**FIT2099 Assignment 2**

**Design Rationale**

**Tutorial 9 Team 9**

**Prepared by:**

**Joanne Ang Soo Yin 30513723**

**Nelly Tay Yee Ting 30312523**

**Jonathan Wong Leong Shan 31435297**

**DinoActor abstract class - extends Actor**

**Subclasses: Stegosaur, Brachiosaur and Allosaur**

All dinosaurs share a lot of common attributes and methods, such as age, printing out a message when they get hungry etc. An abstract class DinoActorthat inherits from Actor is created. In turn, all Actors which are dinosaurs, namely *S*tegosaur, Brachiosaur and Allosaur, will inherit from DinoActor*.*

Some considerations made for the design include:

* Don’t repeat yourself principle (common methods and attributes can be shared) and improve maintainability by utilizing abstraction
* Liskov’s Substitution Principle and the Principle of Sound Contracting
  + The subclasses of DinoActor have constructors that have 2 parameters less than DinoActor

**Behaviours and Actions**

DinoActors functionality - feeding, breeding, attacking, getting pregnant then laying eggs, have a dedicated class that implements Behaviour and class that extends Action.

|  |  |
| --- | --- |
| Breeding | BreedingBehaviour and BreedingAction |
| Feeding (on its own, not fed by player) | Feeding Behaviour and FeedingAction |
| Fed by player | FedByPlayerBehaviour and PlayerFeedAction |
| Attacking another DinoActor | AttackBehaviour and AttackAction |
| Pregnancy | PregnancyBehaviour and LayEggAction |
| Following a target | (i) FollowMateBehaviour  (ii) FollowFoodOnGroundBehaviour  (iii) FollowFoodOnPlantBehaviour  (iv) FollowVictimBehaviour  and MoveActorAction |

Classes that implement Behaviour have an advantage, which is its getAction method can return an Action or null whenever appropriate.

Leveraging on this, we implemented the DinoActor’s behaviour, such that all conditions that need to be checked before an Action can be returned is packed in its getAction method. If all necessary conditions are met, the appropriate Action is returned. If not, null can be returned. It also helps us to do any necessary updating of the Actor's state.

For example, PregnancyBehaviour and LayEggAction:

When the getAction method of PregnancyBehaviour is called, it will check if the actor is a pregnant DinoActor that is due to lay an egg. If it is, LayEggAction is returned, otherwise it will decrement the pregnant DinoActor’s pregnancy period.

Note that since most of the functionality implemented this way are quite straightforward, in later sections we will only highlight more complex functionality such as the following functionality which involves using inheritance.

The motivation behind this:

* Information hiding is achieved. The DinoActor base class does not need to care about the conditions that need to be for a certain Action to occur, i.e. it does not need to know that a DinoActor needs to be a female to be pregnant.
* Reduce clutter in playTurn method in the DinoActor base class, cleaner code means better maintainability.
* The single responsibility principle from SOLID is also achieved in a sense that the role of a Behaviour-implementing class and their corresponding Actions perform a single role together, simulating a specific behaviour together. Whenever there is something wrong with that behaviour, the programmers can just look into that to find the error.
* Ease of extending code if any conditions need to be added or altered in the future, we can simply add it in the Behaviour class. If it needs to be removed, we just can delete the class and not add that behaviour into the DinoActor’s attribute, the rest of the code is not affected.

**DinoEncyclopedia Enum class**

There are a lot of constant values that we need to keep track of for dinosaur Actors, such as: number of turns till the pregnant dinosaur lays an egg, number of turns till a baby dinosaur reaches adulthood, initial food level etc.

These constants belong to their corresponding dinosaur classes, not to a specific any object. Hence, in order to have a standardized set of values necessary for initialization or any other usage, they are stored in the DinoEncyclopedia class.

The motivation behind this:

* Cleaner code in Stegosaur, Brachiosaur and Allosaur since they can have significantly less fields needed to store constants
* Prevent excessive hard coding
* Having a standardized set of values, which can be accessed by other classes apart from dinosaur Actors making the values uniform throughout the project
* Separation of concerns and single point of change, whenever we want to change a value simply look into this class
* Hence, allows for easier maintenance

**DinoCapabilities Enum class**

In the design of the first assignment, there were several small Enum classes (PregnancyStatus, BreedingCapability)created to be used as Capabilities. We have combined the common Capabilities that DinoActors may possess into a single Enum class so that there will not be too many classes containing merely one or two Enum values.

