Design Rationale / Documentation

Our team focuses on designing a code base that is most scalable (i.e., can implement new actors/functionalities changing as little code as possible), readable (i.e., code can be understood intuitively at first glance), maintainable (changing a part of the code base will cause least amount of damage to other classes) through application of principles and design patterns we have learnt in this unit.

All the implemented classes below only perform one task each to adhere strictly to the Single-responsibility Principle (SRP). This increases readability of the code as it avoids confusion caused when reading the code base.

Classes and objects that are unmodified or trivial will NOT be discussed in this rationale for simplicity. Elaborate explanation for them can be found in the generated JavaDocs.

Before we start, look at these concepts that are shared by the functionalities below that we will not discuss later as it will be repeated many times:

1. Probability wise decisions/actions/statement executions is achieved by having a Random object in the class, generating a pseudorandom number in range [0, 1] and checking if the number is less than or equal to the probability of the event happening. This makes use of the uniformly distributed nature of the generated number. In some cases, a random boolean will be used to simulate a 50% chance of the event happening, the reasoning is same as above.

2. Food level is treated as hit points in this game. In this rationale, we will use the phrase 'food level' and 'hit points' interchangeably.

Second map

The 2nd map is created and added to the GameWorld in a similar fashion to the map given in the base code: converting list of strings to GameMap using FancyGroundFactory, and add it to the GameWorld using world.addGameMap().

The 2 maps can be connected simply by adding MoveActorAction at the connecting edges of the maps. However, this method is inelegant, and not very extensible. What if, in the future, we have an arbitrary number of maps that need to be interconnected?

We added a method connectMaps() to DinosaurMap class that can connect 2 maps by their x-axes or y-axes. Connecting the 2 maps by their x-axes, as specified in the specs, is done by adding Exits to the bottommost row of the new map that have their destination set to the topmost row of the current map, and the other way around. Do the opposite and we can connect the maps by their y-axes. This way, a map can also be connected to itself, so when an actor walks past the West edge then he will end up at the East edge of the same map. This can be done for arbitrary number of maps, as long as their connecting axes are of equal length.

This implementation provides more flexibility in connecting maps. In the case where more maps are added, this method can be called to connect them, in any way the developer wishes.

In the method, we implemented the Fail Fast (FF) principle, where connectMaps() will throw a RuntimeException if the connecting axes of the maps are not of equal length. This is the developer's fault and there is no way to recover from this mistake during runtime (how can the developer swap / alter the map during runtime?), so a RuntimeException is thrown to halt the program immediately. The exception is thrown in connectMaps() as the failure occurred when connecting due to the inequality in axes length. Failing fast and locally here leaves behind an informative stack trace that could be useful during the debugging process, as the stack trace will show the program failed in connectMaps().

The method will also throw IllegalArgumentException if the requested axis to connect by is invalid (when the value of the argument 'by' is not 'x' or 'y'). This also throws a runtime exception, as there is no way to recover from this mistake as the developer has no way to tell the program which axis to connect the maps by during runtime.

This implementation also minimises dependencies (ReD), as the Application class already needs to depend on DinosaurMap to create the map. If MoveActorAction is used to connect the maps, the Application class needs to depend on MoveActorAction too, but in our implementation, creating and connecting the maps both depend on DinosaurMap only.

A more sophisticated game driver

This functionality is implemented in 4 classes: Application, GameWorld, Player, QuitAction class.

To prompt the player for his desired game mode, and to check whether he wants to quit the game from the main menu, we wrote a private method promptGameMode() in Application class that will repeatedly prompt the player for his chosen game mode until the selection is valid. The entire method body of the main() method is wrapped in an infinite loop that will only break when the game mode chosen using promptGameMode() is to quit the game. If the chosen game mode is Challenge Mode, promptGameMode() updates the maxTurns and winningPoints attributes by repeatedly prompting the user for their values, until the input is valid. These values will then be passed to the GameWorld constructor to create the world. Otherwise, if the chosen game mode is Sandbox, the values of these 2 attributes will be set to null, signifying there is no ending condition, and the game will not end unless the player chose to, with the in-game menu.

The GameWorld class extends the World class. The constructor takes in 2 extra parameters, maxTurns & winningPoints. If the world is running in Challenge Mode, these values will be checked in the overridden run() method to check whether the game should be ended due to the player winning/losing the game, within the while(stillRunning()) loop. Depending on how the game ended, a suitable message (Player won/lost/quitted) will be displayed to the UI, and the overridden run() method ends its execution and control will be returned to the Application class main() method.

The above process will be repeated, until the player chose game mode 3, quitting the game, where the infinite loop in the main() method will be broken, and the program will terminate.

We noticed the World needs to have at least a Player in it to run. Therefore, to allow the player to quit the game (i.e. stops the World from running), the QuitAction assists the operation by removing the Player from the World. QuitAction is added to the player's list of actions via the playTurn() method, so the player has the choice to quit the World every turn. Notice that this only quits the World, which ends the current while(true) iteration in the main() method. Another iteration will start and the game will prompt the player for game mode until the player chooses to quit the game completely from the main menu (see above).

Dinosaurs

This includes the explanation for these 5 classes: Dinosaur, Stegosaur, Brachiosaur, Allosaur, Pterodactyl. Understanding of this part of the rationale is necessary to understand the rest of it. This is a mandatory read.

dinoCounter, dinoId

For Allosaurs to track which Stegosaur has been attacked, we tagged each Dinosaur with a unique ID. To implement this, we instantiated a protected static integer field dinoCounter and set its initial value to 0, and another protected final integer field dinoId. Every time we create a Dinosaur, we set its dinoId to the current dinoCounter+1, and increment the dinoCounter by 1. This enables us to track the dinosaurs individually, that can be used to implement other functionalities like Allosaur attacking Stegosaur, and helps avoiding confusion when debugging.

actionFactories

An array containing Behaviours that will be looped through in playTurn() which is called every turn. Behaviours will be called one by one to get their action using Behaviours' getAction() method. If it returns null, that means this Behaviour cannot be performed by this Dinosaur at this turn, so the next Behaviour will be called. This system is a switch to decide which Action should the Dinosaur perform at this turn. As it is an array, the order matters.

age

Age of the dinosaur. If the Dinosaur is created as a baby, the age is set to 0, else if the Dinosaur is created as an adult, the age will be set to the number of turns it will turn into an adult (e.g. adult Stegosaur has initial age 30, as per the specifications). This will later be checked to see whether the Dinosaur is an adult (add capability ADULT), as some Dinosaurs portray different behaviours when their growth stage is different (e.g. Allosaurus' attacks only deal 10 damage when they are baby as opposed to 20 when they are adults).

Private static final fields in Dinosaur's subclasses

Those are constants specific to the Dinosaur subclass and all instances of it share the same values that will not change and will be used to create new instances of these classes or used in checking. They are placed at the very top of the class just below the class name for easy viewing and editing. This helps calibrating the balance of the game as we do not need to go through the code lines by lines to change the values. Instead, we just need to check the values placed at the very top. These fields include the name of the Dinosaur, their display character, maximum hit points, the age at which they will turn adult, their babies' and adults' starting hit points.

Relationship between Dinosaur abstract class and their subclasses

All repeated code or common functionalities are factorised to the Dinosaur abstract parent class to adhere to the Don't Repeat Yourself (DRY) principle which increases the scalability and maintainability of our code as changes only need to be done in the Dinosaur class and nowhere else. This reduces the probability of human error occurring. This approach also reduces dependencies as all dependencies are grouped at the Dinosaur class, and all their subclasses are only dependent on their parent class and the classes that provide them exclusive behaviours. This is an application of the Reduce Dependencies (ReD) which increases the modularity, and flexibility of our program, as the classes can be migrated to another project without having to take much of the other classes with it.

Private constructors and public static factory method in Dinosaur's subclasses

The constructors in the subclasses are kept private and the Dinosaurs are instantiated using the exposed public static factory methods. This is because we take 2 arguments when constructing a Dinosaur. One is a String, indicating the growth stage ("baby"/"adult") of the created Dinosaur, the other one is an optional argument (implemented by method overloading), a character indicating the gender ('M'/'F') of the created Dinosaur. If we use constructors to create the instance, we are forced to call the constructor of the super class (Dinosaur, in this case) as the 1st statement, which is unapplicable in this case as we need to first check whether it is a male or female or an adult or baby before creating the instance. Therefore, we had to resort to public static factory method. Another advantage of the static factory method is that in the future, we can modify it so that it does not have to return an instance every time it is called. The static factories also throw errors when the input String or character is incorrect, and this provide a useful message to the programmer when debugging as the program is failed immediately and locally, so the exception raised will point directly at the faulty code. This is an application of the Fail-fast (FF) principle.

