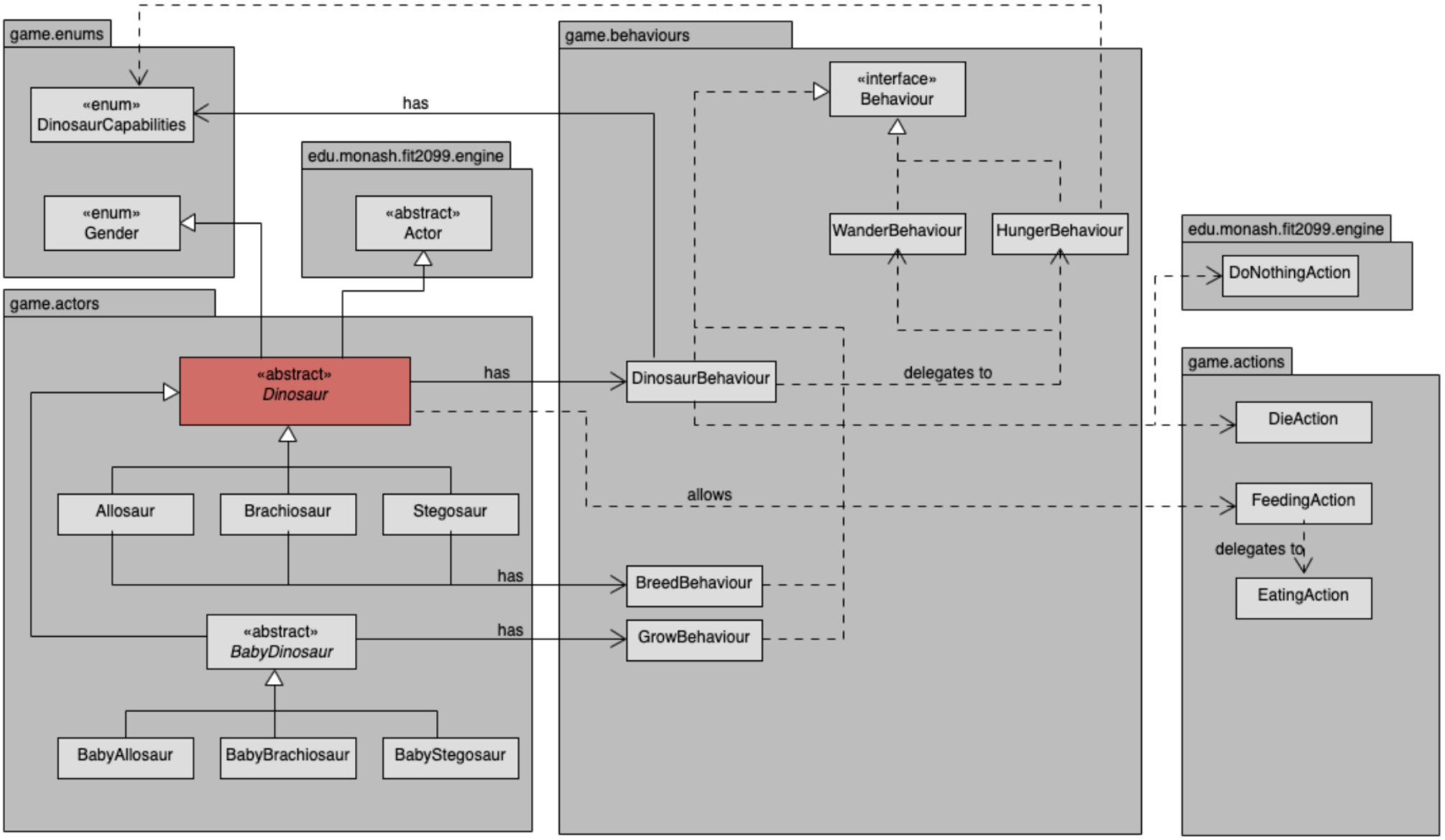
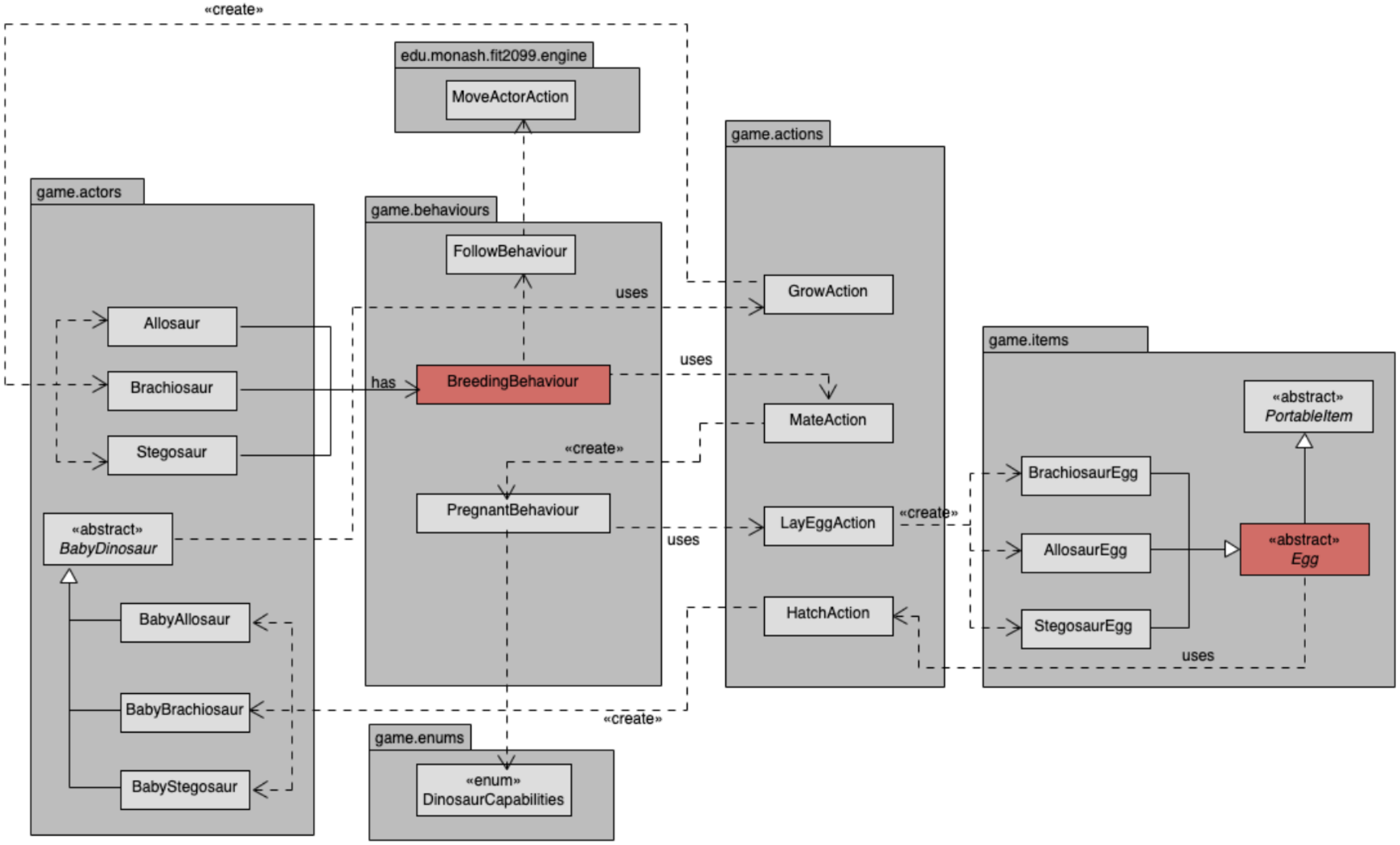
