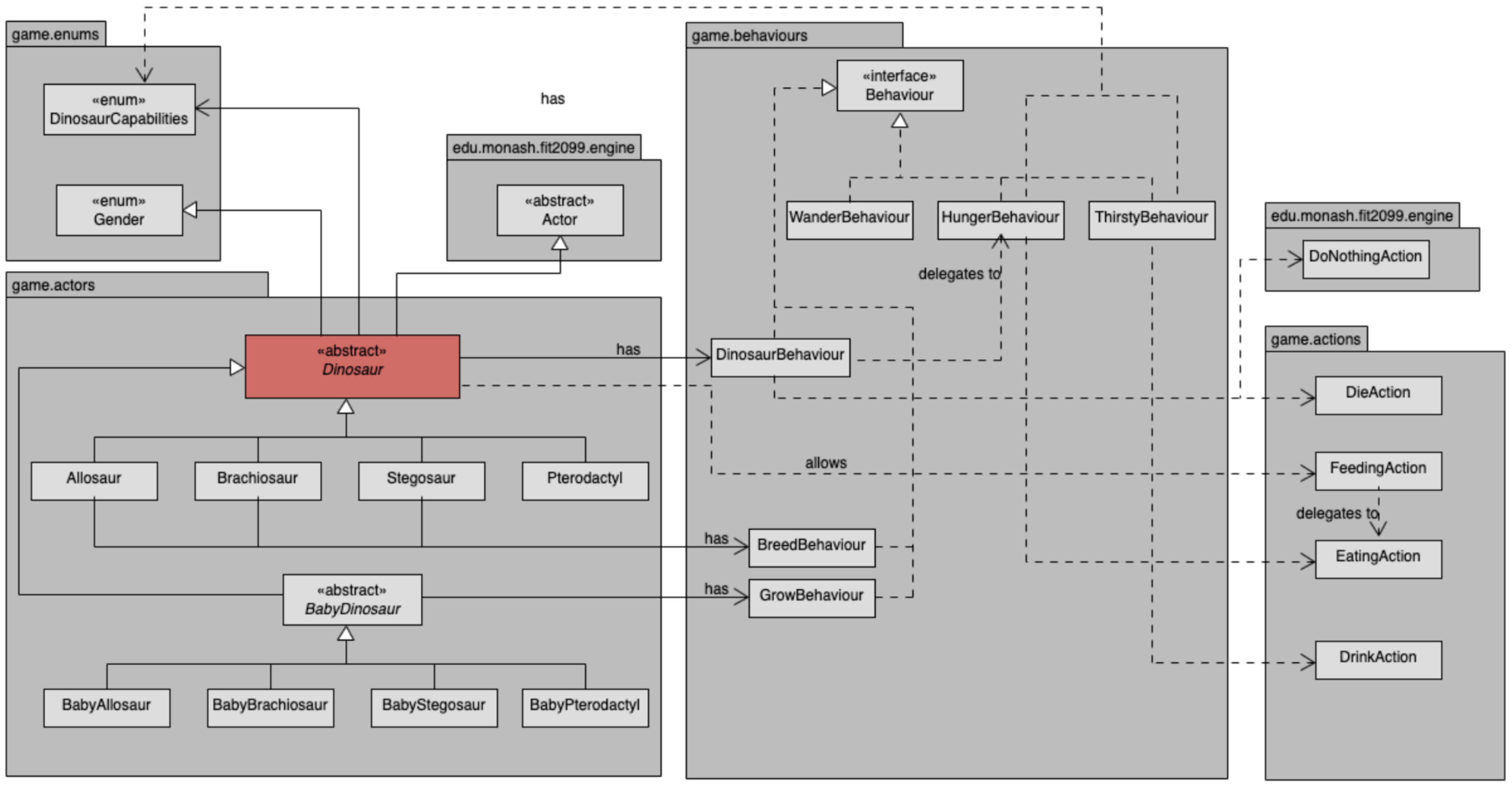
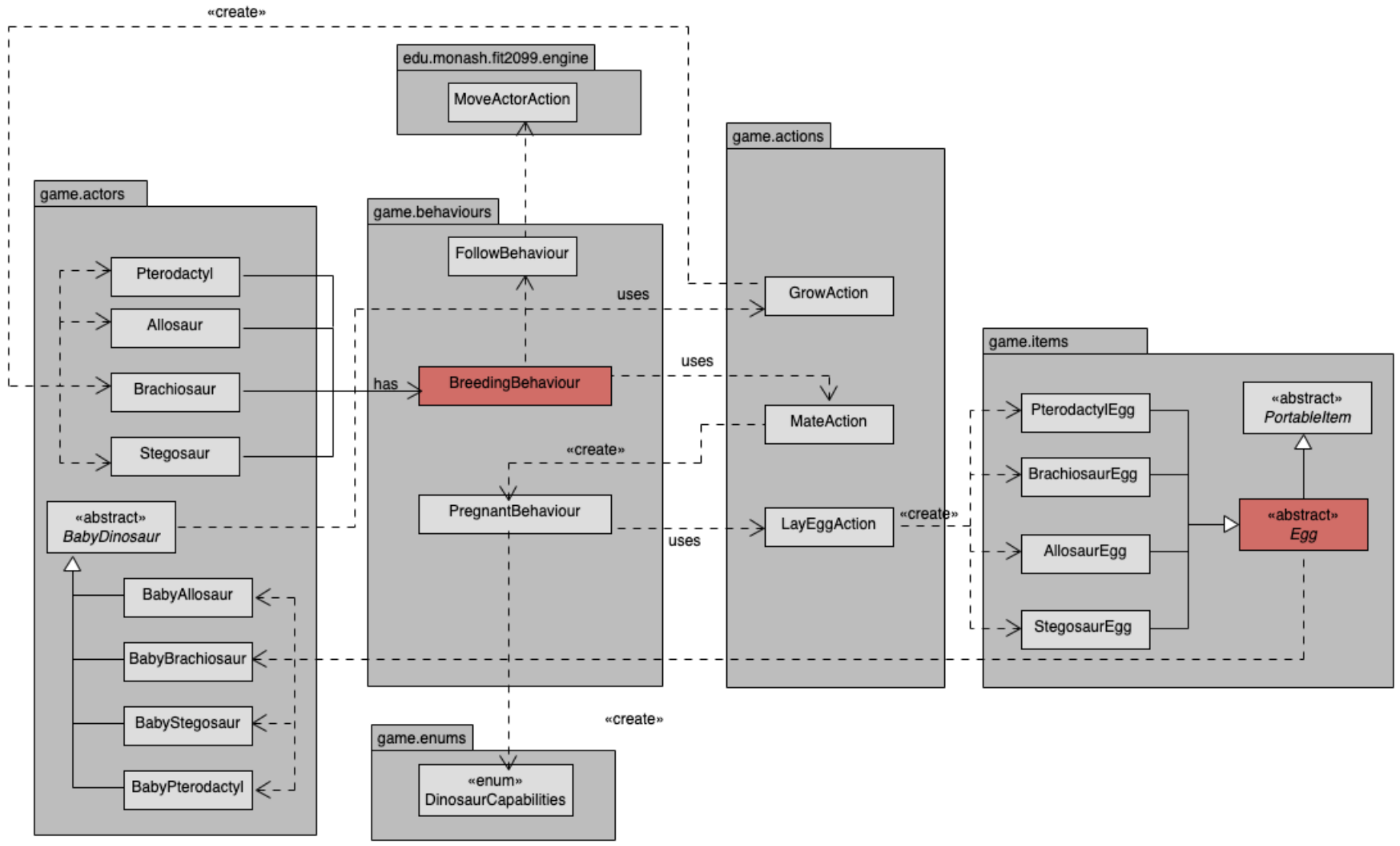
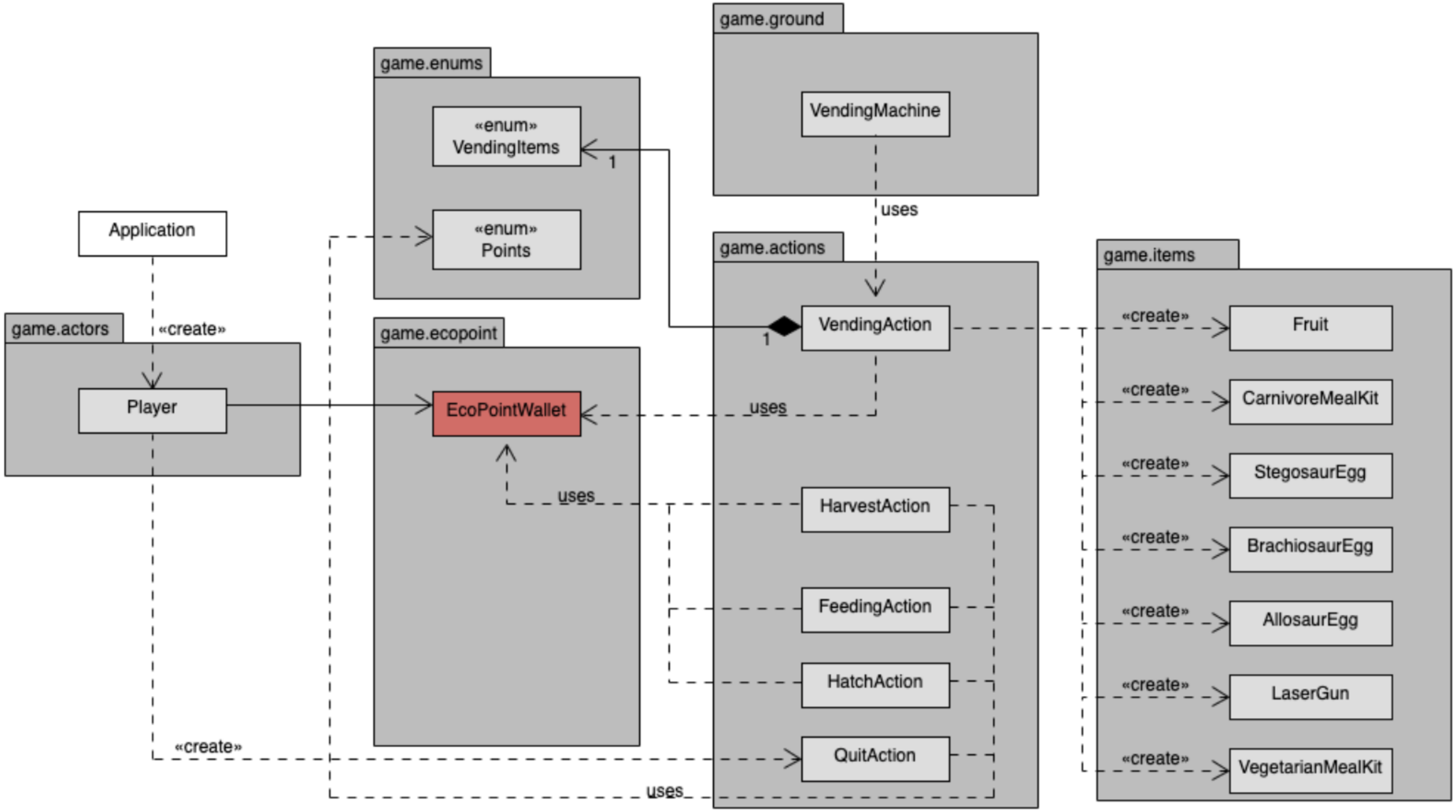
Dinosaur Class Diagram