Checker boolean methods

We implemented the following public abstract methods that return boolean values as checkers: isHungry(), isThirsty, isAdult(), canBreed(), isAlive(), isGivingBirth(), isConscious() (overridden), isSafeToEatFrom(). As abstract methods, the implementation is not provided and is forced to be provided by the Dinosaur's subclasses which represent a Dinosaur species each. The implementation supplied by the subclasses defines under what condition the Dinosaur species is hungry, is considered adult, exceeds the food level threshold to breed, is alive, and is delivering a baby. The definitions of these methods will be used in the subclasses to decide when the Dinosaur needs to have a capability added to it which will then be used by other classes to perform some Actions/Behaviours (more discussed later).

playTurn() method in Dinosaur class

This method is called every turn, therefore logics that need to be updated every turn will be added to this overridden method. Below are the logics:

- Checks if has grown up

  The age is incremented. Call isAdult() method to check if the Dinosaur is grown up. If yes, drop the 'Baby ' prefix in the Dinosaur's name, add ADULT capability to the Dinosaur to allow it to mate with others (for Allosaurs, allow it to deal 2x damage to Stegosaurs) and change the display character (displayChar) to upper case.
- Checks if is fertile (able to mate with others as an individual)

  Call isFertile() to check if is able to mate. If yes, prints a descriptive string, then add FERTILE capability that acts as a flag to be used in MatingBehaviour to find potential partners. If no, removes the capability FERTILE. If it is not fertile, MatingBehaviour will return null.
- Checks if is unconscious or is dead

  Hurts the Dinosaur by 1 hit points using hurt() method inherited from Actor class. Call isConscious() to check whether the Dinosaur is unconscious. If yes, prints a descriptive message, increment the number of turns being unconscious, and checks whether the Dinosaur is still alive by calling isAlive() (based on the turns being unconscious). If no, create an instance of DieAction and execute immediately to replace the Dinosaur with its Corpse. Otherwise, if the Dinosaur is unconscious but not dead, return

DoNothingAction for it to not move during the next turn. If the Dinosaur becomes conscious, reset turns being unconscious.

- Checks if is hungry and/or thirsty

  Decrements food and water level by 1 each. Calls isHungry() to check whether the Dinosaur is hungry. If yes, prints a descriptive message. The same is done for isThirsty(). Since location is passed in to playTurn() as an argument, it can be used to print the coordinates of the Dinosaur by calling its x() and y() methods.

- Loop through actionFactories to choose Action to perform for this turn

  Discussed earlier in 'actionFactories' subsection.

Overrides toString() method

The toString() method is overridden so that when printing the Dinosaurs, it will include their ID as well. This allows players to trace their desired Dinosaurs, and the developers to debug their game.

ActorInterface

getHitPoints(), getMaxHitPoints()

0 parameter methods. They are added here because without them being here, we need to downcast to be able to retrieve the Actors' hit points and max hit points or modify the engine code. It makes sense to put it here so every Actor can have these methods as all Actors have hit points. They are given default implementation to return null.

getCorpse()

0 parameter methods. No implementation provided. Returns the Corpse of the Actor with valid stats initialised. Placed here because all Actors should be able to die. This avoids the use of downcasting when retrieving the Corpse of Actors.

Visitor Pattern

Visitor, Visitable interfaces & ConsumableItem class

These 2 interfaces are used to implement Visitor design pattern. Visitor interface is used to declare the visit operations for all the types of Visitable classes. Visitable interface declares the accept operation and is the entry point which enables an object to be visited by the Visitor object.

The Visitor interface defines 4 visit() methods, one for each Dinosaur species. The implementation is kept abstract and the classes that implement this interface will be forced to provide the implementation details.

The Visitable interface only defines 1 accept() method, which takes in a Visitor type argument as input. It calls the visit() method of the Visitor and passes in itself of its instance type (instead of its reference type) to get the implementation details meant for its class. This is a form of implicit type checking. The classes that call the accept() method does not need to be altered as the type checking is done implicitly. This is an implementation of the Open–closed Principle (OCP) of which the classes that call the accept() method of Visitable is closed for modification to prevent the risk of breaking the code logic, and they are open for extension – any addition of classes that implement the Visitor interface can be used in these classes.

The Visitor and Visitable classes are designed to be generic, so any future classes that require this functionality can just implement them while supplying the type of it.

In our implementation, the Dinosaur class implements the Visitable interface while the ConsumableItem class implements the Visitor interface of type integer array. The ConsumableItem subclasses' visit() methods return the food and water healing points for each Dinosaur species in the form of a tuple. This information is obtained and used by calling the accept() method of the Dinosaur subclasses. This way, only the abstract classes is tied to the interfaces via weak inheritance, which reduces the number of dependencies of their subclasses. This is an implementation of the Reduce Dependencies (ReD) principle.

By implementing this design pattern, explicit type checking using instanceof in the previous implementation is replaced with implicit type checking, which is a form of loose coupling that increases the scalability of the system.

SearchableGround

This class extends Ground class, and is a remake of Plant class from the previous rationale. The Plant class is too forced as in it only works for classes that produce Fruits. Our current implementation of SearchableGround has 4 methods:

- getFoodPoints(Actor target, boolean scoutOnly): Returns food points depending on the difference between the current and maximum food level of the target and the amount of Items it is currently holding. It will attempt to return the least food points that is able to heal the Actor to maximum food level. If that is not possible, it will the maximum food points it can afford. The amount of Items stored is then reduced accordingly, if scoutOnly is set to false.
- getWaterPoints(Actor target, boolean scoutOnly): Same as the above, except for water points.
- getItemList(): Return the list of Items the SearchableGround is currently holding.
- removeItems(int amount): Remove amount amount of Items from the list, returns the number of Items it has successfully removed.

GameLocation

We found a bug in our previous implementation where we scan the whole map for Locations with Actors, place them in a map, then find the one with the closest distance to this Actor. This implementation is inefficient (imagine the map having size of 1e10 x 1e10, the space and time complexity is over the roof) and will not find the closest Actor if there are obstacles in between. For example, if there is an Actor 10 blocks away from you, another Actor 2 blocks away from you but blocked behind a 10000000 blocks long wall, our old implementation will show the one behind the wall as the closest target. In this case, the Behaviour will be stuck as it always marks the one behind the wall as target, but it cannot close the distance between them. Therefore, we propose using the Breadth First Search (BFS) algorithm to fix this bug.

This algorithm that will be used by a lot of Behaviours later is implemented within the Location class. This is because a class already needs to depend on Location class if it needs to search for something, so if BFS is implemented within Location class, there will be no added dependencies to all these classes. This is an implementation of Reduce Dependencies (ReD).

After a lot of experiments, we found out the best way to implement this is the Iterator pattern. This way, the classes that uses BFS can simply loop through the starting Location without knowing the implementation logic, which means the BFS implementation can be swapped out at any time without modifying the classes that use it. This reduces the risk of breaking the code, and is a form of implementation of Open–closed Principle (OCP), where the classes that use BFS are closed for modifications but open to extensions (changing from BFS to DFS in some cases maybe?).

GameLocationIterator class that extends Iterator class has constructor that initialises a Queue to store in order the Locations to visit next, and a HashSet visited to store the explored Locations. The overridden hasNext() and next() method is the main logic of BFS, of which we will not dive into details here, this is not an algorithm unit. One thing to note is that the BFS Locations are computed dynamically, so if the loop is broken early, the rest of the Locations will not be explored, and this saves time and space complexity.

GameLocation extends Iterable class to allow itself to be iterated through, and the iterator() method is overridden to return a new GameLocationIterator. Iterates through this object iterates through Location from nearest to farthest from the starting Location.

This is a form of abstraction, where the concrete implementation is hidden from the developer. We will not discuss this again later.

WaterBody interface

This interface is implemented by all Ground subclasses that could store water and allow other Actors to drink from.

