FIT2099 Assignment 3: Extended Edition Design Rationale

Team: **Tute03Team100**

Team members:

Student Name	**Student ID**
Tan Ke Xin	30149258
Marcus Lim Tau Whang	30734819

Introduction

In Assignment 3, our team has discussed and made changes according to the feedback given by the teaching team. With that, our application now is an improvised and extended version of the previous system.

Object-Oriented Design Principles

Dinosaurs - the main stars of the assignment/game. We know that the dinosaurs have some additional features that a Player Actor does not have. For example, pregnancy status, maturity status (baby/adult), and breeding ability. Therefore, we've created a Dinosaur **abstract class** to function as a base for subclasses of each dinosaur type , i.e. classes Stegosaur, Brachiosaur, Allosaur, and Pterodactyl. With this Dinosaur class, we could add additional features/attributes only particular to the dinosaurs instead of all Actor instances, while still maintaining the shared attributes of all Actor instances (e.g.: hitPoints, displayChar) among them. Also, the purpose of using an abstract class instead of an actual class is to prevent the direct instantiation of a Dinosaur instance.

By having this abstract class, we've successfully achieved the **'Reduce dependencies' (ReD)** design principle and also the **Dependency Inversion Principle** (of **SOLID Principle**). We now will have more flexibility in switching between / updating functionalities. Indirectly, **'Polymorphism'** is achieved as well, since we are now able to pass different data types to classes. This also promotes **code reusability**.

In addition to the above, we know that inheriting an Interface, which is a protocol/contract such that classes that implement this interface must also implement all methods in the interface, would serve the same purpose as extending a base class. However, we've decided that an **abstract class would be better than an interface in this scenario**. This is because we could create attributes of the dinosaur without initializing a value to them yet, and also immediately implement some concrete methods, e.g. getters and setters for each attribute in the base class (Dinosaur class). If we use an interface, all classes that implement that interface would have to override all of its methods, making some methods eg the getter and setter methods appear in all three classes, which greatly defies the **DRY principle** (more explanation below) and make our code messy.

Besides that, by inheriting classes (parent-child relationship), e.g. : the four dinosaur classes inheriting Dinosaur and Actor class, Tree and Bush class inheriting Ground class etc, we can greatly reduce repetitive code for 1) methods that have similar functionality, 2) objects/instances that have similar attributes. Methods/attributes that are shared among parent(super) classes and children(sub) classes need only be implemented or declared once in the parent class. If we want to modify some functionality of the methods for the child class, then we could override the methods by changing the method signature and body; similarly if we want to add extra attributes in the child class, we could just add them in, without modifying the parent class. With this, we've just achieved the **'Don't Repeat Yourself' (DRY)** design principle. This also promotes **code readability** and makes updating or debugging our code easier, since any updates would only need to be done in one particular class, instead of changing it in every class that has that method.

Moreover, we've created a few actions classes, specifically to handle one action per class. For example, EatAction handles the process of eating, BreedAction the process of mating, SearchFruitAction the process of searching and picking fruit. With this, we've **increased cohesion** (**GRASP principle**) of our program, since the code in each of these classes is united and focuses on performing only one common task. This makes our program easier to maintain, extend and test because we could easily locate certain functionality due to the separation of concerns. Classes like these also allow us to achieve the S in SOLID Principle, which is the **Single Responsibility Principle.**

Last but not least, we've also created DinosaurGameMap, DinosaurLocation and DinosaurWorld classes, which extends GameMap, Location and World classes respectively. This allows us to make extensions to the game map, location and world, without modifying the source code. Therefore, we followed the **Open-Closed Principle** of the SOLID Principle. The usage of the Dinosaur abstract class as mentioned above also illustrates the following of this Open-Closed principle.

**note: sections/words that are highlighted in yellow represent important explanations of the parts of what we've updated from the previous assignments, as we're doing Assignment 3.*

Game Design:

Quick view of all newly added objects and their display character:

Map Objects	Stegosaur	Brachiosaur	Allosaur	Pterodactyl	Vending Machine	Bush
Display character	d	b	a	p	$	v

Portable Items	Fruit	Stegosaur corpse	Brachiosaur corpse	Allosaur corpse	Pterodactyl corpse	Egg	Meal Kit	Laser Gun*	Fish
Display character	f	)	(	%	/	e	m	!	h

*Previously we defined laser gun's display character as '~', which now contradicts with the newly added, specified lake instance's display character. Therefore, we updated the laser gun's display character from '~' to **'!'**

Application driver class (*game package)*

In order to fulfill the new requirements: a second map & a more sophisticated game driver, we made changes to the Application class. Now, the player has the ability to quit, or choose a game mode (either Challenge or Sandbox mode) to play. This is done by using a switch-case statement, in a do-while block. We also added a new map that has the same size as the existing map in this class, which consists of Ground instances (dirt, trees and lakes).
In terms of code, we've created a **runGame** method. This method takes in an integer as input parameter to determine the game mode (1 for Challenge mode, 2 for Sandbox mode). Based on the game mode, the method would display suitable messages, and ask and validate user input (if any). Then, it will create the two new maps, add world objects (Player, Actor and Ground instances), and call the run method from DinosaurWorld class to run the game.

