FIT2099 Assignment 3
Design Rationale

Tutorial 9 Team 9

Prepared by:

Joanne Ang Soo Yin 30513723

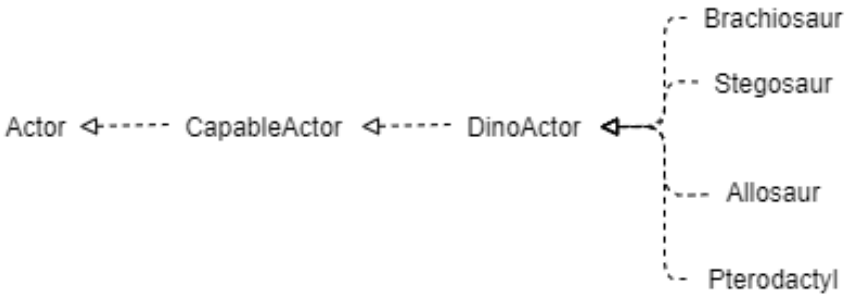
Jonathan Wong Leong Shan 31435297

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 classso 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)	(i) FeedOnActorBehaviour (ii) 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	(i) FollowBirthingSpotBehaviour (ii) FollowFoodBehaviour (iii) FollowMateBehaviour (iv) FollowVictimBehaviour (v) 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 needto know that a DinoActor needs to be a female to be pregnant.
- Reduce clutter in playTurn method in the DinoActor base class, cleaner code meansbetter

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 findthe 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 followis 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 methodand 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 parameterswhile 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.

AttackBehaviour – implements Behaviour and AttackAction – extends Action

If Allosaur is hungry or when a Player who has decided to attack a DinoActor finds another on the location adjacent to it, it will check whether they can attack it or not. This is done so by calling getAction on AttackBehaviour in the getAllowableActions method in DinoActor class. We use capabilities and getter methods to check whether the target (DinoActor to be attacked) has the capabilities, DinoCapabilities.CONSCIOUS and DinoCapabilities.CAN_BE_ATTACKED. This helps to reduce dependencies since the classes

do not depend on Allosaur, Brachiosaur or Stegosaur classes to check these capabilities.

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.

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 aboveall. Due to the usage of Capabilities and getter methods to check whether the DinoActor hasthe 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.

DrinkingBehaviour – implements Behaviour and DrinkingAction – extends Action

These classes are an extension for Assignment 3 which are for the functionality where DinoActors are able to now drink from lakes. In the behaviour class, two methods which are isThirsty() from DinoActor and hasWater() from the Lake class which implements this method from DrinkingGround interface.

This reduces dependencies from the three classes, Allosaur, Brachiosaur and Stegosaur as we are using methods from the related abstract classes and interfaces.

FeedOnItemBehaviour and FeedOnActor– implements Behaviour and FeedingAction – extends Action

The behaviour classes for the feeding package are split into two which are FeedOnItemBehaviour and FeedOnActorBehaviour which obeys the Single Responsibility Principle of the SOLID principle as one is responsible for feeding on items such as eggs, fruits, and corpse (cannot be eaten whole by DinoActor). FeedOnActorBehaviour is responsible for when the DinoActor can feed on a DinoActor corpse whole.

In FeedOnActorBehaviour class, we check if the target is considered a Food object whereas in FeedOnItemBehaviour class, we check if the item in the same location as the DinoActor is of Food type or if the ground of the DinoActor's location is of Food type. Ground of Food type are trees and bushes.

In both behaviour classes, we also check if the DInoActor is hungry and the food is edible by the DinoActor. To check these conditions, we have used methods such as isHungry() from DinoActor abstract class and canEat() from the Food interface. These reduce dependencies as these classes do not need to depend on the subclasses Allosaur, Brachiosaur or Stegosaur.

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 (which extends the World class). This method is called if it rains for every turn in the run method of Jurassic World.

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.