This defines 3 methods:

- hasSips(): returns true if there is at least 1 sip for an Actor to drink from
- addSips(int amount): adds amount of sips to the place
- decrementSips(): decrease the amount of sips available by 1, used when an Actor drank from it

Currently, in our game, the only Ground that can store water is Lake, therefore only Lake class implements this interface.

Instead of checking whether a Ground is Lake, we check whether it is a WaterBody, as there can be more classes that implement this interface in the future. This way, we depend more on the abstractions (an interface in this case) rather than concretions, which increases the scalability of the system, as less changes are required when a new WaterBody class is introduced. This is an implementation of the Dependency Inversion Principle (DIP).

FollowBehaviour

The FollowBehaviour provided in the base code was modified to our own needs as it is not located in the engine code nor it was stated that modification to this class is forbidden in the specification.

The modifications done to FollowBehaviour are implementing ComputeDistance interface and instead of following a specific target Actor, we change it into following a specific Location. The reason for implementing ComputeDistance is to allow FollowBehaviour to use the method distanceBetween() so that it can calculate which exits is closer to the desired destination. This also adheres to the Don't Repeat Yourself (DRY) principle as it reduces repeated code where ShootAction class also requires to find distance between target and shooter. Lastly, the reason for changing the FollowBehaviour from following an Actor to follow a Location was because there are other Behaviours that requires something to move closer to a Location but not an Actor. With this modification, the FollowBehaviour is able to assist many other Behaviours in closing in the distance so they are able to proceed with their desired motives (e.g. eating, drinking, recharging skill, etc.).

Dirt, trees and bushes

- At the beginning of the game (and at the beginning of each turn), each square of dirt has a 1% chance to grow a bush.
- On any turn, any square of dirt that is next to at least two squares of bush has a larger (10%) chance to grow a bush.
- In any square of dirt that is next to a tree there is no chance for a bush to grow.

In the ConwayLocation provided as part of the demo package, we are shown how the Ground of each Location can be updated every tick, which is very similar to the requirement of this task. In our implementation, we created a GameLocation class that extends the Location class from the engine package. We factorised the Ground updating code snippet from the overridden tick() method into a method updateLocation() and call it in tick() (update every turn) and DinosaurMap (previously named JurassicMap) to update 1% of Dirt to Bush before the start of the 1st turn, as per the requirement. In addition, the Actors with CRUSH_GROUND capability trampling Grounds with CRUSHABLE capability and return it to Dirt logic is also included as part of the updateLocation() method because it has to be updated per Location per turn. This reduces repeated code as both update Locations before start of the game, and during the game share the method updateLocation(), and is an application of the principle Don't Repeat Yourself (DRY).

- On any turn, any tree has a 50% chance to produce one ripe fruit and a bush 10%.

We created a SearchableGround class that extends Ground class which represents all Grounds that can be searched for Items, so any Ground terrain that grows or produce Items (e.g. Tree, Bush, Lake) that is added in the future can inherit from the SearchableGround class to reduce repeated code (which indirectly increases scalability and maintainability as the system is less error-prone — we do not need to change all classes that share the same logic, change once, change everywhere), which is an application of the DRY (Don't Repeat Yourself) principle. We store the Fruits in it in an itemList. To achieve the requirement of this task, in the tick() method of Tree class that will be called every turn, a random boolean will be generated, and if the value is true (50% chance), a Fruit instance will be added to the itemList, and also increment the eco points of the Player by 1 (will be discussed in details in 'Eco points and purchasing'). Similar logic is used to produce Fruit in Bushes, except we roll a double and check whether it is smaller than 0.1. We shifted away from the implementation last assignment, of which we call SearchableGround the Plant class. The logic is similar.

- On any turn, any ripe fruit in a tree has a small (say, 5%) chance to fall. Dropped fruit will sit on the same square as the tree.

Ground class has a public tick() method that is called by Location class every turn. Therefore, we can incorporate the Fruit dropping logic in this method, together with the provided Tree aging logic. Like the proposed implementation in the previous rationale, for each Fruit in the Tree, they have an individual 5% chance to drop. This is implemented by looping through NumberRange between 0 and number of fruits (represented by size of the itemList), and on each iteration, we generate a random double, and if its value is less than 0.5 (5% chance), a Fruit instance will be dropped on that Location by calling location.addItem(itemList.remove(0)), as the tick() method takes a handy argument location which is the Location the Tree is at. This also makes sure the Item is dropped on the same Location the Tree is at, which satisfies the latter requirement.

- Fruit left on the ground will rot away in 15 turns.

In the tick() method in Fruit class, which will be called every turn, the number of turns it has been on the Ground will be incremented by 1, and once it reaches 15, the argument currentLocation the tick() method takes in, which represents the Location the Fruit is at, if it is on the Ground, will be used to call currentLocation.removeItem(this) and the Fruit will be removed from the game to simulate 'rot away'.

- The player can pick up fruit that is lying on the ground or from a bush.

The Fruit class extends ConsumableItem, with portable attribute set to true. By game rule, any Item that has portable attribute set to true can be picked up and dropped through PickUpItemAction in the engine package, so no implementation needed on our end. Picking up fruits from Bushes will be discussed in the next subsection below.

- The player can try to pick fruit from a tree or bush in the same square. This has a chance of failing (say, 60%) with a message such as "You search the tree or bush for fruit, but you can't find any ripe ones."

An Action class has been created specifically for the Player to perform this action — SearchItemAction. It is designed exclusively for searching Items from SeachableGround. The Action could not be added by overriding the allowableActions() method inherited from the Ground class as this method is only called when the Actor is not in the same Location (i.e. same square) but in the 8 adjacent Locations to the Ground. Therefore, it is added to the Player's actions attribute via playTurn() method by first checking whether the Ground the Player is standing at is a SearchableGround. If yes and the itemList of the SearchableGround is not empty, the SearchItemAction will be added for each Item in the itemList to the actions attribute and the Player get to select via the UI. When executing the SearchItemAction, it checks if the random double generated has value less than 0.4 (simulate 40% probability). If yes, the

Item (Fruit in this case) will be removed from the itemList via groundHere.getItemList.remove(item) and added to the Player's inventory via player.addItemToInventory(item). The only Player in the game also earns 10 eco points via GameWorld.getPlayer().earnEcoPoints(10) (discussed in detailed in 'Eco points and purchasing). A descriptive string will be returned to be printed in the UI to show whether the Player has successfully obtained an Item from the SearchableGround or not. This implementation is better the previous in now all Grounds that hold Items can be searched, if the Player is on the same square, instead of binding the Fruit to the Plant, and have an exclusive SearchFruitAction only to achieve 1 objective. This way, if there are more Grounds that are searchable in the future is added, this system will work for them too, if they extend SearchableGround class.

Lakes, water and rain

Like Trees and Bushes, Lakes extend SearchableGround as well, as they have Fish and they are stored inside of the Ground, not on the Location.

- Each lake covers 1 square on the map. Use the tilde character (~) to represent water/lake and place some pools of water in the map.

A Lake class that extends SearchableGround class and implements WaterBody interface is created, with displayChar set to '~'. It is added by swapping some Dirt's display characters for Lake's in the creation of the map in Application class. They are placed near the centre of the 1st map.

- Each lake starts with a capacity of 25 sips.

Sips is stored as a private integer attribute in Lake class. During creation of Lake, in the constructor, the value of sips is initialised to 25. This implementation saves space, as integer takes up less space than an instance. Having a Sip class is redundant, as it only stores values, and the values stored is same across all instances of it. Lake implements WaterBody, which forces implementation of methods hasSips() that checks whether the Lake is available to drink from, addSips(int amount) that adds sips to the Lake, and decrementSips() that reduce the number of sips of the Lake by 1. This will be used in task 'Thirsty Dinosaurs' and will be explained in detail there.

- Every 10 turns, with a probability of 20%, the sky might rain which adds water to all lakes. The amount of water that is added to the lake is calculated by multiplying the rainfall by 20 sips where the rainfall is a random value between 0.1 and 0.6 exclusive.