Player class (*game package)*

We've added some extra actions the Player can perform, in the playTurn method. Mainly, the Player can now choose to move to the 2nd map, when he is at the North edge of the 1st map (i.e. y coordinate=0); or move to the 1st map, when he is at the South edge of the 2nd map (i.e. y coordinate = the map height). In the playTurn method, we'll also now increment the player's number of moves, which represents the n-th move of the player, to be used to check if the player has reached the maximum number of moves allowed (for Challenge mode).

Dinosaurs *(dinosaurs package)*

Status Enum Class

An alive dinosaur could either be a baby or an adult, thus enumeration is used here. An Enum class named Status is created with values: ***BABY, ADULT.*** With this, we can easily know the maturity status of a dinosaur by calling **x.hasCapability(BABY)**/ **x.hasCapability(ADULT)** (where x is a dinosaur) and see whichever returns true.

Upon assignment 3, we've also added ***STEGOSAUR, BRACHIOSAUR, ALLOSAUR, PTERODACTYL*** enum values, and each of these values is to be added as a capability to the appropriate Dinosaur instance, for example Stegosaur would have a Status.STEGOSAUR capability. This is so that we can easily differentiate and know the type of each Dinosaur instance just by using hasCapability, and not using 'instance of'. To add on, previously in our code we've used the display character of each object (Actor/Item/Ground) to determine what each object is, eg if getDisplayChar() = 'd', we know that it's a Stegosaur. We realised that using these literals/'magical strings' way of checking is very inefficient, and should we have to change the display character of an object, we have to check each and every class and make the necessary changes, since no IDEs could refactor such a change. Therefore, by using such Enum capabilities, refactoring and further development would be much easier if there are more items added to the game, as we don't need to purposely look back at what each display character represents.

Lastly, ***ON_LAND*** and ***ON_SKY*** enums are added, and they are used for Pterodactyls, to indicate whether the Pterodactyl is on the land/ground, or flying in the sky.

Dinosaur (abstract) Class

In this Dinosaur abstract class, we have a count each for unconsciousness, pregnancy, and baby, to keep track of the number of turns of that status of this dinosaur. Each dinosaur would also have a gender represented by a M/F, an array list of Behaviour, and an indicator whether it is pregnant (true for pregnant; false otherwise). When creating a new dinosaur, it would have Wander behaviour and SearchNearestFood behaviour; its gender is randomly selected, it won't be pregnant, and its unconscious and pregnant count are set to 0.

Besides that, we've also created a method called **eachTurnUpdates**, which takes in an integer babyCount as input parameter (its usage shown below). As the name suggests, this method mainly handles these updates on each turn:

- If dinosaur is conscious: deduct a hit point (food level) and water level
    - Else: increment its unconscious count
- If dinosaur is pregnant: increment its pregnant count
- If dinosaur is currently a baby AND its baby count is greater than/equal to babyCount: transform it into an adult (using the Enum Status.ADULT) and reset its baby count to 0
    - Else: increment its baby count

So as we can see, the input babyCount represents the maximum number of turns the dinosaur can stay as a baby, and since each dinosaur has different babyCount, we need to take it as input to make suitable changes on each dinosaur.

Lastly, this class also contains all **concrete getter and setter methods** to all attributes, so that classes that inherit this class wouldn't need to implement them once more.

Stegosaur, Brachiosaur, Allosaur, Pterodactyl Class

- **Constructor**

Each of these classes's constructor takes in an Enum as input parameter, to determine if the dinosaur instance being instantiated is an adult or a baby. All relevant attribute values (e.g. hit points) are initially set as the adult's initial values. Only after that would we check if the input parameter indicates that it's a baby, we overwrite the dinosaur's initial hit points, and also set the baby count to 1. The only exception is Allosaur class. Since Allosaurs only hatch from eggs, and can't be instantiated early in the game, therefore all its values are automatically set to the baby's initial values.
Lastly, in each of the dinosaur classes' constructor, we also added the respective dinosaur type Enum as a capability for that dinosaur instance. For instance, the Brachiosaur class would have a line: 'addCapability(Status.BRACHIOSAUR)' .