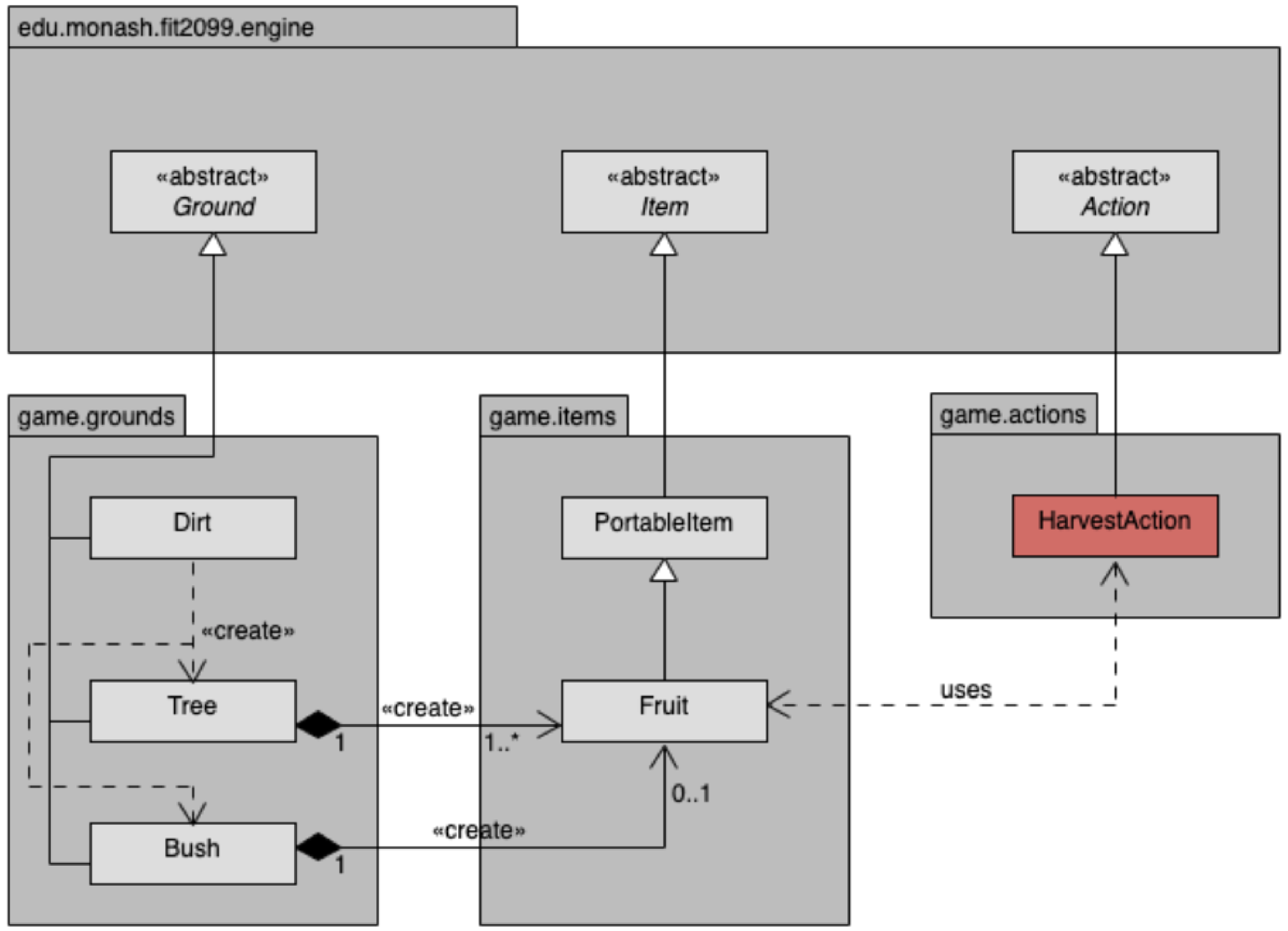
Breeding Class Diagram



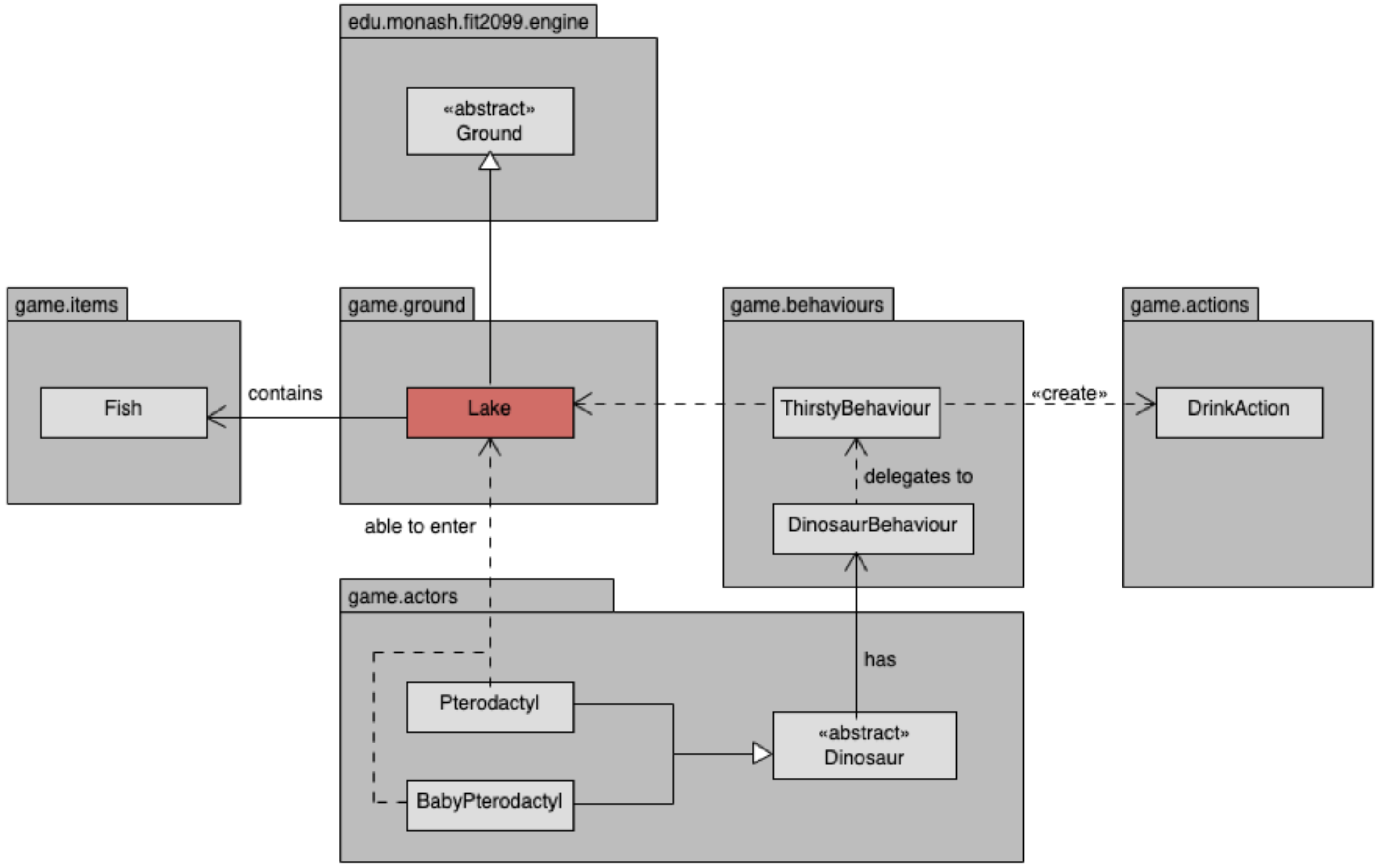
EcoPoint Class Diagram



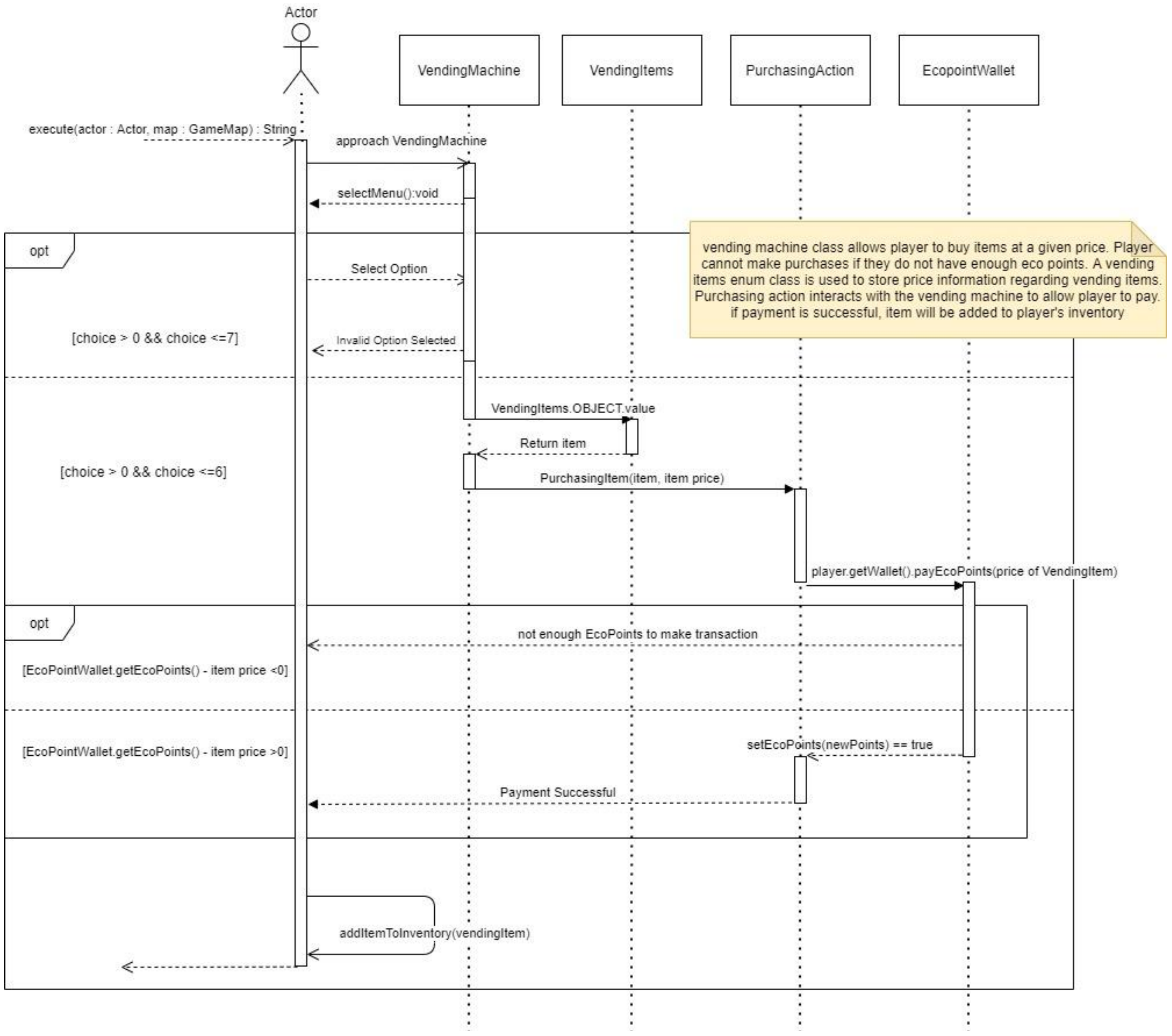
Plants Class Diagram



Lake Class Diagram

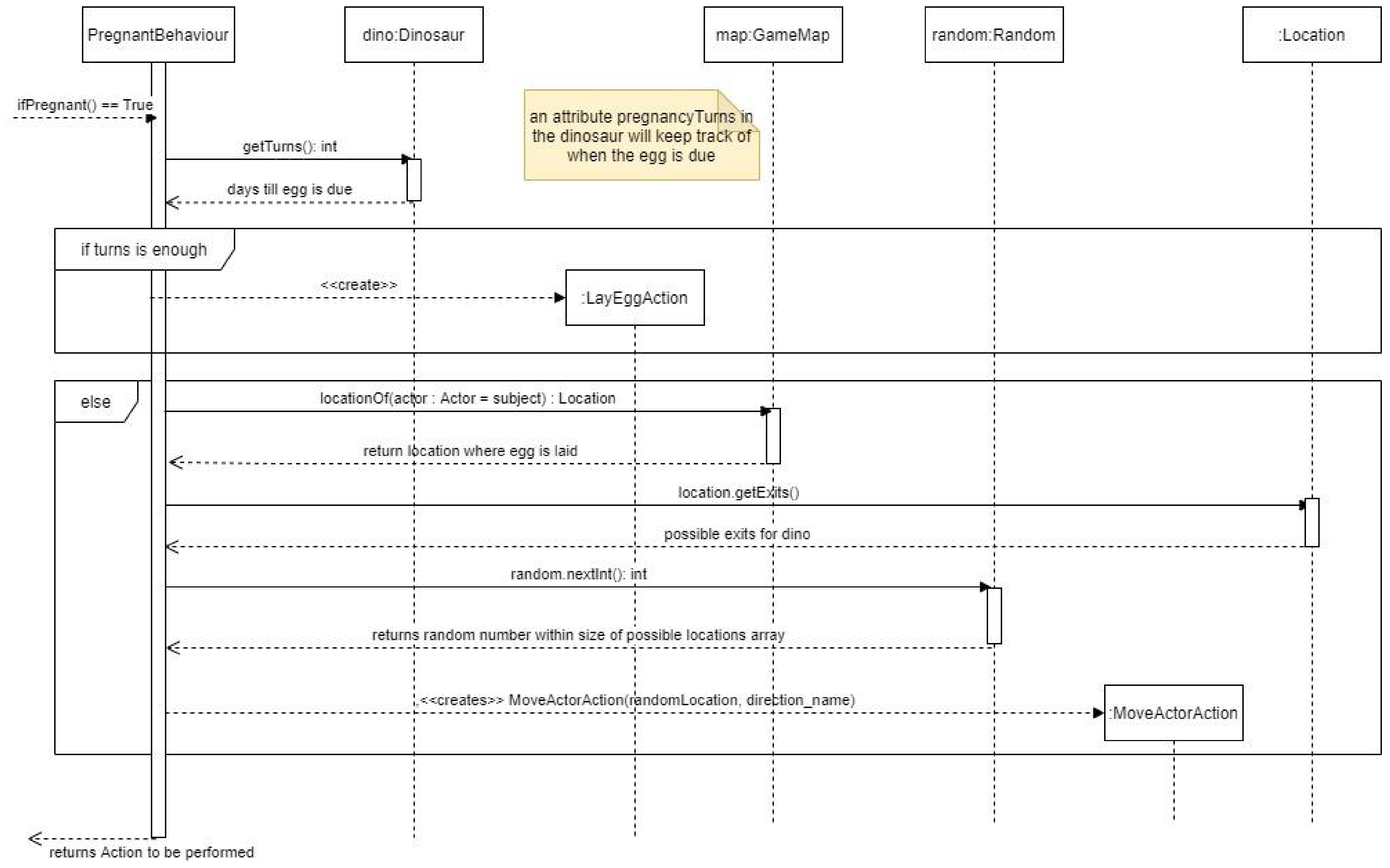


Vending Machine Interaction Diagram

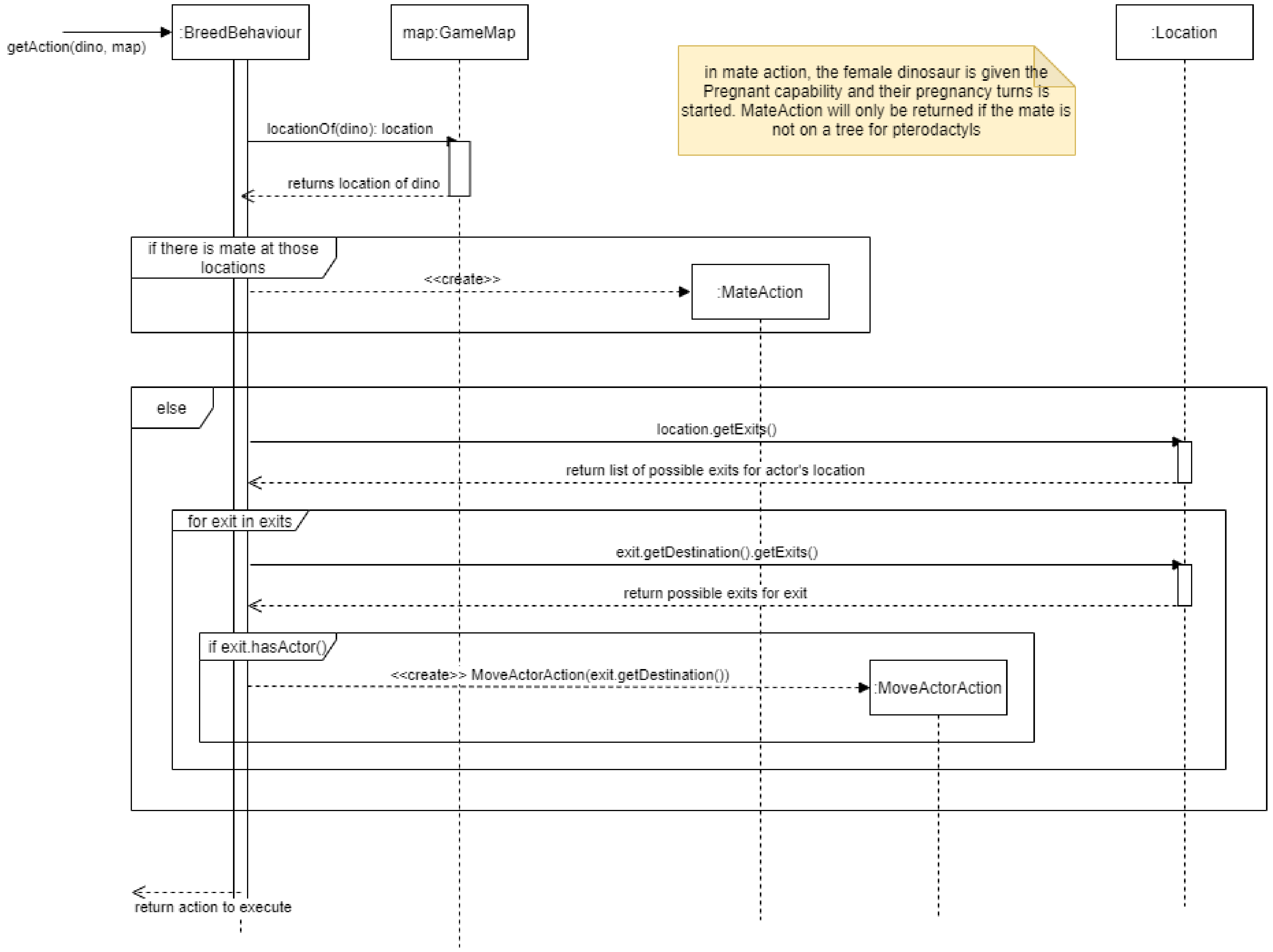

Actor
VendingMachine
VendingItems
PurchasingAction
EcopointWallet
execute(actor : Actor, map : GameMap) : String
approach VendingMachine
selectMenu():void
opt
Select Option
[choice > 0 && choice <=7]
Invalid Option Selected
VendingItems.OBJECT.value
Return item
[choice > 0 && choice <=6]
PurchasingItem(item, item price)
player.getWallet().payEcoPoints(price of VendingItem)
opt
not enough EcoPoints to make transaction
[EcoPointWallet.getEcoPoints() - item price <0]
[EcoPointWallet.getEcoPoints() - item price >0]
setEcoPoints(newPoints) == true
Payment Successful
addItemToInventory(vendingItem)

vending machine class allows player to buy items at a given price. Player cannot make purchases if they do not have enough eco points. A vending items enum class is used to store price information regarding vending items. Purchasing action interacts with the vending machine to allow player to pay. if payment is successful, item will be added to player's inventory