This is implemented in the extended GameWorld class. Private integer attribute turns used to implement this task and 'A more sophisticated game driver' is added. It keeps track of how many turns have passed since the start of the game, and is incremented by 1 after each turn, in World.run() method. When the value of turns is a multiple of 10, a uniformly distributed pseudorandom double in range [0, 1] will be rolled and check whether the value is smaller than 0.2, to simulate 20% chance to rain. If it is, sips will be added to the lake, and the number of sips added is computed by rolling a pseudorandom double between [0, 1], dividing it by 2 so its range is between [0.0, 0.5] then offsetting it by +0.1 by adding 0.1 to it, so its range is [0.1, 0.6]. This value will be multiplied by 20 to determine how many

sips is added. When ticking over the maps in the run() method, the Locations of the map will be looped through to check whether the Ground there is a WaterBody, if yes, sips will be added, and the number of sips added will depend on the calculations earlier. When it rains, a message will be printed to the UI, to demonstrate the functionality during the interview.

- Each lake can also hold up to maximum 25 fish, each fish providing 5 food points. Lakes start the game with only 5 fish. Each turn, there is a probability of 60% for a new fish to be born (increasing the number of fish of lake by 1).

The constructor of Lake class initialises the itemList by creating a list, then adding 5 copies of Fish instances to it. In the tick() method of Lake class that will be called every turn, the length of the itemList that is storing the Fish instances will be checked, if it is smaller than 25, and the pseudorandom double in range [0, 1] rolled is less than 0.6, a new Fish instance will be added to the itemList. This ensures there will not be more than 25 fish in a Lake.

Hungry dinosaurs

Brief description:

In each turn, Dinosaurs will decrement 1 hit point to simulate the real world where living things can get hungry, and eventually get unconscious and die if not fed. This is done in the Dinosaur class playTurn() method. Every Dinosaur species (represented as Dinosaur subclasses) have different hunger thresholds, when their food level goes below that threshold, they will be hungry and display HungryBehaviour (otherwise this Behaviour will return null, and the Dinosaur will attempt to execute the next Behaviour in its actionFactories). If they are unable to feed themselves before their hitPoints reaches 0, they will become unconscious (unable to do anything, performs DoNothingAction). After several turns (different for all species) being unconscious, they will die and turn into a Corpse (executes DieAction, with dropCorpse sets to true so a Corpse will be dropped where it died).

HungryBehaviour iterates through Locations from nearest to farthest from the hungry Dinosaur, and checks whether each of them is a valid food source for the hungry Dinosaur or not (either on or in ground food). If the hungry Dinosaur is standing on the valid food source, it will eat it by calling the corresponding Actions. Otherwise, it will attempt to approach the valid food source using FollowBehaviour.

There are 2 types of food sources, on ground food and in ground food. On ground food represents food that are in the Location and is stored within the list of items there. In ground food represents food that are stored in itemList inside of SearchableGround. To check whether there is any valid on ground food for the hungry Dinosaur in a Location, the Items on that Location will be iterated through, and check whether their classes match any of the Item class that the Dinosaur can eat. To check whether there is any in ground food for the hungry Dinosaur in a Location, the getFoodPoints() of SearchableGround is called, and if there is no valid food, it will return null, otherwise return the food healing points available to the Dinosaur.

- A stegosaur should start out with a ‘food level’ of 50 out of a maximum of 100. This should be decreased by 1 on every turn. If the food level gets to zero.

Implemented by setting the Stegosaur class private static integer fields ADULT_STARTING_HP to 50, and MAX_HP to 100. Detailed explanations in ‘Dinosaurs’ section, under ‘Private static final fields in Dinosaur’s subclasses’.

- Food level should decrease by 1 on every turn. If the food level gets to zero, the stegosaur becomes unconscious and cannot move or act unless it is fed. After 20 turns of unconsciousness, the stegosaur dies.

Implemented in Dinosaurs playTurn(). Detailed explanations in 'Dinosaurs' section, under 'playTurn() method in Dinosaur class'.

- A stegosaur that is hungry (i.e. if its food level is below 90) should move towards a food source and eat it.

Stegosaurs utilises the HungryBehaviour it has in its actionFactories to search for food. The HungryBehaviour is only active when the Dinosaur is hungry. If a food source is adjacent to itself, it will go and eat it, otherwise it will move towards a valid food source. Reasoning is stated in this section's brief description of HungryBehaviour.

- Stegosaurs are herbivores and can eat fruits only from bushes or a fruit laying on the ground under a tree. They can't eat from trees because their necks are short and their jaws are too weak to bite through branches. When a stegosaur eats a fruit from a bush or a fruit laying on the ground under a tree, that fruit should disappear and should increase the stegosaur's food level by 10.

In HungryBehaviour, when determining whether the Location is a valid in ground food source, Bush with Fruits in it will be labelled valid because getFoodPoints() will not return null, as Stegosaur has capability SHORT. However, since Stegosaur does not possess the capability TALL, the getFoodPoints() method of Trees will return null, meaning it cannot eat from it, and HungryBehaviour will label that Ground as invalid, so the hungry Dinosaur will not attempt to move towards it. If the Location has at least a Fruit in it, since Fruit is one of the valid foods for Stegosaur, stored within the HashSet initialised in getValidFoods() in Stegosaur class, the Stegosaur will attempt to move towards it using FollowBehaviour, and when the Stegosaur is on it, it will eat it by calling EatItemOnGroundAction, where the Items stored in the Location will be iterated through, and the Stegosaur will accept the Item via the accept() method. This is part of the Visitor pattern. If the food is invalid, it will return null. The loop will continue until the Stegosaur found what it came for, and obtain the food healing points via the accept() method. The Stegosaur will be healed by the healing points and if the Dinosaur does not have capability TINY (exclusive to Pterodactyl as of now), the Item will be removed from that Location.

- If the player is standing next to a stegosaur and holding fruit, they should be able to feed it to the stegosaur. Fruit given directly by the player should increase the stegosaur's food level by 20.

This is achieved through adding FeedAction to the getAllowableActions() of Dinosaur class (so all Dinosaurs can be fed) whenever the Actor adjacent to the Stegosaur is a Player. FeedAction gets the healing points from the Item using Visitor pattern's accept() method. A tuple will be returned, containing in order food and water healing points. It heals both the target food and water healing points and removes the Item from the Player's inventory. This implementation is better than the previous as it is possible that in the future, Items that can heal both food and water healing points at the same time will be added to the game.

- When a stegosaur becomes hungry, a suitable message should be displayed (e.g. Stegosaur at (19, 6) is getting hungry!)

The message is printed under playTurn() method inherited from Dinosaur class after checking if the Dinosaur is hungry. The Location argument passed into the playTurn() method allows us to find the coordinates of the Dinosaur easily by calling map.locationOf(this).x() and map.locationOf(this).y(). Detailed explanation on how our playTurn() implementation works and reasoning and principles applied can be found at 'Dinosaurs' section, under 'playTurn() method in Dinosaur class'.

Thirsty Dinosaurs

Brief overview:

Living beings in the real world will also get thirsty and unconscious, then die if no water is drunk after a period (15 turns according to the specs of this game). So, in each turn, the dinosaur's water level will decrement by 1, by calling dehydrate() in Dinosaur.playTurn() method. Depending on the species of the dinosaur (subclasses of the Dinosaur), they will have different thirsty thresholds, and if below that level, a message will be displayed indicating they are thirsty and their ThirstyBehaviour will be active. The ThirstyBehaviour will urges the dinosaur to move towards a valid water source and restore their thirst by drinking when standing next to the water source. If they were unable to drink any water from a water source before their water level reaches 0, they will go unconscious. After a few turns, they will die and turn into a corpse (by executing DieAction, with dropCorpse set to true to leave a Corpse behind). The GameWorld now has a raining event (occurring once with a probability of 20% every 10 turns) to refill the sips (drinkable water for dinosaurs) in lake and restore dinosaurs' thirst by 10.

- A dinosaur that is thirsty (water level below 40) should move towards a lake and drink water.

This is done using ThirstyBehaviour, which they will approach the nearest (Ground of type WaterBody) containing sips when their water level is below threshold (40 in our implementation). The search is done by iterating through Locations from nearest to farthest from here using BFS and check each of them whether they are a valid water source. Once a valid water source is found, the Dinosaur will approach it using FollowBehaviour (modified to move towards a location) and proceed to drink the sips from next to the water source by calling DrinkAction in ThirstyBehaviour. If no valid water source can be found, the Dinosaur will execute its next available Behaviour.

- A dinosaur should start out with a "water level" of 60 out of a maximum of 100. Brachiosaurs have a maximum capacity of 200. The water level should decrease by 1 on every turn.