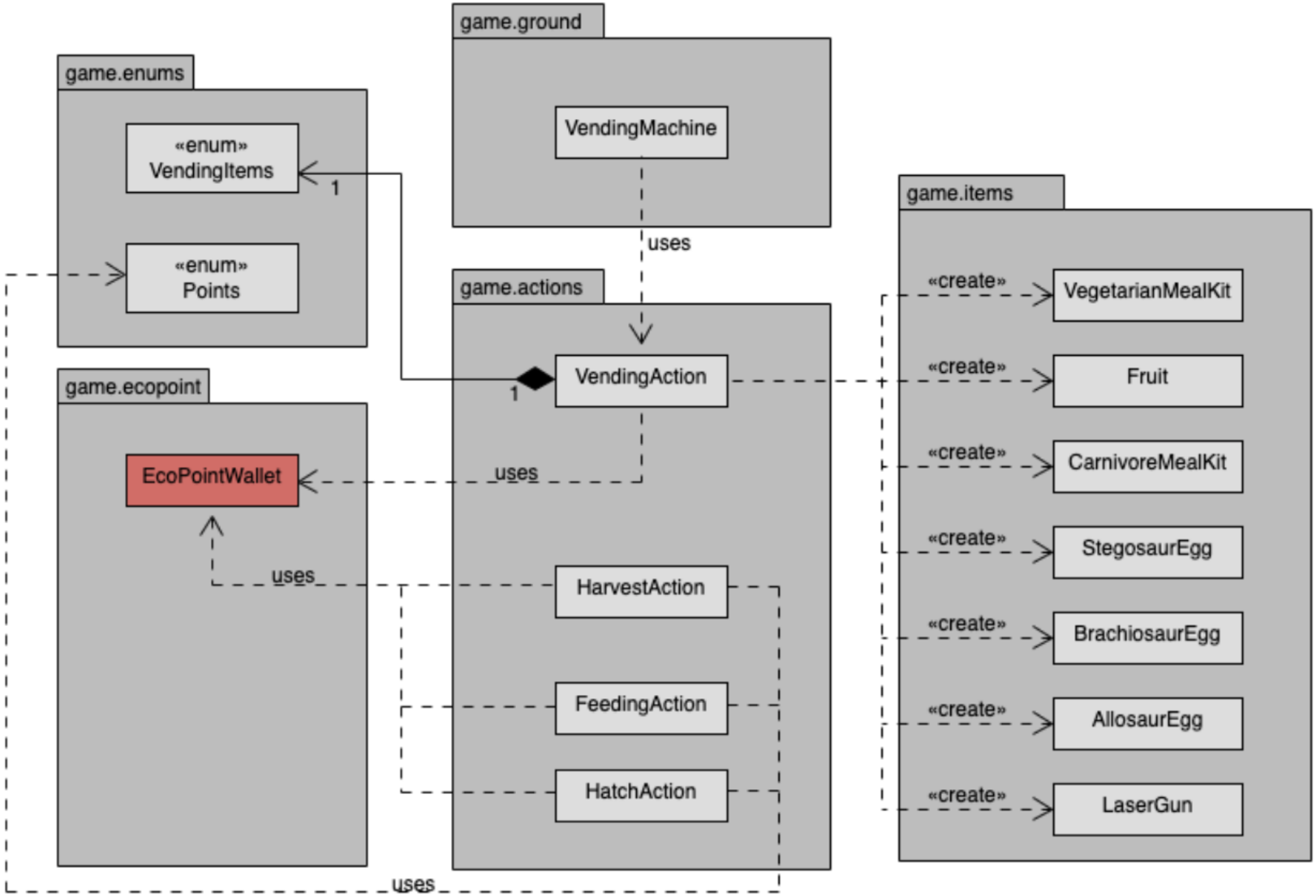
# Dinosaur Class Diagram



