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

Inheritance

Inheritance is used in the design. When one class extends from another class, it becomes the child class of the class it is extending from. By extending from the parent class, the child class will be able to inherit the attributes and methods from the parent class. As there will be similarities between the child and parent class, these methods and attributes can be reused, thus reducing repetitive code which complies to the *Don't Repeat Yourself* principle.

Classes involved in inheritance:

Child class	**Parent class**
BreedingAction	Action
AttackAction	Action
BuyAction	Action
CheckInventoryAction	Action
FeedingAction	Action
LayEggAction	Action
PlayerFeedAction	Action
SearchItemAction	Action
DinoActor	Actor
FollowFoodOnGroundBehaviour	FollowBehaviour
FollowFoodOnPlantBehaviour	FollowBehaviour
FollowMateBehaviour	FollowBehaviour
FollowVictimBehaviour	FollowBehaviour
Allosaur	DinoActor
Brachiosaur	DinoActor
Stegosaur	DinoActor
Player	Actor
CapableGround	Ground
Bush	CapableGround
Tree	CapableGround
VendingMachine	CapableGround
Dirt	Ground
Corpse	PortableItem
Egg	PortableItem
Fruit	PortableItem
MealKitsItem	PortableItem
LaserGun	WeaponItem

Enum

Enum classes are used in the design. There are a lot of fixed sets of constants used throughout the program which include attributes and methods. By using Enums, attributes are not needed to store the fixed values in certain classes. It is also more type-safe compared to using constants as it gives a more descriptive type name. Methods that are constant can also be accessed easily through the Enum class instead of needing to go through each class to find a method we need. It is cleaner, more readable and more maintainable to use Enum classes.

Enum classes:

- DinoCapabilities
- DinoEncyclopedia
- FoodType
- TerrainType

Symbols

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

Egg	o
Stegosaur	Baby stage – s Adult stage – S
Allosaur	Baby stage – a Adult stage – A
Brachiosaur	Baby stage – b Adult stage – B
Fruit on ground	F
Bush	Small – ~ Big – * Has fruit – ^
Tree	Small – + Medium – t Big – T Has fruit – &
Corpse	%

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 DinoActor that 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

- Declaring methods and attributes in the smallest scope possible, to ensure proper abstraction is achieved.

Note that most of the methods can be shared among the dinosaurs through the base class DinoActor, though if there are methods / attributes relevant for a certain type of dinosaur only, those field will be in the most specific class instead of in DinoActor. This is to ensure good design, by making sure classes are responsible for their own attributes.

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.

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.

playTurn method in DinoActor

The playTurn method in DinoActor will determine the Action to return for each dinosaur in the end. We have implemented it such that the priority of Behaviour is in this order, where the first one is the most important:

1) Pregnancy Behaviour – A DinoActor prioritize laying an egg when it is time to do so above all
2) BreedingBehaviour
3) AttackBehaviour
4) FeedingBehaviour
5) FollowMateBehaviour
6) FollowFoodOnGroundBehaviour
7) FollowFoodOnPlantBehaviour
8) WanderBehaviour

If all of the behaviours return null when getAction is invoked on them, this means that no appropriate Action can be returned, hence DoNothingAction will be returned instead.

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

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.

BreedingBehaviour and BreedingAction

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.

PregnancyBehaviour and LayEggAction

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.

AttackAction and AttackBehaviour

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:

1. DinoCapabilities.CONSCIOUS
2. 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.

FeedingAction and FeedingBehaviour

If a DinoActor finds a food item (Egg, Corpse, Fruit) or a plant (Tree or Bush) on the location adjacent to it, it will check whether they can feed on it or not. This is done so by calling getAction on FeedingBehaviour in the getAllowableActions method in DinoActor class. A carnivorous DinoActor which is the Allosaur can feed on Eggs and Corpses only whereas a herbivorous DinoActor can only eat Fruits which can be found on the ground, in trees and in bushes. Trees can only be searched for fruits by Brachiosaur and bushes can only be searched for fruits by Brachiosaur. Different food items may increase hitpoints of a DinoActor

by varying amounts depending on food items and type of DinoActor. In these two classes, we have used Capabilities to check types of food items and DinoActor as well as getter methods to check whether the DinoActor has the capability of :

1. DinoCapabilities.CARNIVORE
2. DinoCapabilities.HERBIVORE
3. DinoCapabilities.CAN_REACH_TREE (For Brachiosaurs.Stegosaurs do not have this capability)

when the actor is downcasted from Actor type to DinoActor type.

For the item(s), they are downcasted to PortableItem where we can check their capabilities:

1. FoodType.CARNIVORE
2. FoodType.HERBIVORE

For trees and bushes, we downcast them from Ground to CapableGround and check their capabilities which are:

1. TerrainType.HAS_FRUITS
2. TerrainType.BUSH
3. TerrainType.TREE

This helps to reduce dependencies since the classes do not depend on Allosaur, Brachiosaur or Stegosaur classes.

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.
    a) 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.
    b) 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

Corpse class - extends PortableItem

The display character of Corpse is '%'.

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 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.

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.

CapableGround – extends Ground

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 CapableGround

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 CapableGround, 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.

Fruit – extends PortableItem

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.

Probability

This class only has one method:

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.

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
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

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

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.

MealKitsItem

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).

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

There is one important method in this class:

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.

PlayerFeedAction

This 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

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.