Every dinosaur class private static integer fields ADULT_STARTING_WL is set to 60 and MAX_WL is set to 100, except Brachiosaur with MAX_WL of 200. Detail explanation can be found under 'Dinosaurs' section 'Private static final fields in Dinosaur's subclasses' subsection.

- If the water level gets to zero, the dinosaur becomes unconscious and cannot move or act unless it is watered. After 15 turns of unconsciousness, if no rain occurs, the dinosaur dies. Each time the sky rains, unconscious dinosaurs due to thirsty must get a water level equals to 10 and return to normal.

Implemented in Dinosaur's playTurn(). Detailed explanations in 'Dinosaurs' section, under 'playTurn()' method in Dinosaur class. The food and water level of the Dinosaur will be decremented by 1 each turn with methods hurt() and dehydrate(). isConscious() is then called to check whether the Dinosaur is unconscious by checking whether it has hitpoints and water level above 0. If no, prints a descriptive message, increment the number of turns being unconscious by 1, and checks whether the Dinosaur is still alive by calling isAlive() (based on the turns being unconscious). If no, create an instance of DieAction with dropCorpse set to true and execute immediately to replace the Dinosaur with its Corpse. Otherwise, if the Dinosaur is unconscious but not dead, return DoNothingAction for it to not move during the next turn. If the Dinosaur becomes conscious, reset turns being unconscious. If it is raining (More on 'Lake, water and rain' section), the dinosaur's water level will increase by 10 and it will be checked again next turn whether it can regain conscious with isConscious() in the next turn. We decided to not blindly set the water level to 10 if it is unconscious due to thirst, because the Dinosaur might be extremely thirsty (current water level lower than –10) and a mere rain should not be enough to wake it up. However, if needed, it can be replaced with that implementation easily, by setting each Dinosaur actors water level to Math.max(10, getWaterPoints()) when looping through each Location.

- When a dinosaur stands next to a lake it increases the dinosaur's water level by a maximum of 30 water–level points from each lake at a time (per turn). Brachiosaurs can increase their water level by 80 points in one turn.

DrinkAction checks if there are any Location next to the Location where the Dinosaur stands that is a WaterBody and has sips. If there is, a sip will be removed from that WaterBody and the Dinosaur's thirst will be quenched by the defined amount. If the dinosaur has capability TALL, of which Brachiosaurs have, it will restore a water level of 80. Otherwise, the water level of the Dinosaur will increase by 30.

- When a dinosaur becomes thirsty, a suitable message should be displayed (e.g. Stegosaur at (19, 6) is getting thirsty!).

Detailed explanation can be found at 'Dinosaur' section, under 'playTurn() method in Dinosaur class'.

- Land-based creatures cannot enter water, and trees/bushes cannot grow there.

This is done by overriding the method canActorEnter() inherited from the Ground class by returning true only when the actor has capability of FLY otherwise false. With this, no dinosaurs can enter the Lake unless it is currently flying. Trees and Bushes cannot grow on Lakes by default, unless an implementation like 'Dirt, trees and bushes' is done, so nothing needs to be done.

- If you find that your dinosaurs need more water than they can drink from lakes and you don't want to (or don't have time to) experiment with animals' water capacity, you can add a "water bottle" to the vending machine, but you are not required to do this.

A WaterBottle class that extends ConsumableItem is added. Its visit() method for all types of Dinosaur returns waterHealingPoints equal to of the Dinosaur's maximum water points, to heal them fully. More information on how the WaterBottle is sold from the VendingMachine can be found in 'Eco points and purchasing' section. They can be fed to Dinosaurs via FeedAction. Detailed explanation on this can be found under 'Hungry dinosaurs' section.

Brachiosaur

- You will also create a small heard of brachiosaur which are also herbivores (2 males and 2 females).

Implemented by calling the public static factory method in Brachiosaur class with signature getNewDino(String growthStage, char gender) in Application class. Detailed reasoning on why we chose to do it this way is under 'Dinosaurs' section, under 'Private constructors and public static factory method in Dinosaur's subclasses'. This adheres to the Fail-fast (FF) principle, as an invalid argument passed into the method will immediately crash the game and return a useful error message.

- In these game, these long neck dinosaurs can only eat fruit from trees. A brachiosaur can eat as many fruits it finds in a tree in a single turn, but each fruit only increases their food level by 5 since they digest fruits poorly.

Detailed explanation and rationale can be found under 'Brief overview' in 'Hungry dinosaurs'.

- If they step on a bush, there is a 50% chance they will kill the bush.

This is implemented in 'Dirt, trees and bushes' under the 1st subtask. A detailed explanation is included there.

- A brachiosaur should start out with a "food level" of 100 out of a maximum of 160.

Implemented by setting the Brachiosaur class private static integer fields ADULT_STARTING_HP to 100, and MAX_HP to 160. Detailed explanations in 'Dinosaurs' section, under 'Private static final fields in Dinosaur's subclasses'.

- A brachiosaur is hungry if its food level is below 140.

Implemented in Dinosaur class isHungry() abstract method that is overridden in Brachiosaur class to return true when hitPoints < 140 and playTurn(). Detailed explanation on reasoning at 'Dinosaurs' section, under 'Checker boolean methods' and 'playTurn() method in Dinosaur class'.

- Their food level should also decrease by 1 on every turn. If the food level gets to zero, the brachiosaur becomes unconscious and cannot move or act unless it is fed. After 15 turns of unconsciousness, the brachiosaur dies.

This is exactly the same as the implementation reasoning provided in 'Dinosaurs' section, under 'playTurn() method in Dinosaur class'.

- They can be fed by the player. If this happens, each fruit increases the dinosaurs' food level by 20.

This is achieved through adding FeedAction to the getAllowableActions() of Dinosaur class (so all Dinosaurs can be fed) whenever the Actor adjacent to the Brachiosaur is a Player. An instance of FeedAction will be added for each ConsumableItem that the Brachiosaur can eat (checking done through brachiosaur.accept(item), if null means inedible) the Player has. FeedAction heals the target and removes the Item from the Player's inventory.

Pterodactyls

- Pterodactyls are small flying dinosaurs. Pterodactyls can only fly over 30 squares before landing on top of a tree or a corpse. After that, they need to land and walk to a tree before they can fly again.

When Pterodactyl is on a Ground with capability RECHARGE_FLIGHT, its flyingTurns attribute will be set to 30 via setFlyingTurns(). Otherwise, the flyingTurns will decrement by 1 every turn. When it hits 0, the capability FLY of the Pterodactyl will be removed. To recharge its flight skill, it will display RechargeSkillBehaviour, where it will attempt to approach a Ground with RECHARGE_FLIGHT capability, in this case Trees, via FollowBehaviour. Once it reaches there, the FLY capability of the Pterodactyl will be restored. RechargeSkillBehaviour is coded to be dynamic, so if in the future, any other Dinosaurs have similar needs, even though the GroundCapability they are looking for is different, they can still implement this behaviour as well.

- They can traverse water squares.

We overridden the canActorEnter() method of Lake, and make it so only Actors with capability FLY can enter it. This means Pterodactyls can only enter water squares when they are flying.

- They can eat fish which they would catch from the surface of the water by dipping their long beaks in as they fly over. Every time they fly over any lake square they can catch 0, 1 or 2 fish and increase 30 water points.

As discussed earlier, Lakes is only accessible to flying Actors. When Pterodactyls are flying, they can enter the Lake. Lake extends SearchableGround too, so when Pterodactyls are on top of Lakes, and is hungry, HungryBehaviour will lead the Pterodactyls to the Lake with Fish (via HungryBehaviour isInGroundFood()), and the Pterodactyls will feed on it via EatFromSearchableGroundAction, where it will query the Ground with this Pterodactyl to check how much food and water points can be obtained from feeding on this Lake using SearchableGround class getFoodPoints() and getWaterPoints() methods. These 2 methods check how much food and water is left for the Pterodactyl, removes the corresponding Items from the itemList then returns the healing points. The points will then be added to the Pterodactyl. Detailed implementation and rationale can be found at ‘Hungry dinosaurs’ section ‘Brief overview’.

- Add Pterodactyls eggs to your vending machine for a similar price to Stegosaurus. Hatching and feeding Pterodactyls should gain eco points in the same way as Stegosaurus.

Implementation details in 'Eco points and purchasing' section, under VendingMachine.