Corpse class – extends FoodItem

The display character of Corpse is '%'. For assignment 3, we have modified the constructor so it now initializes the Corpse with methods implemented in FoodItem class such as setForCarnivores() and setBigSize() (if DinoActor can be eaten whole or not). Final and static values are added to this class for each type of DinoActor which are then used in the dictionary of the Corpse for easy retrieval of the respective values.

The design of this class obeys the Single Responsibility principle, as its only role is to represent a dead DinoActor (Corpse) that will be removed after a certain number of turns depending on the type of DinoActor. A static Map containing the different number of turns is used in this class linked with their respective DinoActor type. This allows easy and convenient modifications to be done as the class carries out only one functionality. This class extends PortableItem which allows the Corpse to be eaten and removed easily without any extra changes.

Egg class – extends FoodItem

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 FoodItem

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.

MealKitsItem – extends FoodItem

Only the constructor is used here and it only used to call the superclass constructor with a parameter list. There will be two different types of meal kits - Carnivore meal and Vegetarian meal kit, which will be differentiated through the usage of the values from the FoodType Enum (see FoodType class above).

Fruit – extends FoodItem

This class have one additional attribute which is the groundTime that is used to keep track of the how long the Fruit has been on the ground. This class is extended from the PortableItem class instead of the Item class. Item can be portable or not portable, directly inheriting from the PortableItem class allows the Fruit to be initialized to be portable. This prevents us from initializing the Item as not portable, therefore, there won't be situations where we need change the portability and this prevents the risk of privacy leaks.

There are two tick methods given to us from the Item class in engine. This class complies to the Liskov Substitution Principle as the input parameter accepted by the method overridden by the subclass is the same as the one from the superclass. This allows the subclass to behave the same way as the parent class and prevents the application from breaking.

Making use of this tick method also reduces dependencies. The time the Fruit spent on the ground is being keep track of in this class instead of related classes such as the Tree class.

The "nature" part of the game:

Dirt – extends Ground

The display character of Dirt is '.'. This class does not have any additional attributes and overrides only one method from Ground which is the tick method.

This class complies to the *Single Responsibility Principle* as its only responsibility in the whole game is to decide whether or not a Bush will grow on each turn. It also takes the *Liskov Substitution Principle* into account as the input parameter values and return values are the same as its parent class's. Thus, when an instance of a base class is expected, the instance of this class can also be used, and the program does not break when this is done.

FertileGround – extends Ground, implements Food

This is an abstract class that inherits all functionalities of Ground and works as the parent class for Bush and Tree. The main reason that this class is necessary is to reduce redundancies. Tree and Bush class have a lot of methods with the same implementations and only differ in the naming. By

making use of this class, we are complying to the rule of *Don't Repeat Yourself*. In some of the methods, for example:

1. isTree method

The role of this method is to check whether the Ground type is of Tree type by using the hasCapability method to see if the location is of TerrainType.Tree. From here, it can be seen that we are making sensible use of the provided code which is the hasCapability method that has been provided by the Ground class in the engine.

Bush and Tree – extend FertileGround

Display character of the Bush is '~' when it is still young and will change into '*' when it is fully grown. Display character of the Tree is '+' at the start. When it is still young, the character will be 't' and changes into 'T' when it is fully grown. Both of the classes have two additional attributes which are the age, that is used to keep track of how old is are they, and an ArrayList that stores Fruit item grown on the Bush/Tree as time ticks. All of these further justifies why we made an abstract parent class for the two classes as they share common attributes and methods. The only difference is their terrain type and their tick method.

Attributes such as the displayChar is inherited from the parent class of FertileGround, which is the Ground that resides in the engine. We are yet again making use of given attributes instead of redefining our own that has the same functionality. The class integrates with the existing system and reduces redundancies at the same time.

Utility

Methods in this class:

1. generateProbability method

As the name suggests, it generates a probability. This is done through getting a random float number and check if it is less the input value. This method is declared as static as there is no need to create a new instance every time this method is needed. This results in a cleaner and less convoluted code. Having a class to specifically check for probability of whether certain code should run is useful in a way such that duplicate code can be reduced as it is needed throughout different classes.