PregnancyBehaviour and Egg Laying Interaction Diagram (Updated to add random class and to reflect new methods used)

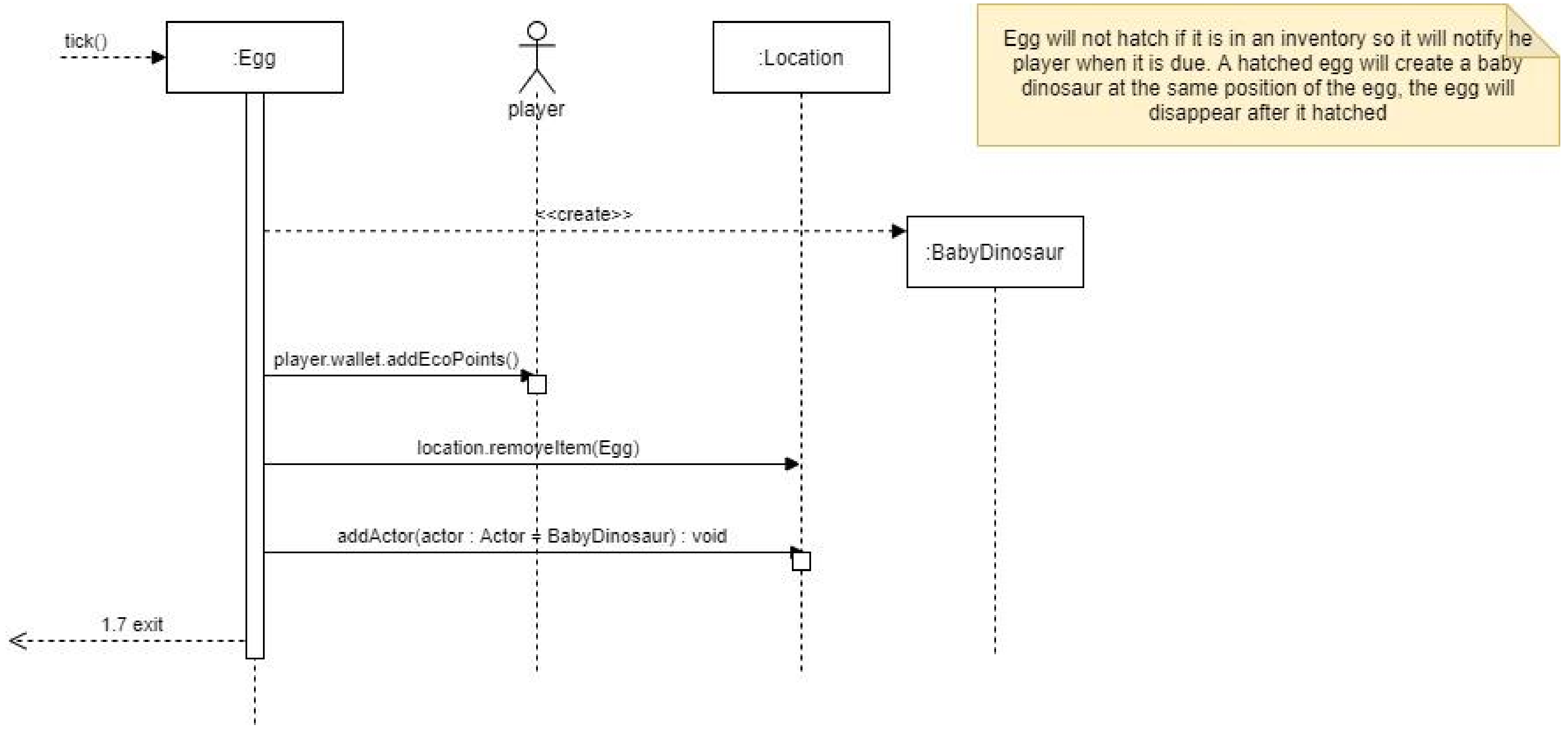

PregnantBehaviour
dino:Dinosaur
map:GameMap
random:Random
:Location
ifPregnant() == True
getTurns(): int
days till egg is due
an attribute pregnancyTurns in the dinosaur will keep track of when the egg is due
if turns is enough
<<create>>
:LayEggAction
else
locationOf(actor : Actor = subject) : Location
return location where egg is laid
location.getExits()
possible exits for dino
random.nextInt(): int
returns random number within size of possible locations array
<<creates>> MoveActorAction(randomLocation, direction_name)
:MoveActorAction
returns Action to be performed

BreedBehaviour Diagram Updated the note to reflect Pterodactyl choice of mate



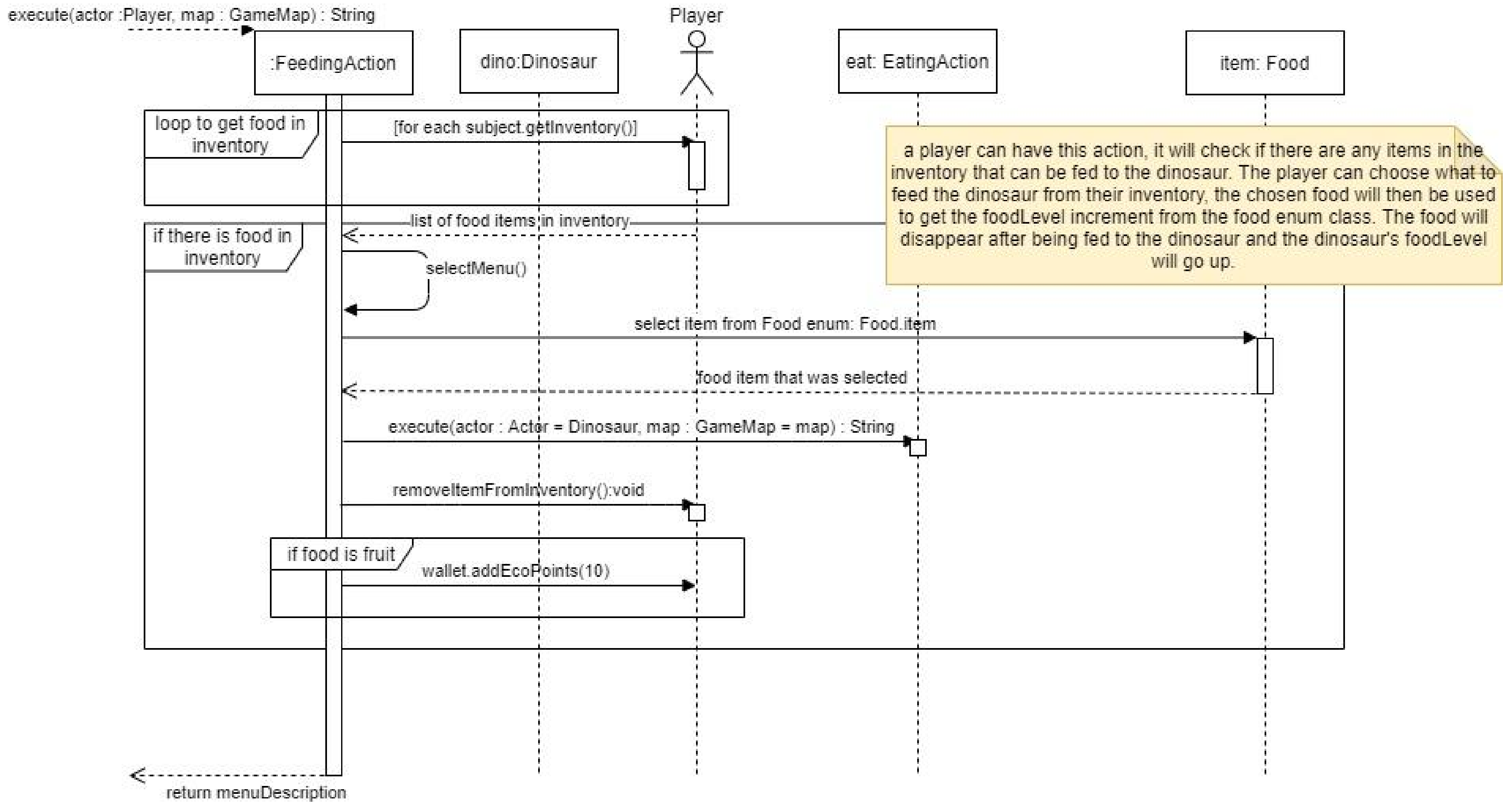
getAction(dino, map)
:BreedBehaviour
map:GameMap
:Location
in mate action, the female dinosaur is given the Pregnant capability and their pregnancy turns is started. MateAction will only be returned if the mate is not on a tree for pterodactyls
locationOf(dino): location
returns location of dino
if there is mate at those locations
<<create>>
:MateAction
else
location.getExits()
return list of possible exits for actor's location
for exit in exits
exit.getDestination().getExits()
return possible exits for exit
if exit.hasActor()
<<create>> MoveActorAction(exit.getDestination())
:MoveActorAction
return action to execute

Egg Hatching Interaction Diagram (Changed from HatchAction because Egg is an item and does not execute action)