- Pterodactyls live and breed at the top of the trees there can be only one in one tree at a time). They can breed (similar conditions as the Stegosaur) only at the trees (i.e., two Pterodactyls resting at the top of contiguous trees). When female Pterodactyl is pregnant and about to lay an egg, it must be on top of the tree to lay its egg. It cannot lay an egg on the ground or on the water.

We made it so the Pterodactyl isFertile() and isGivingBirth() only returns true when it is on top of a Ground with capability RECHARGE_FLIGHT, which in this case, a Tree. This made sure they can only mate and lay Egg on Trees. This implementation blends well with the breeding logic we implemented in the previous assignment and no change is required. This is an implementation of the Open-close Principle (OCP) where the MatingBehaviour and MateAction is closed for modifications, but extension can be done on the newly created Pterodactyl class and the behaviours of these 2 classes will change accordingly. Implementation details in 'Breeding' section.

- Pterodactyls would also eat dinosaurs who were already dead. They can land and stay on top of a corpse if there are no other dinosaurs around. Pterodactyl cannot eat a whole corpse in one turn. Those corpses are way too big for Pterodactyl's beak. Pterodactyl can eat a corpse slowly, that is only 10 food points per turn. When the corpse is completely eaten, it will disappear from the map.

Corpses and Eggs are added to the HashMap returned by getValidFoods(), which declares the Pterodactyl can eat Corpses and Eggs that are on the Ground (not to be confused with food inside of SearchableGround). Before starting to eat, inside of EatItemOnGroundAction, it will perform pre-eating action, for Pterodactyls that is landing (stop flying, remove capability FLY). As the visit() method of Corpse class for Pterodactyls only return 10 food points instead of the full food points and decreases the food point of the Corpse by 10, the Pterodactyl will not be able to finish eating the Corpse in one turn. Inside of the Corpse tick() method, it will check whether it still have healingPoints (can be seen as durability of the Corpse) left. If no, same as rotting, it will be removed from the map by calling currentLocation.removeItem(this). A Pterodactyl that has landed when eating the Corpse can be eaten

by Allosaurs (more in ‘Allosaurs’) since it is not flying. In the next turn, since the Pterodactyl’s flyingTurns is not fully depleted, it will start flying again.

- The other Dinosaurs that eat Pterodactyl’s corpse gain 30 food points.

Implementation details in ‘Death’ section. The getCorpse() method of Pterodactyl initialises and return a Corpse with maxAge 40 and healingPoints 30.

Breeding

Brief description:

MatingBehaviour, MateAction and LayEggAction are the core classes for this task. We implemented MatingBehaviour and store them inside an array of Behaviours in each of the Dinosaurs. In MatingBehaviour, Locations nearest to farthest from this Dinosaur will be iterated through and if there is an Actor at the Location, it will check whether this Dinosaur and that Actor are compatible with each other. If they are, and they are next to each other, they will mate by calling MateAction, otherwise this Dinosaur will attempt to approach that Dinosaur. Compatibility between Dinosaurs is determined by the isCompatibleWith() method declared under MatingBehaviour, where it will check whether both Dinosaurs are fertile, are of opposite gender and same species. This checking is done by calling corresponding methods in Dinosaur class. If no compatible partner is found, MatingBehaviour will return null, and the Dinosaur will try to execute the next available Behaviour. MateAction makes the female partner pregnant, by adding PREGNANT capability to it. The pregnant Dinosaur will lay an Egg after a certain number of turns depending on the species. The Eggs laid are left on the Ground for a set length of time (depending on the species) before it hatches into a baby Dinosaur. The hatched baby Dinosaur will then take the 'baby' attributes of its own Dinosaur species and live normally (wander and eat Fruit). After several turns, the baby Dinosaur will grow into an adult and can breed with other adult Dinosaurs of same species. Below we will discuss the implementations in detail:

- If a stegosaur or a brachiosaur is sufficiently well-fed, i.e. has a food level over 50 for stegosaurus or 70 for brachiosaur, it has a chance to breed.

An abstract method canBreed() is defined in Dinosaur class hence all its subclasses have to provide the concrete implementation by supplying the food level threshold of which only Dinosaurs with food level greater than it can breed. It returns true when the Dinosaur's food level is higher than the breeding threshold. This method is included as a criteria that the Dinosaur class method isFertile() will check. isFertile() checks whether the Dinosaur is able to breed with others. In Dinosaur class, we have an array of Behaviours named actionFactories that will be looped through and each Behaviour will be tested if it can return an Action, which means it is a valid Action for the Dinosaur this turn. If yes, the loop will break, and the Dinosaur will perform the returned Action. If no, the loop will check if the next Behaviour can be performed. If all of them failed, the Dinosaur will perform WanderBehaviour. Therefore, order matters.

- A dinosaur that wants to breed will try to move towards another dinosaur of the opposite sex if there is one nearby. Once in an adjacent square, the dinosaurs will mate (only with those of their same species).

This is done with the combination of Dinosaur, MatingBehaviour and MateAction classes. An array that stores Behaviours, actionFactories will be iterated through in the Dinosaur class playTurn() method that is called every turn, the first Behaviour that returns an Action will be the Behaviour that is performed by the Dinosaur at that turn. Inside MatingBehaviour, Location from nearest to farthest from this Dinosaur will be iterated through, and each Location will be checked if it contains an Actor, and that Actor isCompatibleWith() this Dinosaur. If yes, it will check whether they are adjacent to each other. If yes, they will mate by calling MateAction, otherwise it will use FollowBehaviour to approach the partner Dinosaur. If a partner cannot be found, the MatingBehaviour will return null so the next available Behaviour will be executed. MateAction adds the PREGNANT capability to the female of the pair.

- Ten turns (for the stegosaur) and thirty turns later (for the brachiosaur), the female of the pair will lay an egg.

This is achieved through checking within the Dinosaur class (of which all Dinosaur species inherits from) and LayEggAction that handles all Egg laying logic. All Dinosaurs have an integer attribute named turnsBeingPregnant that counts how many turns it has been since it is pregnant. After a set number of turns, which is different for all Dinosaur species, the Egg will be laid. Therefore, an abstract method isGivingBirth() is declared within Dinosaur class to force all Dinosaur subclasses to provide the number of turns before the parent will lay the egg, which returns a boolean value telling whether the turns is up. turnsBeingPregnant is incremented each turn the Dinosaur is pregnant in playTurn() and isGivingBirth() checks if LayEggAction should be called. If the turns is up, LayEggAction will be instantiated by passing in the Egg using Dinosaur's getEgg() method as argument and during execution of LayEggAction the passed Egg will be added at the parent's location with map.locationOf(parent).addItem(egg) to simulate the egg laying action required by the task, remove the PREGNANT capability the parent has, then return a descriptive String showing what has happened. The String is then printed in the menu in playTurn() of Dinosaur.

- Eggs will hatch after a while into a baby dinosaur.

This is implemented in the Egg class' tick() method that is called every turn, the number of turns it has been on the Ground (named age in the code) will be incremented by 1. When it reaches a certain threshold (depending on the species of the Egg), it will hatch by calling the hatch() method, which will add the baby Dinosaur to the Location by calling currentLocation.addActor(babyDinosaur), then remove the Egg from the Location by calling currentLocation.remove(this). The Player earns eco points

depending on which Dinosaur species Egg is hatched. This functionality is not factorised into an Action because only Eggs can be hatched, there is no need to create a separate class if the only class that will call it is the Egg class. This reduces unnecessary dependency, which is an application of Reduce Dependencies (ReD), that will increase the modularity of the code, as when we withdraw the Egg class and integrate it into another project, we do not need to take HatchEggAction (that is not implemented) with us (that might require other classes too).

- Baby dinosaurs are hungry: its starting food level should only be 10.

In our implementation, the constructors of the Dinosaurs subclasses are kept private, and 2 public static factory methods used to instantiate the Dinosaur subclasses' objects are introduced in each of the subclasses of Dinosaur class. This is to perform error checking on the provided arguments, a String growthStage that will dictate whether a baby or adult Dinosaur will be constructed, and a character gender that will dictate whether the Dinosaur created will be a male or female. The latter is optional as we have 2 static factory methods. If the latter is not provided, the gender will be randomised. If any of the provided String or char is invalid, then the method will throw a runtime exception IllegalArgumentException to fail the game fast and locally for easier debugging as the error log will show that the program failed in this constructor where the wrong argument is supplied. This adheres to the Fail Fast (FF) principle. The starting food level (or hit points) and other parameters shared by all Dinosaurs of same species of all Dinosaurs subclasses are included as private static final integer attribute, placed at the top, just under the class name. This is to make the calibration of the game easier, as the parameters of all Dinosaurs can be seen at first glance and the programmer does not need to scroll around to find where the value is stored. The starting food level is one of those private static final attributes and are initialised in these static factory methods by taking the value from the private static final attribute.