- Static ***dinosaur*****Count** variable

Each of these classes would have a static variable, called ***dinosaur*****Count**, where dinosaur is replaced by stegosaur/brachiosaur/allosaur/pterodactyl. This static variable is mainly used to keep track of the number of each type of dinosaur instantiated, and used to be part of the name of the dinosaur, so that each dinosaur would have a unique name,eg Stegosaur1, Allosaur3. This is also the reason why it has to be static, so that each dinosaur instance of that class would share the same amount of dinosaurCount.

- **getAllowableActions**

In each of these classes, we've overridden the getAllowableActions method, to add suitable allowable actions for each dinosaur. All dinosaurs would be able to be fed by

Player, therefore they would have a FeedAction. Only Stegosaurs can be attacked, (by Allosaurs or Player), hence Stegosaur class would also have an AttackAction allowed.

- **playTurn**

Methods such as eat(), breed() mentioned in Assignment 1 are all replaced by Action classes of their own, and are returned/handled in this playTurn method.

This playTurn method will be called each turn, and it is used to, in each turn, handle all updates of the dinosaur (this is when **eachTurnUpdates** method in Dinosaur class is called), find all possible actions, and determine which action to be performed by the dinosaur.

The **priority of actions/behaviours** are as follows (most prioritised to least):

Lay egg → Breed → Attack (only for Allosaurs) → Drink → Eat (prioritise food that can fill up most hit points) → Follow (follow another nearby, same specie dinosaur to prepare for breeding) → Search for nearest lake → Search for nearest food source → Wandering around → Do nothing

Each action/behaviour has its own class that inherits the Action class/implements the Behaviour class.

As we can see from the priority of actions/behaviours above, we can see that **drinking water is prioritised over eating**. This is because the amount of water level increased when the thirsty dinosaur finds a nearby lake and drinks from it, is quite high (as compared to hit points increased when eating). To be more specific, Stegosaur, Allosaur and Pterodactyl will increase their water level by 30 while Brachiosaur will increase by 80. Having said that, if the dinosaur is thirsty and hungry, once it finds a nearby lake and drinks water from it, it can then happily move around the map to find a suitable food source, and slowly eat the food which gradually increases the food points, with no need of worrying for water all the time.

Also, this playTurn method would check the dinosaur's food and water level and act accordingly should the dinosaur be unconscious. Dinosaurs that are unconscious can only do nothing and stay in its location. If a dinosaur is unconscious due to hunger, it has to wait to be fed by a Player; if it's due to thirst, it has to wait for a rain to occur.

Lastly if it remains unconscious for a specific number of turns, due to thirst or hunger, it will die (DieAction is returned).

- **Extra explanation:**

- **Brachiosaur Class:**

Brachiosaurs have a 50% chance to step on a bush and kill it. Hence, in the playTurn, if a Brachiosaur steps on a bush and probability is met, the bush location will be set to a Dirt instance, using **setGround** method from Location class.

- **Allosaur Class:**

For Allosaurs that can attack Stegosaurs, we've overridden the Allosaur class' **getIntrinsicWeapon** method, to return a new IntrinsicWeapon, which deals 20 damage for an adult Allosaur, and 10 for a baby.

Besides that, in the Allosaur class we can see a **ConcurrentHashMap** of <String, Integer> pair. Compared to a normal HashMap, a ConcurrentHashMap prevents the system from throwing a ConcurrentModificationException when we make changes to the hash map during the game. But more importantly, this hash map is used to store the name of the Stegosaur attacked by this current Allosaur, and the number of turns of cooldown, since we know that an Allosaur can't attack the same Stegosaur in the next 20 turns after an attack. We can tell that this update has to be made on each turn, thus the update would appear in Allosaur's playTurn method. It will check, for each Stegosaur in the hash map, if its cooldown turn =20, meaning now is the 20th turn, and if it is, remove the Stegosaur's name from it; else, just increase the cooldown count.

- **Pterodactyl Class:**

A Pterodactyl can only fly for 30 turns, which then it has to find a tree to rest on and recharge. In our code, we implemented in a way that if 30 turns are reached but its current square location is a lake, it has no choice but to keep on flying until it reaches land, in which after that, we remove the Status.ON_SKY capability and replace it with Status.ON_LAND, to indicate that this Pterodactyl is on land (and Allosaurs can eat it). Then, if it finds a tree, it will rest on it. Upon resting its flyCount is reset back to 0, it now has the ON_SKY capability and not ON_LAND, and it can continue flying in the next turn, since resting takes a turn.