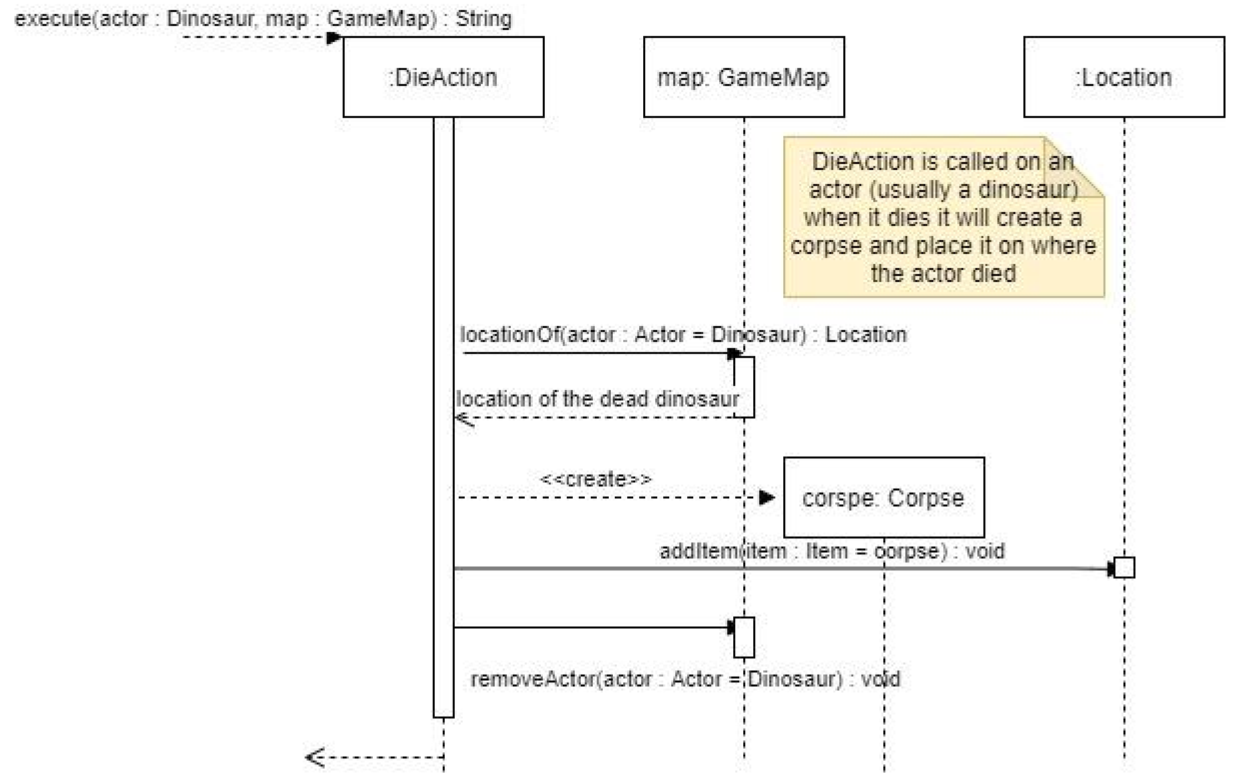


tick()
:Egg
player
:Location
Egg will not hatch if it is in an inventory so it will notify he player when it is due. A hatched egg will create a baby dinosaur at the same position of the egg, the egg will disappear after it hatched
<<create>>
:BabyDinosaur
player.wallet.addEcoPoints()
location.removeItem(Egg)
addActor(actor : Actor = BabyDinosaur) : void
1.7 exit

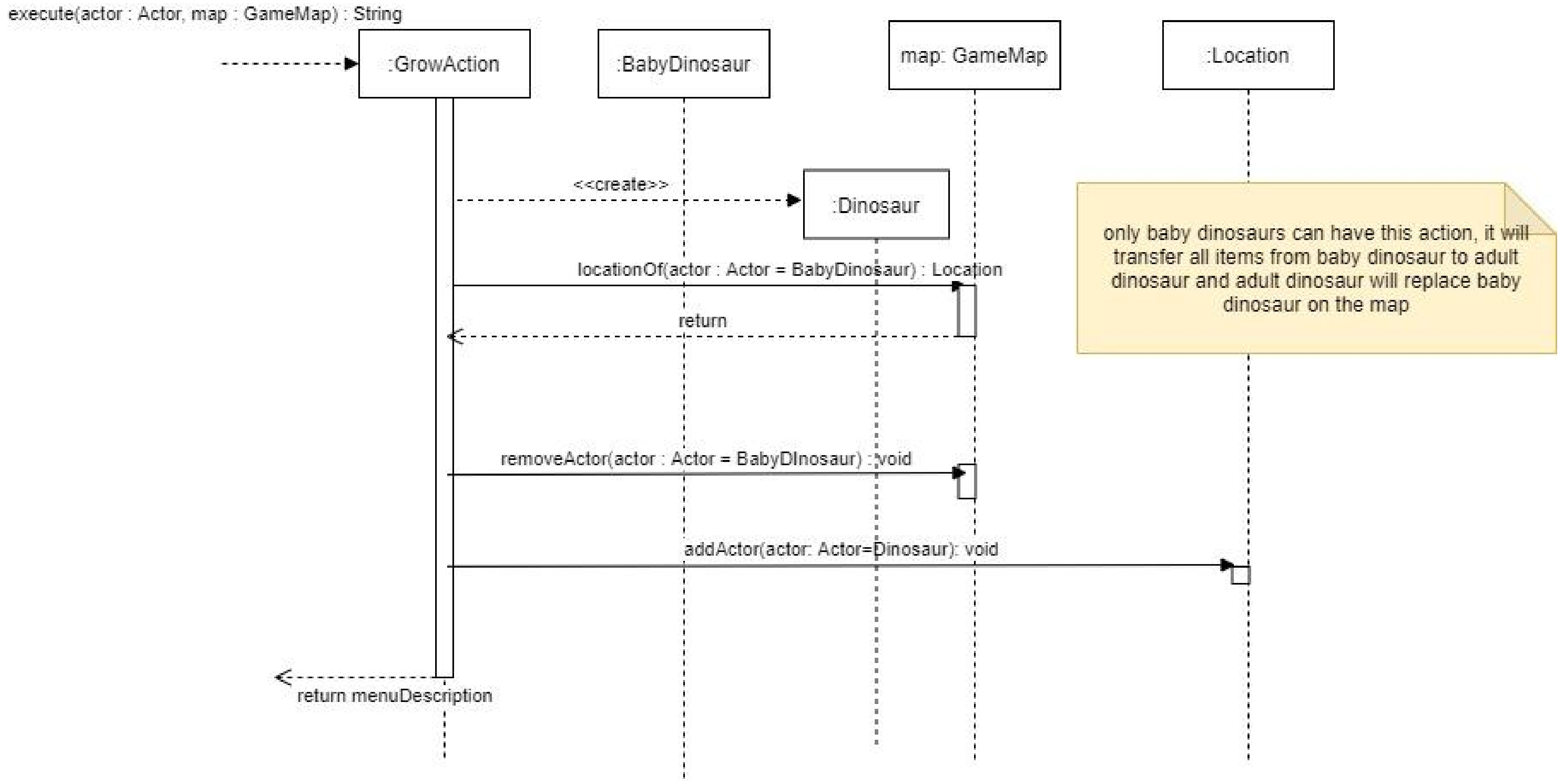
<u>FeedingAction Interaction Diagram</u> (Updated version made to include Food enum)


execute(actor :Player, map : GameMap) : String
:FeedingAction
dino:Dinosaur
Player
eat: EatingAction
item: Food
loop to get food in inventory
[for each subject.getInventory()]
a player can have this action, it will check if there are any items in the inventory that can be fed to the dinosaur. The player can choose what to feed the dinosaur from their inventory, the chosen food will then be used to get the foodLevel increment from the food enum class. The food will disappear after being fed to the dinosaur and the dinosaur's foodLevel will go up.
list of food items in inventory
if there is food in inventory
selectMenu()
select item from Food enum: Food.item
food item that was selected
execute(actor : Actor = Dinosaur, map : GameMap = map) : String
removeItemFromInventory():void
if food is fruit
wallet.addEcoPoints(10)
return menuDescription

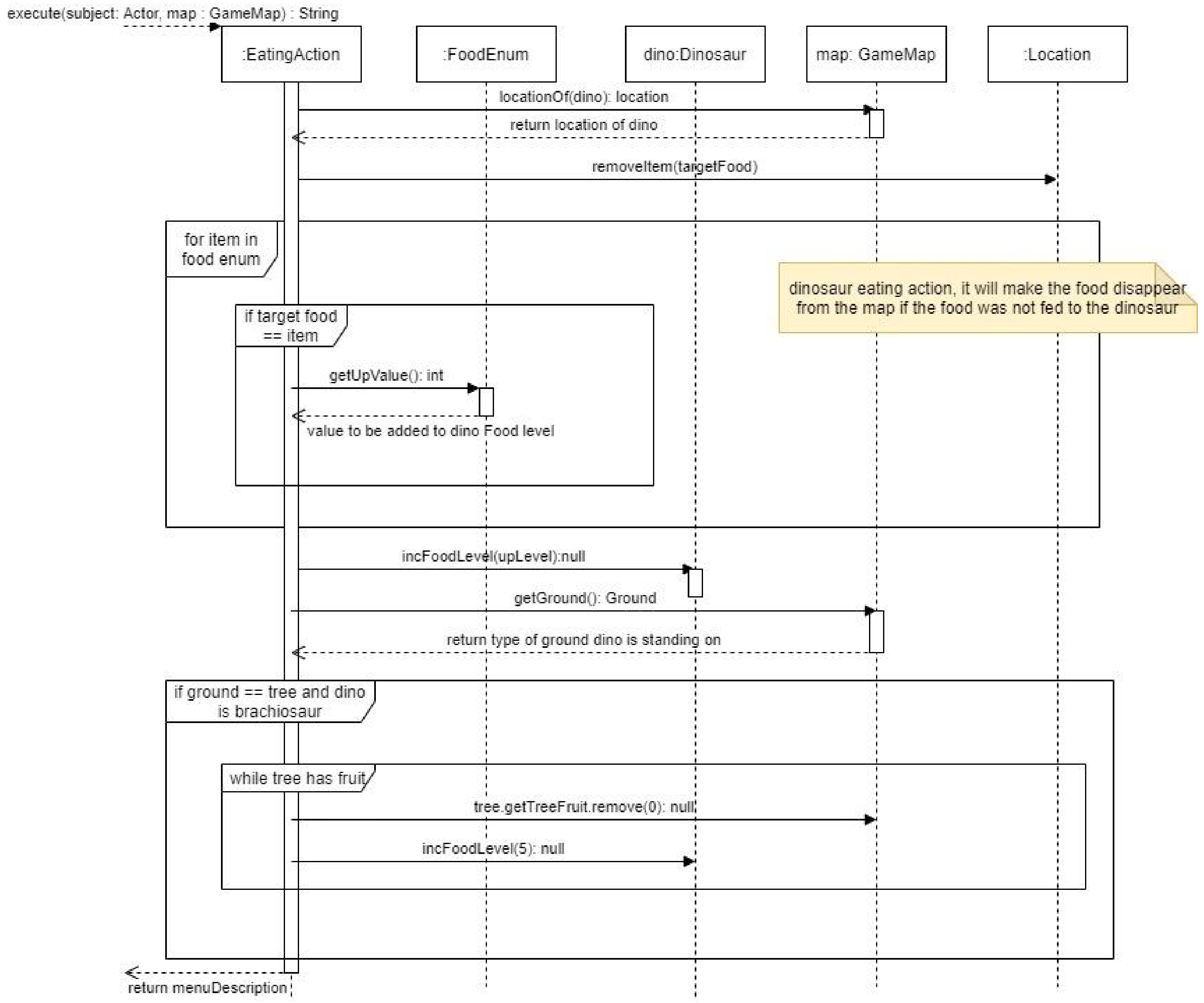
<u>DieAction Interaction Diagram</u> (New version is simplified because Dinosaur does not have inventory)


execute(actor : Dinosaur, map : GameMap) : String
:DieAction
map: GameMap
:Location
DieAction is called on an actor (usually a dinosaur) when it dies it will create a corpse and place it on where the actor died
locationOf(actor : Actor = Dinosaur) : Location
location of the dead dinosaur
<<create>>
corspe: Corpse
addItem(item : Item = corpse) : void
removeActor(actor : Actor = Dinosaur) : void

