FIT2099 Assignment 3

Design Rationale

Tutorial 9 Team 9

## Prepared by:

### **Joanne Ang Soo Yin 30513723**

**Jonathan Wong Leong Shan 31435297**

### **Nelly Tay Yee Ting 30312523**

## Symbols

For easy reference, the symbols that will be displayed on the console for are as such:

|  |  |
| --- | --- |
| Stegosaur | Baby stage – s Adult stage – S |
| Allosaur | Baby stage – a Adult stage – A |
| Brachiosaur | Baby stage – b Adult stage – B |
| Pterodactyl | Baby stage – p  Adult stage – P |
| Fruit on ground | F |
| Bush | Small – ~ Big – \*  Has fruit – ^ |
| Tree | Small – + Medium – t Big – T  Has fruit – & |
| Lake | ~ |
| Egg | o |
| Corpse | % |
| MealKitsItem | = |
| Bridge | ! |

## 

## For easy reference, the diagram above is to illustrate the inheritance relationship between the classes involved in creating dinosaur classes.

## CapableActor Abstract class – extends Actor

## This is an entirely new class created in Assignment 3. The purpose of this class is to store a lot of small methods to check whether an Actor has certain capabilities from the DinoCapabilities Enum class, then decide what to do next. For example, using the isConscious method, we will check whether a DinoActor is conscious and if it is, it will not be allowed to do anything.

## Although the methods are checking for capabilities relevant to a DinoActor, this class is not combined with the DinoActor class. The reasons are as follows:

## prevent DinoActor from becoming a very large class, making code neater

## there might be a possibility that when the system is extended, new kinds of Actors will be added and they might share the same capabilities as dinosaurs. Then, those Actors can inherit from this class too to share the methods in this class. DinoCapabilities should be renamed appropriately if that happens of course.

## 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 checking methods in CapableActor 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.
* Adheres to cohesion, since this class is solely for all Enums that should be used as capabilities for DinoActors.

## DinoActor Abstract class – extends CapableActor

## Subclasses: Stegosaur, Brachiosaur, Allosaur and Pterodactyl

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 DinoActor that inherits from CapableActor is created for all dinosaurs, namely *S*tegosaur, Brachiosaur, Allosaur and Pterodactyl to inherit from.

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. For example, the subclasses of DinoActor have constructors that have 2 parameters less than DinoActor
* Reusing logic to reduce chances of error. For all methods that exist in DinoActor, only necessary methods will be overridden in the child classes.

Note if there are methods / attributes relevant for a certain type of dinosaur only, those fields will be in the most specific child class instead of in DinoActor. This is to ensure good design, by making sure classes are responsible for their own attributes. For example, Pterodactyl which is the only DinoActor that can fly, keeps track of its own flyingEnergy, while Allosaur which is the only DinoActor that can attack Stegosaurs, keeps track of the victims it has attacked on its own.

## 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. 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 need significantly less fields to store constants
* Prevent excessive hard coding
* Having a standardized set of values, which can be accessed by other classes apart from DinoActor, making the values uniform throughout the project uniform
* Separation of concerns and single point of change, whenever we want to change any of the values, simple look into this class
* Consequently, easier maintenance

## Behaviours and Actions

Significant common functionalities of DinoActors have a dedicated class that implements Behaviour and / or a class that extends Action.

|  |  |
| --- | --- |
| Breeding | BreedingBehaviour and BreedingAction |
| Feeding (on its own, not fed by player) | 1. FeedOnActorBehaviour 2. FeedOnItemBehaviour   and FeedingAction |
| Drinking | DrinkingBehaviour and DrinkingAction |
| Fed by player | FedByPlayerBehaviour and PlayerFeedAction |
| Attacking another DinoActor | AttackBehaviour and AttackAction |
| Pregnancy | PregnancyBehaviour and LayEggAction |
| Following something | 1. FollowBirthingSpotBehaviour 2. FollowFoodBehaviour 3. FollowMateBehaviour 4. FollowVictimBehaviour 5. FollowWaterBehaviour   and DynamicMoveAction |
| Fleeing from other DinoActors | EvadeDinoBehaviour and DynamicMoveAction |
| Wandering about | WanderBehaviour and DynamicMoveAction |

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 or DynamicMoveAction is returned, otherwise it will decrement the pregnant DinoActor’s pregnancy period.