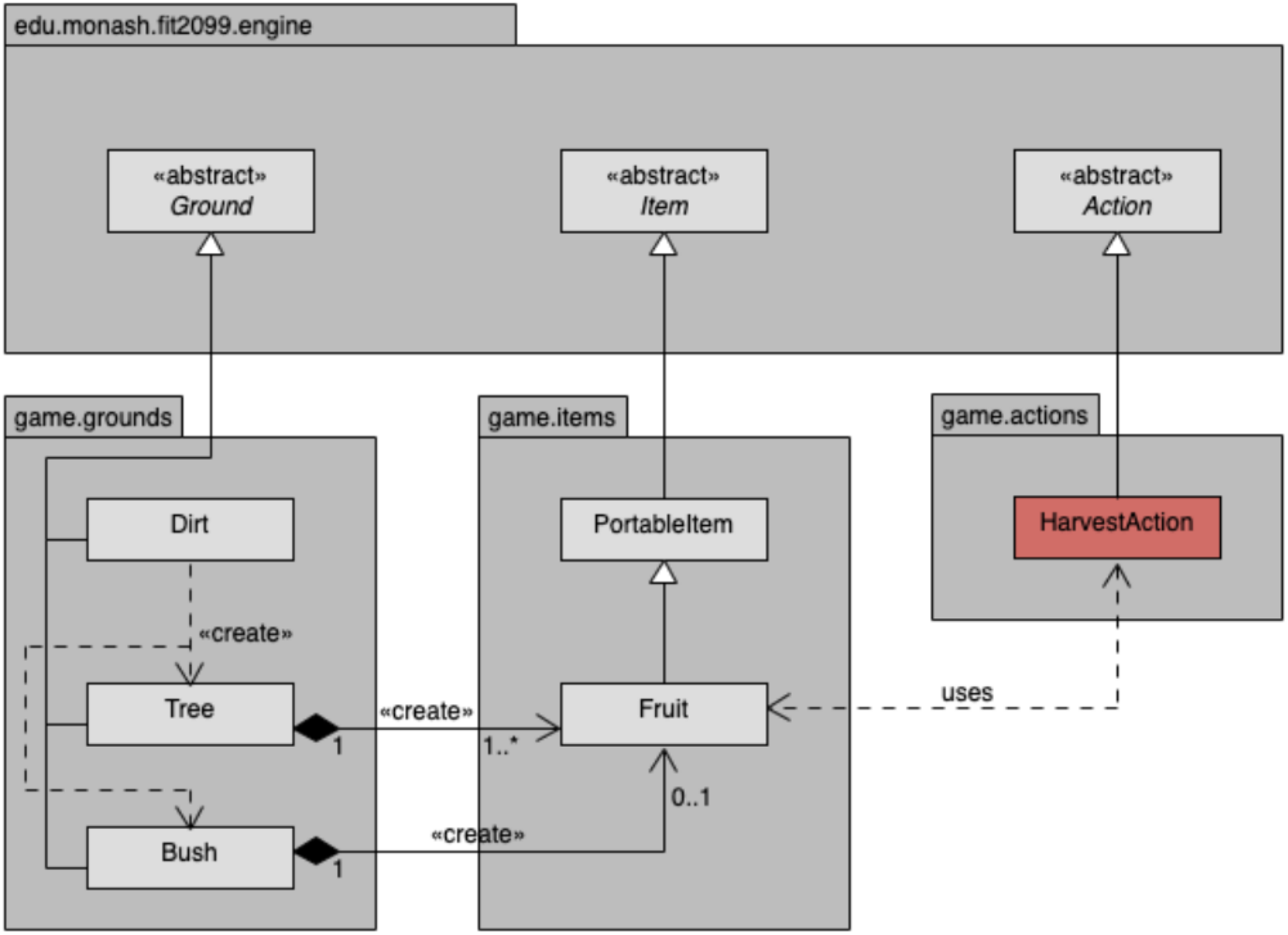
## Breeding Class Diagram



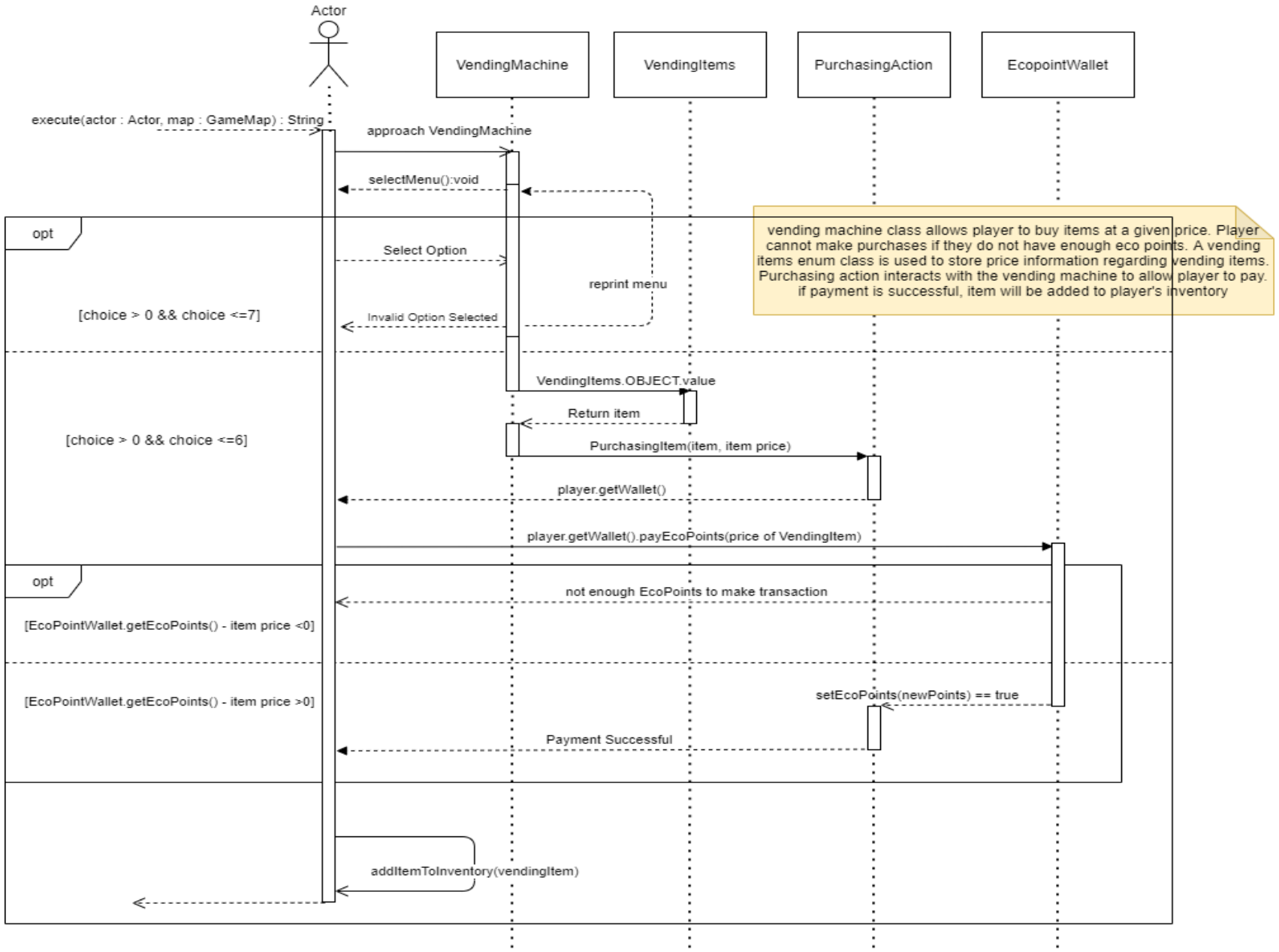
**EcoPoint Class