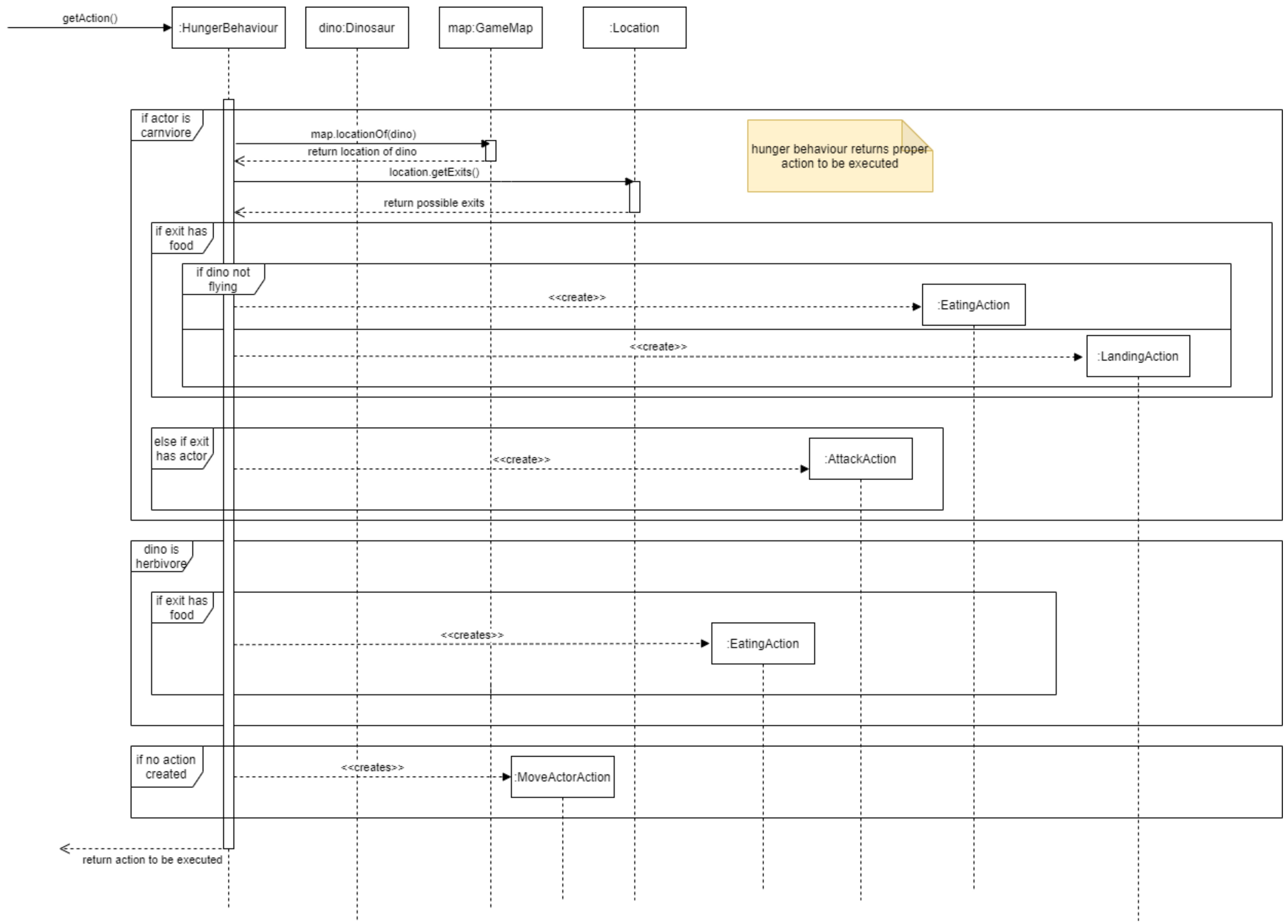
GrowAction Interaction Diagram (Updated version is shorter because it does not have the removal of items from inventory)


execute(actor : Actor, map : GameMap) : String
:GrowAction
:BabyDinosaur
map: GameMap
:Location
<<create>>
:Dinosaur
only baby dinosaurs can have this action, it will transfer all items from baby dinosaur to adult dinosaur and adult dinosaur will replace baby dinosaur on the map
locationOf(actor : Actor = BabyDinosaur) : Location
return
removeActor(actor : Actor = BabyDInosaur) : void
addActor(actor: Actor=Dinosaur): void
return menuDescription

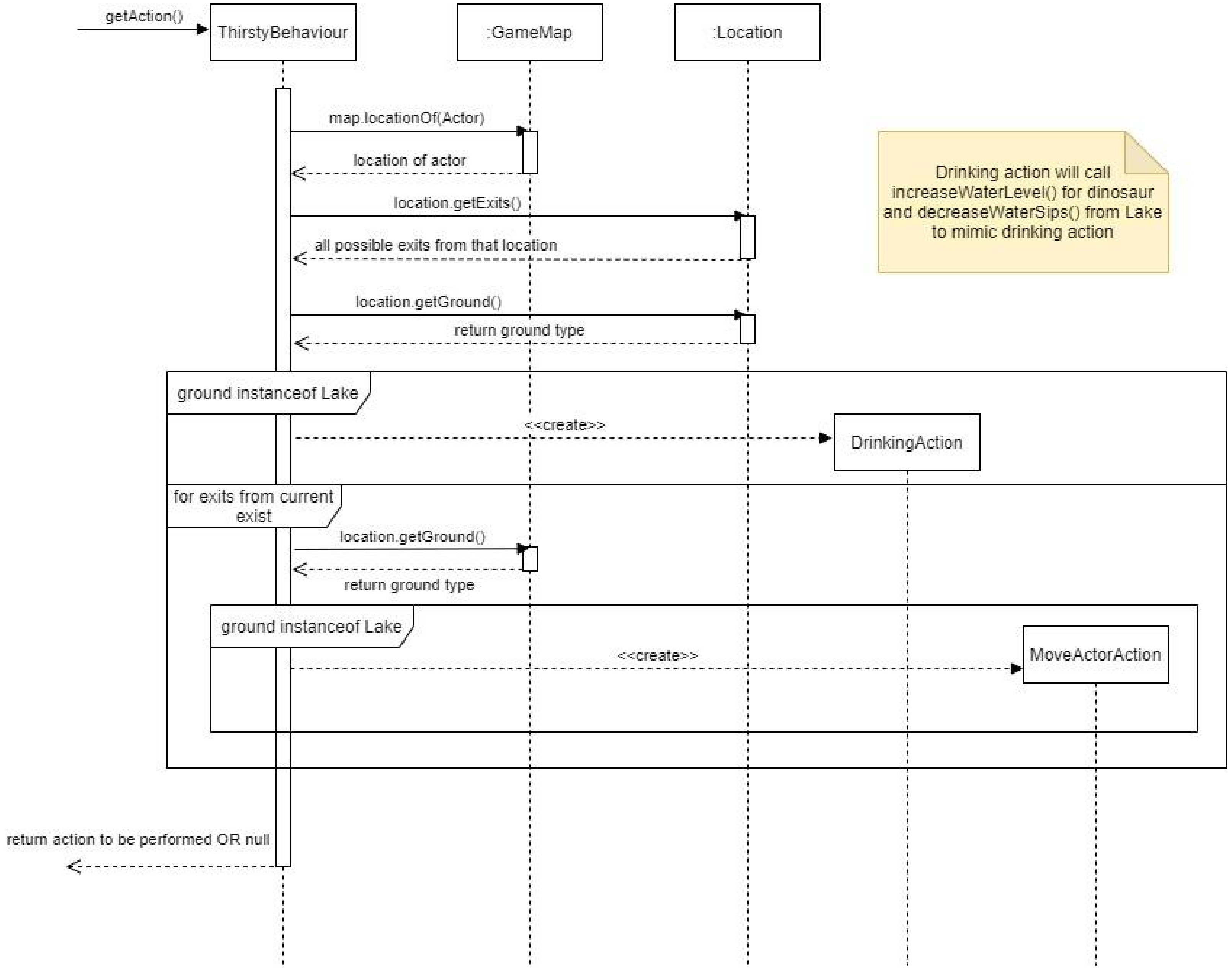
EatingAction Interaction Diagram


execute(subject: Actor, map : GameMap) : String
:EatingAction
:FoodEnum
dino:Dinosaur
map: GameMap
:Location
locationOf(dino): location
return location of dino
removeItem(targetFood)
for item in food enum
dinosaur eating action, it will make the food disappear from the map if the food was not fed to the dinosaur
if target food == item
getUpValue(): int
value to be added to dino Food level
incFoodLevel(upLevel):null
getGround(): Ground
return type of ground dino is standing on
if ground == tree and dino is brachiosaur
while tree has fruit
tree.getTreeFruit.remove(0): null
incFoodLevel(5): null
return menuDescription

HungerBehaviour Interaction Diagram (updated for landing action)



ThirstyBehaviour (DrinkingAction, ThirstyBehaviour) ADDED



Design Rationale

Hungry dinosaurs (DRY)

```
public abstract class Dinosaur extends Actor
```

An abstract Dinosaur class is created and extends the Actor class. An attribute of the class, ArrayList<Behaviours>, stores all the behaviours of dinosaurs such as wander and hunger behaviour which include in the DinosaurBehaviour.

```
public class DinosaurBehaviour implements Behaviour
```

A DinosaurBehaviour class is created and implements the Behaviour interface class. It contains the basic behaviour of dinosaur and baby dinosaur as well such as wander and hunger behaviour. It also contains the die and do nothing action.

```
public class FeedingAction extends Action
```

A FeedingAction class is created and extends the Action class. If the food from inventory is edible by the targetDinosaur, the player/actor will feed the targetDinosaur then the food will be removed from the inventory of player/actor.

```
public class EatingAction extends Action
```

An EatingAction class is created and extends the Action class. When a target food is fed to the dinosaur, it will eat the target food, the target food will be removed from the current location of the dinosaur and the food level of the dinosaur will increase based on the value of the target food.

```
public class DieAction extends Action
```

A DieAction class is created and extends the Action class. When a dinosaur becomes unconscious due to 0 food level, it will die after a certain amount of rounds based on different species. Then, the dinosaur will be removed from its current location and a new Corpse object item will be added to the same location.