Other requirements, such as Pterodactyls only allowed to mate on trees, can eat corpses on the ground if no dinosaurs are around, and so on, are also implemented in this class.

Bush, Tree, Fruit, Lake *(ground package)*

We added Capabilities(which is an Enumeration) to these classes. Since these capabilities are known and not going to change, an Enum class named **Status** was created to indicate the Capabilities/Status of all Ground instances. The Status can be divided into:

- DEAD : unable to grow fruits
- ALIVE : able to grow fruits
- ON_TREE : fruits are on tree
- ON_GROUND: fruits dropped on the ground
- BUSH: a bush
- TREE: a tree
- LAKE: a lake
- DIRT: a dirt

(Each respective instance would have a capability of itself, e.g. a Lake instance would have a Status.LAKE capability, a Bush a Status.BUSH capability etc. This is used to differentiate the Ground objects, by using the method hasCapability)

The proposed methods mentioned in Assignment 1 (e.g.:hasFruitOnGround(), removeFruit() and hasFruit() methods) are removed. To illustrate, **Dirt** class has a Capability of DEAD because of its inability to grow fruits, **Tree,Bush** classes a Capability of ALIVE because of its ability to grow fruits. While ON_TREE capability is added if the fruits are grown on a tree, otherwise ON_GROUND status is added to indicate fruits dropped from tree to the ground, or basically any fruit that is on the ground.

In the given assignment specification, the probability of growing a bush from dirt is 1%. Our team revised it to **0.5%.** This is because 1% is inappropriate since bushes will be growing at a very fast pace, leading to the possibility of having an imbalance game to increase.

Since some dinosaurs have limitations and are only able to eat from ground, we can simply check if the ripe fallen fruit is on ground with **x.hasCapability(ON_GROUND)** (where **x** is a Fruit instance), true will be returned if it is on the ground, otherwise false. This also shows that the proposed hasFruitOnGround() method is not needed at all, reusing the available code base will tackle the problem nicely.

Besides, to check if the tree has any ripe fruits and is ready to be eaten by a dinosaur, we can check the last item in the list of items on the current ground. Consequently, we check if the item has the Capability of being ON_TREE (e.g.: **x.hasCapability(ON_TREE)** where x is a Fruit instance). If the probability of ripe fruit falling from tree to ground is met, we can achieve this by: **x.removeCapability(ON_TREE)** (where x is a Fruit instance) and subsequently update it to on ground by: **x.addCapability(ON_GROUND)** (where x is a Fruit instance). Again, here shows that we reuse the available code in the engine and the proposed hasFruit() method is not required.

Once a fruit is eaten by a dinosaur, the EatAction will handle all the required processing. Therefore, removeFruit() method that was suggested in Assignment 1, is being replaced with the **EatAction.**

To handle possible actions that might happen in class Bush, similar concepts are applied. Since fruits grown from bushes will only be on ground, **x.hasCapability(ON_GROUND)** (where x is a Fruit instance) will be used to validate.

Previously, our suggested solution is to use static variables named bushAlive or bushDead to check if the bush is stepped and killed by Brachiosaur. Now, our improved solution is to handle this situation directly in the **Brachiosaur's playTurn() method**.

Lake class

To ensure that there are enough lakes on the game map, a number of lakes has been added to the game map. Having said that, each horizontally 10 square away and vertically 5 square apart, there will be a lake. Since land-based creatures are **not allowed** to enter the lake, we've overridden the **canActorEnter** method and provided our implementation in it. To handle the situation where the sky rained and added water to the lake, we used an if condition. Moreover, each lake can hold a maximum of 25 fish and there is also a probability of 60% that a new fish is born. If probability is met, a new Fish instance will be created and added to the current lake.

Portable Items Classes *(portableItems package)*

ItemType Enum Class

As the name suggests, this enum class has enum values of each Item type, ie EGG, FRUIT, CORPSE, MEALKIT, FISH, and is to be used as a capability for the relevant item. For example, each Egg instance would have EGG capability, each Corpse instance would have CORPSE capability, in addition to the CorpseType Enum capability (explained below) etc. Again, such enum values are to be used to check what type of Item instance an Item is, by using hasCapability.

Corpse Class

When a dinosaur remains unconscious for a certain number of turns (varies for each dinosaur), it will die and turn into a corpse (which is a portable item). A stegosaur's corpse is represented by a ')' in the game map, a brachiosaur's corpse is represented by a '(', lastly '%' represents an Allosaur's corpse.