The motivation behind utilizing Capabilities:

* Integrates well with the existing system, can utilize hasCapability and addCapability method.
* These capabilities with their accompanying accessors (small functions that return true if the DinoActor has such a Capability, false otherwise) will help to prevent excessive usage of the instanceOf keyword at multiple classes. Most of the time, we only need to downcast once to the base class, making the design more Object-Oriented. Though many accessors are needed, these small methods are safe and will not cause side effects.
* Adhering to cohesion, since this class is solely for all Enums that should be used as Capabilities for DinoActors.

**Dinosaurs growing up**

The dinosaur actors shall be represented in the console with the first letter of their names, whereby the lowercase form represents a baby dinosaur and the uppercase form represents a grown up dinosaur, eg: ‘a’ - baby Allosaur, ‘B’ - adult Brachiosaur, ‘s’ - baby Stegosaur

An age attribute is added to the DinoActor class, with appropriate methods to increment the age and change the displayChar in the playTurn method. This is good design since each class should be responsible for their own properties.

**Following**

This section includes the explanation of all DinoActors following mates, food or victims (Allosaur following Stegosaur to attack it when hungry). The original class FollowBehaviour was modified significantly, since at first it **only detected Actors that were at eight immediate locations around it.** Note that after modification, **a suitable range** is set for different behaviours such that it will search locations for targets to follow **within the range MIN\_RADIUS to MAX\_RADIUS**. Modifications have also been made to make it **more flexible and be able to follow Items or Ground (Tree or Bush that has fruit) as well.** This helps to reduce redundancies significantly, and will also make any future extensions easy to implement. To achieve this, FollowBehaviour is made into an abstract class, which will be the base class for four other subclasses.

**Follow Behaviour abstract class - implements Behaviour**

**Subclasses: FollowFoodOnGroundBehaviour, FollowFoodOnPlantBehaviour, FollowMateBehaviour and FollowVictimBehaviour**

The FollowBehaviour class is modified to contain methods that compose the core logic of the following functionality, with some abstract methods that need to be implemented by its child classes.

Methods lookAround and getSpottedLocations are methods that work together to obtain the locations which are x number of squares away from the actor. Methods moveCloser and distance will work together to determine the next step the Actor should take to move closer to the target.

Allowing an Actor to follow another Actor, Item or Ground is achieved by having two abstract methods:

1. motivatedToFollow - checks whether an Actor has fulfilled conditions that make it actively look for a target to follow, e.g. DinoActor only follows food when it is hungry etc.
2. findTarget - given a location, determine whether what the DinoActor intends to follow is on that location. If yes, the location that the target is on will be returned, otherwise it will return null.

The code in getAction pieces everything together, steps are as follows :

1. Check if the actor wants to follow something, by calling motivatedToFollow method. If yes, proceed to step 2, otherwise null is returned.
2. Starting from radius x = MIN\_RADIUS to start searching, get all locations that are x squares away from the Actor
3. For all the locations found, check one by one whether the Actor will find its target on the location by calling the findTarget function.
   1. If a location is returned eventually and not null, the moveCloser function will be called, which returns a MoveActorAction that helps the Actor move towards the target.
   2. If after looping through all locations and the Actor does not find its target, steps 1 - 3 will be repeated with radius x + 1, until a MoveActorAction is returned or radius exceeds MAX\_RADIUS.

As mentioned above, all subclasses will override the abstract motivatedToFollow method and findTarget method to enable tailoring of the following behaviours specific to the motivation of the DinoActor.

To emphasize, the principles considered into this part of the design include:

* Adherence to the Don’t Repeat Yourself Principle by utilizing abstraction, inheritance and overriding
* Easy to extend code if we want the dinosaurs to follow other things in the future
* Liskov’s Substitution Principle and the Principle of Sound Contracting
  + The subclasses of FollowBehaviour have constructors that have 0 parameters while FollowBehaviour has 3 parameters

**Egg class - extends PortableItem**

The display character of Egg is ‘o’.

The design of this class adheres to the Single Responsibility principle, as its only role is to represent the Egg entity that will eventually hatch into another DinoActor. Static constants were used in this class to store the number of turns that an Egg needs to wait till it hatches for each dinosaur species with its corresponding number of EcoPoints. This is to prevent hard coding literals and provide a single point of change. This class also integrates well with the base code since extending PortableItem allows an Egg to be picked up by Player directly with the existing engine, without extra changes.

.