Rationale: All dinosaur species of classes does not extend the Actor class but extends the abstract Dinosaur class to adhere to the Don't Repeat Yourself (DRY) OOP principles. Thus, the child classes of Dinosaur such as Allosaurs, Brachiosaur and Stegosaur that inherits the common attributes and the basic behaviours with minimal code.

Brachiosaur

Brachiosaur have the similar behaviour as other dinosaurs where being able to feed, breed and grow, so it contains basic DinosaurBehaviour. However, brachiosaur is a herbivore and long neck dinosaur where it only eats fruits from trees. If brachiosaurs step on a bush, the bush will die in 50% chance. It can eat as many fruits as it wants but each food only increases the food level by 5, if the fruits are fed by players the food level increases by 20. 2 males and 2 females of brachiosaur are required to be created in the game.

```
public class Brachiosaur extends Actor
```

A Brachiosaur class is created and extends the Actor class. A boolean instance variable of isLongNeck will be initialized with true, which means it can only eat fruits from trees.

```
public class HungerBehaviour implements Behaviour
```

```
If the current dinosaur is a Brachiosaur, isLongNeck is true:
  If there's a tree in the current location and the tree has fruits: return EatingAction, Else: move to the nearby
  location that has a tree which contains fruits on it and return EatingAction.
Elif food level is > 140: return wanderAction
Else: return DoNothingAction
```

<u>Rationale</u>:

We decided to implement the Brachiosaur class as an extension of the abstract Dinosaur class. This is to adhere to the DRY OOP principle. While extending the abstract class, we also made sure to only add functions to the Brachiosaur class and not remove any functionalities that are present in the Dinosaur class so as to follow the Liskov Design principle. By applying the DRY principle and Liskov Design Principle, the Brachiosaur class was easily created and only required a few additional functions that are specific to the Brachiosaur class.

VendingMachine(Eco points and purchasing)

`public class VendingMachine`

VendingMachine class is put in place for players to make purchases. It is included in the ground package since it will be in a fixed position throughout the game. The VendingMachine class functions similarly to a real vending machine in which the player is presented with a menu.

For example:
1. Allosaur Egg (200 points)
2. VegetarianMealKit (300 points)
3. Laser Gun (100 points)
Item selected: <Prompt for user input>

The user can then enter the number which represents the item they want to purchase. From there, the VendingMachine class accesses the VendingItems enumeration class to get the latest cost of the item selected.

Rationale: We decided to set VendingItems as an enumeration class because it will provide an easy way to update price of items if and when a vending item is needed outside of the VendingMachine class, removing the need to repeat ourselves in other classes when the price of an item changes.

`public class PurchasingAction extends Action`

After the VendingMachine gets the price of an item, it performs the player's purchasing action. A purchasing action was created to reduce the dependency of a purchasing functionality to the vending machine.

Rationale: This reduced dependency will allow the user to make purchases outside of the vending machine e.g between players if needed in future implementations. The purchasing action is also extended from the Action abstract subclass. This abstraction is made in accordance with the Liskov Design Principle, which will allow us to use important methods like execute(actor, map) while being able to increase functionality according to the purchasing function.

`public class EcoPointWallet`

EcoPointWallet. The EcoPointWallet uses the Object Oriented Programming concept of encapsulation. To keep track of the EcoPoints that a player has, we decided to implement an EcoPointWallet class where each player will have one instantiated object of this class. Within the EcoPointWallet class, there will be methods in place to make sure that a player never has negative EcoPoints. In the case an outside class is attempting to take more points that are available, the system will end the transaction. This attribute of the EcoPointWallet class follows the Fail Fast design principle and will return an error message “Player does not have enough points” before stopping the transaction.

The EcoPointWallet is also made to mimic a real life wallet, and can be easily extended to include more complex actions like borrowing between players etc because of the encapsulation. Furthermore, all methods within the EcoPointWallet class are made in accordance with the command-query principle. The methods focus on letting outside classes take EcoPoints from the player’s wallet like in the VendingMachine class. All these methods allow outside classes to have controlled access to the private attributes of the EcoPointWallet class like the number of EcoPoints currently stored. The setter and getter methods provide an initial level of protection against privacy leaks.

After the money is taken out of the player’s EcoPointWallet (player paid for the item), the selected item will be added to the player’s inventory. The class of the item added will depend on the item selected, i.e. an AllosaurEgg object will be added to the player’s inventory if that is what they purchased.

Rationale: The EcoPointWallet class was created in accordance with the FailFast principle and the command-query principle which will make the class simple and efficient in application. This simple application, when applied with the use of accessor and mutator methods on its private attributes will work together to provide basic protection against privacy leaks. This will make for a good software because it ensures that the private data is kept safe while maintaining the simplicity of the program in runtime.

Breeding(pregnant behaviour, mateAction, layEggAction)

```
public class mateAction extends Action
```

In order to simulate the breeding process, we have created three additional classes. The process begins with the Dinosaurs approaching each other. When they are close enough to each other, the mateAction is called. However, we have designed the mateAction to fail as soon as the gender of the dinosaurs is discovered to be the same. The check made early in the mating process allows the code to follow the Fail Fast Principle of OOP Design.

```
public class PregnantBehaviour extends Behaviour
```

Following the mating, the female of the pair gains the pregnantBehaviour in her DinoBehaviour array. A species specific Egg object will also be created and assigned to the dinosaur’s private attribute of egg. This will allow her to keep track of when an egg is due to be laid. Setters and getters enforce privacy while it allows classes outside of Dinosaur to see whether the dinosaur is pregnant or not, and if they can mate and otherwise. The Liskov Design Principle is used here when extending the Behaviour class allowed the Dinosaur to perform multiple actions like eat and attack all while being ‘pregnant’ with an egg. The female of the two will continue carrying an egg and keep track of the egg’s delivery date with the Egg’s tick function.

Rationale: All of the classes created to enable the breeding function are all extended by some type of abstract class. The reason behind extending each of these abstract classes is that it will adhere to the Don’t Repeat Yourself Object Oriented programming (OOP) principle. This reduces the complexity of code and also makes maintenance of code in the future exponentially easier.

Egg/Birth (GrowAction, BabyDinosaur)

```
public abstract class Egg extends Item
```

To simulate the hatching and growing process, we decided to create an abstract Egg class. This class was created because it would allow the Egg to exist away from the mother in situations where the egg is laid on the ground or when an egg is bought from the VendingMachine.

Additionally, the Egg abstract class also extends the Item abstract class. We decided to extend the Item class because of the tick function that was available in Item class so the Egg was able to determine on its own when it was time to hatch. The methods in the Egg class were also created in accordance with the command-query principle so there can be clarity in the functionality of all methods. Most of the methods in the Egg class are extended from the Item class, and if they are not, they are setters and getters for the Egg’s private attributes like HatchCountdown. By using setters and getters, we are able to provide a first level of protection against privacy leaks.

Rationale: The independent egg class reduced dependency allows the egg to be used in multiple ways which makes it easier to add features to the gameplay. The Egg class is made to be abstract so it could be extended by other egg types to include species specific information like hatch countdowns. The reason the attributes of the egg class are all private, and that only accessors and mutators are used to access those values is so as to protect the system against any privacy leaks which might leak sensitive information about the egg to users.

```
public class GrowAction extends Action
```

When it comes to the Egg specific actions, we created the GrowAction class which extends the abstract Action class by adding functionalities specific to the growing of an egg.

Rationale: The Liskov Design Principle is used while extending the action class in order to provide additional functionality to GrowAction class. Extending the abstract Action class also allowed us to follow the 'Don't Repeat Yourself' Principle of OOP since we were able to call the super class's execute function instead of having to implement a new one from scratch in each of the new classes.

```
public abstract class BabyDinosaur extends Actor
```

After the Egg hatches, it will create a new BabyDinosaur object that corresponds to the species. The baby dinosaur will then keep track of the turns made since it became a baby dinosaur, and when a limit is reached, the growAction method will be called and it will become a grown dinosaur.

Rationale: We created BabyDinosaur in response to the DRY principle as there are many species of dinosaurs that have specific attributes as babies but a majority of the baby dinosaurs function similarly, an abstract class that would be extended made more sense. The transition from baby dinosaur to adult dinosaur is made extremely simple because baby dinosaur class and dinosaur class both extend the Actor class, which implies that they have mainly similar functions.

Allosaurs

Allosaurs have the similar behaviours as Stegosaurs where being able to feed, breed, and grow, so it contains BreedBehaviour and DinosaurBehaviour. However, allosaurs are able to attack stegosaurs and they are able to eat corpses and eggs. A method is created in HungerBehaviour class with condition to check whether the dinosaur is carnivores, if true that means it's capable of attacking Stegosaurs in a nearby location. If there is no AttackAction is returned, then if there's food in the current location the dinosaur will eat it, otherwise move to the closest location that contains food.

```
public class HungerBehaviour implements Behaviour
```

If carnivore: return AttackAction if there's a stegosaur is nearby.

If the current location has food: EatingAction, Else: return MoveActorAction to move to a nearby location that has food. This if else statement applies to both herbivores and carnivores.

Rationale: HungerBehaviour is implemented which can apply to both herbivores and carnivores, thus it is object oriented for both herbivores and carnivores dinosaurs.

Death

`public class DieAction extends Action`

A DieAction class is created and extends the Action class. When a dinosaur dies, the dinosaur will be removed from its current location and a Corpse object will be added to the same location.