In order to maintain a balanced ecosystem in this game, we've decided that **Stegosaur corpses will remain for 20 turns, Brachiosaur corpses 25 turns, and Allosaur corpses 30 turns.**

In our opinion, Stegosaur corpses should remain in the game for the shortest turns, as compared to the other two dinosaurs. This is because Stegosaurs grow fastest in the map: their eggs are laid fastest (only 10 turns needed), and they grow from baby into adult fastest as well (only 30 turns needed), as compared to the other two dinosaur types. As a result, shortening the duration of the Stegosaur corpse in the game map would prevent overly high chances for Allosaurs to find food, which indirectly prevents Allosaurs from mating and giving birth to more Allosaurs too often, thus giving an ecological balance in space.

In this game, the only way for Brachiosaurs and Allosaurs to die is when they fail to find any food supply, consequently become unconscious and then die after a fixed number of turns of unconsciousness (as compared to Stegosaurs, which could die due to hunger, or being attacked by Allosaurs or a Player with the laser gun). Having said that, Brachiosaur will turn into a corpse after 15 turns of unconsciousness while Allosaur needs 20 turns. We are also aware that eating a Brachiosaur corpse will fill up the Allosuar's food level to its maximum, while eating an Allosaur corpse will increase its food level by 50 only. Due to this, we should remain Brachiosaur corpse for a shorter number of turns. This is to prevent an Allosaur from always choosing to eat Brachiosaur corpses, as compared to Allosaur corpses, leading Allosaur corpses to be less useful (not bringing positive effect) in the game. So, with this proposed number of turns, we are able to ensure that the system is well-maintained.

However, it's also important that neither corpses of the three dinosaurs are left for too short periods of time. As corpses are one of the food supplies to Allosaurs which are carnivores, this provides more (but not overly excessive) opportunities for Allosaurs to eat, leading the chances of them being unconscious (which will possibly lead to extinction) to be greatly reduced.

Pterodactyl corpses act similarly to Stegosaur corpses.

And finally, we've also added an **edibleCount** variable, to indicate the number of turns this corpse can still be eaten. (EatAction would explain how it is used)

CorpseType Enum Class

This class contains enum values of ALLOSAUR, BRACHIOSAUR, STEGOSAUR, PTERODACTYL, and each dinosaur Corpse instance would have its respective CorpseType as capability.

Egg & EggType Enum Class

In the Egg class, if it reaches a specific count, then the egg of the respective dinosaur will hatch and turn into a baby dinosaur. An appropriate message will be displayed on the console to inform the player that a dinosaur just hatched at the particular position on the game map.

Since an Egg instance could be either Stegosaur/Brachiosaur/Allosaur's egg, enumeration is also used here. An Enum class named EggType is created with: ***STEGOSAUR***, ***BRACHIOSAUR***, ***ALLOSAUR, PTERODACTYL***, and is added to be a capability for the respective egg. With that, we can know the type of egg instance by checking its capability.
For example, if **x.hasCapability(EggType.STEGOSAUR)** returns true, then we know that **x** is a stegosaur egg.

Also, we've decided that Stegosaur eggs would hatch after 40 turns, while Brachiosaur eggs 30. Allosaur eggs will hatch after 50 turns, as given in the assignment specs.

MealKit & MealKitType Enum Class

Since a MealKit instance could be either Vegetarian or Carnivore, enumeration is also used here. An Enum class named EggType is created with:***VEGETARIAN***, ***CARNIVORE***. With that, we can access the MealKit instance by **x.hasCapability(MealKit.VEGETARIAN/CARNIVORE)**.

Consequently, the MealKitType Enum class will be used to add these Capabilities to the MealKit instance.

Fish class

This is a class that represents a fish instance in the lake. It currently does not have any specific implementation.

DinosaurGameMap, DinosaurLocation and DinosaurWorld Class
(game package)

We realized the mistake of using the given gameMap as we might provide our own implementation in the gameMap. Hence, to increase completeness of the game, add extra features, and prevent making any edits in the source code that might crash the engine/system, we created new classes named **DinosaurGameMap, DinosaurLocation** and **DinosaurWorld.**

DinosaurGameMap class

In every 10 turns, there is a probability of 20% that the sky might rain. Thus we decided to implement this functionality in the DinosaurMap class. We overridden the **tick()** method and provided condition checking in it. To illustrate, if probability of raining is met, boolean variable **isRained** will be updated to **True**, otherwise remain as False.