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 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.

## DynamicMovement interface

## When attempting to make the Pterodactyl fly while it still has energy to fly and walk otherwise, we realized this is a good chance to use an interface. Hence, this interface is created for DinoActors that have different types of movement, so that the type of movement (eg: walking or flying) can be decided dynamically.

## DynamicMoveAction – extends MoveActorAction

This class depends on the DynamicMovement interface and uses its methods to simulate the DinoActor using different kinds of movement whenever it still has energy to do so. If the DinoActor only has one kind of movement, the default mode of movement “walking” is used.

The benefit of using this interface is:

* Dependency inversion is achieved. Ease of extending the system, if other DinoActors need different kinds of movement too, they can just implement the DynamicMovement interface implement this interface. Other parts of the code do not need to be altered.

## WanderBehaviour – implements Behaviour

## This class which provided with the engine code was only slightly modified to replace returning MoveActorAction with DynamicMoveAction to allow DinoActor to wander using its proper mode of movement dynamically.

## Following

This section includes the explanation of all DinoActors following mates, food or victims (Allosaur following Stegosaur to attack it when hungry), water and birthing spot to lay egg on. 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: FollowFoodBehaviour, FollowMateBehaviour, FollowVictimBehaviour, FollowWaterBehaviour and FollowBirthingSpotBehaviour**

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

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.

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. For example, FollowWaterBehaviour will override the motivatedToFollow to return true when the Dino

There are a lot of benefits with this implementation. 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. Note that this is proven in Assignment 3, where classes FollowWaterBehaviour and FollowBirthingSpotBehaviour could be easily added, without changing anything in the FollowBehaviour class.
* 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.

## EvadeDinoBehaviour – implements Behaviour

This class allows Pterodactyl to avoid other DinoActors when it lands on a spot with a Corpse on it. Initially, we considered making an additional interface for it, but considering only Pterodactyl requires this functionality and to prevent speculative generality and overengineering.

## BreedingBehaviour – implements Behaviour

## and BreedingAction – extends Action

If two DinoActors are at adjacent locations, it will be checked whether they can breed or not. This is done so by calling getAction on BreedingBehaviour in the getAllowableActions method in DinoActor class. Note that the female DinoActor will have a 50% chance of getting pregnant once the breeding occurs. Due to the usage of Capabilities and getter methods to check whether the DinoActor has the capability of DinoCapabilities.CAN\_BREED, when only need to perform downcasting once (from Actor to DinoActor). This helps us to reduce dependencies successfully, since the classes do not need to depend on Stegosaur, Brachiosaur or Allosaur.

For Assignment 3, to fulfill the requirement of only allowing Pterodactyls to breed on trees,

the method canBreedHere from the Ground interface is utilized. Based on the code from Assignment 2, BreedingBehaviour is easily extended to add an additional condition to check whether both dinoActors are at locations where they can breed for BreedingAction to be returned.

## PregnancyBehaviour – implements Behaviour

## and LayEggAction – extends Action

After breeding, if a female DinoActor becomes pregnant, it will prioritize laying an egg above all. Due to the usage of Capabilities and getter methods to check whether the DinoActor has the capability of DinoCapabilities.PREGNANCY, when only need to perform downcasting once (from Actor to DinoActor). This helps us to reduce dependencies successfully, since the classes do not need to depend on Stegosaur, Brachiosaur or Allosaur. Note that a pregnant DinoActor will not be able to breed.