Diagram**



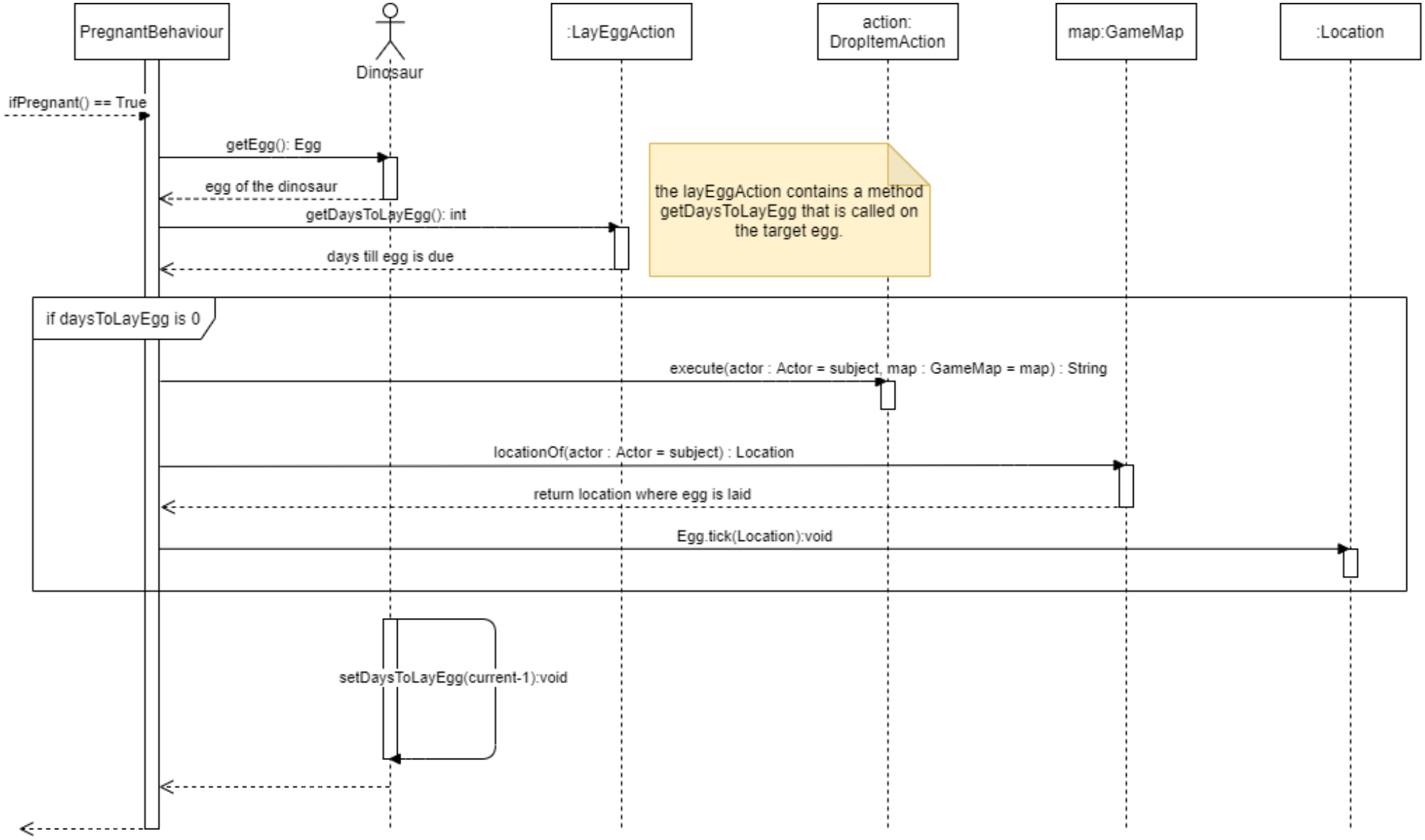
**Plants Class Diagram**



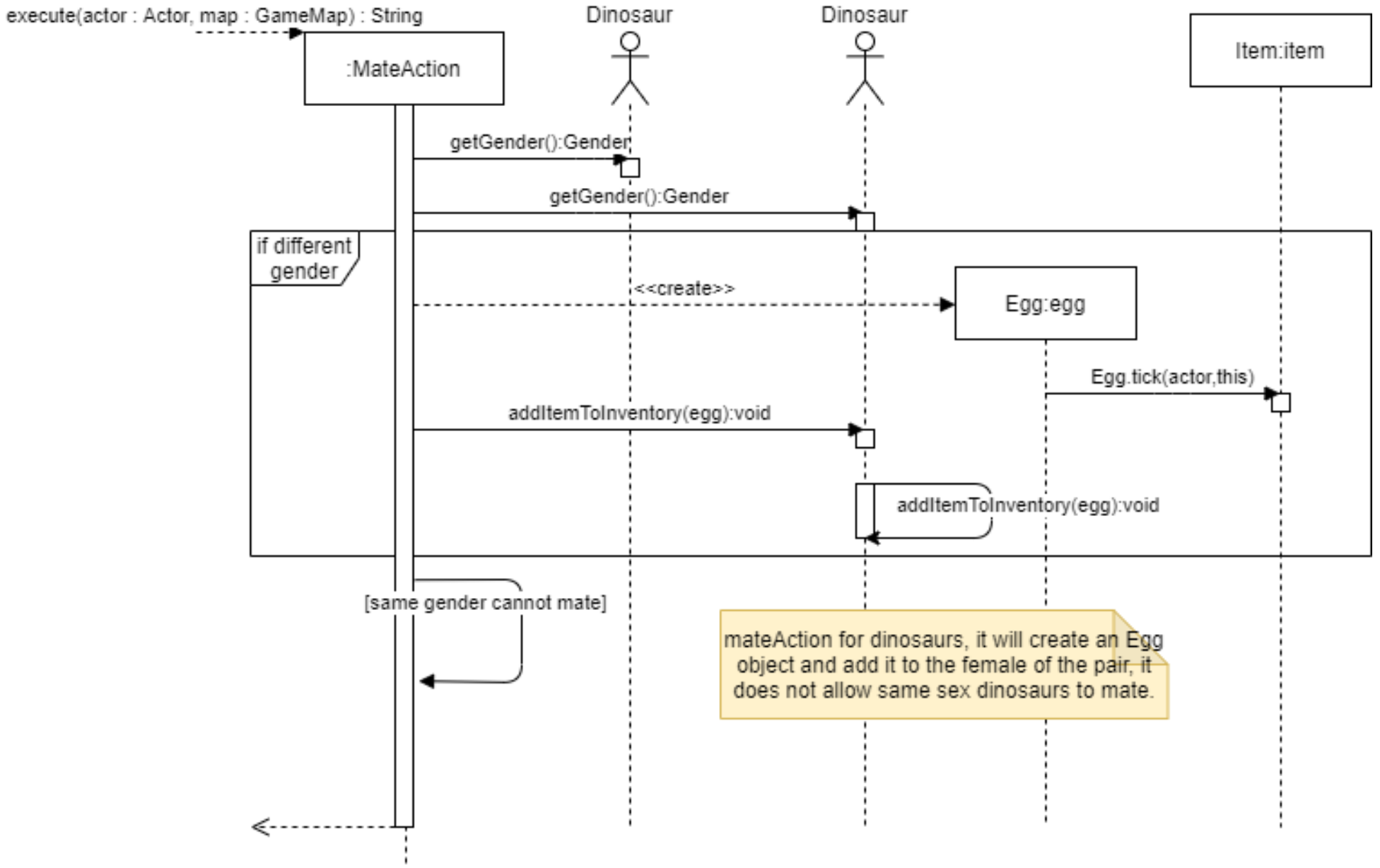