DinosaurWorld class
In this class, we've overridden the run() method from World, to do some further checking in each turn of the game. Mainly, in each turn it will check whether:
- The player has decided to quit the game (which will bring him back to the game menu)
- The player has reached the maximum number of moves allowed and if so, whether he has earned enough eco points (Challenge mode)
- The player has earned enough eco points without reaching the maximum number of moves allowed (Challenge mode)

And do the necessary processing (quit the game and display suitable messages). Of course, the usual ticking of all objects in the world is still done.

DinosaurLocation class
For now, there is no specific implementation in this class.

Action *(actions package)*
All classes in this package inherit the Action class.

AttackAction class
AttackAction is created to handle situations where an Actor attacks another Actor; execute() method is overridden to have its own implementations. If an actor

successfully attacks the dinosaur and the dinosaur's hit points reach less than/equal to 0, the dinosaur will die and turn into a corpse. Therefore, to increase code reusability, a call will be made to DieAction.execute() method to process the required changes. We've also added a line of code to heal the actor by the amount of damage done to the attacked target, to cater for the requirement of 'Allosaur increasing their food level by 20, each time it attacks a Stegosaur'. Moreover, by overriding the menuDescription() method, messages will be displayed on the console, informing the player which dinosaur was killed.

BreedAction class

BreedAction is implemented to determine which dinosaur should be the one that is pregnant. Having said that, we again override the execute() method to update the pregnancy count accordingly. This action will be called in each dinosaur's playTurn() method if the dinosaur is well-fed, not pregnant and found a mating partner near them. Besides, we also overridden the menuDescription() method to display a message stating which dinosaur is getting pregnant.

DieAction class

DieAction is created to handle cases where a dinosaur is dead, either its food level remains as 0(unconscious) for a certain number of turns, or it is killed. This action will be called in each dinosaur's playTurn() method. Here, we override the execute() method to perform required processing. To illustrate, if a Stegosaur dies from hunger, its display character on the map will be changed from 'd' to ')' which indicates a stegosaur's corpse. Subsequently, this stegosaur will be removed from the game map and its corpse will be added into the list of items on the current location.

DrinkAction class

This class is created to handle situations when a Dinosaur is thirsty and found a lake to drink water from. We overridden the **execute()** method to update the water level of the respective dinosaur accordingly. Amount of water a dinosaur can consume is different for different types of dinosaurs. For example, a Brachiosaur will increase its water level by 80 if it found a lake nearby. Whereas for the other 3 dinosaurs(Stegosaur, Allosaur & Pterodactyl), it will increment their water level by 30 only. This action will be called in each dinosaur's playTurn() method if the dinosaur's water level is lower than its bare minimum and it finds a lake nearby.

EatAction class

EatAction has **2 constructors,** and their respective input parameters are:

1. Item item, boolean fedByPlayer

2. Actor actor, boolean fedByPlayer

The boolean fedByPlayer is true if it is fed by the player, false otherwise.
Meanwhile, for the constructor that takes in the Item instance, the Item represents the item that the dinosaur will eat. Similarly the one with the Actor instance represents the dinosaur is going to consume an Actor, but this is only used when an Allosaur is going to eat a live Pterodactyl.

This eat action will be called in each dinosaur's playTurn() method if they found a food source near them, or in the FeedAction class when Player decides to feed it. Once again, we overridden the execute() method to handle each eating case and display appropriate messages accordingly. For example, if a stegosaur ate from bushes or fruits on the ground, its hit points would increase by 10. The item eaten will then be removed from the game map and a message: "**StegosaurX ate a fruit on the bush or a fruit laying on ground under a tree.**" will be displayed to inform the player. For cases where the dinosaur is fed by Player, the fed item will be removed from the Player's inventory, but this is handled in the FeedAction class.

Last but not least, for cases of eating corpses, Allosaurs that can devour the whole corpse at once will restore (the corpse's edibleCount * 10) hit points, while Pterodactyls will only restore 10 hit points each time, and after that the edibleCount is deducted by 1. After the Allosaur eats the corpse, the corpse is removed from the game; but for Pterodactyl, the corpse is only removed when the edibleCount = 0.