- Baby dinosaurs cannot breed.

In Dinosaur class' playTurn() method, if the Dinosaur's age is incremented to adult threshold, the isAdult() method will return true, the displayChar will be updated to uppercase and the 'Baby ' prefix in their name will be dropped. Dinosaurs that are not FERTILE (able to mate as an individual, requirements are the Dinosaur must be conscious, has capability ADULT, has food level higher than breeding requirement and is not pregnant), will not be able to breed, as in the MatingBehaviour's isCompatibleWith() method will check if both Dinosaurs are fertile, of opposite gender and of same species. If not, they will not be able to trigger MateAction, and MatingBehaviour will just return null, and the Dinosaur will perform other Behaviours this turn instead.

- After 30 turns for the stegosaur and 50 turns for the brachiosaur, the baby dinosaur should grow into an adult.

Implemented in Dinosaur class' playTurn() method, the age is incremented by 1 each time the method is called, and playTurn() is called every turn. Once it hits the adult age (different for each species, therefore an abstract method isAdult() is forced to be implemented by the Dinosaur subclasses to supply the age when the Dinosaur will turn adult), the 'Baby ' prefix in their name will be dropped, their displayChar will be updated to uppercase. The isAdult() method that determines whether the Dinosaur is an adult based on its age will now return true. This method is used in all the places where a baby Dinosaur could not perform what an adult can. With this method returns true, the Dinosaur will now behave like an adult thanks to the implementation of Behaviours. Note that we removed the capability ADULT as it is redundant as we already have a public boolean checker method.

Eco points and purchasing

Brief description:

Like the proposed implementation in the last assignment, the amount of eco points the Player has is stored as an attribute in the Player class. This makes the most sense as if we have more than 1 Player, we can have separate 'wallet' for each of them, which lays the foundation for multiplayer support, increasing the scalability of the system. Public setters earnEcoPoints() and spendEcoPoints() with guardian code are added to prevent direct access to the ecoPoints attribute that allows setting of invalid values. Public getter method canAfford() is also added to check if the Player can afford the item by comparing the Item's price and the amount of ecoPoints the Player currently has. These methods are an application of the concept Encapsulation and implementation hiding as the attribute ecoPoints is kept private while having public getters and setters to compare and change the value of the private attribute. This improves maintainability, flexibility and reusability as the user does not need to know what is happening behind the scenes and can treat the methods as black boxes (only precondition and postcondition matters).

- Eco points are gained whenever any of the following happens:
  - A ripe fruit is produced by a tree (1 point).
    - In Tree class, the overridden tick() method, when the number of fruits is incremented, call GameWorld.getPlayer().earnEcoPoints(1) to add 1 eco point to the player.
  - A ripe fruit is harvested from a bush or a tree (10 points).
    - In execute() of SearchFruitAction (discussed in details in 'Dirt, trees and bushes'), after a Fruit has been plucked from a Plant (only Tree/Bush as of now), GameWorld.getPlayer().earnEcoPoints(10) is called to add 10 eco points to the Player.
  - Fruit is fed to a dinosaur (10 points).
    - In execute() of FeedAction, if the food fed to the target Actor is a Fruit, call GameWorld.getPlayer().earnEcoPoints(10) to add 10 eco points to the Player.
  - A stegosaur hatches (100 points).
    - In the Egg class' tick() method that is called every turn, the number of turns it has been on the Ground (named age in the code) will be incremented by 1. When it reaches a certain threshold (depending on the species of the Egg), it will hatch by

calling the hatch() method, which will add the baby Dinosaur to the Location by calling currentLocation.addActor(babyDinosaur), then remove the Egg from the Location by calling currentLocation.remove(this). The Player earns 100 points by calling GameWorld.getPlayer().earnEcoPoints(100). Detailed explanations on hatching logic can be found under 'Breeding'.

- A brachiosaur hatches (1000 points).
  - Like above, except eco points earned is 1000.
- An allosaur hatches (1000 points).
  - Like above.

– You must place a vending machine on the map.

In the Application class, a VendingMachine object is passed into the FancyGroundFactory object groundFactory as an argument. The FancyGroundFactory class is responsible for getting the constructor from the passed VendingMachine object and store it inside of a HashMap which will then later be used to create the VendingMachine via the method newGround() in GameMap class while looping through all Locations. We then add the displayChar of VendingMachine '$' near Player's spawn on the String map in Application class, so a VendingMachine terrain will be created at the Location we added it at.

– This vending should sell:
  - Fruit (30 points)
  - Vegetarian meal kit (100 points)
  - Carnivore meal kit (500 points)
  - Stegosaur eggs (200 points)
  - Brachiosaur eggs (500 points)
  - Allosaur eggs (1000 points)
  - Laser gun (500 points)

The VendingMachine class constructor initialises the HashMap productsOnSale that will store the String names of products on sale as key, and the prices as values, then put each of the Item sold above into it. We also created a private method nameToItem() that takes the name of the Item as input and returns the a new instance of the Item. In the overridden allowableActions() method, the productsOnSale HashMap is iterated through and the String key is translated to the corresponding Item using nameToItem() and the Item and price is used to create BuyActions, so that for each Item sold by the

VendingMachine, there will be a choice in the UI menu. BuyAction extends Action class, when executed, checks whether the Player can afford the Item via player.canAfford(price). If the Player can afford the Item, it will spend the right amount via player.spendEcoPoints(price), and the Item will be added to the Player's inventory via player.addItemToInventory(item). A descriptive string will be printed on the UI to show what happened. The BuyAction is factorised from the Player class into an Action because the Player's turn should end immediately after he has bought an Item and it is also to increase the modularity of the code as there might be more Actors that can buy Items from the VendingMachine in the future, perhaps NPC actors? PterodactylEgg and WaterBottle is sold in the VendingMachine now too, with prices 400 and 100 respectively, as per the requirement of assignment 3.

- Vegetarian and carnivore meal kits are items that the player can feed to a vegetarian or carnivorous dinosaur. The meal kit will fill the target dinosaur up to its maximum food level and then disappear.

We created 3 classes: MealKit that extends ConsumableItem with portable attribute set to true to allow picking up and dropping it, and VegetarianMealKit and CarnivoreMealKit that extends MealKit. Unlike the previous rationale, we now propose using Visitor pattern to achieve this task. There are 4 visit() methods inherited from Visitor interface, one for each Dinosaur species. In VegetarianMealKit class, visit() methods for Stegosaur and Brachiosaur are overridden to return food points equivalent to their maxHitPoints while in CarnivoreMealKit class, visit() methods for Allosaur and Pterodactyl are overridden to return their maxHitPoints. A FeedAction instance is added for each ConsumableItem that the Player has that the Dinosaur can eat (target.accept(item) does not return null) in the Dinosaur's getAllowableActions() method. FeedAction queries the food and water points the target can heal by eating the Item using Visitor pattern accept() method. It then heals the target Dinosaur's food and water points, and remove the Item from the Player's inventory.

- The laser gun is a weapon that does enough damage to kill a stegosaur in one or two hits.

LaserGun class that extends WeaponItem class is created. Capability RANGED_WEAPON is added so that it can be used as a ranged weapon in ShootAction. ShootAction's acquireTarget() returns the Actor closest to the shooting Actor, by iterating through Locations from nearest to farthest from the Player and return the first location that contains an Actor. ShootAction's execute() method has 50% chance to kill the target Actor (by dealing as much damage as the maximum hit points of the target Actor), and 50% to damage the target by 80% of their max hit points. This is to ensure the Action kills the target in 1 or 2 hits. If the target is unconscious (hit points hit 0) after the Action, DieAction will be

immediately instantiated and executed to replace the target Actor with its Corpse. This is to ensure the target Actor will die immediately after the attack instead of dying in the next turn (as it can be fed by the Player with MealKit to regain full hit points) in their playTurn() method.

Allosaurs

Brief description (PreyingBehaviour):

Allosaurs are carnivores, so it will feast on meat instead of Fruits. To implement this, we have created PreyingBehaviour for carnivores (can be added to future carnivorous Dinosaurs) to prey on other Dinosaurs tagged with capability PREY.

