The engine also follows the Interface Segregation Principle. This is evident in the Capable, Weapon and Printable interface. Anything that needs to be displayed on the map needs to implement the Printable interface, while the ones that doesn't will not need it, which follows the principle: We are not forced to implement and depend on methods we won't use. Further elaboration would be weapon. In a game there will be items that are not weapons and items that can be used as a weapon. For items that are not weapons, it would not need to implement the Weapon interface, while to make an item into a weapon, it just need to implement the Weapon interface.

Due to this, every time I want to add a new weapon into the class, all I need to do is have a new class extend Item and implement Weapon and that's it, simple to use, which makes us happy with the engine.

Dependency Inversion Principle

The engine is designed with dependency inversion principle in mind. The high level modules all depend on an abstraction and so do the low level works. To elaborate, the World class depends on the abstract Actor class to process all the Actors we have. The low level modules are our own implementation of Actor which is done by extending it, which also depends on the abstraction. Due to this, the World doesn't need to depend on our custom Actor, which reduces modifications needed when adding functionality, which helps us to reduce bugs when developing a game. This also means that the World doesn't need to depend on our custom Actor class since all it needs is the Actor, which via Liskov Substitution Principle, also acts as an Actor.

Dependency Inversion Principle 2 / Miscellaneous

The engine helps a lot with reducing dependencies. The creator of the engine has achieved this via the use of another class: The Capability class. This class helps store all the enumerations a Ground or Actor can have, and makes it so much easier to check if an Actor or Ground can do something, instead of having to check if the object is an instance of a class. For example, for the Jurassic Park game, we can have enum of TEAM_HERBIVORE and TEAM_CARNIVORE. Each dinosaur will have its own respective enum inside its capability

attribute, and we can just check which enum it has to execute respective behaviour. All they needed to do was just depend on an enumeration class, instead of two Dinosaur classes. This helps to reduce dependency and reduce the chance of errors.
Due to the capability class, a lot of limitations of the engine is solved by this, which means we don’t need to modify the engine class, which is good.