# Vending Machine Interaction Diagram



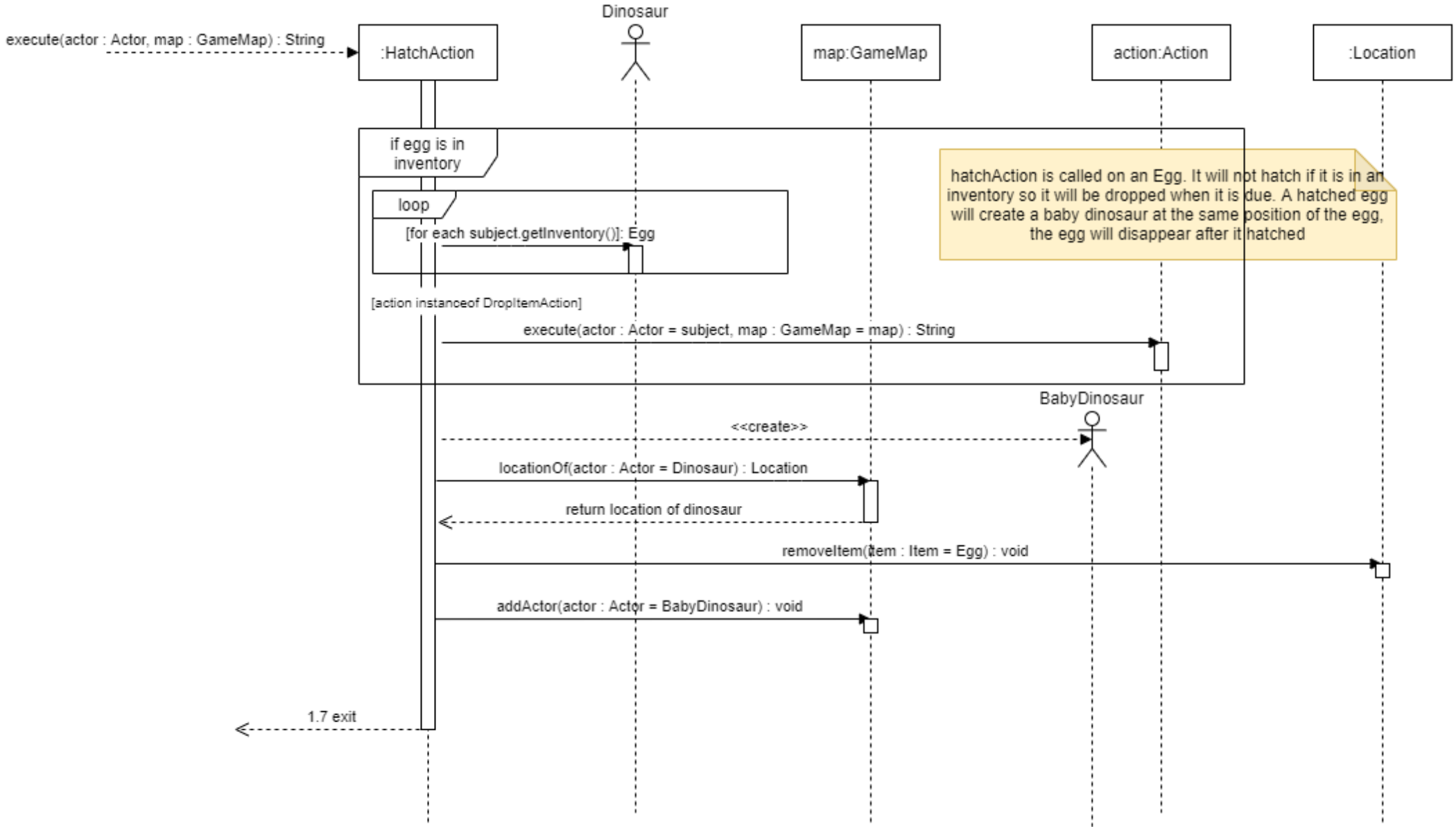
**PregnancyBehaviour and Egg Laying Interaction Diagram**