This implementation is very scalable as we can easily add PreyingBehaviour to the actionFactories of other carnivorous Dinosaurs. This will make them chase after Dinosaurs tagged with capability PREY whenever they are hungry. They will find the nearest edible prey by iterating through the locations from the nearest to farthest from its location using BFS and when a valid prey is found it will either follow them using FollowBehaviour or attack them using AttackAction or eat them directly if they are tiny (have capability TINY). The hunter will attack the PREY and heal the damage it dealt. However, if the prey has capability TINY and is not flying, the Allosaur is able to eat the prey as a whole and instantly restore its own food level to the maximum.

Once it has attacked the prey, the carnivore dinosaur should not be able to attack the same dinosaur for 20 turns, and we implement it by having the PreyingBehaviour implement Huntable interface which forces the implementation of getBlacklist() and addToBlacklist(). These methods allow PreyingBehaviour to access and modify the blacklist that maps the ID of its prey to the number of turns since it has last attacked it. Whenever the hunter attacks a prey, the prey's ID will be recorded as key with an integer starting 1 as value into the blacklist, representing the number of turns since it has last attacked it, and incrementing that integer every turn. Any prey in the blacklist is immune to the hunter. When the number of turns reaches 20, the prey will be removed from the blacklist and the hunter will be able to attack it again.

If the prey's hitpoints exceeds or reached 0 after an attack from a carnivore dinosaur, it will turn into a Corpse and the carnivore dinosaur will view it as food source. HungryBehaviour, as discussed earlier, will lead the hunter to eat the Corpse if it is still hungry.

- Like stegosaurs, allosaurs must be able to feed, breed, and grow.

Detailed explanation about Dinosaur's Behaviours can be found at 'Dinosaurs' section, under 'actionFactories') and aging logic in playTurn() method inherited from Dinosaur class (detailed explanation can be found at 'Dinosaurs' section, under 'playTurn() method in Dinosaur class").

- If they go near a stegosaur, they will attack it reducing by 20 the food level of the stegosaurus (the allosaurus increases their food level by 20).

Allosaur class getIntrinsicWeapon() overridden method heals itself by 10 then return 10 as damage if the Allosaur is not adult, otherwise, it heals itself by 20 and return 20 as damage. AttackAction is modified to incorporate this change. If the attack misses, it will not get the weapon, so it will not call getIntrinsicWeapon(), therefore the Allosaur will not heal itself if it misses its attack.

- If the stegosaurus doesn't die in the attack, the allosaurus cannot attack the same stegosaurus in the next 20 turns.

Detailed explanation in brief description of PreyingBehaviour in this section. Each Dinosaur is tagged with a unique ID, and the Allosaur keeps track of which Stegosaurs that it cannot attack by storing the information as a HashMap in PreyingBehaviour's flagCandidate. The key is the ID of the Stegosaur it is banned from attacking, and the value is the number of turns since it last attacked it. PreyingBehaviour will be called on every turn, so the HashMap's value will be looped through and incremented by 1 every turn. When it reaches 20, it will be removed. In PreyingBehaviour, the Allosaur will only look for preys that IDs are not stored in the HashMap.

- If it dies, it can keep feeding from the corpse. If they go near a dead stegosaur or brachiosaur, they will move toward it and eat it. They can eat eggs for an increase of 10 (food level). They can eat dead Allosaurs or Stegosaurus for an increase of 50 in their food level. They fill their maximum food level (100) if they feed from a brachiosaur corpse.

Allosaurs prefer living prey. If PreyingBehaviour fails to find any Dinosaurs tagged with capability PREY. It will try to find other food sources (Egg, Corpses, defined under the HashMap returned by getValidFoods()) via HungryBehaviour. Detailed explanation can be found in 'Hungry dinosaurs' section, under 'Brief overview'.

- Allosaurus cannot attack living brachiosaur.

Detailed explanation in brief description of PreyingBehaviour in this section. Brachiosaur does not have capability PREY. Therefore, PreyingBehaviour will not view Brachiosaurs as valid preys.

- Allosaurs do not appear on the map at the start of the game. Their eggs can be purchased from the vending machine for 1000 points each.

As this is a requirement of assignment 2, for assignment 3, we have added Allosaurs to the game to demonstrate their ability to eat Pterodactyls as a whole.The only way to get it into the game is by earning eco points as a Player, purchase the Egg from the VendingMachine (explained in 'Eco points and purchasing'), drop it on the Ground and wait for it to hatch into a baby Allosaur.

- Adult Allosaurs have a maximum food level of 100.

Implemented by setting the Allosaur class private static integer field MAX_HP to 100. Detailed explanations in 'Dinosaurs' section, under 'Private static final fields in Dinosaur's subclasses'.

- Allosaurs can breed if their food levels are above 50. Allosaurs can lay eggs, 20 turns after mating.

Implemented by overriding the canBreed() and isGivingBirth() methods to return true if hitPoints > 50 and turnsBeingPregnant > 20 respectively. Detailed explanations in 'Dinosaurs' section, under 'Checker boolean methods'.

- Eggs will hatch after 50 turns, into a baby allosaurus.

Allosaur's overridden getEgg() method returns an Egg object with a new baby Allosaur in it by creating a new Egg instance using Egg's constructor. In our previous implementation, we have 3 types of Egg subclasses. However, we found out they have minor differences and do not need a class each as it is redundant. Therefore, we refactored all 3 types of Eggs into the Egg class, and make it a concrete class. The Egg constructors utilise polymorphism to create Egg for a specific Dinosaur type. This way, we can see the stats of all types of Dinosaur Eggs all in one place as well. The Egg object is used in playTurn() method to call LayEggAction to lay this Egg when 50 turns is up. LayEggAction adds the Egg at the parent Dinosaur's current location, then removes the parent's PREGNANT capability.

- Baby allosaurus cannot breed but they can attack stegosaurus (with only an increase of 10 food points while they are babies).

As discussed earlier, MatingBehaviour will return null if the Dinosaur's isAdult() returns false. Therefore, baby Allosaur that attempt to breed using MatingBehaviour will fail. Detailed explanations in 'Dinosaur' section, under 'Checker boolean methods'. Allosaur class getIntrinsicWeapon() overridden method

heals itself by 10 then return 10 as damage if the Allosaur is not adult, otherwise, it heals itself by 20 and return 20 as damage. AttackAction is modified to incorporate this change. If the attack misses, it will not get the weapon, so it will not call getIntrinsicWeapon(), therefore the Allosaur will not heal itself if it misses its attack.

- Baby allosaurus' starting food should only be 20. After 50 turns, the baby allosaurus should grow into an adult.

Implemented by setting the Allosaur class private static integer fields BABY_STARTING_HP to 20 and ADULT_AGE to 50. Detailed explanations in 'Dinosaurs' section, under 'Private static final fields in Dinosaur's subclasses'.

Death

- Make sure that when a dinosaur dies, the corpse remains in the game for a set period of time (e.g. 20 turns for an allosaurus or stegosaurus and 40 turns for a brachiosaur).

We followed the previous rationale, we added a class Corpse that extends ConsumableItem with its portable attribute set to true, so the Player can pick up and drop, as it can be fed to the Dinosaurs by the Player. The constructor takes 3 arguments: a String name (name of the dead Actor), an integer maxAge (the maximum number of turns it can stay on the Ground before it rots away) and an integer healingPoints (the amount of hit points a carnivorous Actor can heal should he consume this Corpse). Actor class' getCorpse method is forced to be overridden by its subclasses, it returns a Corpse with the correct stats initialised, with the Corpse constructor. This is used to complete this task — Stegosaur's getCorpse initialises a Corpse with maxAge set to 20, the same logic applies to Brachiosaur (40 turns) and Allosaur (20 turns). Since Corpse implements ConsumableItem, it inherits the 4 visit() methods from the Visitor class as well, one for each Dinosaur species. The visit() method for Allosaur will simply return the healingPoints it has while for Pterodactyl, it will only return 10, and reduce the healingPoints of itself by 10. In the overridden tick() method, it will check whether its age has exceeded the maxAge or if the healingPoints have depleted beyond 0. If any of the two is true, it will remove itself from the map. Applying the Visitor pattern in this case allows the implementation to be more flexible, as there can be lines of code exclusive to one type of Visitable.