FeedAction class

This class is used by the Player instance, to handle its feeding action and also the fed dinosaur's eating action (a call to EatAction). FeedAction's constructor accepts an Actor instance, which is the to be fed dinosaur, as input. The execution of this action would first display what the Player has in his/her inventory, then prompt a user input to indicate which food item the Player would like to feed. After that, it will check if the item is suitable to be fed to the dinosaur, eg herbivore dinosaurs can only eat fruits and vegetarian meal kit, and if it is, this action will remove that item from the Player's inventory, and call an EatAction for the dinosaur, with the food item as input parameter. After a successful feed, a message saying the Player fed which dinosaur, along with the output from the EatAction, would be displayed.
In addition, we've used a **do..while** loop, and a **try..catch** block in this class, so that exceptions that are thrown when there is invalid input, would be caught and not

make the game crash, instead make the game keep on running until the user enters a suitable input.

LayEggAction class

An action class for a dinosaur to lay eggs. This class would look at the dinosaur's type (via its capabilities), and determine which egg to be laid in that dinosaur's location. The way to differentiate the type of dinosaur egg is explained here, using Enums. This class would also set the dinosaur's pregnancy status to be false, and its pregnant count back to 0, after laying an egg. Lastly, a message of the dinosaur name and the location of the egg laid is displayed in the output.

PurchaseAction class

PurchaseAction is called when the Player wants to buy items from a vending machine, whenever the Player is adjacent to a vending machine. When executed, firstly the Player's eco points would be shown, followed by the vending machine menu. Then, upon the user entering its option on which item he wishes to purchase, if the player has enough eco points to purchase the item, its eco points would be deducted that item's price amount, and a new instance of that item would be added into the Player's inventory. At last, the display would output a message indicating what item the Player purchased (if any).
Similar to FeedAction, we've used do..while, try..catch here as well to catch exceptions and keep the game running.

SearchFruitAction class

SearchFruitAction handles the action of Player searching and picking fruit on a bush/tree in the same square. Each time this action is called, a String input parameter is required to the constructor, indicating which plant type (a bush or a tree) is being searched for fruit. Searching for fruit has a 60% chance of failing, and we used Math.random() to handle it. An output message of whether the Player successfully searched and picked the fruit is displayed after each execution.

SearchNearestFoodBehaviour class *(game package)*

This behaviour class is a class that finds the nearest food source and moves the dinosaur actor one step closer to that food source. Its getAction method would return a MoveActorAction if a suitable food source is found for the actor, else it would return a null. Its getAction method is also called from each of the 4 dinosaur classes (Stegosaur, Brachiosaur, Allosaur, Pterodactyl) in their playTurn() method.

SearchNearestLakeBehaviour class *(game package)*

This behaviour class is a class that finds the nearest lake and moves the dinosaur actor one step closer to that particular lake. The getAction() method would return a MoveActorAction if there is a lake nearby, otherwise it would return null. Its getAction method is also called from each of the 4 dinosaur classes (Stegosaur, Brachiosaur, Allosaur,Pterodactyl) in their playTurn() method.

Vending Machine class *(game package)*

A vending machine instance is placed on the map in (12,5). It extends the Item class, but is not portable. Its display character is '$'.

Laser Gun class *(game package)*

Extends WeaponItem class. A laser gun deals 70 damage, is portable, and displayed with a '!' character. Used by Player to attack Stegosaurs.

Engine Recommendations

Firstly, we concluded that the introduction of abstract classes is very useful. For example, in the game engine, classes such as Actor,Item,Ground,WeaponItem are declared as abstract classes. With that, each implementation of the child-classes are hidden and only functionality of the abstract class is shown/known. To illustrate, we can easily extend the Actor class and create our own Dinosaur class and provide our own implementations by overriding the necessary methods. Since we can easily extend child classes, code reusability is promoted, the chances of creating repetitive methods that have the same functionality is greatly reduced, leading us to achieve the D.R.Y principle. Besides, the game is now extendable and maintainable in an easier manner.

To add on, we feel that **separation of concerns** is portrayed well in the engine. For example, everything related to the display in the console is only located in the Display class; everything related to the World is in World class, and doesn't mix with other classes, e.g. Location class. Thus, we can clearly see that each class is only in charge of its own concerns and responsibilities, in a way that these classes overlap with other classes as little as possible. This greatly increases cohesion, and reduces coupling. However, we feel that the engine still has high coupling, but we understand that the whole game is large and complex, and that these couplings are inevitable. In addition to this, **encapsulation** is also implemented well in the engine. For example, every class' private attributes that may be required elsewhere have suitable getters and setters for them. This prevents external access and modifications of the class' attributes, without going through suitable methods to perform them.