2. getIntegerInput method

This method is to get a user input and validate it to make sure it is an integer before returning the input value. It makes use of the fail fast principle as it checks for error and immediately outputs a suitable message to ask for a new user input if the input value is invalid. Having this method here also adheres to the Don't Repeat Yourself principle as any other classes in the game will be able to reuse the method.

Player actions and related classes:

EcoPoints (currency used in the game to buy items)

All of the methods and attributes are declared as static in this class. Some motivations behind this are:

- All the instances of the class share the same static variable. This allows the ecoPoints used by the player to be updated consistently as changes to the ecoPoints value happen in several different classes at every turn.
- There is no need to create a new instance every time a method from this class is needed as it will be the same for all instances of the class. This makes the code cleaner and readable.

FoodType and TerrainType - enum class

FoodType values: CARNIVORE, HERBIVORE, BIG, SMALL

TerrainType values: BUSH, TREE, VENDING_MACHINE, HAS_FRUITS

As mentioned above, Enum classes are used to make the code cleaner and more maintainable as values are also standardized so they are the same and can be used in different classes. The values above are given to the edible items, dinosaurs and ground as a capability. This allows the food and dinosaurs to be easily matched. When comparison is needed, the hasCapability method can be used to check for the category the item or dinosaur is in which reduces the risk of getting type errors.

BuyAction – extends Action

The purpose of this class is to simulate the action of buying from the vending machine. An important method in this class is the following method:

1. execute method

In this method, a menu of items will be shown to allow the player to choose from to buy and input a number corresponding to the item on the menu shown. Once the choice is made the VendingMachine is called to process the input, update the player's EcoPoints (the currency used in the game), and add the item to the player's inventory.

The menu is shown in a table – like form. System.out.format is used because it allows a formatted string to be returned by given locale, format and arguments. It is a convenient way to output strings in the format we need.

LaserGun – extends WeaponItem

Here, the only method used is the constructor. As the specifications require the player to be able to kill the Stegosaur in one or two hits, the attributes of the LaserGun is initialized to have a damage of 50. The initialization is done through using the keyword super to get a superclass constructor with a matching parameter list and entering the values for each parameter to initialize them.

VendingMachine

This class is to simulate a vending machine that sells a list of items. It is used together with the BuyAction class. Here, we used the fail fast principle when the accepting the input from the player. Once the input is accepted, and the choose method in this class is invoked, it will immediately decide whether the input is expected, if it is an unexpected input, a message will immediately be output to the user and a false is returned.

Same as the previous classes, this method is declared as static to increase readability and make the code cleaner. Furthermore, no initializations are required so it would be redundant to create a new instance of the classes.

Player – extends Actor

1. playTurn method

This method selects and return an action to perform on the current turn. The BuyAction is added into the list of Actions. This way of adding an action is different from the way actions such as AttackAction, PlayerFeedAction and so on is due to the fact that the VendingMachine is not an object that would appear in the GameMap, so there are no such things as allowable actions for the VendingMachine. Thus, it needs to be added directly via the playTurn method in the Player. At the same time, the current EcoPoints is also printed here as we need to know the EcoPoints for every turn and having it placed before the actions menu can allow the Player to decide whether they want to buy an item or continue with other actions.

FedByPlayerBehaviour – implements Behaviour

and PlayerFeedAction - extends Action

The purpose of having the FedByPlayerBehaviour is to make sure that the Actor that feeds the target (DinoActor), is Player.

The PlayerFeedAction class has an additional attribute called target, which is set to final as this value will never change. The purpose of this class is to allow the Player to feed the Dinosaurs. Method used here is:

1. execute method

In terms of feeding, the dinosaurs are placed in two different categories: Stegosaurs and Brachiosaurs are Herbivores and Allosaurs are Carnivores. Therefore, to feed the dinosaurs, we must first determine the type of dinosaurs they are. Once that is determined, the Player needs to go through their inventory to look for the right food to feed them. This is where the FoodType class will come in handy. Rather than checking for which instance they belong such as MealKitsItem or Fruits which are both herbivore food, the hasCapability method is used to check which food type they belong, i.e. FoodType.HERBIVORE or FoodType.CARNIVORE. This is definitely a more concise and clean way to compare the types and is less error prone. Furthemore, the fail fast principle is being make use of as messages are output every time we go into an if or else condition. If there are logic errors anywhere in the code, these messages, while it is output to inform user of the result of their chosen action, it also helps us programmers to determine where the error might be.