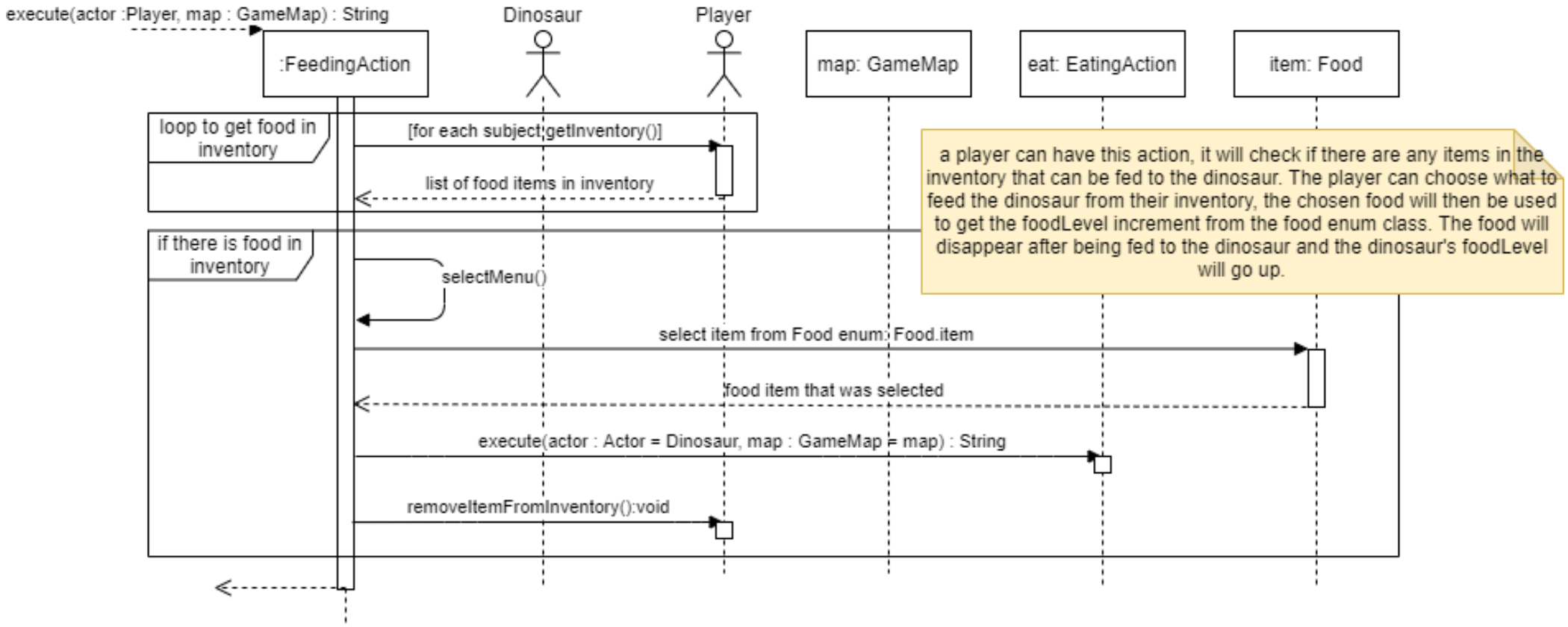
**MateAction Interaction Diagram**



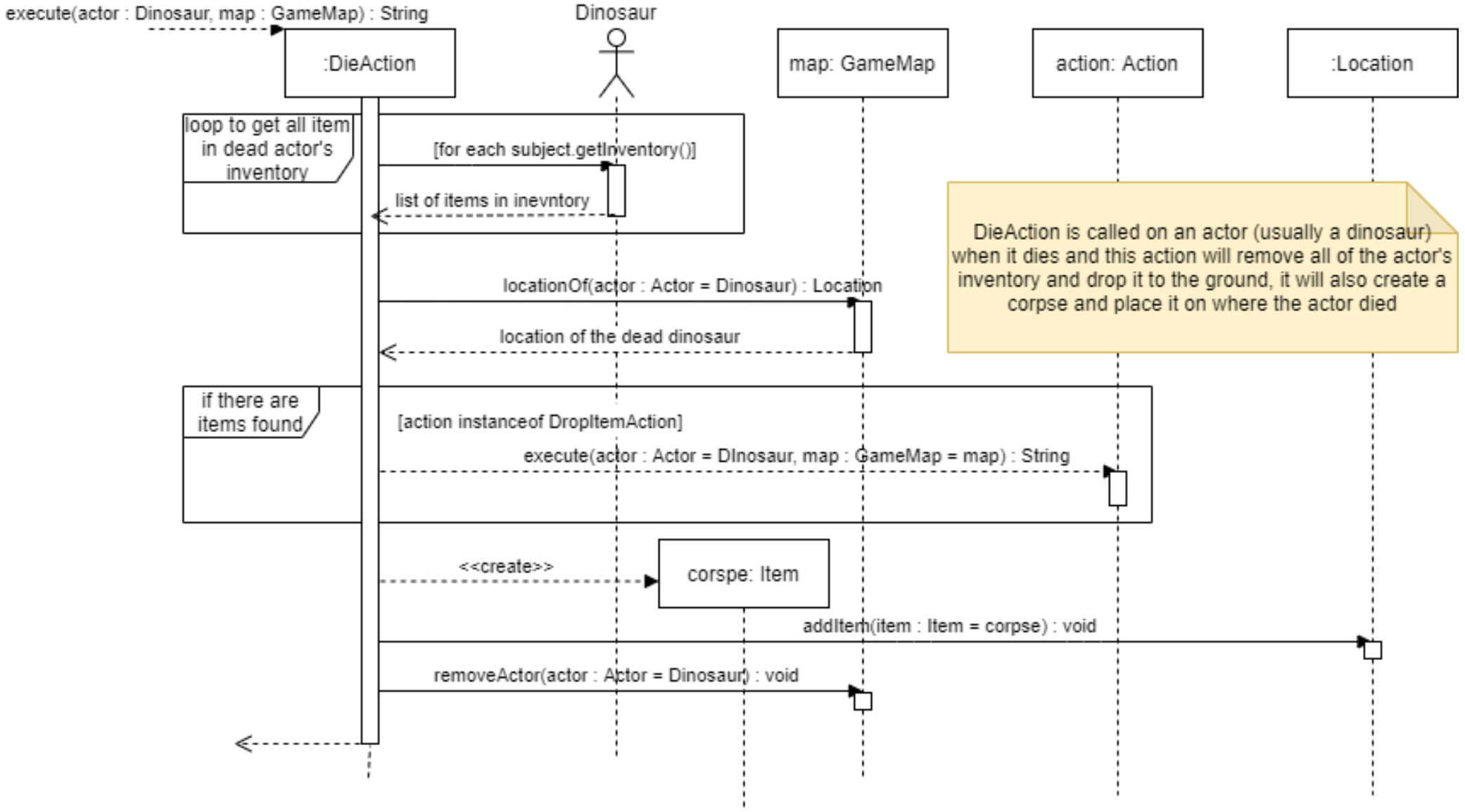
## HatchAction Interaction Diagram



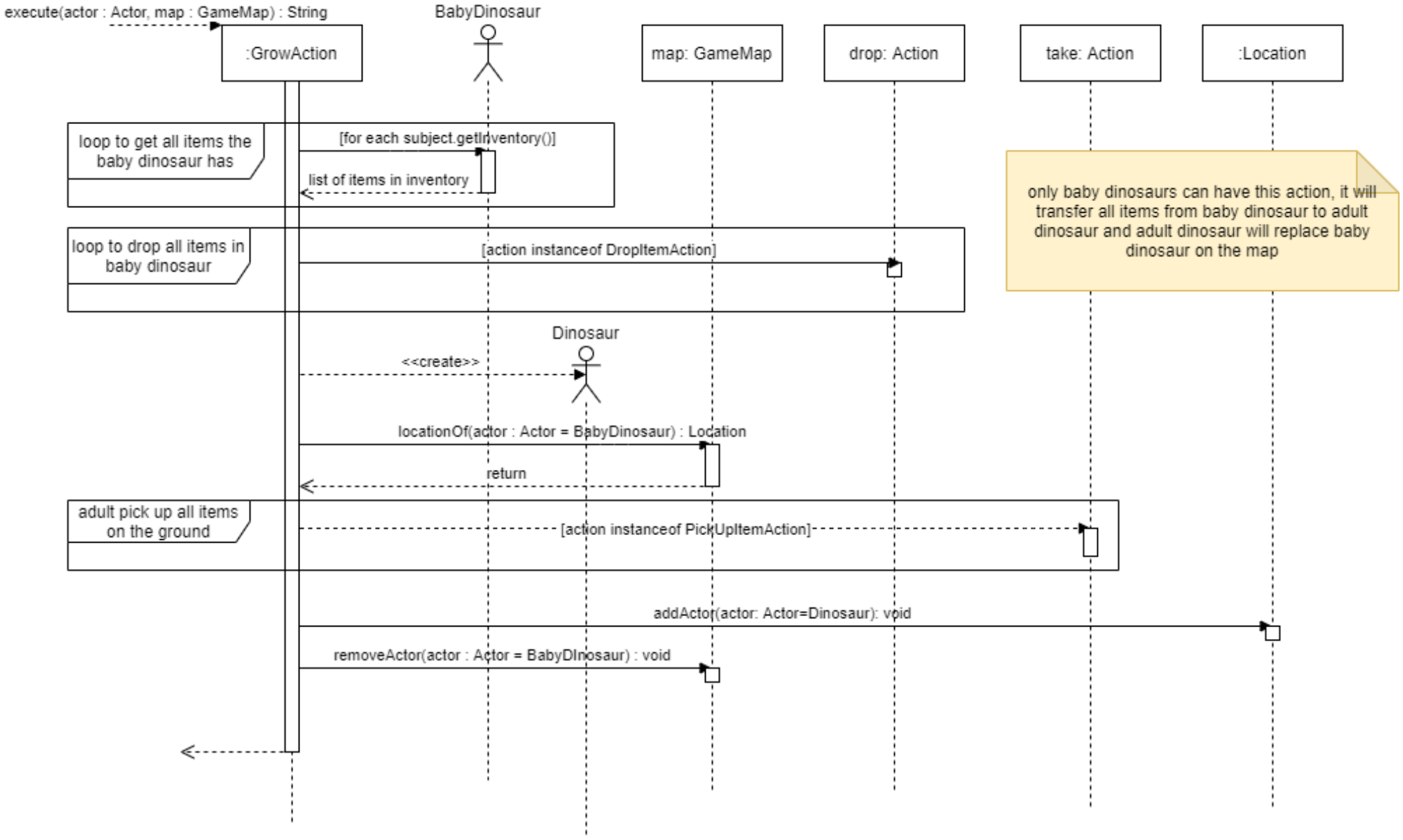
**FeedingAction Interaction Diagram**



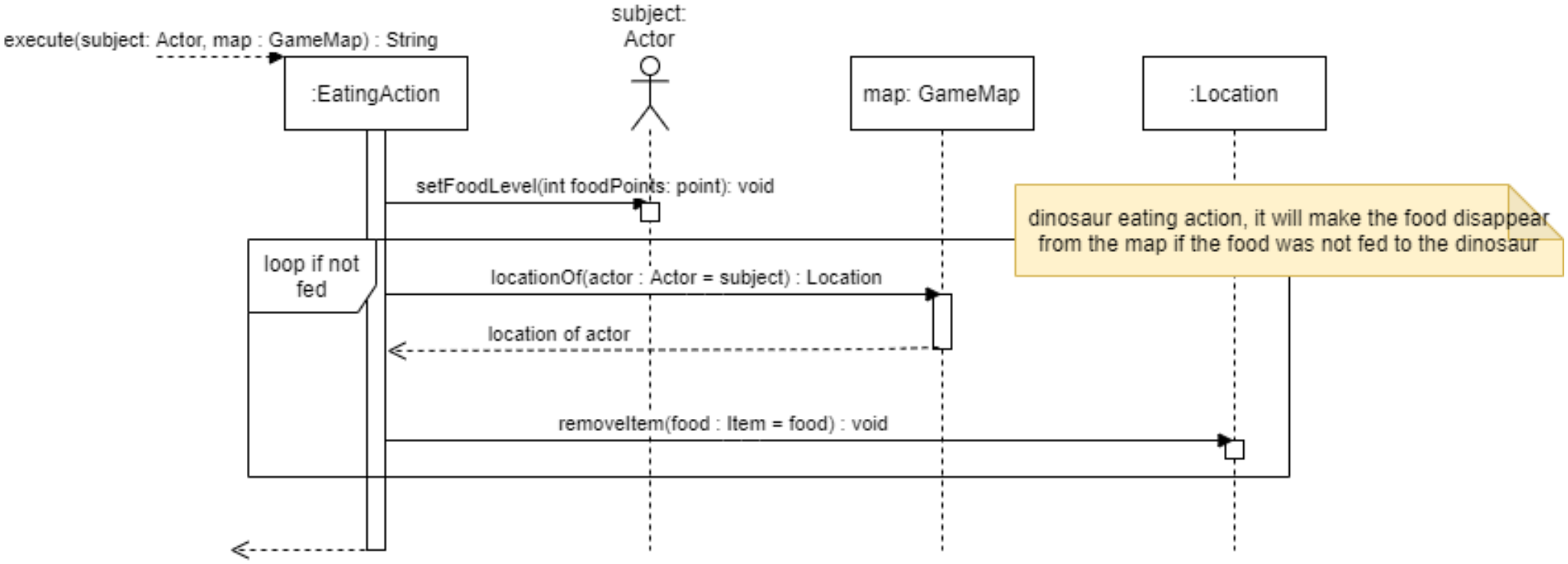
**DieAction Interaction Diagram**



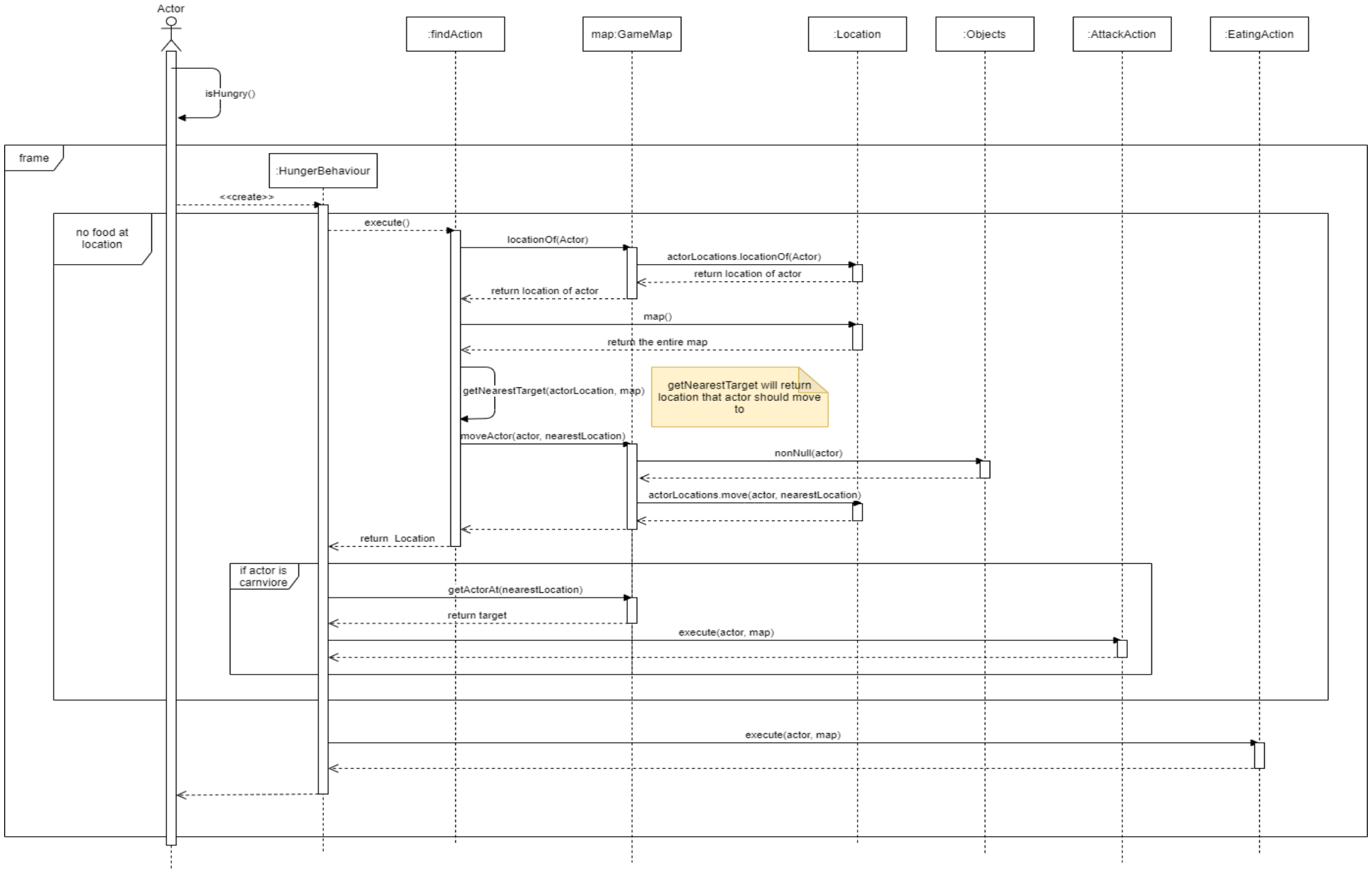
## GrowAction Interaction Diagram



**EatingAction Interaction Diagram**



# HungerBehaviour Interaction Diagram



# 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 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

**Rationale**:

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 and 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 (HatchAction, 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 HatchAction extends Action` **AND** `public class GrowAction extends Action`

When it comes to the Egg specific actions, we created the HatchAction and GrowAction classes, both of which extend the abstract Action class by adding functionalities specific to the hatching and growing of an egg. Extending the abstract class was extremely important for the HatchAction specifically because it required the Egg object to be placed on the ground since an egg cannot hatch in a player’s inventory.

**Rationale:** The Liskov Design Principle is used while extending the action class in order to provide additional functionality to HatchAction and GrowAction classes. 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.