For Assignment 3, to fulfill the requirement of only allowing Pterodactyls to lay egg on trees,

the method canLayEggHere from the Ground interface is utilized. Based on the code from Assignment 2, PregnancyBehaviour is easily extended to add an additional condition to check whether the pregnant DinoActor is at a location where it can lay an egg. If yes, LayEggAction is returned, otherwise a different Action (Action returned by calling getAction on FollowBirthingSpotBehaviour) is returned for the Pterodactyl to find a tree to lay its egg on.

## Utilizing Actor Interface

## A method, doWhenRaining is added into this interface for all Actors to be able to do something when it rains. This is implemented for the DinoActors to be able to drink water when it rains, while Player does nothing. The purpose of adding it into the interface is to utilize the existing mechanism to prevent downcasting references and introducing additional dependencies into the Jurrasic World class.

## Utilizing Ground Interface

## A method, canBreedHere and canLayEggHere is added into this interface in order to work with the BreedingBehaviour and PregnancyBehaviour to determine which classes that extend Ground are suitable to breed on / lay an egg on.

## The purpose of adding it into the interface is to utilize the existing mechanism to prevent downcasting references and introducing additional dependencies into the BreedingBehaviour and PregnancyBehaviour. If it is not implemented this way, additional checks on what kind of DinoActor and what kind of ground it is on need to be added, making code unnecessarily messy, especially if new types of Ground or Actors are added.

## Food Interface

When attempting to incorporate new functionalities of Assignment 3, where Pterodactyl can be eaten directly by Allosaur, we realized that using an interface to identify all possible food sources would produce better code. This interface is implemented by anything that represents food:

* FoodItem: all items that are considered food, i.e. Fish, Corpse, Egg, Fruit and MealKitsItem
* Tree, Bush and Lake, since fruits can be eaten from Tree And Bush, while Fish can be eaten from Lake
* Pterodactyl, since it can be eaten whole by Allosaur

The benefit of using this interface is:

* Dependency inversion is achieved. Ease of extending the system, when a new kind of Item, Ground or even Actor needs to be considered as Food, simply implement this interface. Code in FeedingAction and FeedingBehaviour do not need to be altered.
* Simplifies code in FeedingAction significantly, since we won’t have a lot of checking to determine what kind of food is edible by the Actor and how much the hitPoints should increase by.

## PortableItem abstract class – extends Item

In assignment 2, it was assumed that this class would only be inherited by Items that were considered food, hence some methods suitable for food items only were placed here. In assignment 3, we have decided that it was best to make a clearer distinction between strictly portable items and food items that are also portable. Hence, this class was reverted to its original implementation as it was provided from the start.

## FoodItem abstract class – extends PortableItem implements Food

All items that are considered food will extend this class, i.e, Egg, Fish, Corpse, Fruit and MealKitsItem. This way, all items that are considered food will be able to share common methods by utilizing inheritance, and we can prevent downcasting to references.

## Egg class – extends PortableItem

The display character of Egg is ‘o’. For assignment 3, the code of this class is slightly modified so that an additional constraint is added, to prevent the Egg from being eaten by a DinoActor of the same species. This is to prevent carnivorous dinosaurs from eating its own Egg as soon as the Egg is laid.

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.

## Fish class – extends PortableItem

## Though Fish can be represented as an integer in lake, a separate class that extends item is still created for it. The reason of this that we need it to be Fish to be an Item for Lake’s Food interface implementation. This class carries only a single responsibility of representing a Fish that can be eaten by carnivorous DinoActors.

## DrinkingGround Interface

This interface should be implemented by all classes that have water that the DinoActors can drink from, which is Lake in assignment 3. By using this interface, we are practicing the dependency inversion principle. In the event that other Ground that contains water, maybe pond or waterfall, needs to be added to the system, the code in DrinkingAction and DrinkingBehaviour can be reused readily without modification. We simple need to ensure the added classes implements this interface.