Once the correct food is matched with the correct dinosaur, that item will be removed from the inventory and the dinosaurs will have their hitPoints healed (if it is not max). If a fruit is fed, the EcoPoints will increase by 10.

SearchItemAction – extends Action

SearchItemAction class extends the Action class. As mentioned above, the main reason for extending from another class is to reuse the attributes and methods that are similar between the classes.

In this class, we are making full us of the methods that are introduced in the other classes to check for things such as the terrain type instead of using instanceOf. This makes the code cleaner and more readable as it is more descriptive. The effects of Liskov Substitution Principle are shown here: the ground, although it is of Bush/Tree type, does not get any error although it is declared as Ground type. This is due to the classes complying to the rules of using the same input parameter when overriding the methods in the parent class. Thus, allows the application to not break.

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.

Lake – extends Ground, implements Food, Drinking Ground and Rain

Lake is a class that represents the Lake in the game which allows the dinosaurs to drink water from and there are fishes in the Lake that can provide food points. Rain is a class to manipulate the “weather” in the game. This class decides whether rain will fall or not every 10 turns with 20% probability of rain falling.

The Lake class implements Food and DrinkingGround interfaces which adheres to the Open/Closed principle from SOLID design principles. By making use of these interfaces, the class is open for extension but closed for modification. New features can be added easily and the substitutions can be done without changing any code that uses them. When it rains in the game, the water level of the Lake will increase and there is a 60% probability that a new fish will born every turn. The status of the weather is accessed from the getStatus method in Rain class. Fish are represented as integers in this class although there is a Fish class to reduce dependencies. This way, the tick method in Lake will not need to depend on the Fish class, this makes the code more readable and modifications can be done easily. This also avoids the need to depends on Lists or ArrayLists which makes it less error prone as we do not need to manipulate lists to store the fishes.

JurassicWorld – extends World

The purpose of extending from the World is to make use of the run method in the class. It is where the whole game takes place and the only place where the number of turns can be calculated without corrupting the main usage of the class. The total number of turns is also calculated here to decide whether the player met the target eco points or not in certain number of turns. As rain needs to fall globally in the game, the method to decide whether rain falls will need to be called in the loop before each game. There are no other place which is suitable to implement these two functionality.

A more sophisticated game driver

Earlier in Assignment 2, the player has means of terminating the game and they can only play the game in the default Sandbox mode, which is not user friendly. In order to fulfill the requirements of Assignment 3, an option to choose from Challenge or Sandbox mode, or to quit the game is added.

- **Modifications to Application**
    - The driver now runs in a loop and will only stop when the user chooses to close the game entirely.
    - The chooseMode method is added to output the instruction of the game and display the mode options for user to choose. getIntegerInput is used here to validate each user input which adheres to the fail fast principle.

- **Modifications to JurassicWorld**
    - Methods to set the mode, target eco points and allowed moves are added. In the loop in the run method, the total number of turns is calculated (i.e. the number of times the loops ran), if Challenge mode is chosen, the loop will stop (by removing Player off the map) when the target eco points is met within the allowed number of moves or when the allowed number of moves is exceeded before meeting the target eco points. Using the method to remove player from the map to terminate the loop is integrating to the existing game engine.

- **Making use of QuitGameAction**
    - This action is added to allow the Player to quit a game halfway by removing the Player from the map. This will not terminate the program as the loop in Application will only stop when mode 3 (The mode to close the program) is chosen. It will instead stop the current game and give the player the option to play another game. This again intergrates the game engine as the game is quit by removing the player from the map using the methods given in the engine