Besides, we also concluded that the game engine given fulfilled the SOLID principle very well, leading us to extend the game easily. To be specific, **the Single Responsibility Principle** is achieved. For example, the Actor abstract class strictly represents only 1 actor/entity in the game. With that, it gave us flexibility to make changes in the child classes accordingly or extend easily, without worrying about the impact of changes in other classes. Now, classes will be compact and easily-understandable where each class is responsible for a single problem or functionality. Indirectly, it increases readability of code as well. Subsequently, we are aware that whenever we want to introduce new functionalities to the game, we can get it done by simply extending related classes and overriding some functions (NOT modifying). Refactoring is then minimized and it eventually achieved the second principle - **Open Closed Principle**.

Besides that, since most of our classes that represent main objects of the game extend from parent classes in the engine, substituting instances of those subclasses to the engine's parent classes should not be a problem. Thus, we could achieve the third principle: **Liskov's Substitution Principle.** In addition, since we have the privilege to choose which class to extend, we also have the flexibility to implement only the required interfaces. With that, we will only implement necessary methods, reducing code redundancy and achieving **Interface Segregation Principle** simultaneously.

Lastly, along with the concept of abstraction in the engine class, the fifth principle: **Dependency Inversion Principle** is achieved, since any high or low level modules now depend on abstractions. In other words, any future modifications would only need to be done on the abstraction layer (abstract class/interface) without modifying the high level modules. For instance, with the introduction of the Actor class of the engine, if we're now allowed to add features for all Actor instances (i.e. now allowed to modify the engine and its classes), we could just update the Actor class without the need to update/modify all other child classes of the Actor class, since all the child classes are dependent on the parent, the Actor class. This creates flexibility for future edits, and also better code readability and reusability. Not to mention, the DRY Principle is once again obeyed.

The engine class provided to us is a very good example on how principles of Object-Oriented Programming are achieved. However, we think that there are also a few improvements that can be done in the game engine, which makes the engine slightly more complete for future usages.

Firstly, in the **Display class**, instead of having only a **readChar()** method that reads the first character of the user entered string, we should provide more flexibility in reading user input. In other words, the Display class could also have a method that reads in (and process) strings. This extension allows the game/system to have situations where the user can input more than just one character (which could possibly make the game more fun). For example in the Application class' runGame method, if the user chooses Challenge mode, they are required to enter the eco points goal and maximum number of moves allowed. This requires the system to read in more than just one character (unless everything is bound to a single digit, which doesn't make sense), and due to the lack of this functionality, we had to create a new Scanner object just to handle required non-character inputs. If the game/system can read in both 'char' and 'String' (which really is just a list of 'char') and do the necessary processing afterwards, it would be much more complete and flexible.

Moreover, we think the idea of the **hotkey() method**(in the Action class) can be improved. Now, the system might encounter a situation where there are no free characters available to use anymore to option an action. Consequently, the game cannot be continued and affects the gaming experience, as options that can be chosen are now limited. We are also aware that we should avoid using any magic

numbers/strings as well. This is discouraged as it might disrupt other programmers from understanding what each literal number/string stands for. Not only that, it will be more problematic and harder to maintain if the literals appear more than once in the system. Since we are the developers of this game, we can indeed define all the possible actions starting with a named constant, instead of randomly assigning a character in a-z for each action. For example, all attack actions could start with an ‘A’, feeding actions could start with a ‘F’. If an action can be done to more than one different actor, e.g. the Player can feed 2 different dinosaurs, we could append a number, make it as ‘F1’ and ‘F2’, each representing a feed action to a dinosaur. Also, we could sort the actions in ASCII order when displaying the options in the console as well, so that all related actions are grouped and classified together.
Having said that, we can now maintain the game in a better way. By doing so, we can also ensure that the game will be more user-friendly; ensuring all programmers who read the code will be able to identify quickly/have a better understanding of which constant represents which actions.

Last but not least, we think that having a **new method to get all respective objects in the map**, for example a method to get all Actor instances in the map, would be greatly beneficial. As for now, we could only loop through the whole map to know what object lies in that location. This is very inefficient since the same process has to be done again and again in whichever class that has to do it (since we’re not allowed to modify the engine). One example of this in our code is in the SearchNearestFoodBehaviour and SearchNearestLakeBehaviour class. Since we don't have such a method, we have to iterate through the whole map, checking whether that location has a suitable food/lake, and compare with all other locations, to get a minimum distance, i.e. the nearest food/lake.

We think that said method could be included in the World class, since that class is in charge of running the game map. There could be respective methods for getting all Actors, Items, Ground objects like Trees or Bushes etc., by returning a list of the locations of all those objects (or anything suitable). Even though by achieving this we still need to iterate the whole map, which doesn't improve complexity, having a function in charge of this would significantly promote code reusability, readability, and make modification easier as any changes would only need to be done in one function, in one class.