Rationale: DieAction class extends the abstract class by adding functionality specific to the death. Extending the abstract class is important for the DieAction due to the requirement of removing the dead dinosaur and adding a new Corpse object in the same location. The Liskov Design principle is applied in this while extending the action class in order to provide additional functionality to DieAction class.

Lakes, water and rain

`public class Lake extends Ground`

A Lake class is created and extends the Ground class, it contains water and fishes. All dinosaurs will be able to drink the water by being beside it, only pterodactyl dinosaurs are able to traverse the lake and eat the fishes in the lake as they are the only flying dinosaurs. There is a probability of 60% that a new fish is born in the lake. Overriding the tick() method to decrease the waterSips by 1 each turn and it has to check the probability to increase the number of fishes as well and Lake may dry up if water sips are zero.

```
public class DinoGameMap extends GameMap
public class DinoWorld extends World
```

A rain() method is created in DinoGameMap to make the game map to experience the rain; in rain(), it increases the water sips of all lake in the game map and the water level of the unconscious dinosaur.
DinoWorld class is created to extend World, override the run() method and add new functionality for rain into it, where it has to check the condition where every 10 turns there's a chance to rain and if it rains, it'll loop through the gameMap to call rain() from DinoGameMap.

Rationale: Lake class extends the Ground abstract class by adding functionality specific to the lake, water where DinoGameMap class extends the GameMap abstract class and DinoWorld class extends the World abstract class by adding functionality specific to the rain. It's important to extend the abstract class for the requirement of overriding the tick(); the lake or GameMap can also experience the tick() and increase the number of fishes for Lake class and increase the waterSips for all lakes in GameMap for each turn. This applies the Liskov Design Principle and Polymorphism in order to provide additional functionality to Lake class and GameMap class.

Thirsty dinosaurs

A new attribute is added into the Dinosaur class to state the water level of dinosaurs. All dinosaurs have the same initial water level of 60 and the same minimum water level of 40 before getting unconscious, however the maximum water level differs based on different species of dinosaurs. The water level should be handled similar to the food level where it will decrease by 1 on each turn and dinosaurs become unconscious when water level is 0.

```
public class ThirstyBehaviour extends behaviour
public class DrinkAction extends action
```

ThirstyBehaviour class is created, if there's a lake around the current location of dinosaurs and the lake contains water sips then return DrinkAction() to increase the water level of dinosaurs and decrease the waterSips of the lake, else call moveCloser() which move the actor closer to the location where contains lake.

Rationale: Instead of combining the Thirsty behaviour and Hunger behaviour into one class, we decided to separate them by creating additional ThirstyBehaviour and DrinkAction class to handle the thirsty dinosaurs for drinking only. This applies the single responsibility principle where each class has a single purpose and all its methods should relate to function as fewer responsibilities leads to fewer opportunities to introduce bugs during changes.

Pterodactyl

```
public class Pterodactyl extends Dinosaur
public class BabyPterodactyl extends BabyDinosaur
public class PterodactylEgg extends Egg
```

Pterodactyls are small flying dinosaurs and carnivorous, they are able to traverse into lakes and eat the fishes. A new FLY capability is added to DinosaurCapabilities for pterodactyl. A Pterodactyl and BabyPterodactyl class are created and add FLY and CARNIVORE to its capabilities, a PterodactylEgg class are created as well, the eco points for pterodactyl egg is similar to Stegosaur. The feature of pterodactyl able to traverse into lakes will be handled in the Lake class.

```
public class LandingBehaviour extends Behaviour
public class LandingTakeOffAction extends Action
```

Pterodactyls can fly over 30 squares before landing on top of a tree or a corpse then they need to land and walk to a tree before they can take off again. Thus, LandingBehaviour and LandingAction are created to handle this case. LandingBehaviour class allows the pterodactyl to land on tree or corpse if they are standing on tree or corpse then calls the LandingTakeOffAction(), else move closer to place where it contains tree or corpse. LandingAction class handles the land or take off action and return the description of action. The rest of the behaviours are handled by the previous implementation for dinosaurs.

Rationale: Overriding a method of canActorEnter() in Lake class to check if the actor is a Pterodactyl or BabyPterodactyl is better than implementing a new abstract method in Dinosaur class to check whether the dinosaur is flight-based. This is because the Pterodactyl species is the only actor that can enter the lake, thus no further implementation is required for this. Creating Landing action instead of putting the same code in pregnant behaviour, landing behaviour, and hunger behaviour also follows the Don't Repeat Yourself principle and the Single Responsibility Principle where Landing action is the only class that can land a dinosaur.

Second Map

A new FancyGroundFactory is initialised with all types of Ground that would appear on map and a new GameMap is initialised with a List<String> and it has the same dimension as the existing map for the second map. A linkMapVertically() is created in the Application class by adding exits that are in the direction of North, North-East and North-West are required to be added for the first row of the below map, exits that are in the direction of South, South-East and South-West are required to be added for the last row of the above map. Therefore, Actors on the map will be able to travel between both maps in one world where it can exit at the borders of 2 maps.

Rationale: We decided to create a linkMapVertically() method in Application instead of hardcoding the code to link both maps vertically. This applies to the Open-closed principle where it's open for extension and closed for modification. If more new maps are required to be created and they are the same dimension as the existing map, we can easily link the game map by calling the linkMapVertically() instead of hardcoding it for the new map.

A more sophisticated game driver

The required elements of the sophisticated game driver was mostly done by putting all world-building elements to a while loop that is terminated when the user selects the "End game" option.

```
public class QuitAction extends Action
```

A QuitAction class is created and extends the Action abstract class. When the execute() method is called, it'll return the actor(Player) from the dino game map and return the descriptive string that states whether the player has won or lost the game. The run() method on DinoWorld will terminate the game.

```
public class Player extends Actor
```

The playTurn() in Player class is modified by adding a if statement; if the game is a challenge mode and there's no challengeRound left, then it'll check the Player's current eco point and compare to the fixed challengePoints to determine whether the player has won or lost the game and return the QuitAction().

Recommendations for extensions to the game engine

1. **Move closer function in Location class.**
   a. Problems this extension will solve
      i. Removes the need to implement same version of this method in any instance which would require a user to move closer to look for a target
   b. Advantages
      i. Removed need to implement the same method in multiple classes will be following the DRY principle
      ii. A standardised format for location selection which can be used to reduce the complexity of possible student-implemented version of this method
   c. The design of proposed extension
      i. A moveCloser(Actor target) and moveCloser(Ground target) will be added to Location
      ii. Method signatures:
         1. `public Action moveCloser(Actor actor)`
         2. `public Action moveCloser(Ground target)`
      iii. In both versions of the method, there will be two for-each loops that will loop through each exit of the exits of the current location.
2. **Add Edible interface**
   a. Problems this extension will solve
      i. An edible interface that can be implemented by any class will be helpful in giving the objects of those classes mandatory values and methods like getFoodValue and allow it to be used in EatingAction(for this case) or any situation where an object of a class would be allowed to be eaten by other elements of the game
   b. Advantages

   i. An advantage of this extension is that there would be a standardised approach to the implementation of points and eating in these games, and would follow Liskov's Design Principle where the child classes can have extra methods but still be able to perform all the methods in its parent classes
   ii. Using an interface is also beneficial because a class can have multiple inheritance for interface which would make anything and everything in the game possibly edible.
   iii. This interface will also reinforce the Open-Closed Principle where any extensions to this interface can be made, but not any changes like removing a method can be made. This will make sure that all the classes that inherit this edible interface will not run into issues of maintaining and running a function that no longer is useful.
   iv. An interface will also adhere to the Interface Segregation Principle where this interface will only be inherited by those classes of elements that will be edible in the game. Putting getFoodPoints() and getIsEdible() in other classes like Player or Item would make it semi-redundant, which is not optimal.

c. Possible disadvantages
   i. It is possible that the method might be misinterpreted and will be misused in classes that extend this interface, making a possibility of the code breaking.

d. The design of proposed extension
   i. A getFoodPoints() method and a getIsEdible() method in the Edible interface
   ii. Method signatures:
      1. `int getFoodPoints()`
      2. `boolean getIsEdible()`

3. Get actor locations in map

a. Problems this extension will solve
   i. This extension will make it possible to access the location of all actors in the game without having to extend the world or map class, hence reducing the load during runtime

b. Advantages
   i. Can be used in additional implementation of any case that would affect all actors in the game. An example of this in the case for dinosaurs would be something like a mass extinction where a majority of dinosaurs will die. In other implementations, there could be instances like a global pandemic or a heatwave that can wipe out everything on the map and effectively end the game.
   ii. Reduces dependencies because the Map and world classes do not have to be extended to gain access to the actorLocations ArrayList.

iii. Privacy. Implementing this method in the engine class would allow developers to create a method that can return the actor locations in a controlled and secure way to prevent any possibility of privacy leaks.

c. The design of proposed extension

i. An additional function in the GameMap class which can return the instance variable actorLocations of GameMap.

ii. Method signature:

1. `public ActorLocations getActorLocations()`

iii. The reason behind making the method public is because it needs to be called in outside classes in order to simulate global events.

iv. Placing this method specifically in the GameMap class is because the GameMap class is already accessed in most Action classes. This would maintain the dependencies between classes.

4. Add and Link game map function in World class

a. Problems this extension will solve

i. This additional functionality will make it possible to expand the game map easily as there are multiple game maps that are available in a game and allows the Actor to cross all those game maps easily.

b. Advantages

i. Instead of hardcoding by adding exits to the borders of both maps, these functions can be called to link the same dimension maps together. This applies the open-closed principle that allows us to add more new features and doesn't change the way we use the existing code.

c. The design of proposed extension

i. A modification function in the World class which links the maps together.

ii. Method signatures:

1. `public void addLinkGameMap(GameMap gameMap1, GameMap gameMap2, boolean Vertically)`
2. `private void addLinkGameMapVertically(GameMap aboveGameMap, GameMap belowGameMap)`
3. `private void addLinkGameMapHorizontally(GameMap leftGameMap, GameMap rightGameMap)`

iii. The boolean Vertically parameter of addLinkGameMap is to determine which function(addLinkGameMapVertically() and addLinkGameMapHorizontally() to call in addLinkGameMap().

iv. The reason behind making addLinkGameMap() public because it can be called in the Application class and addLinkGameMapVertically() and addLinkGameMapHorizontally() as private because it will be better to be able to access only the addLinkGameMap().